USER MANUAL

PathWave Test Sync Executive 2022

User Manual

PATHWAVE

This User Manual describes the PathWave Test Sync Executive programming environment, which is based on Keysight's Hard Virtual Instrument (HVI) technology. HVI enables you to develop and execute synchronous, real-time operations across multiple instruments. The real-time sequencing and synchronization capabilities of PathWave Test Sync Executive make it a powerful tool for Multi-Input Multi-Output (MIMO) applications that require tight synchronization and real-time control and feedback.

KEYSIGHT TECHNOLOGIES

Notices

Copyright Notice

Manual Part Number

KS2201-90000

Published By

Keysight Technologies
1400 Fountaingrove Parkway
Santa Rosa,
CA 95403-1738

Edition

Edition 2022_U0_00, June, 2022
Keysight Technologies, USA

Regulatory Compliance

This product has been designed and tested in accordance with accepted industry standards, and has been supplied in a safe condition. To review the Declaration of Conformity, go to http://www.keysight.com/go/conformity.

Warranty

THE MATERIAL CONTAINED IN THIS DOCUMENT IS PROVIDED "AS IS," AND IS SUBJECT TO BEING CHANGED, WITHOUT NOTICE, IN FUTURE EDITIONS. FURTHER, TO THE MAXIMUM EXTENT PERMITTED BY APPLICABLE LAW, KEYSIGHT DISCLAIMS ALL WARRANTIES, EITHER EXPRESS OR IMPLIED, WITH REGARD TO THIS MANUAL AND ANY INFORMATION CONTAINED HEREIN, INCLUDING BUT NOT LIMITED TO THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. KEYSIGHT SHALL NOT BE LIABLE FOR ERRORS OR FOR INCIDENTAL OR CONSEQUENTIAL DAMAGES IN CONNECTION WITH THE FURNISHING, USE, OR PERFORMANCE OF THIS DOCUMENT OR OF ANY INFORMATION CONTAINED HEREIN. SHOULD KEYSIGHT AND THE USER HAVE A SEPARATE WRITTEN AGREEMENT WITH WARRANTY TERMS COVERING THE MATERIAL IN THIS DOCUMENT THAT CONFLICT WITH THESE TERMS, THE WARRANTY TERMS IN THE SEPARATE AGREEMENT SHALL CONTROL.

KEYSIGHT TECHNOLOGIES DOES NOT WARRANT THIRD-PARTY SYSTEM-LEVEL (COMBINATION OF CHASSIS, CONTROLLERS, MODULES, ETC.) PERFORMANCE, SAFETY, OR REGULATORY COMPLIANCE, UNLESS SPECIFICALLY STATED.

Technology Licenses

The hardware and/or software described in this document are furnished under a license and may be used or copied only in accordance with the terms of such license.

U.S. Government Rights

The Software is "commercial computer software," as defined by Federal Acquisition Regulation ("FAR") 2.101. Pursuant to FAR 12.212 and 27.405-3 and Department of Defense FAR Supplement ("DFARS") 227.7202, the U.S. government acquires commercial computer software under the same terms by which the software is customarily provided to the public. Accordingly, Keysight provides the Software to U.S. government customers under its standard commercial license, which is embodied in its End User License Agreement (EULA), a copy of which can be found at http://www.keysight.com/find/sweula. The license set forth in the EULA represents the exclusive authority by which the U.S. government may use, modify, distribute, or disclose the Software. The EULA and the license set forth therein, does not require or permit, among other things, that Keysight: (1) Furnish technical information related to commercial computer software or commercial computer software documentation that is not customarily provided to the public; or (2) Relinquish to, or otherwise provide, the government rights in excess of these rights customarily provided to the public to use, modify, reproduce, release, perform, display, or disclose commercial computer software or commercial computer software documentation. No additional government requirements beyond those set forth in the EULA shall apply, except to the extent that those terms, rights, or licenses are explicitly required from all providers of commercial computer software pursuant to the FAR and the DFARS and are set forth specifically in writing elsewhere in the EULA. Keysight shall be under no obligation to update, revise or otherwise modify the Software. With respect to any technical data as defined by FAR 2.101, pursuant to FAR 12.211 and 27.404.2 and DFARS 227.7102, the U.S. government acquires no greater than Limited Rights as defined in FAR 27.401 or DFAR 227.7103-5 (c), as applicable in any technical data.

Safety Notices

CAUTION

A CAUTION notice denotes a hazard. It calls attention to an operating procedure, practice, or the like that, if not correctly performed or adhered to, could result in damage to the product or loss of important data. Do not proceed beyond a CAUTION notice until the indicated conditions are fully understood and met.

WARNING

A WARNING notice denotes a hazard. It calls attention to an operating procedure, practice, or the like that, if not correctly performed or adhered to, could result in personal injury or death. Do not proceed beyond a WARNING notice until the indicated conditions are fully understood and met.

The following safety precautions should be observed before using this product and any associated instrumentation.

This product is intended for use by qualified personnel who recognize shock hazards and are familiar with the safety precautions required to avoid possible injury. Read and follow all installation, operation, and maintenance information carefully before using the product.

WARNING

If this product is not used as specified, the protection provided by the equipment could be impaired. This product must be used in a normal condition (in which all means for protection are intact) only.

The types of product users are:

- Responsible body is the individual or group responsible for the use and maintenance of equipment, for ensuring that the equipment is operated within its specifications and operating limits, and for ensuring operators are adequately trained.
- Operators use the product for its intended function. They must be trained in electrical safety procedures and proper use of the instrument. They must be protected from electric shock and contact with hazardous live circuits.
- Maintenance personnel perform routine procedures on the product to keep it operating properly (for example, setting the line voltage or replacing consumable materials). Maintenance procedures are described in the user documentation. The procedures explicitly state if the operator may perform them. Otherwise, they should be performed only by service personnel.
- Service personnel are trained to work on live circuits, perform safe installations, and repair products. Only properly trained service personnel may perform installation and service procedures.

WARNING

Operator is responsible to maintain safe operating conditions. To ensure safe operating conditions, modules should not be operated beyond the full temperature range specified in the Environmental and physical specification. Exceeding safe operating conditions can result in shorter lifespans, improper module performance and user safety issues. When the modules are in use and operation within the specified full temperature range is not maintained, module surface temperatures may exceed safe handling conditions which can cause discomfort or burns if touched. In the event of a module exceeding the full temperature range, always allow the module to cool before touching or removing modules from chassis.

Keysight products are designed for use with electrical signals that are rated Measurement Category I and Measurement Category II, as described in the International Electrotechnical Commission (IEC) Standard IEC 60664. Most measurement, control, and data I/O signals are Measurement Category I and must not be directly connected to mains voltage or to voltage sources with high transient over-voltages. Measurement Category II connections require protection for high transient over-voltages often associated with local AC mains connections. Assume all measurement, control, and data I/O connections are for connection to Category I sources unless otherwise marked or described in the user documentation.

Exercise extreme caution when a shock hazard is present. Lethal voltage may be present on cable connector jacks or test fixtures. The American National Standards Institute (ANSI) states that a shock hazard exists when voltage levels greater than 30V RMS, 42.4V peak, or 60VDC are present. A good safety practice is to expect that hazardous voltage is present in any unknown circuit before measuring.

Operators of this product must be protected from electric shock at all times. The responsible body must ensure that operators are prevented access and/or insulated from every connection point. In some cases, connections must be exposed to potential human contact. Product operators in these circumstances must be trained to protect themselves from the risk of electric shock. If the circuit is capable of operating at or above 1000V, no conductive part of the circuit may be exposed.

Do not connect switching cards directly to unlimited power circuits. They are intended to be used with impedance-limited sources. NEVER connect switching cards directly to AC mains. When connecting sources to switching cards, install protective devices to limit fault current and voltage to the card.

Before operating an instrument, ensure that the line cord is connected to a properly-grounded power receptacle. Inspect the connecting cables, test leads, and jumpers for possible wear, cracks, or breaks before each use.

When installing equipment where access to the main power cord is restricted, such as rack mounting, a separate main input power disconnect device must be provided in close proximity to the equipment and within easy reach of the operator.

For maximum safety, do not touch the product, test cables, or any other instruments while power is applied to the circuit under test. ALWAYS remove power from the entire test system and discharge any capacitors before: connecting or disconnecting cables or jumpers, installing or removing switching cards, or making internal changes, such as installing or removing jumpers.

Do not touch any object that could provide a current path to the common side of the circuit under test or power line (earth) ground. Always make measurements with dry hands while standing on a dry, insulated surface capable of withstanding the voltage being measured.

The instrument and accessories must be used in accordance with its specifications and operating instructions, or the safety of the equipment may be impaired.

Do not exceed the maximum signal levels of the instruments and accessories, as defined in the specifications and operating information, and as shown on the instrument or test fixture panels, or switching card.

When fuses are used in a product, replace with the same type and rating for continued protection against fire hazard.

Chassis connections must only be used as shield connections for measuring circuits, NOT as safety earth ground connections.

If you are using a test fixture, keep the lid closed while power is applied to the device under test. Safe operation requires the use of a lid interlock.

Instrumentation and accessories shall not be connected to humans.

Before performing any maintenance, disconnect the line cord and all test cables.

To maintain protection from electric shock and fire, replacement components in mains circuits – including the power transformer, test leads, and input jacks – must be purchased from Keysight. Standard fuses with applicable national safety approvals may be used if the rating and type are the same. Other components that are not safety-related may be purchased from other suppliers as long as they are equivalent to the original component (note that selected parts should be purchased only through Keysight to maintain accuracy and functionality of the product). If you are unsure about the applicability of a replacement component, call an Keysight office for information.

WARNING

No operator serviceable parts inside. Refer servicing to qualified personnel. To prevent electrical shock do not remove covers. For continued protection against fire hazard, replace fuse with same type and rating.

PRODUCT MARKINGS:

The CE mark is a registered trademark of the European Community.

Australian Communication and Media Authority mark to indicate regulatory compliance as a registered supplier.

This symbol indicates product compliance with the Canadian Interference-Causing Equipment Standard (ICES-001). It also identifies the product is an Industrial Scientific and Medical Group 1 Class A product (CISPR 11, Clause 4).

South Korean Class A EMC Declaration. This equipment is Class A suitable for professional use and is for use in electromagnetic environments outside of the home. A 급 기기 (업무용 방송통신기자재) 이 기기는 업무용 (A 급) 전자파적합기기로서 판 매자 또는 사용자는 이 점을 주 의하시기 바라 며 , 가정외의 지역에서 사용하는 것을 목적으 로 합니다.

This product complies with the WEEE Directive marketing requirement. The affixed product label (above) indicates that you must not discard this electrical/electronic product in domestic household waste.

Product Category: With reference to the equipment types in the WEEE directive Annex 1, this product is classified as “Monitoring and Control instrumentation” product. Do not dispose in domestic household waste. To return unwanted products, contact your local Keysight office.

This symbol indicates the instrument is sensitive to electrostatic discharge (ESD). ESD can damage the highly sensitive components in your instrument. ESD damage is most likely to occur as the module is being installed or when cables are connected or disconnected. Protect the circuits from ESD damage by wearing a grounding strap that provides a high resistance path to ground. Alternatively, ground yourself to discharge any built-up static charge by touching the outer shell of any grounded instrument chassis before touching the port connectors.

This symbol on an instrument means caution, risk of danger. You should refer to the operating instructions located in the user documentation in all cases where the symbol is marked on the instrument.

This symbol indicates the time period during which no hazardous or toxic substance elements are expected to leak or deteriorate during normal use. Forty years is the expected useful life of the product.

Contents

KS2201A - PathWave Test Sync Executive User Manual

This User Manual describes the PathWave Test Sync Executive programming environment, which is based on Keysight's Hard Virtual Instrument (HVI) technology. HVI enables you to develop and execute synchronous, real-time operations across multiple instruments. The real-time sequencing and synchronization capabilities of PathWave Test Sync Executive make it a powerful tool for Multi-Input Multi-Output (MIMO) applications that require tight synchronization and real-time control and feedback.

NOTE PathWave Test Sync Executive (KS2201A) is **not compatible** with the older M3601A. You cannot use them together and they cannot run the same Sequences.

Chapter 1: Introduction

This chapter introduces Keysight KS2201A, PathWave Test Sync Executive and HVI technology.

Keysight PathWave Test Sync Executive Overview

PathWave Test Sync Executive is a programming environment based on Keysight's Hard Virtual Instrument (HVI) technology, that enables you to develop and execute synchronous real-time operations across multiple instruments.

The real-time sequencing and synchronization capabilities of PathWave Test Sync Executive make it a powerful tool for Multi-Input Multi-Output (MIMO) applications that require tight synchronization and real-time control and feedback. For example:

- Radar.
- Bit error testing.
- Communication systems.
- Massive-scale quantum physics experiments.

PathWave Test Sync Executive supports:

- Multi-chassis configuration.
- HVI sequence design using an Application Programming Interface (API) for Python.
- Programming of multiple instruments.
- Execution of time-deterministic sequences of operations.
- Precision synchronization and execution.

About HVI Technology

HVI technology enables you to program one or more instruments to execute time-deterministic sequences of operations with precise synchronization. It achieves this by deploying a code executable onto the hardware of each instrument. This executes on an HVI Engine, which is an IP block that is integrated into the instrument. The code executes on these Engines in parallel, across multiple instruments.

The user-defined hardware operation of a group of instruments is called a Hard Virtual Instrument or just HVI. The sequences of operations or instructions executed by the HVI engines are called HVI Sequences. The operations and instructions that make up sequences are known as HVI Statements.

When creating an HVI, you can include any instrument that has HVI support. For example, the Keysight M3xxxA family of PXI instruments is one product family with HVI support, the M5302A instrument also has HVI support. This User Manual includes code examples of the HVI Instrument-specific API that complement the code examples that explain the functionality of the HVI-native API.

HVI Application Programming Interface

The HVI API is the set of programming classes and methods that enable you to create and program an HVI instance. HVI API 2022 supports the Python and C# languages. Unless otherwise noted, this document refers to the Python API in explanations.

Python Help

A complete description of the HVI Python API is provided in the help file provided with the PathWave Test Sync Executive installer. It is found inside the installation directory for PathWave Test Sync Executive inside the *api\python\Help* subdirectory, by default this is:

```
C:\Program Files\Keysight\PathWave Test Sync Executive 2022\api\python\Help
```

Alternatively, you can enter *Python API Help* into the Windows Search.

C# Help

The HVI API documentation for C# is located at:

```
C:\Program Files\Keysight\PathWave Test Sync Executive 2022\api\dotNet\Help
```

API Use Model: The HVI-native API and the HVI Instrument Specific API

Each instrument extends the HVI API functionality with an instrument specific API. The HVI API is common to all products and only the instrument specific HVI API is different, depending on the instrument. It is important to differentiate between the HVI-native API features and the instrument-specific extensions. The extensions enable a heterogeneous array of instruments and resources to coexist in a common framework.

The HVI-native API exposes all HVI functions and is a common API for all products. It defines the base interfaces and classes that are used to create an HVI, control the hardware execution flow, and operate with data, triggers, events and actions, but it alone does not include the ability to control instrument-specific operations. The HVI API defines the hard virtual instrumentation framework, and it is the job of the instrument-specific HVI API extensions to enable instrument functions in an HVI. These functions are exposed by the instrument-specific add-on definitions. This is done by an HVI instrument add-on API provided by each instrument that describes the instrument-specific resources and operations that can be executed or used within HVI sequences:

HVI instrument-specific definitions are listed in your Instrument documentation. For a list of supported instruments see Appendix A: Supported Instruments.

Chapter 2: Install PathWave Test Sync Executive

This chapter explains how to install PathWave Test Sync Executive and related required components.

It contains the following sections:

System Requirements

This section describes the system requirements for PathWave Test Sync Executive.

PathWave Test Sync Executive Installation Requirements

To install PathWave Test Sync Executive you require the following:

- Python 3.7.x or higher, 64-bit.
- Keysight PathWave Test Sync Executive installer.

To install these, see Install Main Components.

Additional Components Required

To run PathWave Test Sync Executive with hardware, you require:

- One or more PXIe chassis.
- One or more PXIe instruments.
- Associated software, libraries, drivers, and firmware.

Chassis

PathWave Test Sync Executive is compatible with any PXIe chassis, however Keysight recommends the following Keysight chassis so you can make use of their capabilities and multi-instrument and multi-chassis scalability:

- M9019A.
- M9018B.
- M9010A.
- M9046A.

These chassis include an enhanced PXI trigger bridge that provides the capabilities required by PathWave Test Sync Executive to provide support for multi-segment/chassis operation. You can use other chassis without limitation for single segment operation, and you can also use other chassis for multi-segment/multi-chassis operations, but these impose limitations on the complexity of the HVI sequences that you can execute.

For most chassis, the enhanced PXI trigger bridge functionality is delivered by a firmware update, see your chassis user manual for details. The PathWave Test Sync Executive programming examples show how to verify the correct firmware version for specific chassis. The programming examples are described in Appendix B: Additional Documentation and Examples .

NOTE The Programming Examples are often updated so ensure you check for the latest versions.

Instruments

PathWave Test Sync Executive works with a number of PXIe instruments.

For more information see the *PathWave Test Sync Executive Release Notes* and Appendix A: Supported Instruments.

Older versions of HVI technology

PathWave Test Sync Executive (KS2201A) and the previous version M3601A, are not compatible. You cannot use them together.

If you use M3601A, the additional components required by HVI use different versions, so they must be reinstalled every time you change between running M3601 and KS2201A.

Install Main Components

This section explains how to install the main components of PathWave Test Sync Executive, it contains the following sections:

1. Install Python 3.7.x, 64-bit.
2. Install PathWave Test Sync Executive.
3. Manual Installation of Python APIs.

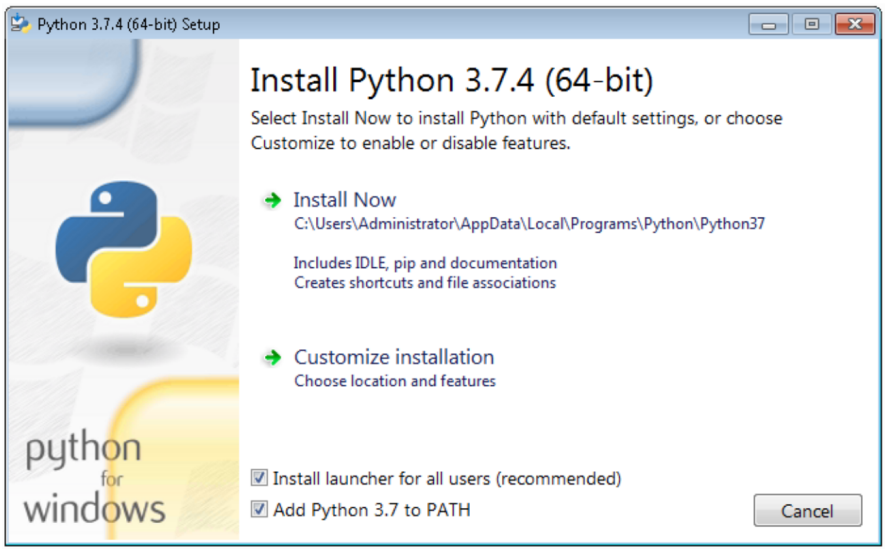
NOTE PathWave License Manager must not be running when you install PathWave Test Sync Executive.

If PathWave License Manager is running, you must close it before installing the main components.

1: Install Python

PathWave Test Sync Executive requires 64-bit Python. Versions 3.7, 3.8, 3.9, and 3.10 are supported along with their sub-versions. Multiple versions can also be supported.

1. Download the Python installer from the Python web site: python.org.
2. Run the installer.
   a. Add Python 3.x to the PATH system Variable. To do this, ensure the check box **Add python 3.x to PATH** is checked. This is shown in the following screenshot:



2: Install PathWave Test Sync Executive

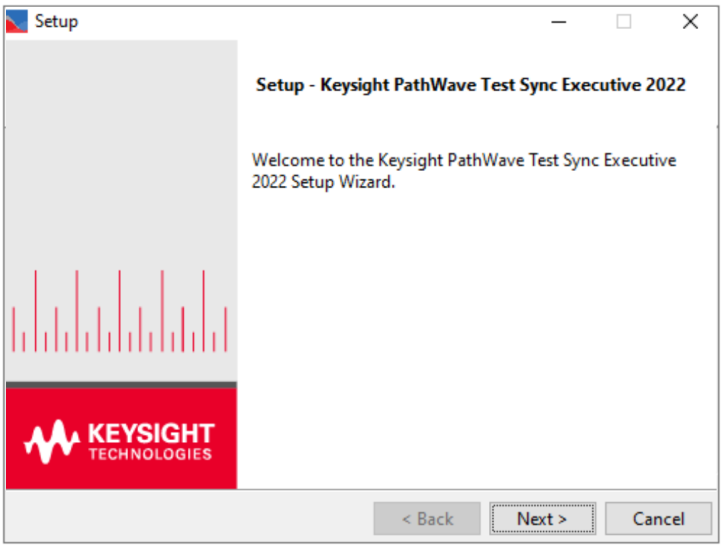
Use the following procedure to install PathWave Test Sync Executive:

NOTE You must install Python 64-bit before installing PathWave Test Sync Executive.

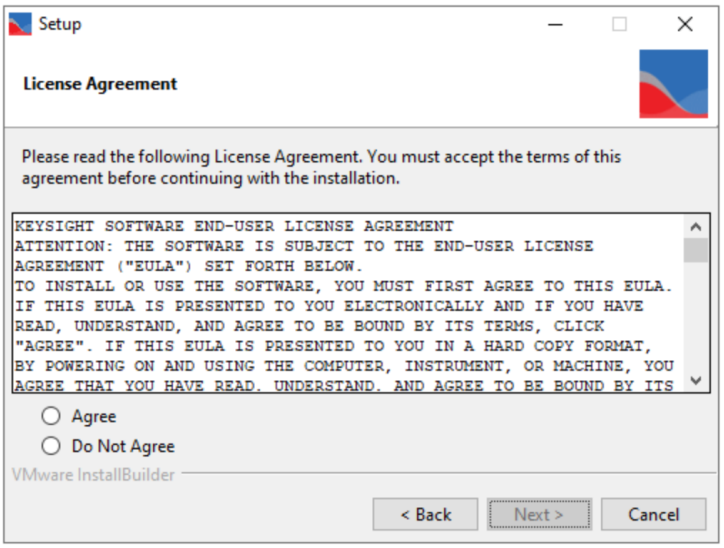
If PathWave License Manager is running, you must close it before installing PathWave Test Sync Executive.

Execute the installer file:

The **Setup** screen is shown:



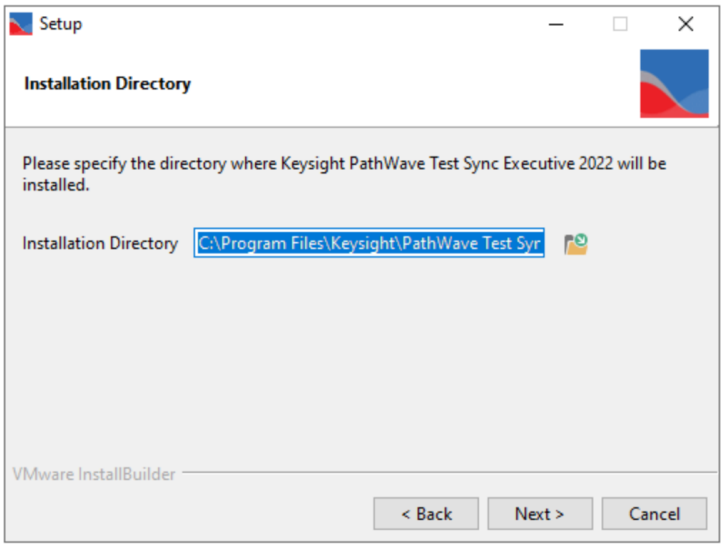
The next screen is the License Agreement screen. You must accept the license to continue:



You can change the installation directory on the Installation Directory screen.

By default, PathWave Test Sync Executive is installed to:

```
C:\Program Files\Keysight\PathWave Test Sync Executive 2022
```

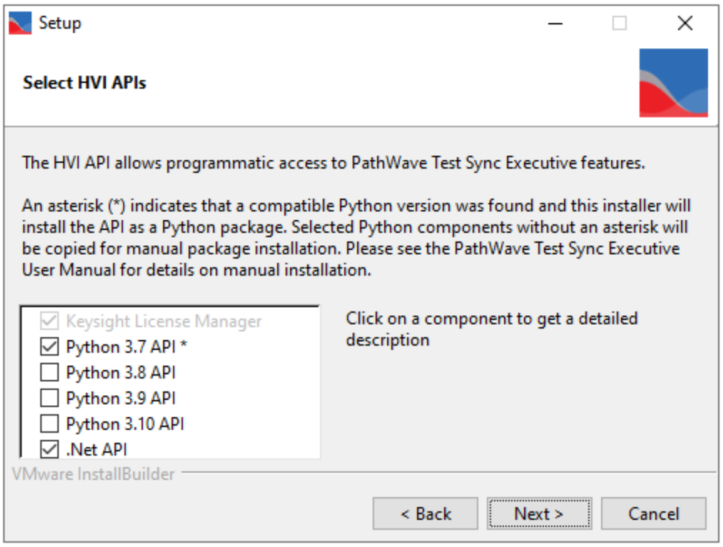


The Select HVI APIs screen enables you to select the Python API versions you want to install.

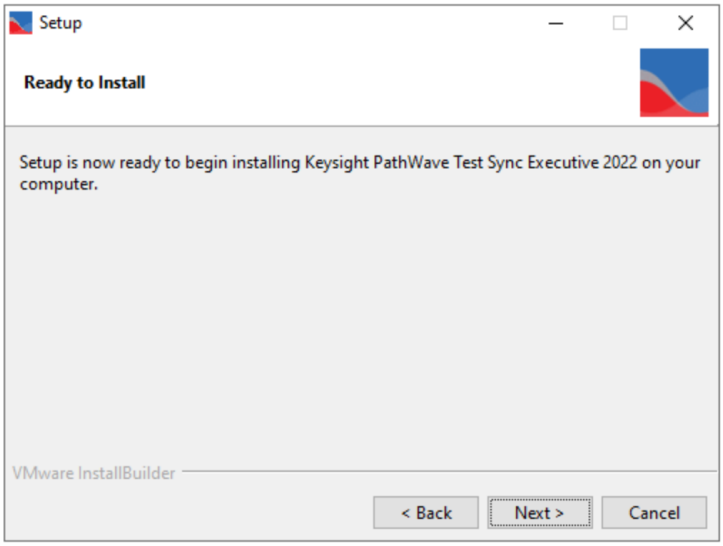
If a Python version component is marked with an asterisk and selected, the installer will install the Python package.

If the Python version component is *not* marked with an asterisk, but is selected with a check mark, an additional step is required; see Manual Installation of Python APIs below.

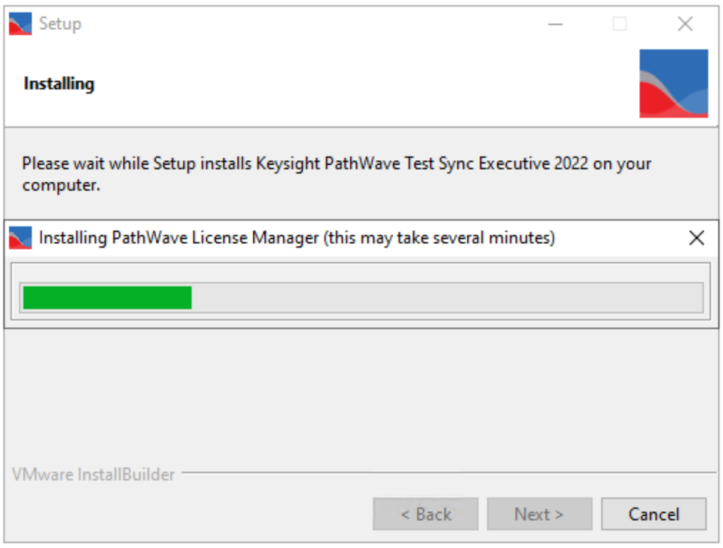
Required components are selected by default and you cannot de-select them.



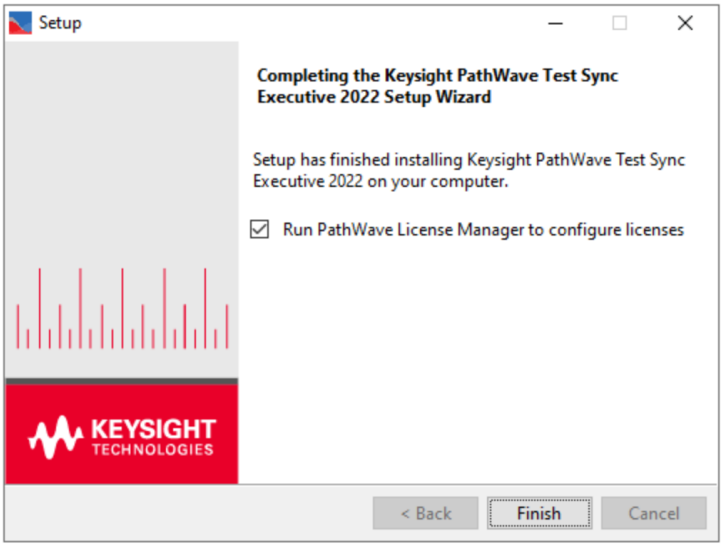
When you have selected the components, the next screen is Ready to Install. Select Next to install PathWave Test Sync Executive.



The Installer first installs the License manager. It then installs PathWave Test Sync Executive:



The following screen is shown when the installer has completed installing: Select Finish to close the installer.



Manual Installation of Python APIs

If you selected Python APIs when installing PathWave Test Sync Executive that were not automatically installed, you can complete the installation process with the `pip` command.

For example, to install Python APIs for Python 3.9, type the following command at a command prompt:

```
py -3.9 -m pip install "C:\Program Files\Keysight\PathWave Test Sync Executive 2022\api\Python\Python39"
```

Install Additional Components

To use PathWave Test Sync Executive, you require both hardware and software.

To work with PathWave Test Sync Executive, instruments and chassis require minimum specific software and firmware versions. These are listed on line at: Instrument and Chassis Software and Firmware Requirements for KS2201A .

Ensure you have all the following components and they are all up to date:

- Keysight IO Libraries.
- Keysight Instrument Drivers, Libraries, and Software Front Panel.
- Keysight Instrument FPGA Firmware.
- Keysight Chassis Family Driver.
- Keysight Chassis Driver and Firmware.

Install Keysight IO Libraries

Install the IO Libraries. These are available at Keysight IO Libraries Suite.

Install Keysight Instrument Drivers, Libraries, and Software Front Panel

To install the instrument drivers and libraries, install the software for your instruments:

- For the M5302A instrument see: M5302A Software.
- For the M3xxxA instruments see: Keysight SD1 Software.

NOTE Ensure you check the driver release notes, so that your drivers that are compatible with the version of PathWave Test Sync Executive you have installed.

Update Keysight Instrument FPGA Firmware

You can update the FPGA firmware of your PXI instruments from your Software Front Panel. For information about how to install SW and FPGA firmware for Keysight instruments, see the instrument documentation:

These are available at Keysight PXI Products.

NOTE Ensure you check the firmware release notes, so that you install firmware that is compatible with the version of PathWave Test Sync Executive you have installed.

Install Keysight Chassis Family Driver

Install the Chassis Family Driver, which is available at Keysight PXI Chassis. When you install the Keysight Chassis Family Driver, PXIe Chassis Software Front Panel software is automatically installed.

Update Keysight Chassis Firmware

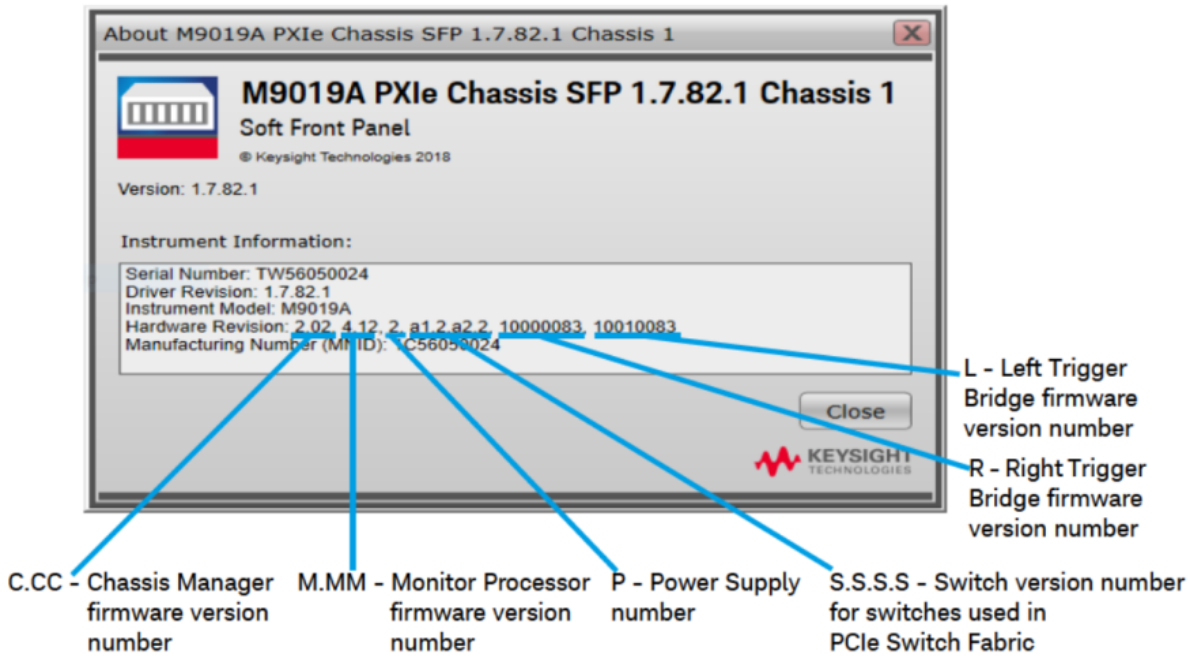
In PXIe Chassis Software Front Panel, you can:

- Check the chassis firmware version in the help window.
- Update the chassis firmware with the Utilities window of PXIe Chassis SFP.

You can use the Utilities window of PXIe Chassis SFP to update the chassis firmware. For more information about updating Chassis firmware, see *PXIeChassisFirmwareUpdateGuide.pdf* at Keysight PXI Chassis.

NOTE Ensure you check the firmware release notes, so that you install firmware that is compatible with the version of PathWave Test Sync Executive you have installed.

The following screenshot shows an example of the chassis firmware version shown in the help window of the PXIe Chassis SFP. In this case the chassis is a Keysight Chassis model M9019A.



The following screenshot shows a breakdown of components of different versions of the M9019A chassis firmware:

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

Chapter 3: Installing Licenses

This chapter provides a brief introduction to PathWave Test Sync Executive licensing. It contains the following sections:

PathWave Test Sync Executive License Requirements

Each instrument used in your HVI implementation must be licensed to be used with PathWave Test Sync Executive.

There are 2 types of licensing for instruments:

1. For instruments with no -HVx option installed, you require 1 license for each instrument (including Sync Modules)
2. For instruments with the -HVx option (-HV1 or -HV2) installed, a single license covers all of the instruments with the -HVx option in the same chassis.

The following table shows an example of the number of licenses required for a single chassis system:

Chassis	Number of Instruments without -HVx option	Number of Instruments with -HVx option	Licenses required
A	1	4	2 • 1 license for the instrument without the -HVx option. • 1 license for the 4 instruments with the -HVx option.

The following table shows an example of the number of licenses required for a 3 chassis system:

Chassis	Number of Instruments without -HVx option	Number of Instruments with -HVx option	Licenses required
A	1	4	2 • 1 license for the instrument without the -HVx option. • 1 license for the 4 instruments with the -HVx option.
B	4	0	4 • 1 license each for the 4 instruments without the -HVx option.
C	2	4	3 • 1 license each for the 2 instruments without the -HVx option. • 1 license for all the instruments with the -HVx option.
Total licenses required			9

NOTE The `-HVx` option was previously required to be purchased for an instrument to be used with PathWave Test Sync Executive.

The -HVx option is now deprecated, but existing instruments with the -HVx option are still supported.

- Keysight M3xxxA PXI Instruments used the `-HV1` option.
- Keysight M5302A Digital I/O instruments previously used the `-HV2` option.
- Keysight M9415A VXT Vector Transceiver uses the `-HV2` option.

Licenses and Processes

All HVI instances running in the same process share the same licenses, but HVI instances running in different processes require different licenses.

For example, if you have 3 HVI instances running in a single process, the licenses are reused.

The following table shows the number of licenses required for scenarios where these are 1 or 3 processes:

Description	HVI instance 1	HVI instance 2	HVI instance 3	Licenses required
3 HVI instances in the same process	3	6	10	10
3 HVI instances in 3 different processes	3	6	10	19

Supported Licensing Modes

The following types of licenses are supported:

Commercial licenses:

- Node-Locked, perpetual and 6, 12, 24, and 36 months, subscription.
- USB Portable, perpetual and 6, 12, 24, and 36 months, subscription.
- Floating/Networked, perpetual and 6, 12, 24, and 36 months, subscription.
- Transportable, perpetual and 6, 12, 24, and 36 months, subscription.

Trial licenses:

- 30 days Node-locked.

NOTE

- To obtain a trial or commercial license, see the product download page.
- As part of the licensing process you will require a Host ID (probably a Mac address) for your workstation. The product license manager might display this, if not, the help or documentation for the license manager shall tell you how to obtain a Host ID.

Transportable Licenses

If you want to reconfigure your systems so a different number of chassis are used, you can use a transportable license . These enable you to move your licenses between systems without any need to contact Keysight, so you don't have to keep buying new licenses.

For example, say you have two systems: one with three chassis and a second system with two chassis. If you want to move the third chassis from the first system to the second, the second system will require a third license. The first system has three licenses, but it shall no longer require all three. A transportable license enables you to move the third license from the first system to the second system. You can then use the new configuration without having to buy a new license.

The Licensing Process

The Keysight licensing process uses the following steps:

1. Purchase and fulfillment

   For most Keysight licensed product options, your entitlement certificate is sent to you as a PDF attachment via email immediately after your purchase. In some cases, you receive a paper copy of your certificate with your purchased product. The licensed product options may be software products or upgraded features of an instrument.

2. Getting a license

   Using the entitlement certificate you received when you ordered, you can request your licenses on the Keysight Software Manager web site. To do this, you'll need to choose a host instrument or PC, and provide its identifying information (the Host ID) when you request your licenses. Once you begin the process, Keysight Software Manager will guide you step by step through requesting your licenses and you will receive the license files via email.

   You might need to create a *myKeysight* login when you first go to the Keysight Software Manager site, and you will need to log in anytime you go to the site.

3. Installing your license

   To enable the licensed software, after you receive a license file from Keysight Software Manager, you must install it on your instrument or computer or on a central licensing server accessible from your instrument or computer. If you are installing node-locked or transportable licenses on the same local PC where you execute KS2201A, ensure you place your license files in a public folder, for example, **`C:\Users\public\folder_name`**.

   To install the license:

   1. Install PathWave Test Sync Executive.

   2. Use PathWave License Manager to install your license. The installation process is described in the email that comes with your license.

Installing Licenses with PathWave License Manager

You can install licenses from the PathWave License Manager. This is installed when you install Keysight PathWave Test Sync Executive. You can use a local license on your computer or a floating license from a license server.

Full details describing how to install licenses are provided by email when you purchase a license.

If you are upgrading without purchasing a new license, have a more complex setup, or did not get a licensing email, see the Licensing Quick Start Guide, this provides comprehensive information about the licensing process and how to solve problems.

NOTE If you are upgrading from a previous version of PathWave Test Sync Executive that used a different license manager, Keysight recommends that you keep the old license manager installed.

Potential Conflicts Between License Managers of Different HVI Software

Some previous versions of KS2201A software used a different license manager. Specifically:

- KS2201A Pathwave Test Sync Executive 2021 Release and later use PathWave License Manager (PLM).
- KS2201A Pathwave Test Sync Executive 2020 Update 1 Release uses PathWave License Manager (PLM).
- KS2201A PathWave Test Sync Executive 2020 Release uses Keysight License Manager 6.

The license managers described above are compatible with each other and they can detect and show the licenses installed using the other license managers. For node-locked or transportable licenses, conflicts can arise if any licenses were not installed in a a public folder, for example, **`C:\Users\public\folder_name .`** In this case, the license must be reinstalled from scratch using the license manager of the product the license belongs to.

If you are moving from one HVI software to another version that uses a different license manager, to update the floating license installation on your license server see the instructions provided.

NOTE

- If you need to uninstall any PathWave Test Sync Executive software, always use the provided software uninstaller. Manually uninstalling a license manager can cause corruption to other license managers.
- If you have licenses located in user-specific locations (such as **`C:\Users\fred\Desktop`**), these licenses may not be accessible to the license service created by PathWave License Manager. Using the license manager provided with the appropriate product, remove and reinstall such licenses in a generally accessible location, such as **`C:\Users\public`**

Troubleshooting the License Installation

If you have difficulties with installing or using your licenses see Licensing Quick Start Guide. If the problem persists, please contact Keysight Tech Support and share the log files.

Log files are saved by PathWave License Manager in:

`C:\ProgramData\Keysight\Licensing\Log`

Chapter 4: HVI Elements

This chapter describes the elements that make up an HVI.

It contains the following sections:

About Instruments

Instruments are modules or cards that can capture or generate various kinds of electronic signals. Many kinds of instruments are available with different kinds of functions.

Different kinds of instruments can perform various functions with electronic signals:

- Measure signals.
- Record signals.
- Perform signal analysis.
- Perform signal conditioning.

Some types of instruments can generate different kinds of outputs:

- Signals.
- Voltages.
- Pulses.
- Arbitrary waveforms.
- Digital outputs.

Instruments can be supplied as modules or cards that fit into a chassis. The chassis enables you to fit multiple modules together. The instruments in a chassis are synchronized to a common digital clock reference that is shared by all the instruments. The chassis also offers shared triggering and communication resources.

For this User Manual, the specific instruments referred to are PXI modular instruments that are inserted into a PXI chassis.

For a full list of Keysight instruments, see Keysight.com.

About PathWave Test Sync Executive

PathWave Test Sync Executive enables you to program multiple instruments together. They operate together, tightly orchestrated with other instruments, so they behave like a single instrument.

PathWave Test Sync Executive enhances individual instruments by enabling them to:

- Execute real-time sequences of operations with full time determinism.
- Precisely synchronize instrument operations.
- Fast, real-time hardware exchange of information and decisions between instruments.

You define a new virtual instrument made up of a combination of instruments. This is known as a Hard Virtual Instrument (HVI). Once the HVI resources are defined, you can program multiple instruments to work together as if they were a single instrument.

To program the HVI, you write an application using the HVI API. When you run your application, it generates the HVI instance and the binary code that is executed by the hardware in the instruments.

When creating an HVI, you can include any instrument that supports PathWave Test Sync Executive, such as Keysight's M3xxxA family of PXI instruments.

Each instrument that supports PathWave Test Sync Executive has specific instructions that enable you to use its functionalities within HVI. These instructions are documented in the instrument documentation.

HVI API Language Support

The HVI API is the set of programming classes and methods that enable you to create and program an HVI instance. PathWave Test Sync Executive 2021 and above supports the Python and C# languages.

The C# API is similar to the Python API except for the following differences:

- Class names are in camel case, that is, the beginning of individual words are capitalized.
- Variable names are also in camel case, except the first letter of the first word is not capitalized.
- There are no spaces, underscores, or dashes between words in class names.
- The first letter of methods and functions is capitalized.

The following table shows examples in Python and C#:

Type	Python	C#
Type names	`SystemDefinition`	`SystemDefinition`
Variables	`multi_seq_block_1`	`multiSeqBlock1`
Methods	`add_sync_multi_sequence_block()`	`AddSyncMultiSequenceBlock()`

The following blocks of Python and C# code are equivalent:

Python code:

```
# Add a sync multi-sequence block:
multi_seq_block_1 = sync_while.sync_sequence.add_sync_multi_sequence_block("multi_seq_block_1",
210)
```

C# code:

```
// Add a sync multi-sequence block
var multiSeqBlock1 = syncWhile.SyncSequence.AddSyncMultiSequenceBlock("multiSeqBlock1", 210);
```

A complete description of the HVI Python API is provided in the help file installed with the PathWave Test Sync Executive installer.
It is found inside the installation directory for PathWave Test Sync Executive inside the *api\python\Help* subdirectory, by default this is:

C:\Program Files\Keysight\PathWave Test Sync Executive 2022\api\python\Help

Alternatively, you can enter *Python API Help* into the Windows Search.

The HVI API documentation for C# is located at:

C:\Program Files\Keysight\PathWave Test Sync Executive 2022\api\dotNet\Help

HVI API Use Model

This section describes the HVI API use model, and the steps it involves.

HVI uses a program-within-a-program model, that is, HVI can be seen as a real-time hardware program that runs within a software program.

HVI Use Model Steps

To use the HVI API, your application must follow a series of steps to define and run an HVI instance. These steps are broadly defined by three different classes within the HVI API:

1. SystemDefinition.
2. Sequencer.
3. Hvi.

1 SystemDefinition

You use this class to define all the instrument and platform resources that are required to set up the HVI.

You use this class to define:

- Chassis.
- Interconnects.
- Clocks.
- Synchronous signals.
- Trigger routing.

You also use this class to define the resources that are available on the instruments:

- Engines - IP blocks in the FPGA or instrument hardware that executes HVI sequences.
- Actions - these initiate instrument-specific operations.
- Events - these indicate instrument-specific operations have occurred.
- Triggers - signals used to communicate between instruments.

When you have defined these resources, you must register them within the relevant collections. Collections are special classes that associate resources with individual Engines, so that you can use the resources on those Engines.

2 Sequencer

You use the Sequencer class to program and compile your sequences:

- You add instructions and operations known as statements to sequences. These can be synchronized across instruments or local to a specific instrument.
- You also add and use HVI registers within this class. Registers are small, fast memories on the HVI engines that you can use as program Variables.
- Once you have defined all the Sequences that define your HVI, you must compile it. The compilation process returns a instance of the Hvi class.

3 Hvi

Hvi is the runtime or executable object. With this object, you load the HVI sequences into the relevant engines and execute them.
This object also enables you to interact with the hardware resources assigned to the HVI and initialize all resources before the actual execution happens.

Execution Flow of the HVI

When you run your application, the HVI instance is generated, compiled, and downloaded into the instruments and infrastructure. It is executed across all the instruments and the infrastructure resources, and then the HVI instance takes control of the individual instruments and platform components. The HVI configures the required resources and downloads the hardware programs that, when executed, run on the instruments and platform hardware synchronously.

An application can create multiple HVI instances, but if the resources are shared, only one can be downloaded and executed in hardware at a time. If the HVI instances do not share any resources, they can be executed in parallel.

HVI Engines

For HVI to control an instrument, the instrument requires one or more HVI Engines. An HVI Engine is an Intellectual Property (IP) block that controls the functions of the instrument and the timing of operations. The HVI Engine is included directly in the instrument hardware or it can be programmed into the *Field Programmable Gate Array* (FPGA) in the instrument.

HVI works by deploying a binary executable to each hardware instrument to be executed by the HVI Engine. Different binaries execute on the different HVI Engines in parallel, across multiple instruments.

When you write an application that includes an HVI, you create HVI sequences. These are sequences of HVI statements, these are operations that control the instrument. The HVI sequences are compiled into the binary executables that the HVI Engine executes.

About Instrument FPGAs

An FPGA is an electronic component on the Instrument. The FPGA in an instrument might include pre-programmed IP for the instrument's functionality and this can include HVI IP components and regions you can configure.

In addition to any existing IP and HVI engines, instrument FPGAs include an FPGA sandbox, this is a user-configurable region in the instrument FPGA. You can configure the FPGA Sandbox to implement your own specific functionality. This can include custom logic and memory. To take advantage of this feature, you must use *PathWave-FPGA* to create your design in the FPGA sandbox. For more information see Chapter 5: HVI integration with PathWaveFPGA.

HVI Resources

The HVI Engine executes Sequences that are made up of Statements. These statements or instructions can operate on different resources in real-time. HVI can operate on the following resources:

- HVI Actions.
- HVI Events.
- HVI Triggers.
- Clock signals.
- HVI registers.
- FPGA sandbox registers and memory maps.

Actions, Events and Triggers are concepts within HVI. They are used to initiate operations, wait for operations, send signals, and receive signals.

Actions

HVI actions are digital electronic pulsed or level signals that are sent from the HVI engine to control instrument operations outside of the HVI Engine.

You use actions in HVI sequences to initiate operations. Typically, actions initiate instrument-specific operations. For example, in a digitizer instrument, a `StartAcquisition` action sends a digital pulse to start an acquisition operation.

Events

HVI events are digital electronic pulsed or level signals that are sent to the HVI Engine and used as notifications when instrument operations have occurred outside of the HVI Engine.

You use HVI Events in HVI sequences as notification events that the execution has to wait for. Typically, events indicate instrument-specific operations have occurred. For example, in an AWG, the AWG will send a digital pulse through the `WaveformDone` event when a waveform execution has been completed.

Triggers

HVI Triggers are electronic signals that the HVI engines can send or receive.

HVI Triggers are used to send signals and share data between instruments. You can use these to initiate operations, communicate states, or share information. There are multiple types of triggers depending on how they are connected, for example: front panel triggers (usually a SMA connector on the module's front panel), PXIe triggers (connected to the PXIe backplane of the chassis), and general purpose digital IO (LVDS connector in the module's front panel).

HVI Registers

HVI registers are similar to Variables in a programming language. They hold values that can be modified at runtime and can be used as parameters for instructions and statements. Physically, HVI registers are small hardware memories located in HVI engines. Their contents can be shared between HVI Engines by using specific instructions.

FPGA Sandbox registers and Memory maps

Some instrument FPGAs provide a user-configurable region in the instrument FPGA known as an FPGA sandbox. This enables you to program the instrument with logic that implements your own custom functionality. HVI Registers and Memory Blocks are components in the FPGA sandbox that you can use as resources in your HVI sequences. For more information see Chapter 5: HVI integration with PathWaveFPGA.

For the instruments that support an FPGA sandbox, HVI can support the sharing of data between the sandbox and the HVI engine in an instrument or between the sandboxes of different instruments. This functionality depends of the availability of specific interfaces inside the FPGA Sandbox. To take advantage of these features, you must use PathWave-FPGA to create your design in the sandbox.

NOTE The exact resources available and how they are configured is instrument dependent. Each instrument defines the actions and events available, how it uses triggers and the number and type of registers available. For the specific definitions and availability of resources in each instrument, see your instrument documentation.

HVI Sequences and Statements

You control instruments with HVI Statements. Statements operate on resources such as Actions, Events, and Triggers. There are different types of statements that perform different types of operations. HVI Statements are the building blocks of HVI Sequences. These sequences are compiled in your application and are executed in real-time on the HVI engines.

The following sections describe the different types of sequences and statements.

- HVI Sequences
- HVI Statements

HVI Sequences

An HVI instance consists of HVI sequences, which are the foundations of HVI technology. An HVI sequence is an ordered list of HVI statements with associated timing information. A sequence is executed in a time-deterministic manner by the HVI hardware engine located within an instrument. An HVI instance is made up of one or more sequences that run in parallel and synchronously.

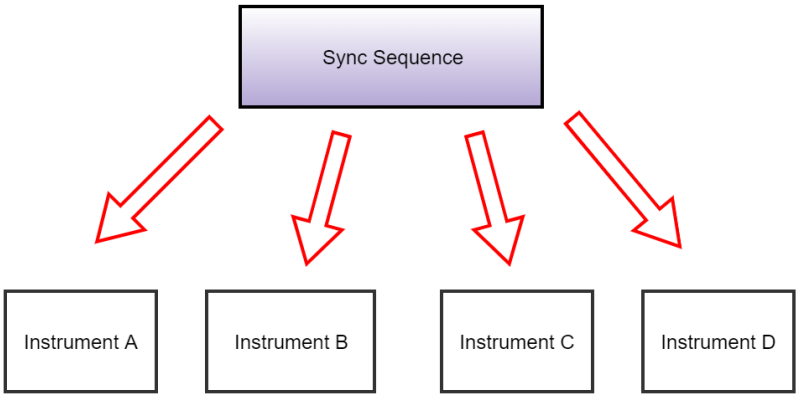
There are two types of sequences:

- Sync sequence.
- Local sequences.

HVI sequences are organized in a hierarchy with Sync sequences at the top.

Sync sequences

A synchronized sequence (called a Sync sequence) contains commands known as Sync statements that execute across multiple instruments:

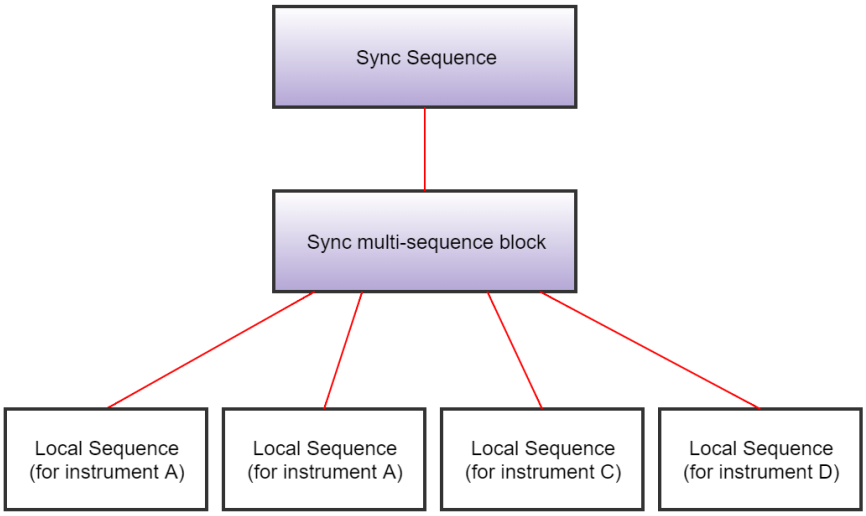


Local sequences

The Local sequences are executed by each individual HVI engine in an instrument.

Local sequences are contained within *Sync Multi-Sequence Blocks*. A Sync multi-sequence block is a type of Sync statement that is contained in a Sync sequence.

The following diagram shows the relationship between a Sync sequence, Sync multi-sequence block, and Local sequences:



HVI Statements

HVI statements are the commands or operations that make up an HVI sequence. HVI sequences are the ordered lists of HVI statements that are executed with precise timing. If you think of an HVI sequence as a poem, the HVI statements are the possible words you can use to write the poem and the HVI API is the language you use to write it. HVI statements are FPGA-level operations that are executed by the HVI engines.

HVI statements are broadly divided into two groups:

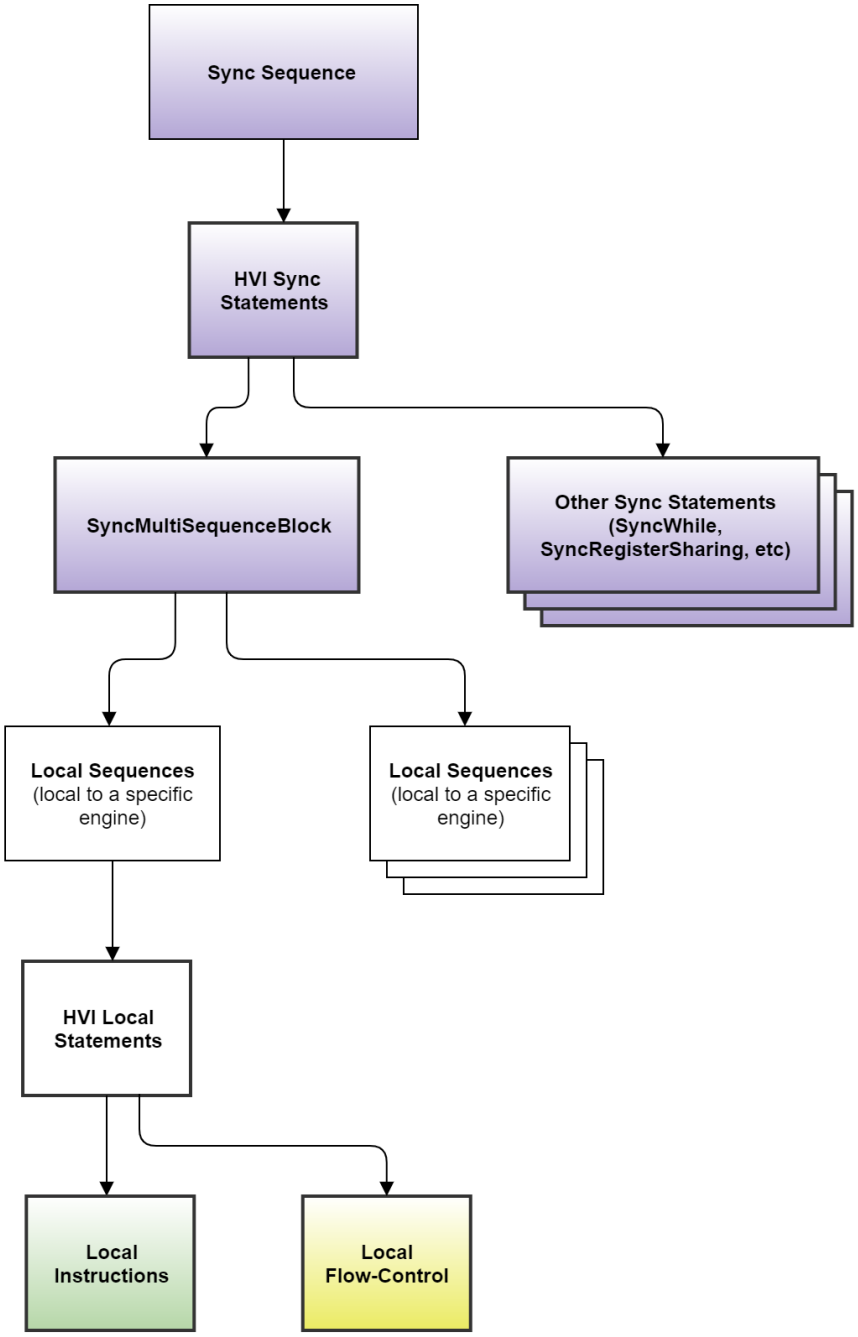
HVI Sync statements

Synchronized (Sync) statements are used to execute operations or control the flow of execution across all HVI hardware engines. Sync statements are executed synchronously among all HVI engines.

HVI Local statements

These are the commands or operations you put in the Local sequences to be executed on a specific HVI engine that is in a specific hardware instrument.

The following diagram shows the different kinds of statements and how they relate to Sync sequences and Local sequences:



HVI Sync statements

These are used to execute operations or control the flow of execution across all HVI hardware engines. Sync statements are executed synchronously among all HVI engines.

HVI Sync statements are contained in a Sync sequence. HVI Sync statements execute across all instruments.

The Sync sequence enables multiple engines to execute statements in lockstep.

The following HVI Sync statements are available:

- Sync while
- Sync register-sharing
- Sync FPGA data-sharing
- Sync multi-sequence block

Sync while

Enables a while loop to execute synchronously on all engines.

The Sync while flow-control enables you to execute a Sync sequence in a loop while a condition is met. The condition is evaluated each time before starting the Sync sequence execution. When the condition is false and the Sync sequence reaches the end, the Sync while jumps out of the loop and the Sync sequence containing the Sync while continues execution with the next Sync statement.

Sync register-sharing

The Sync register-sharing statement enables you to share data from a source register to a destination register in any other HVI Engine.

It enables you to share the contents of N adjacent bits from a source register and write it to a destination register in another HVI Engine in your HVI.

Sync FPGA data-sharing

Enables you to share data from one FPGA Sandbox to one or more other FPGA Sandboxes in different instruments. The data-sharing is orchestrated by the HVI engines of the different instruments.

Data can be shared between instruments in a single chassis or across instruments in multiple chassis. Sync FPGA data-sharing utilizes the *Fast Data Sharing* (FDS) functionality to enable the low-latency transfer of data.

Data is sent 4 bits at a time and can be sent from one to one, or from one to many FPGA Sandboxes.

Sync multi-sequence block

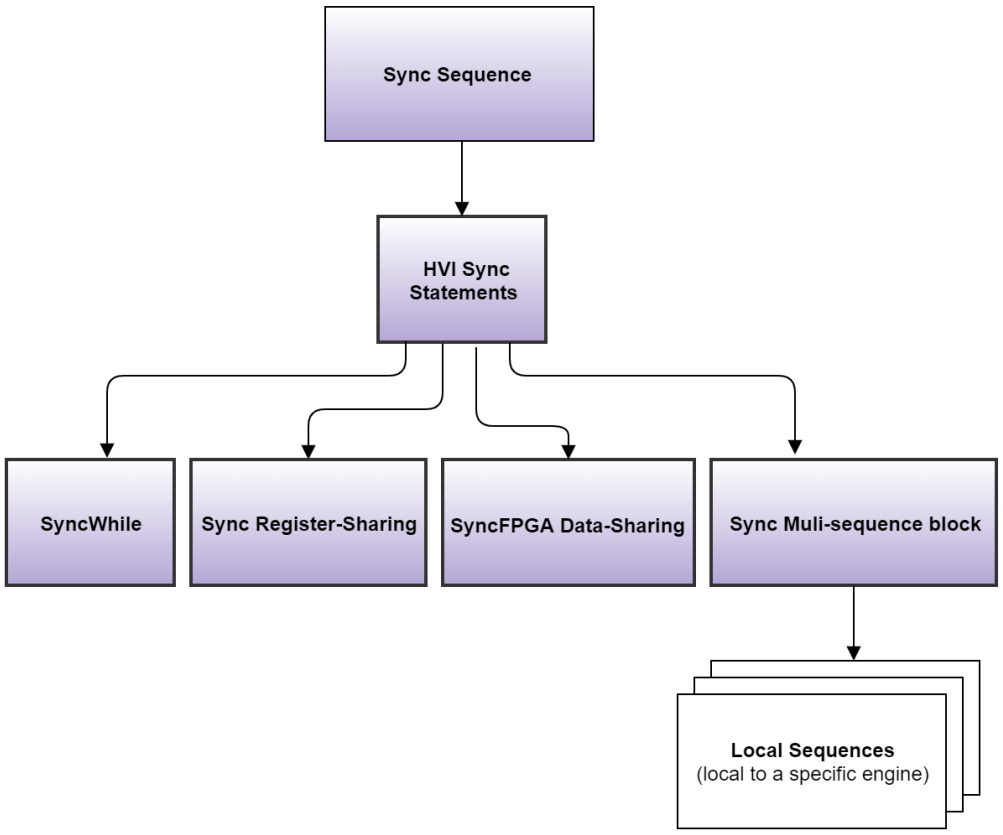
Enables the execution of multiple, simultaneous, engine-specific sequences.

Sync multi-sequence blocks are a type of Sync statement that contain a set of Local sequences. The Local sequences execute on individual HVI Engines within the instruments. All Local sequences contained in a Sync multi-sequence block start and end at the same time.

The Sync multi-sequence block enables you to run different sequences on each engine concurrently. It ensures that the execution of all the Local sequences starts exactly at the same time and that the Sync sequence remains synchronous afterwards. It serves as a boundary between sections and a container where each engine operates individually.

All HVI Local Sequences operate within HVI Sync statements. The HVI Sync statements determine global or synchronized operations, or synchronization points.

The following diagram shows how the HVI Sync statements fit in the Sync sequence:



HVI Local statements

HVI Local statements are the commands or operations that make up Local sequences. These are the commands or operations you put in the Local sequences to be executed on a specific HVI engine in a specific hardware instrument. There are two types of Local statements:

- Local instruction statements.
- Local flow-control statements.

Local instruction statements

These are operations that are executed by the HVI engine in the instrument hardware and do not impact the execution flow.

There are two types of Local instruction statements:

HVI-native instructions

HVI-native instructions are instrument independent, general-purpose instructions present on all instruments, for example, math operations, writing triggers and executing actions. HVI-native instructions are defined by the HVI API.

Instrument-specific instructions

These are instructions that are specific to instruments. You can use these when you program an HVI with those specific instruments.

These instructions can change instrument settings such as amplitude and frequency. They can also trigger instrument functions such as queuing waveforms for playback, outputting a waveform, or triggering a data acquisition.

Instrument-specific instructions are defined by the HVI instrument add-on API and are exposed in each instrument driver as instrument-specific HVI definitions.

NOTE The User Guides for the M320xA PXI AWGs and M310xA PXI Digitizers describe all the HVI instructions available for each of the M3xxxA PXI instruments.

Local flow-control statements

Local flow-control statements are used to control the execution flow within each Local sequence. These statements are depicted with yellow boxes in the HVI diagrams displayed in this User Manual.

These are used to control the execution flow of a specific HVI engine. They are divided into two types:

Wait statements:

Local Wait-for-Event

Waits for a condition that can be determined by an HVI Event, an HVI Trigger, or any logical combination of any of these types of conditions.

Local Wait-for-Time

Waits for an amount of time specified in a register.

Local Delay

Delays a sequence for a time you specify.

Conditional flow-control statements:

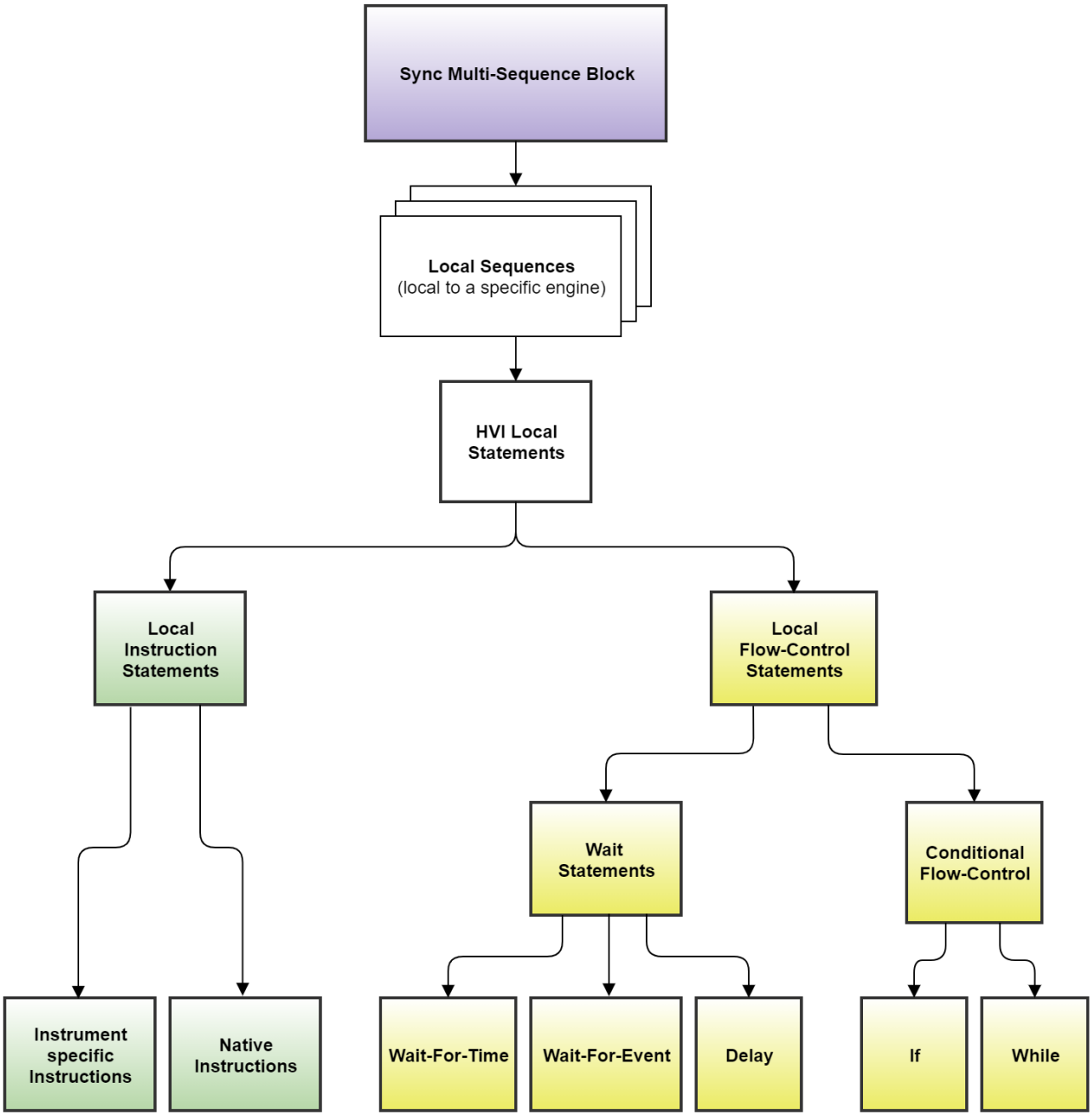
Local If

This acts as an If-Elseif-Else, local If executes one of a set of possible Local sequences depending on the value of a defined condition.

Local While

Executes while a condition is true.

The following diagram shows the different types of Local statements and their relationship to the Local sequences:



HVI Diagrams

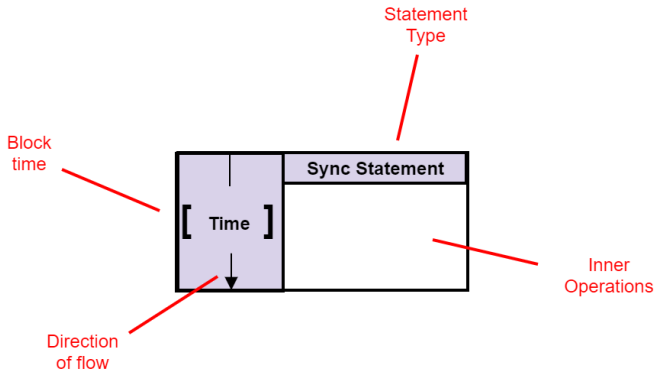
This section shows HVI diagrams. These are used to illustrate HVI sequences.

In the HVI diagrams, the following colors are used to indicate different kinds of statements:

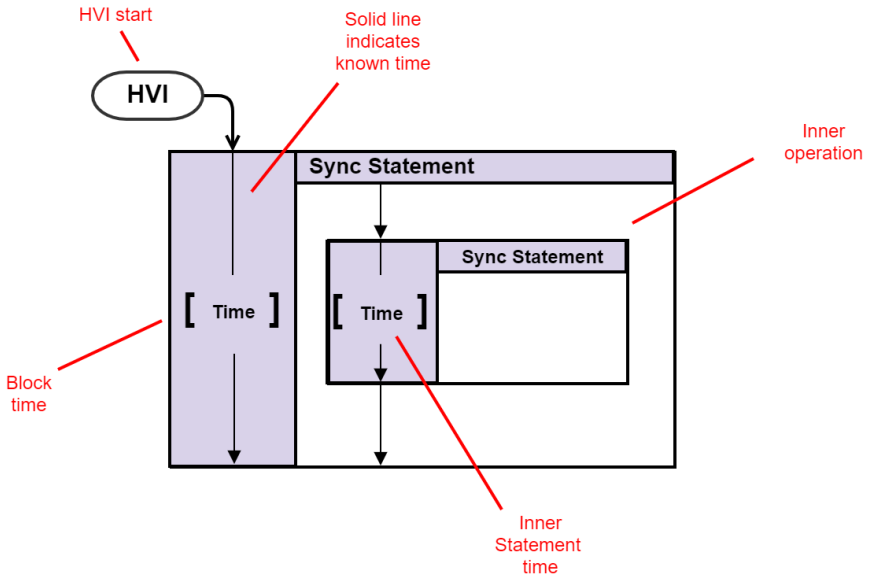
Sync Statements	
Local instructions	
Local Flow-control	

Statement diagram color code

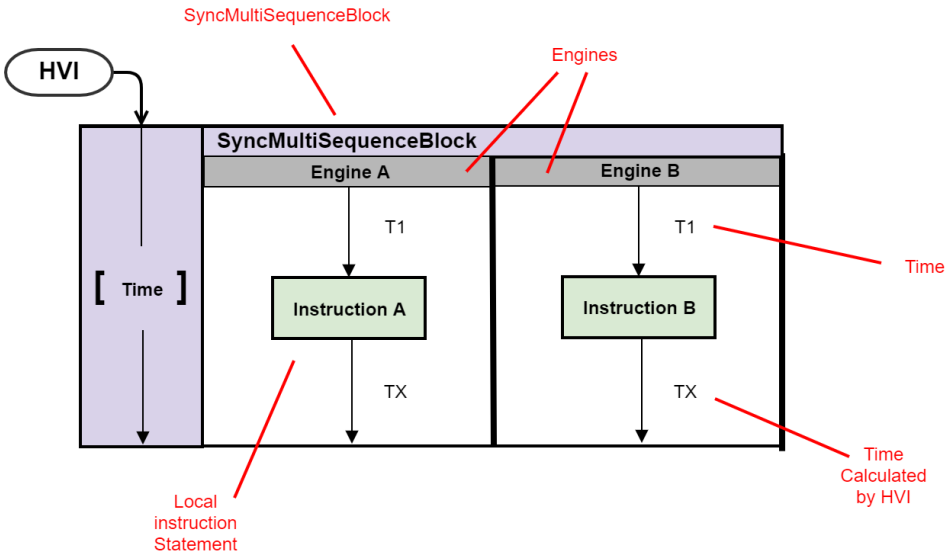
The following diagram shows a single Sync statement with flow and time for the block:



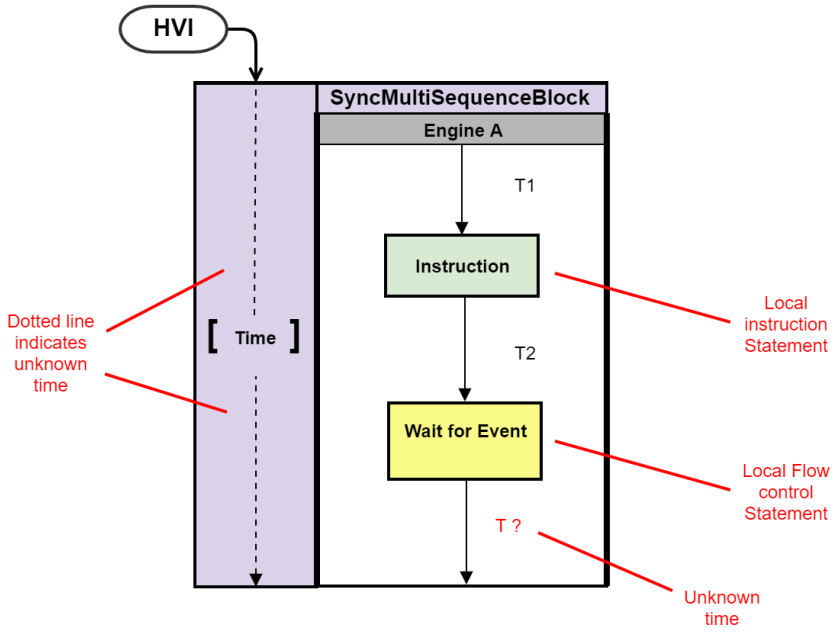
The diagrams can show nesting of statements within statements. For example, the following diagram shows a Sync statement that is within another Sync statement:



Local sequences are placed within their HVI engines in Sync multi-sequence blocks. The following diagram shows a pair of Local sequences with an instruction each inside a Sync multi-sequence block:

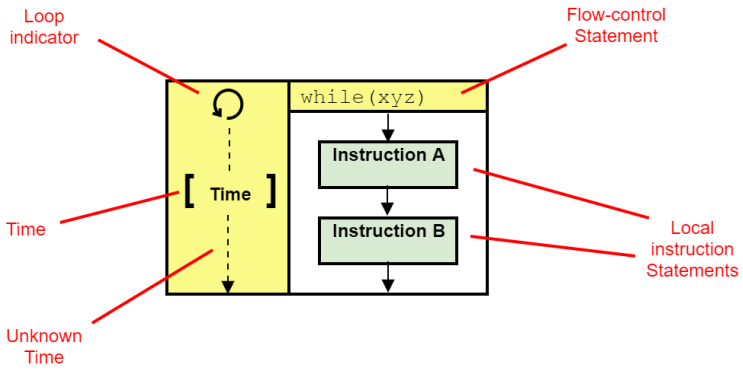


A dotted line indicates that execution time is not known at compile time. This is often the case with flow-control statements. In this case the Wait-for-event statement shall not release until the event occurs. It is not known at compile time when this is, so the time cannot be calculated at compile time.

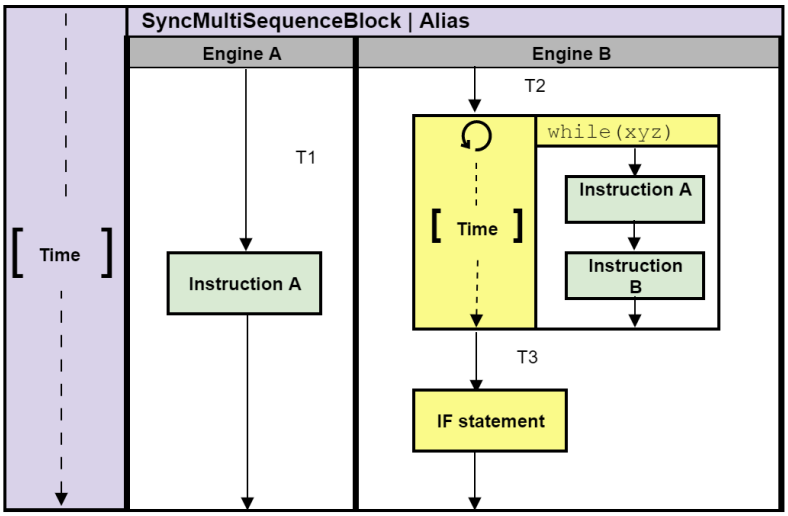


The following diagram shows a Local flow-control statement that encloses a pair of Local instruction statements. The color Yellow indicates a Local flow-control statement.

The circular symbol is a loop indicator that shows that the block iterates.



The following diagram shows a more complex example. The Sync multi-sequence block contains two Local sequences, one per HVI engine. The Local sequences execute operations on their associated HVI engines in parallel.



HVI Timing

This section introduces the basic HVI timing concepts, including:

- HVI Statement Timing Definitions.
- Timing description for different statement types.
- Time Matching of Sequences in Sync Multi-Sequence Blocks.

HVI timing is a complex topic that involves you understanding how to calculate the timing between statements. The calculations required and parameters involved are described in detail in Chapter 9: HVI Time Management and Latency.

HVI Statement Timing Definitions

When you are programming an HVI, you have precise control over the timing of HVI statement execution. To do this correctly, you must understand the following time definitions:

- Start time.
- End time.
- Fetch time.
- Execution time.
- Start delay.

Start time

This is the instant of time when the HVI starts the execution of a statement. You set the Start time when you are programming your sequence by setting a parameter called *Start delay*. HVI either meets the specified time exactly, or it generates an error if it is not possible.

End time

This is the instant of time when:

- The execution of a statement is completed, and the result is available.
- An operation is completed, such as a register update or a trigger value change.

For operations that have a long execution time, the End time indicates when the first result is available, or the operation is complete.

Fetch time

This is the time interval required by the HVI engine hardware to fetch and dispatch a statement for processing. Depending on their characteristics, some statements can take several HVI engine cycles to complete the fetch before the processing can start.

Execution time

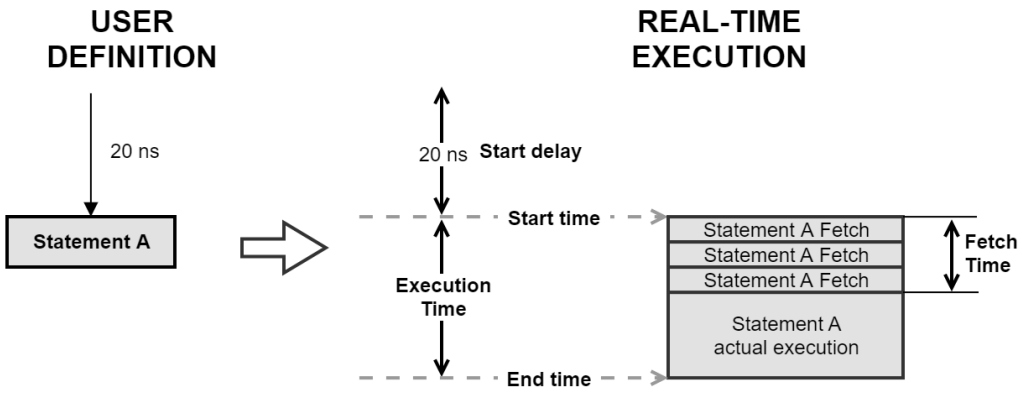
This is the time interval from the Start time to the End time of the statement. This interval is determined by instrument constraints and inherent limits such as propagation delays and resource availability. The Execution time includes the Fetch time.

Start delay

The Start delay defines the period between the execution of consecutive statements. The Start delay enables you to have full control of the timing of operations and ensures there is enough time for correct execution. If the Start delay is not accounted for properly, the HVI sequences shall not behave correctly. Start delay is a parameter that you set in the `add_statement()` methods.

NOTE If you do not specify a valid Start delay, the compiler generates an error and indicates the minimum valid minimum value. For more information, see Chapter 9: HVI Time Management and Latency.

The following diagram shows the HVI statement timing definitions:



Timing Descriptions for Different Statement Types

This section describes statement timing and provides a set of examples. It contains the following subsections:

- Start delay operation for different types of statements.
- Local instruction timing.
- Local flow-control timing.
- Sync statement timing.

Start delay operation for different types of statements

Start delay is always specified between statements, from the previous statement to the current statement.

You define a start delay in one of 2 different ways:

- From the beginning of the previous statement.
- From the end of the previous statement.

The way you define the start delay depends on the type of the previous statement. For example, say you have 2 statements: A followed by B. The Start delay for statement A is already specified and you want to specify the start delay for statement B.

The current statement is statement B, so the start delay of statement B depends on the type of the previous statement A:

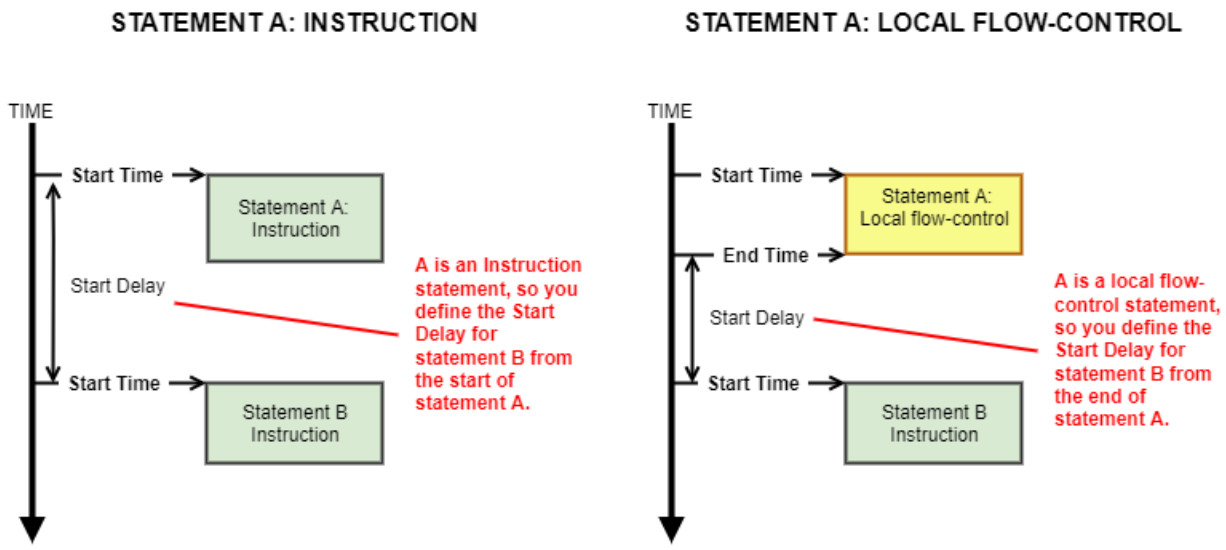
Instruction statements

If statement A is a **Local instruction** statement, the start delay of statement B starts at, and is measured from, the **Start time** of the statement A.

Sync statements and Local flow control statements

If statement A is a **Sync statement** or a **Local flow-control** statement, the start delay of statement B starts at, and is measured from, the **End time** of statement A.

The following diagram shows the different start delay definitions:



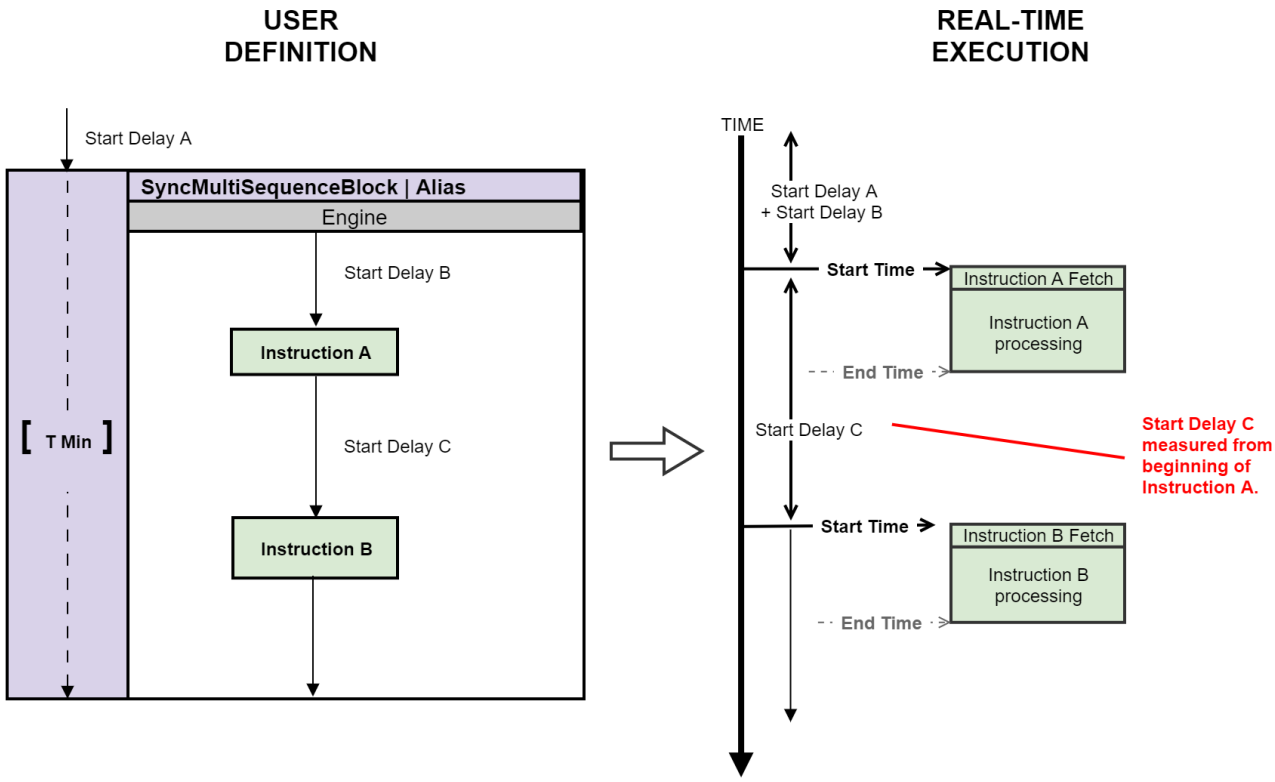
Local instruction timing

The following diagram shows the timing of Local instructions.

For local instructions, the Start delay of the following instruction is measured from the start of the previous instruction. This is possible because once the instruction fetch cycles are completed, the HVI engine is free to fetch and execute another instruction.

It is important to highlight that the Start delay must be greater than or equal to the fetch time of the previous instruction.

The following diagram shows two Local instructions and their timing:



Local flow-control timing

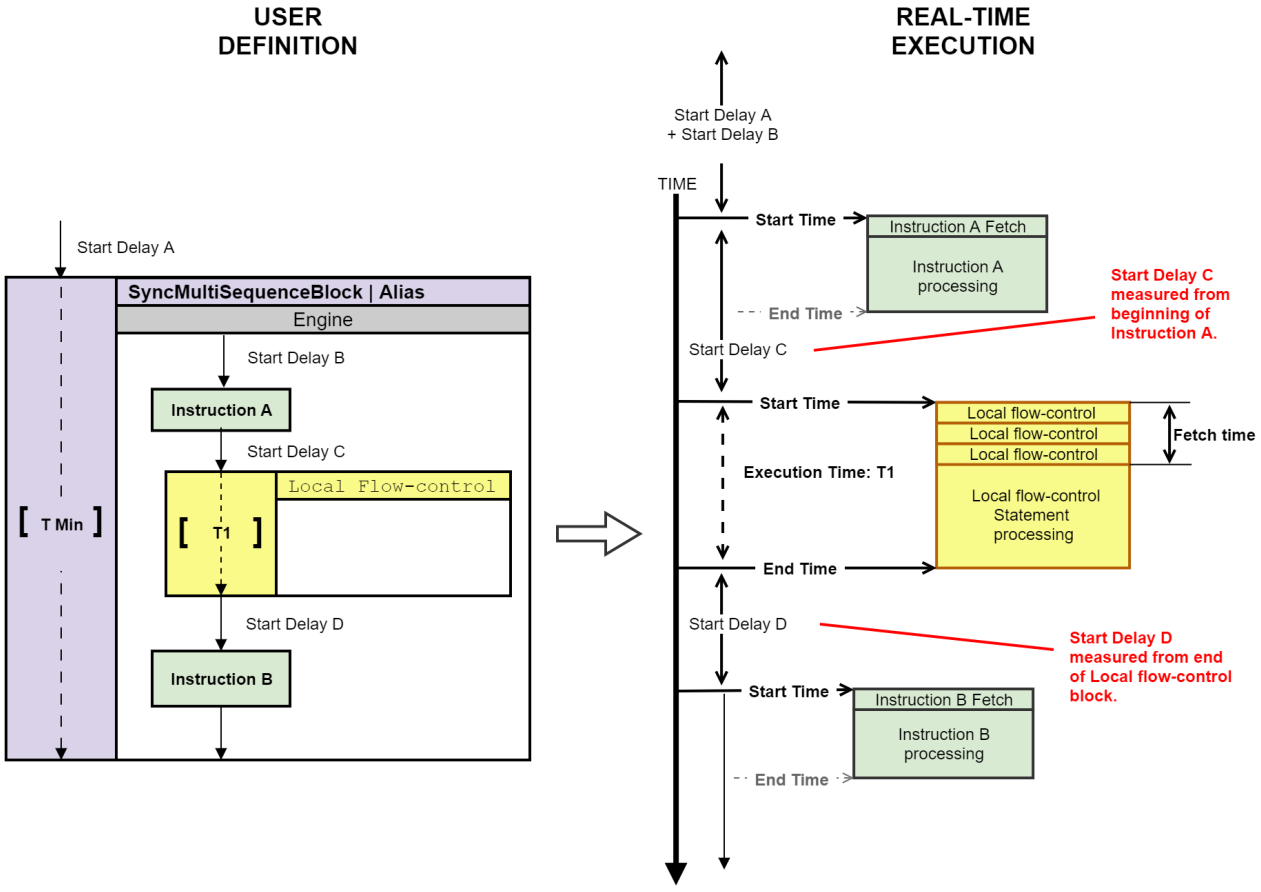
For Local flow-control statements, the Start delay of the next statement is measured from the end of the previous Local flow-control statement. This is because the HVI engine is busy during the execution of the flow-control statement and the execution of a flow-control statements cannot be overlapped with any following statements.

For the Local flow-control statement after instruction A, the Start delay (Start delay C) is measured from the start of the previous instruction (instruction A).

For instruction B, that follows the Local flow-control statement, the Start delay (Start delay D) is measured from the end of the flow-control block.

The execution time of local flow-control statements can be known at compile time, or might be unknown, the dotted line in the diagram below indicates that the execution time of the Local flow-control block T1 is not known at compile time.

The following diagram shows the difference between measuring timing of Local instructions and Local flow-control statements.



Sync statement timing

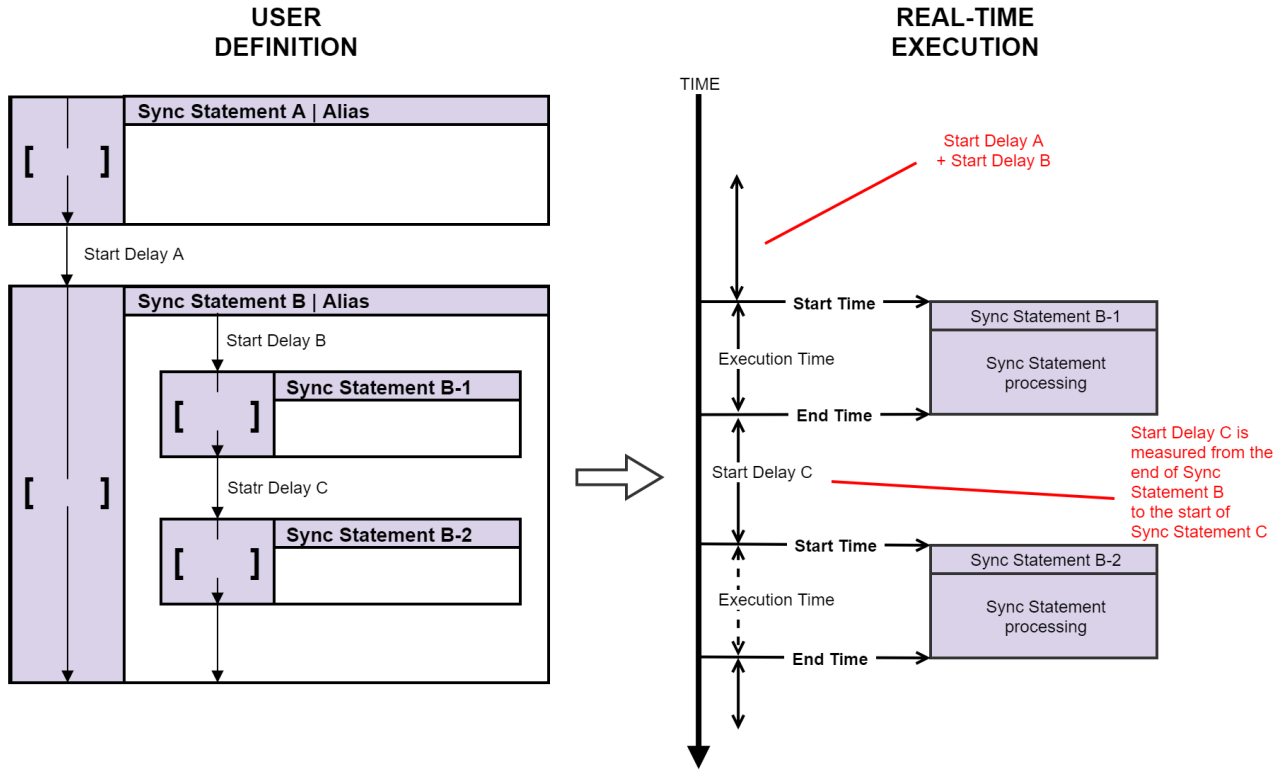
For Sync statements, the Start delay is measured from the end of one Sync statement to the start of the following Sync statement.

The following diagram shows two Sync statements, A and B. Sync statement B is a container for two further Sync statements, B-1 and B-2. The times indicated are Start Delay A, Start Delay B, Start Delay C, T1, and T2.

The time between the end of Sync statement A and the start of Sync statement B-1 is Start Delay A + Start Delay B. The time between the end of Sync statement B-1 and the start of Sync statement B-2 is Start Delay C.

The execution time of Sync Statements can be known at compile time, as shown below with a solid line.

The following diagram shows the timing between Sync statements:



Time Matching of Sequences in Sync Multi-Sequence Blocks

Sync multi-sequence blocks can contain multiple Local sequences, each running on a different engine.

At the start of the Sync multi-sequence block, the Local sequences are synchronized so that they all start simultaneously.

At the end of the Sync multi-sequence block, the sequences are all synchronized to end simultaneously. The individual sequences can have different execution times, so HVI automatically adjusts the timing of each individual sequence to ensure that they all end simultaneously.

The HVI ensures the sequences end at the same time in one of the following ways:

- The end times of the sequences are set to match the longest sequence (minimum execution time).
- The end times of the sequences are set to match a specific execution time that you define.
- The end times of the sequences are set to match at runtime, dynamically. This occurs if any of the sequences includes statements with an execution time that is unknown at compile time.

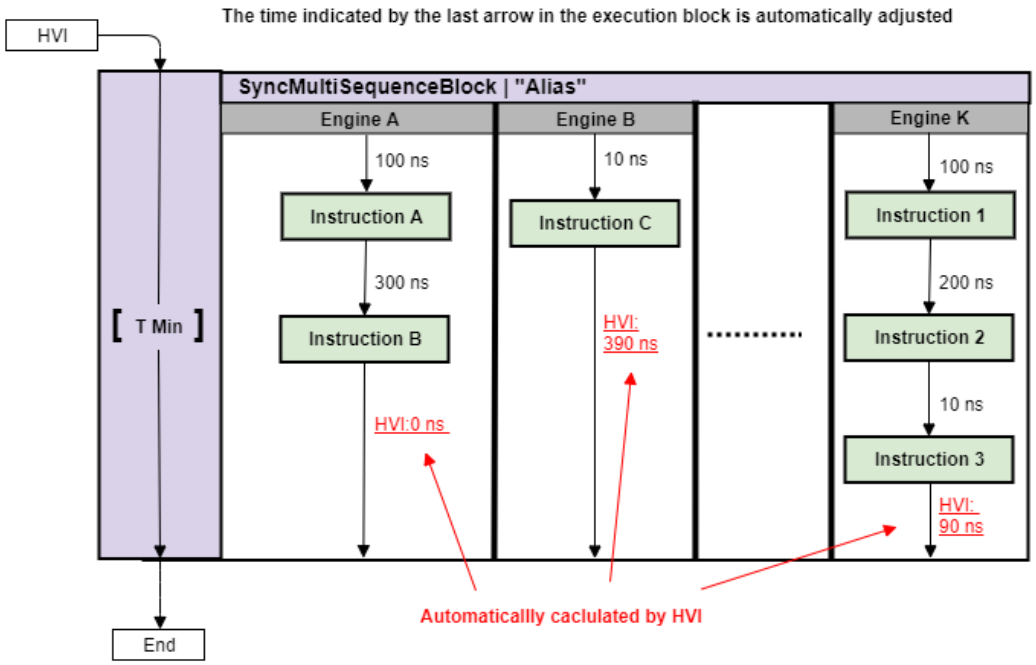
End times of sequences set to match the longest sequence (minimum execution time)

If the execution time of the instructions and flow-control statements in the sequences are known at compile time, then HVI adjusts the final times so that all the sequences in the Sync multi-sequence block end at the same time.

In the following diagram, the time of the Sync multi-sequence block is not specified. In this case the compiler adjusts the total execution time of all sequences to match the longest one. The execution times of the instructions and the delays between them are known, so the timing between them and the timing of the entire sequences can be calculated during the HVI sequence compilation. The Sync multi-sequence block execution time is set to the minimum possible time given by the longest sequence. The different HVI Engine clocking constraints are also taken into consideration.

The total time for Engine A is 400 ns. The HVI calculates the additional times required for the other engines so that they finish at the same time. For Engine B the additional time is 390 ns, for Engine K the additional time is 90 ns.

The following diagram shows a Sync multi-sequence block with minimum execution time:

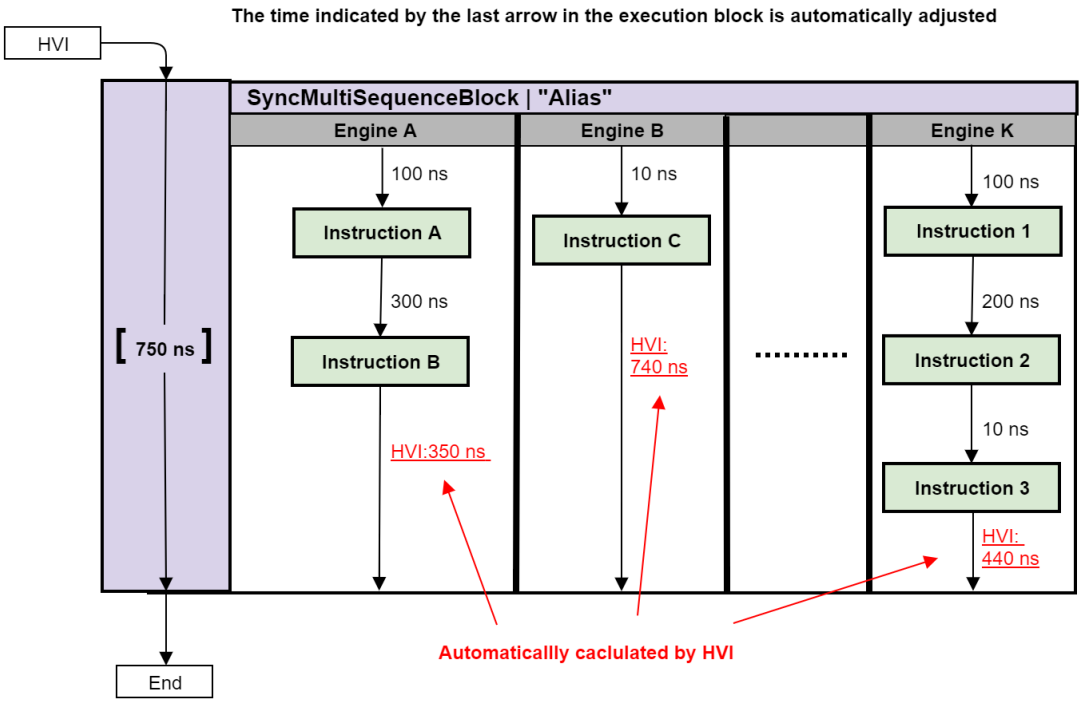


End times of sequences set to match a specific execution time

You can specify a time for the Sync multi-sequence block using the duration property. If the execution time of the instructions and flow-control statements in the sequences are known at compile time, HVI adjusts the final times so that all of the sequences in the Sync multi-sequence block end at the time you specified.

In the following diagram the Sync Multi-Sequence Block duration time is specified at 750 ns. The timing of the instructions and the delays between them are known at compile time, so the execution time for each sequence can be calculated. HVI calculates the additional times required for all the engines to finish at the specified time. For Engine A this is 350 ns, For Engine B this is 740 ns, for Engine K this is 440 ns.

The following diagram shows a Sync multi-sequence block with an execution time specified as 750 ns:



Chapter 5: HVI integration with PathWaveFPGA

This chapter describes PathWave Test Sync Executive integration with PathWave FPGA. It contains the following sections:

PathWave FPGA and HVI Overview

What is an FPGA?

A *Field programmable Gate Array* (FPGA) is a digital electronic component on many Keysight instruments, whose behavior can be modified for different use cases.

Keysight instruments use FPGAs to implement complex functionality and data processing. Some instruments also make a region in the FPGA available to enable the addition of custom logic and real-time processing into the instruments. You can customize the FPGA with PathWave FPGA software.

PathWave FPGA

PathWave FPGA is a graphical software tool that enables you to rapidly customize logic in the *Sandbox* section of the FPGA in supported Keysight instruments. By doing this you can modify or enhance the default behavior of these instruments.

Instruments that support both PathWave FPGA and PathWave Test Sync Executive enable you to combine your customized logic with the real-time capabilities of PathWave Test Sync Executive. For example, you can have DSP processing in the FPGA Sandbox, triggered in real time by an HVI sequence.

FPGA Sandbox

In addition to any existing *Intellectual Property* (IP) and HVI Engines, a Keysight instrument FPGA can include one or more FPGA Sandboxes. An FPGA Sandbox is the region in the FPGA that you can configure using PathWave FPGA.

You can configure the FPGA Sandbox to implement your own custom IP, signal processing and other functionality. This can include custom logic, registers and memory interfaces. HVI can interact with this custom logic using HVI-specific interfaces.

PathWave FPGA includes an *Intellectual Property* (IP) library that includes Logic/Math, Memory, and DSP blocks that you can place in the FPGA Sandbox. The *Real-time HVI* design interface catalog enables you to add memories and registers, and also contains specific HVI interfaces for HVI Actions, HVI Events, HVI Triggers, and FPGA-Instruction statements.

.k7z files

When you have completed your design, PathWave FPGA enables you to easily build a **.k7z file** for the FPGA from your schematic.

The .k7z file contains the bitfile that is used to program the FPGA-Sandbox design into the FPGA. The .k7z file also contains all the information about the FPGA-Sandbox design, such as names, addresses, ranges of the registers, and memory-mapped locations, etc, including resources that are connected to the HVI Engine.

You add the **.k7z** into your HVI instance in the SystemDefinition. This file is used by the HVI to get all the definitions required so you can utilize your customizations.

HVI Resources in the FPGA Sandbox

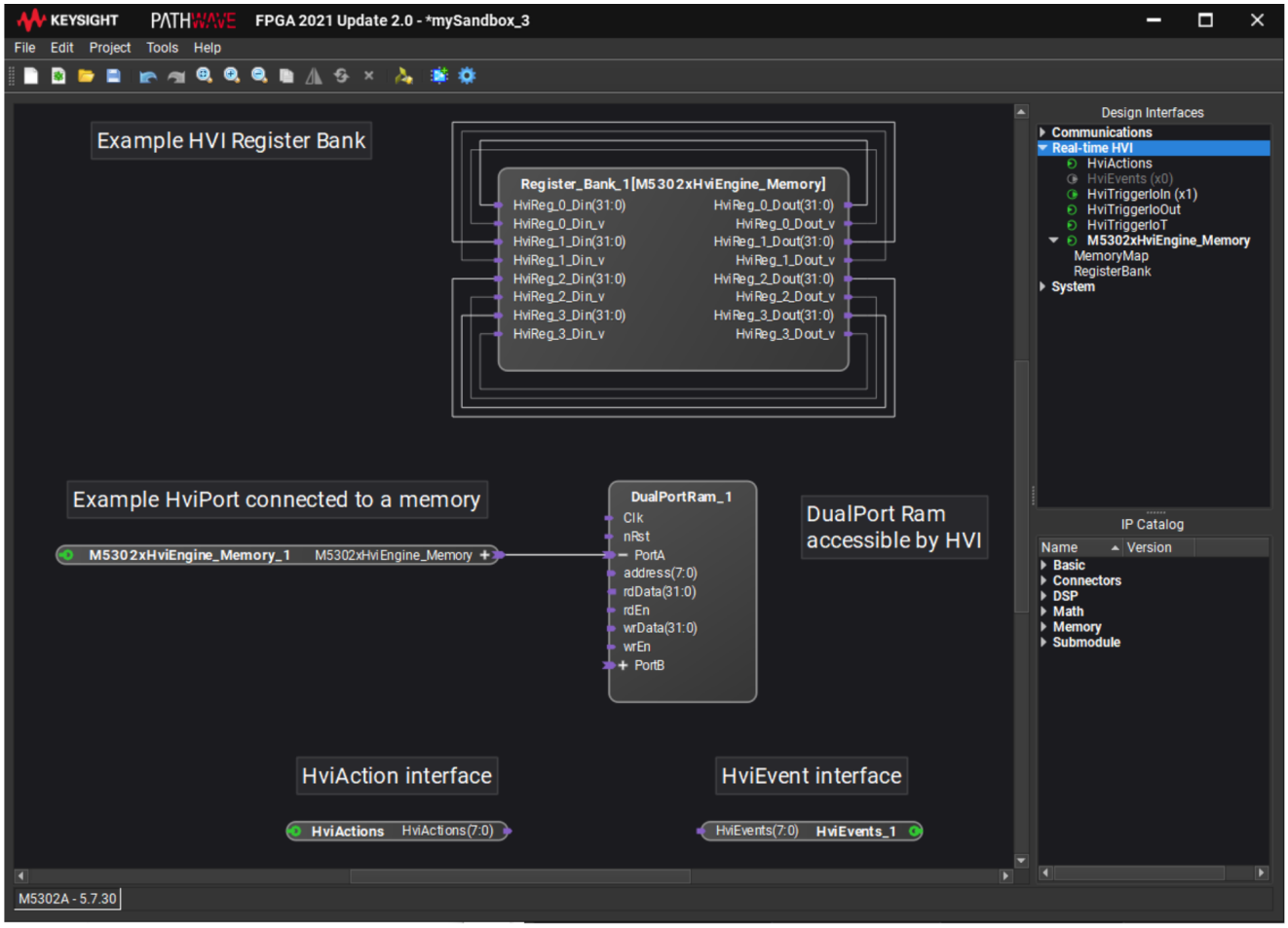
PathWave FPGA enables you to add HVI to your logic design in the FPGA Sandbox. These resources enable your HVI to interact with the logic in the sandbox.

You can add the following types of components and access them from your HVI sequences:

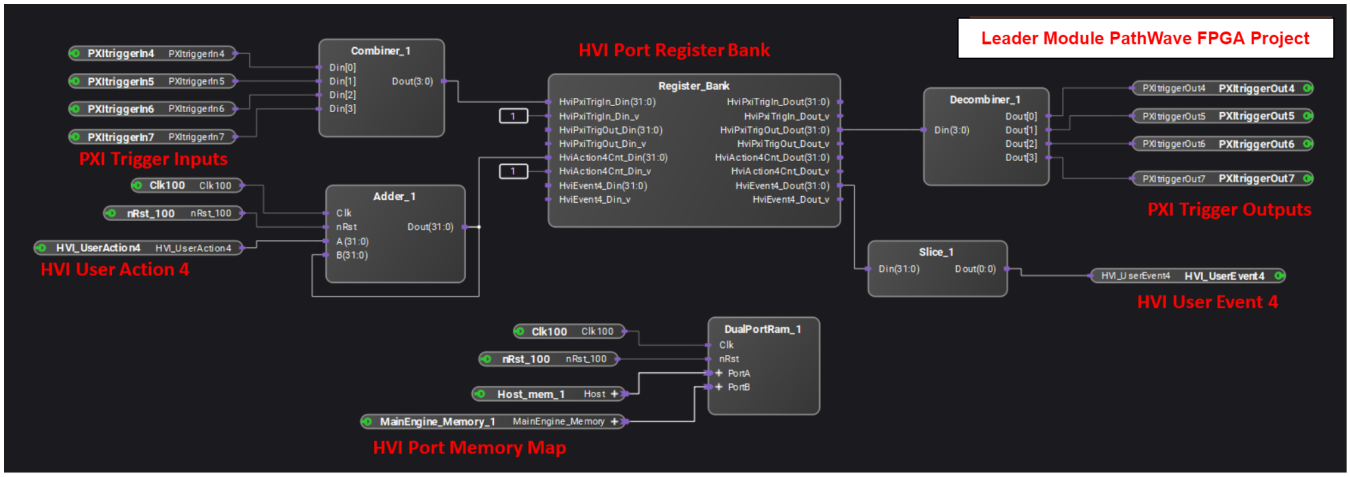
- HVI Sandbox registers.
- HVI Memory maps.
- HviAction interfaces.
- HviEvent interfaces.
- HviTrigger interfaces.
- FPGA *Fast Data Sharing* (FDS) ports.
- FPGA-Instruction (HviFPGAInstructions).

NOTE The exact resources you can add depends on the capabilities of the instrument you are using. For example, FDS ports are only available on instruments that support them.

The following diagram shows a screenshot of PathWave FPGA with some example resources:



The following image is a screenshot of from PathWave FPGA taken from Programming Example 3. The image shows a set of FPGA blocks, a number of HVI resources, and the connections between them in an FPGA Sandbox. For more information about Programming Example 3 see: Appendix B: Additional Documentation and Examples. For information about how to use PathWave FPGA see the documentation at PathWave FPGA.



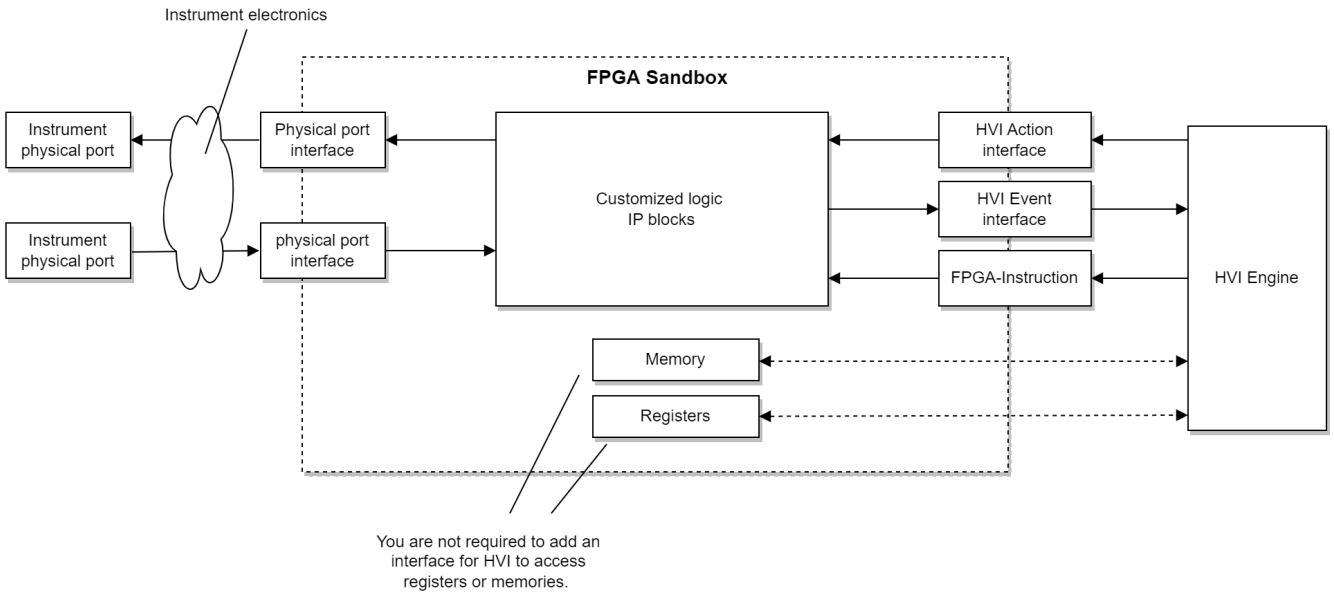
PathWave Test Sync Executive includes a number of HVI instructions that enable your HVI sequences to interact with the IP blocks in the FPGA sandbox. The instructions include access to registers or memory maps, and send data or commands into the FPGA sandbox.

If you want to send an action into the sandbox, you must add an **`HviAction`** interface to the sandbox, you just add this to the design and connect them to your customized logic. In a sequence you are able to interact with the logic using the relevant instructions. Actions in the sandbox are accessed from sequences in the same way you use any other HVI actions, no special instructions are required. **`HviEvents`** and **`HviTriggers`** work the same way.

HVI Sequences and Sandbox resource interaction

When you run your HVI, the HVI engine reads and executes the individual commands within your HVI sequences. When the HVI engine executes an HVI statement that involves interaction with a resource in the FPGA sandbox, the HVI engine communicates with the FPGA Sandbox.

The following diagram shows an FPGA Sandbox that contains custom IP blocks with connections to an HVI engine and the instrument physical interfaces:



Using PathWave FPGA with PathWave Test Sync Executive

The full flow to customize the logic in an FPGA Sandbox and then use these customizations in HVI sequences is:

In PathWave FPGA:

- Open the instrument *Board Support Package* (BSP) using PathWave FPGA.
- Customize the logic by adding logic blocks.
- Add any registers and memories required.
- Add HVI interfaces so HVI can interact with your logic (Actions, Events, Triggers).
- Connect your customized logic to the relevant I/O signals in the HVI interfaces.
- Generate the **`.k7z`** file.

Once you have configured the FPGA with PathWave FPGA, added the relevant HVI interfaces as required, and generated a **`.k7z`** file, you must load the definitions into PathWave Test Sync Executive:

- Load the **`.k7z`** file into your SystemDefinition.
- Write your HVI sequences and use the FPGA resources.
- Load your HVI instance to Hardware (this step loads the **`.k7z`** as required).

You can use the HVI resources in the FPGA in the same way as you use any other HVI resources.

Using FPGA-Sandbox Resources with HVI

This section describes what to do in the different HVI programming stages, so you can use the FPGA customizations you made in PathWave FPGA in your HVI sequences.

Load the k7z file in SystemDefintition

When you have completed and built a design, PathWave FPGA generates a **.k7z** file.

Before you can use any FPGA sandbox resources with HVI, you must first load the **.k7z** into your HVI System Definition.

The **.k7z** file contains a *bitfile*, that is used to load the design into the FPGA Sandbox. The **.k7z** file also contains information about the resources in the design, such as names of ports and interfaces, addresses, ranges of the registers, memory-mapped locations, etc. The **.k7z** file is used to program your customizations into the FPGA and it is also used by HVI to get all the definitions required so you can utilize your customizations.

The following code shows how to load the **.k7z** file:

```
# This must be the name that the instrument has defined for the target sandbox
sandbox_name = "InstrumentSandbox1"# Get Engine Sandbox
sandbox = system_definition.engines[engine_name].fpga_sandboxes[sandbox_name]
# Load the k7z file to HVI
sandbox.load_from_k7z(k7z_file_path)
```

Using FPGA Sandbox resources in an HVI Sequence

When you load the **.k7z** , file into a specific Sandbox, HVI is able to access the resources defined in PathWave FPGA for that specific sandbox, allowing you to use them in your HVI sequences or at runtime.

The following example shows how to get and write to a memory map inside a sequence:

```
# Get Memory map object by name, as this is defined in the PathWave FPGA design
hvi_memory_map = sandbox.hvi_memory_maps["memory_map_name"]
#
# Write Memory Map
engine_sequence = multi_sequence_block_statement.sequences[engine_name]
fpga_array_write_instruction = engine_sequence.instruction_set.fpga_array_write
write_mem_map_instruction_statement = engine_sequence.add_instruction("Write FPGA Memory Map",
10,   fpga_array_write_instruction.id)
write_mem_map_instruction_statement.set_parameter(fpga_array_write_instruction.fpga_memory_
map.id, hvi_memory_map)
write_mem_map_instruction_statement.set_parameter(fpga_array_write_instruction.fpga_memory_map_
offset.id, 0)
write_mem_map_instruction_statement.set_parameter(fpga_array_write_instruction.value.id, 10)
```

Actions, Events, and Triggers are treated in a different way.

The Actions, Events, and Triggers available in the Sandbox are provided by the Instrument the FPGA is located in.

You define and use these the same way as you do other instrument Actions, Events and Triggers. For more information, see your instrument documentation.

The following code shows how to use an action-execute instruction to execute an action:

```
# First, the action that goes to the Sandbox must be added to the Engine:
#
# Get the action ID using the instrument's API, e.g.:
action_id = instrument.hvi.actions.user_sandbox3
#
# Specify a name for the action to be used in the context of your HVI program
action_name = "MySandboxAction"#
# Add the action to the Engine of the instrument
action = system_definition.engines[engine_name].actions.add(action_id, action_name)
#
# Then, use the action in your sequence
engine_sequence = multi_sequence_block_statement.sequences[engine_name]
action_execute_instruction =  engine_sequence.instruction_set.action_execute
action =  engine_sequence.engine.actions[action_name]
instruction = sequence.add_instruction("Execute Action", 10, action_execute_instruction.id)
instruction.set_parameter(action_execute_instruction.action.id, action)
```

NOTE When you write HVI sequences, you must use the same names you used in the PathWave FPGA project to access the HVI FPGA resources memory-maps, registers and FDS ports.

Load to Hardware

The **.k7z** internal *bitfile* is automatically loaded into the hardware at this stage if it is not already loaded. Once it has been loaded, in addition to running the HVI sequence to control FPGA resources in real-time, you can also access some of the HVI FPGA resources from software, for instance writing to the FPGA memory map:

```
# Load or Deploy Hvi instance to hardware. At this step the k7z is loaded, if it is not already
loaded
hvi_instance.load_to_hw()
#
# Write to memory map, in this example 0 is the offset and 1 is the data.
sandbox.fpga_memory_maps["memory_map_name"].write( 0 , 1 )
#
# Run the Hvi Sequence to use/control FPGA Sandbox resources in real-time as described in the
Sequence
hvi_instance.Run()
```

HVI Memory Maps and Register Banks in FPGA Sandbox

This section describes the HVI Registers and HVI Memory maps that you can add to FPGA Sandboxes in PathWave FPGA.

HVI Registers

HVI registers are user-defined hardware registers that are similar to Variables in a programming language. Physically, registers are small hardware memories located in the sandbox in the FPGA. These registers can be accessed and modified by both HVI instructions in real-time during sequence execution, and can be written in HVI software calls.

These registers can be used as destinations or sources of data. The source of the data to be written is a literal or an HVI Register. The destination for the read data is always an HVI Register.

Registers can be treated as signed or unsigned. The register size is 32 bits and numerical values must be within the signed or unsigned range. Registers are not required to be used for numerical values, you can use the 32 bits however you wish. You can add multiple registers at once as a register bank.

The following image shows a register bank in PathWave FPGA:

In the image, `Din_v` and `Dout_v` indicate signals.

`Din_v` is used to specify when a value is valid, so the bank will update the internal value of the register for when it is being read.

`Dout_v` indicates when the `Dout` value is valid, so it can be used by your custom logic.

Register read example

The instruction `fpga_register_read` is an HVI-native instruction that enables you to read from an HVI register in an FPGA sandbox, the destination must be an HVI Register.

The following code example shows an `fpga_register_read` instruction

```
# Read FPGA Register into an HVI Register
#
fpga_register = hvi.sync_sequence.engines["engine_name"].fpga_sandboxes["sandbox_name"].hvi_
registers["sandbox_register"]
readFpgaReg0 = sequence.add_instruction("Read FPGA Register_Bank_HviAction4Cnt", 10,
sequence.instruction_set.fpga_register_read.id)
readFpgaReg0.set_parameter(sequence.instruction_set.fpga_register_read.destination.id, hvi_
register)
readFpgaReg0.set_parameter(sequence.instruction_set.fpga_register_read.fpga_register.id, fpga_
register)
```

Register write example

The instruction `fpga_register_write` is an HVI-native instruction that enables you to write to an HVI register in an FPGA sandbox. The value to be written to the register is taken from an HVI register or from a literal.

The following code example shows an `fpga_register_write` instruction:

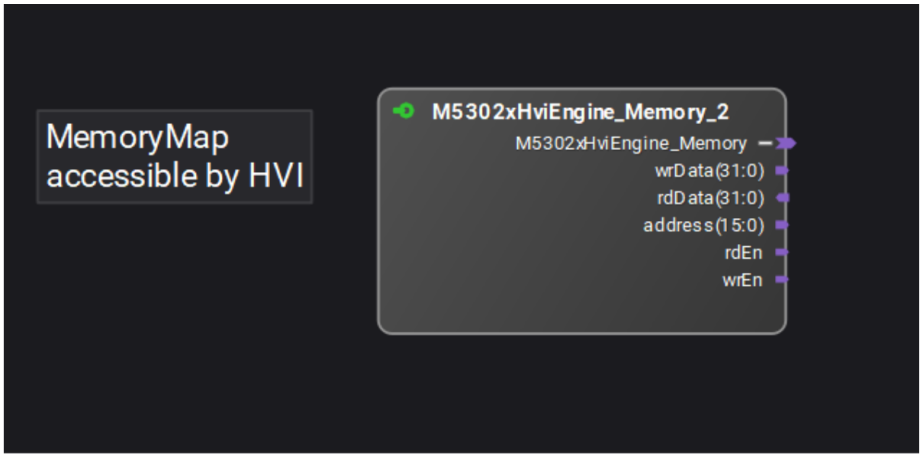
```
# Write to an HVI Register from an HVI Register used in an HVI sequence
#
writeFpgaReg0 = seq.add_instruction("Write FPGA Register_Bank_HviPxiTrigOut", 50,
hvi.instruction_set.fpga_register_write.id)
writeFpgaReg0.set_parameter(seq.instruction_set.fpga_register_write.fpga_register.id, fpga_
register)
writeFpgaReg0.set_parameter(seq.instruction_set.fpga_register_write.value.id, hvi_register)
```

HVI Memory maps

An HVI Memory map is an interface you add to an FPGA sandbox that enables you to to connect HVI sequences to a memory in the FPGA sandbox, or to custom logic that includes a memory block. The interface specifies a location and size that you define. The memories are always accessed 32 bits at a time.

To use the interface in HVI sequences, you must use the same name that you used in PathWave FPGA, otherwise you will not be able to access the memory.

The following image shows a Memory map in PathWave FPGA:



HVI Memory map read example

The following code example shows an HVI Memory map this is read with an `fpga_array_read` instruction. The destination is always an HVI register:

```
# Read Memory Map
#
readMemoryMap = sync_block_1.sequences["engine_name"].add_instruction("Read FPGA Memory Map",
20, hvi.instruction_set.fpga_array_read.id)
readMemoryMap.set_parameter(seq.instruction_set.fpga_array_read.fpga_memory_map.id, hvi_memory_
map)
readMemoryMap.set_parameter(seq.instruction_set.fpga_array_read.destination.id, register)
readMemoryMap.set_parameter(seq.instruction_set.fpga_array_read.fpga_memory_map_offset.id, 0)
```

HVI Memory map write example

The following code example shows an HVI Memory map that is written by a `fpga_array_write` instruction. The source can be a literal or an HVI register:

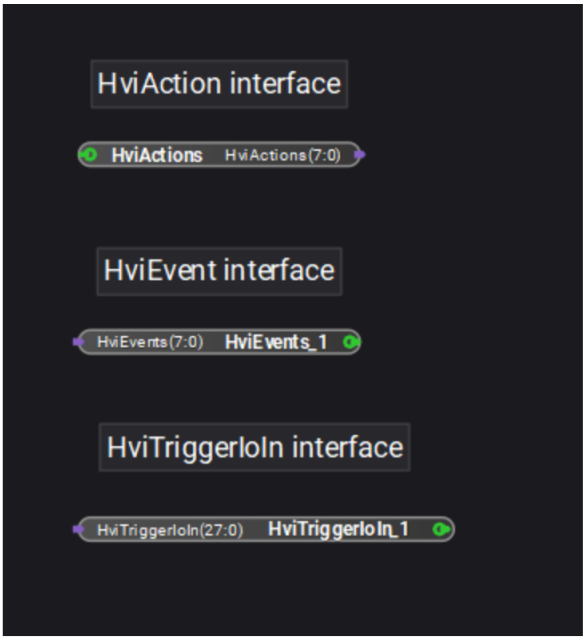
```
# Write Memory Map
#
writeMemoryMap = sync_block_1.sequences["engine_name"].add_instruction("Write FPGA Memory Map",
10, seq.instruction_set.fpga_array_write.id)
writeMemoryMap.set_parameter(seq.instruction_set.fpga_array_write.fpga_memory_map.id, hvi_
memory_map)
writeMemoryMap.set_parameter(seq.instruction_set.fpga_array_write.value.id, register)
writeMemoryMap.set_parameter(seq.instruction_set.fpga_array_write.fpga_memory_map_offset.id, 0)
```

For more information, see HVI API Local Statements.

Actions, Events and Triggers in an FPGA Sandbox

A number of interfaces for Actions, Events, and Triggers are available in PathWave FPGA that you can add to an FPGA Sandbox. These are are provided by the instrument that the FPGA is located in.

The following image shows HviAction, HviEvent, and HviTriggerIoIn interfaces:



Actions

You add Actions to enable you to send signals into the FPGA Sandbox from your HVI sequences.

For example, you can use an action to tell the logic in an FPGA Sandbox to send a signal.

Events

You add Events to inform your HVI sequences of events in the FPGA sandbox.

For example, you can use an event to get the FPGA to inform your HVI that an external signal has been received in the sandbox, or a signal has been generated in the sandbox.

You use the Wait-for-event statement to command your HVI to wait until an event occurs. A signal in the sandbox can initiate the event.

Triggers

You can add Triggers that go into or out of the sandbox.

For example, a trigger going into the sandbox can initiate an event in the sandbox.

You can also use an action to initiate the logic in the sandbox to send a trigger out of the sandbox, or you can use an event that is initiated when a trigger has arrived.

Depending on the Instrument, there may be a number of different trigger types available, for example:

- HviTriggerIoIn.
- HviTriggerIoOut.
- HviTriggerIoT.
- HviTriggerOutToLvds.
- LvdsToHviTriggerIn.

For a list of Actions, Events and triggers available for an instrument, see your instrument documentation.

Using FPGA Sandbox Actions, Events and Triggers in HVI Sequences

You can use the Actions, Events and Triggers that you added to an FPGA Sandbox in your HVI sequences. For Actions, Events and Triggers this is the same as you use any other instrument Actions, Events and Triggers, except for Triggers and Events where you must set the source to be `fpga_sandbox`.

For example, for actions:

- Use the `ActionDefinition` class to define the action.
- Add the definition to the `ActionDefinitionCollection.`
- Add the action to an HVI engine with the add method of the `ActionCollection.`
- In your sequence, add an action with `InstructionsActionExecute.`

FPGA Fast Data Sharing

HVI enables FPGAs on instruments to communicate with each other using *Fast Data Sharing* (FDS) technology.

Fast Data Sharing (FDS) is a technology that enables you to share data between FPGA sandboxes with a known fixed low latency. You can share data during the execution of sequences between sandboxes in different instruments in the same, or different chassis. The data sharing is performed using *Sync FPGA data-sharing* statements.

To take advantage of this FDS, you must use PathWave-FPGA to create a design in an FPGA sandbox that includes FDS ports.

Communication with Fast Data Sharing

FDS enables you to move data such as register values, or values of items such as qubit states. The data can travel between instruments on System Sync cables or on the PXIe DSTARB/C lines inside a chassis. FDS requires *System Synchronization Modules* (SSM) and PXIe instruments that support FDS technology. An advantage of FDS is that it does not use up additional triggers beyond those PathWave Test Sync Executive requires, so you are not required to reserve any additional triggers to use FDS.

HVI supports the following kinds of FDS transfers:

- Sharing FPGA Sandbox data with `SyncFpgaDataSharing` statements.

For FDS enabled instruments, Pathwave-FPGA provides the interfaces to use FDS. Timing and routing information is provided by the instruments.

HVI guarantees that the data is sent in the correct order, and that the communication timing and routing is computed automatically by HVI. HVI also automatically calculates the optimal communication timing to avoid collisions when data is transferred.

Accessing FPGA FDS Ports

When a PathWave FPGA project (`.k7z` file) is loaded, the memory maps, registers, and FDS ports are populated under the sandbox object.

You can access the list of FDS Port locations (`FdsPort` objects) defined in the FPGA sandbox by using an FPGA Sandbox Definition object that is loaded from the `.k7z` file.

The `FdsPort` object enables you to use the FDS port instances placed in the sandbox of a loaded PathWave FPGA project. An `FdsPort` has one property which is the name of the port.

The `FdsPort` can be set as a parameter in a `SyncFpgaDataSharing` Statement.

The following example shows how to use and program the FDS transactions in the `SyncFpgaDataSharing` statement.

The `FdsPortAddress` object enables you to specify both the source and the destination for each FDS transaction. This is done by specifying the name of the FDS port connected in PathWave FPGA to the block transmitting/receiving the data over FDS. The address where the data is read/written is also specified. Once the source and destinations are specified, each transaction can be added to the `SyncFpgaDataSharing` statement by specifying how many bits are shared in each transaction.

Python example:

```
# SyncFpgaDataSharing definition with 3 transactions
#
# Retrieve ports
instrument1_fds_ports = sequencer.sync_sequence.engines[instrument1_engine_name].fpga_sandboxes
[0].fds_ports
instrument2_fds_ports = sequencer.sync_sequence.engines[instrument2_engine_name].fpga_sandboxes
[0].fds_ports
instrument3_fds_ports = sequencer.sync_sequence.engines[instrument3_engine_name].fpga_sandboxes
[0].fds_ports
#
# Sources
instrument1_tx = kthvi.FdsPortAddress(instrument1_fds_ports[instrument1_tx_port_name], src1_
address)
instrument2_tx = kthvi.FdsPortAddress(instrument2_fds_ports[instrument2_tx_port_name], src2_
address)
#
# Destinations
instrument2_rx = kthvi.FdsPortAddress(instrument2_fds_ports[instrument2_rx_port_name], dst2_
address)
instrument3_rx = kthvi.FdsPortAddress(instrument3_fds_ports[instrument3_rx_port_name], dst3_
address)
#
# Adding Sync FPGA Data Sharing statement
fpga_data_sharing = sequencer.sync_sequence.add_sync_fpga_data_sharing("my statement", start_
delay)
# Transaction 1
fpga_data_sharing.transactions.add(instrument1_tx, instrument2_rx, num_bits_to_share)
#
# Transaction 2
fpga_data_sharing.transactions.add(instrument2_tx, instrument3_rx, num_bits_to_share)
#
# Transaction 3
fpga_data_sharing.transactions.add(instrument1_tx, instrument3_rx, num_bits_to_share)
```

FPGA-Instruction

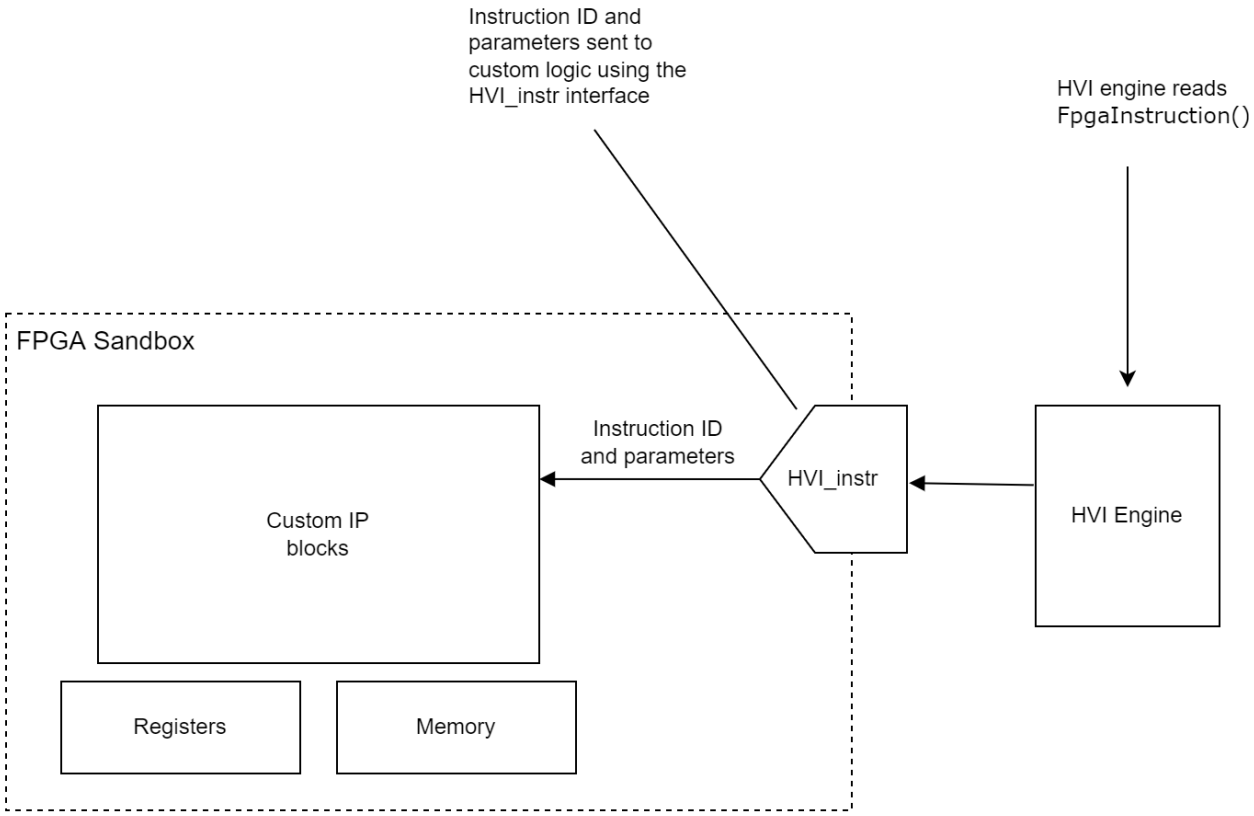
The FPGA-instruction enables you to issue custom commands into an FPGA sandbox to utilize customized logic.

You can customize logic in an FPGA sandbox using PathWave FPGA to create different functions. When you do this, adding an HVI_Instr interface enables you to interface with your logic from HVI sequences.

You use the FPGA-instruction statement in your sequences to issue the commands into the FPGA sandbox to utilize the different functionality. This means you can setup custom commands with different functions in the FPGA sandbox and utilize them in HVI sequences.

When an HVI Engine executes an `FpgaInstruction`, it also reads the parameters and the instruction ID and passes this data to the `HVI_Instr` interface in the sandbox, this interfaces an instruction parser and your logic.

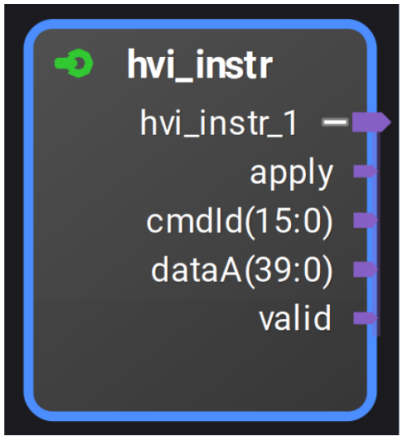
This flow is shown in the following diagram:



Integrating the Fpga-instruction Statement with FPGA sandbox Logic

If you want to issue commands to your logic with the `FpgaInstruction` statement, you must add an HVI instruction interface to the FPGA sandbox. Your logic must receive the parameters provided and then decode and execute the commands. How this is done depends on the instrument you are using, see your instrument documentation for more information.

The following image shows the HVI instruction interface as it appears in PathWave FPGA:



The signals are:

`apply`:

A flag that is typically used to apply stored configuration data in a multi-step setup process.

`cmdId`:

Command identifier. This is useful when more than one command is supported by the custom logic.

`dataA`:

General purpose data, 40 bits wide.

`valid`:

A flag used to identify when the data on the other ports is valid.

The following example shows an FPGA-instruction statement:

```
# Set up local sequence
fpga_inst = local_sequence.instruction_set.fpga_instruction
instruction = local_sequence.add_instruction('fpgaInstruction', 10, fpga_inst.id)
#
port_number = 2
data_a = 1234
command_id = 5
apply = 1
#
instruction.set_parameter(fpga_inst.port_number.id, port_number)
instruction.set_parameter(fpga_inst.data_a.id, data_a)
instruction.set_parameter(fpga_inst.command_id.id, command_id)
instruction.set_parameter(fpga_inst.apply.id, apply)
```

For more information see HVI API Local Statements.

HVI Statements for using FPGAs

PathWave Test Sync Executive includes a number of FPGA-specific HVI statements you can use to interact with the FPGA on an instrument:

Local statements

FPGA register read

The instruction `fpga_register_read` is an HVI-native instruction that enables you read from an HVI FPGA register to a destination HVI register.

FPGA register write

The instruction `fpga_register_write` is an HVI-native instruction that enables you to write an HVI FPGA register placed in an FPGA sandbox. The value to be written to the HVI FPGA register is taken from an HVI register or from a literal.

FPGA memory map write

The instruction `fpga_array_write` is an HVI-native instruction that enables you to write to an HVI FPGA memory map that is located in an FPGA sandbox. The value to be written to the HVI FPGA memory map is taken from an HVI register or from a literal.

FPGA memory map read

The instruction `fpga_array_read` is an HVI-native instruction that enables you to read from an HVI FPGA memory map. The value read from the HVI FPGA memory map is written to a destination HVI register.

FPGA-instruction statement

The `fpgaInstruction` statement enables you to issue commands to your custom FPGA Sandbox logic from within HVI sequences. This is an HVI-native instruction, but it can only be used successfully on instruments that support it.

For more information, see HVI API Local Statements.

Sync statements

Sync FPGA data-sharing statement

The Sync FPGA data-sharing statement enables you to transfer data between FPGA sandboxes.

For more information, see HVI API Sync Statements.

Chapter 6: Multi-Chassis Systems and System Synchronization Modules

This chapter describes how you use System Synchronization Modules to synchronize a Multi-Chassis System. It contains the following sections:

For information about troubleshooting a multi-chassis system, see the *System Setup Guide*.

System Synchronization Modules

This section describes System Synchronization Modules.

KS2201A 2022 release introduces new multi-chassis topologies that use the Keysight M9032A/M9033A PXIe *System Synchronization Modules* (SSMs). The previous means of interconnecting multiple PXI chassis using M9031A modules is discontinued starting from the KS2201A 2022 release. Compared to the discontinued M9031A module, the SSM has a much wider range of functions including:

- Distribution of a precise reference clock.
- Management of *Fast Data Sharing* (FDS).
- Chassis interconnectivity.
- Synchronization of all the PXI instruments in the multi-chassis.

M9032A and M9033A PXIe SSM Overview

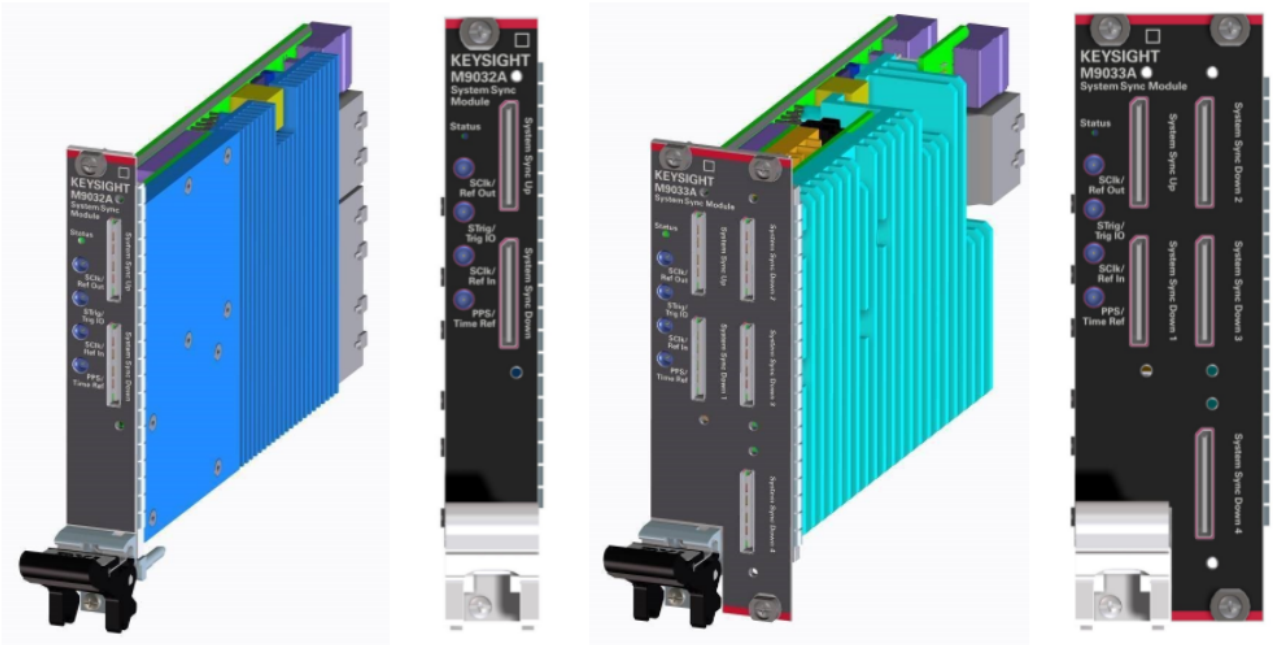
The M9032A/M9033A are PXIe *System Synchronization Modules* (SSM). These include an onboard high-quality 10MHz Oven Controlled Crystal Oscillator (OCXO) to achieve a very precise synchronization among various measurement instruments distributed across different chassis. The M9032A/M9033A System Synchronization Module functionalities can only be successfully deployed on chassis compliant with the *PXI-Express* (PXIe) standard. The SSM must be inserted in the timing slot of the PXIe chassis.

Keysight PXIe System Synchronization Module is available in two form factors, which only differ in their connectivity capabilities:

- M9032A is a one-slot PXIe System Synchronization Module with 1 System Sync Upstream and 1 System Sync Downstream ports.
- M9033A is a two-slot PXIe System Synchronization Module with 1 System Sync Upstream and 4 System Sync Downstream ports.

For further information about these SSMs including detailed performance specifications, see the M9032A/M9033A User's Guide, available at Keysight PXI Products.

The following image shows the physical M9032A and M9033A SSMs:



System Sync Module Connectivity

Front Panel

The M9032A and M9033A Front Panel contains various connectors that can be used for both multi-chassis interconnection and configuration of the reference clock source.

Front Panel SMP IOs

FP (Front Panel) SMP (Sub Miniature Push-on) connectors are:

SClk/Ref Out:

Outputs a copy of the system clock or a reference clock signal.

STrig/Trig IO:

Receives an arbitrary trigger signal.

SClk/Ref In:

Receives the reference clock signal.

PPS/Time Ref:

Receives a Pulse Per Second (PPS) signal.

The front panel SMP connectors can be used to share input and output reference clocks.

System Sync ports

System Sync ports use PCIe *Optical Copper Link* (OCuLink) connectors. System Sync ports are used for chassis interconnection and synchronization in the multi-chassis system. The signals in the System Sync include:

- Clocking (System Sync only).
- Triggering.
- Data.

The different SSM models have the following front panel System Sync ports:

The M9032A has 2 System Sync ports:

- 1 System Sync Upstream.
- 1 System Sync Downstream.

The M9033A has 5 System Sync ports:

- 1 System Sync Upstream.
- 4 System Sync Downstream.

Each System Sync Downstream port can connect to the System Sync Upstream port of another SSM placed in a different chassis. For more information, see the section below about Inter/Intra-chassis Connectivity.

PXIe Backplane DSTAR Connectivity

The M9032A and M9033A are placed in the Timing Slot of a PXIe chassis which enables them to support the DSTAR connectivity built-in the chassis.

DSTARA/B/C are multi-instrument point to point connections inside a chassis. DSTARA is used to carry the clock signal. DSTARB and DSTARC carry trigger or data signals.

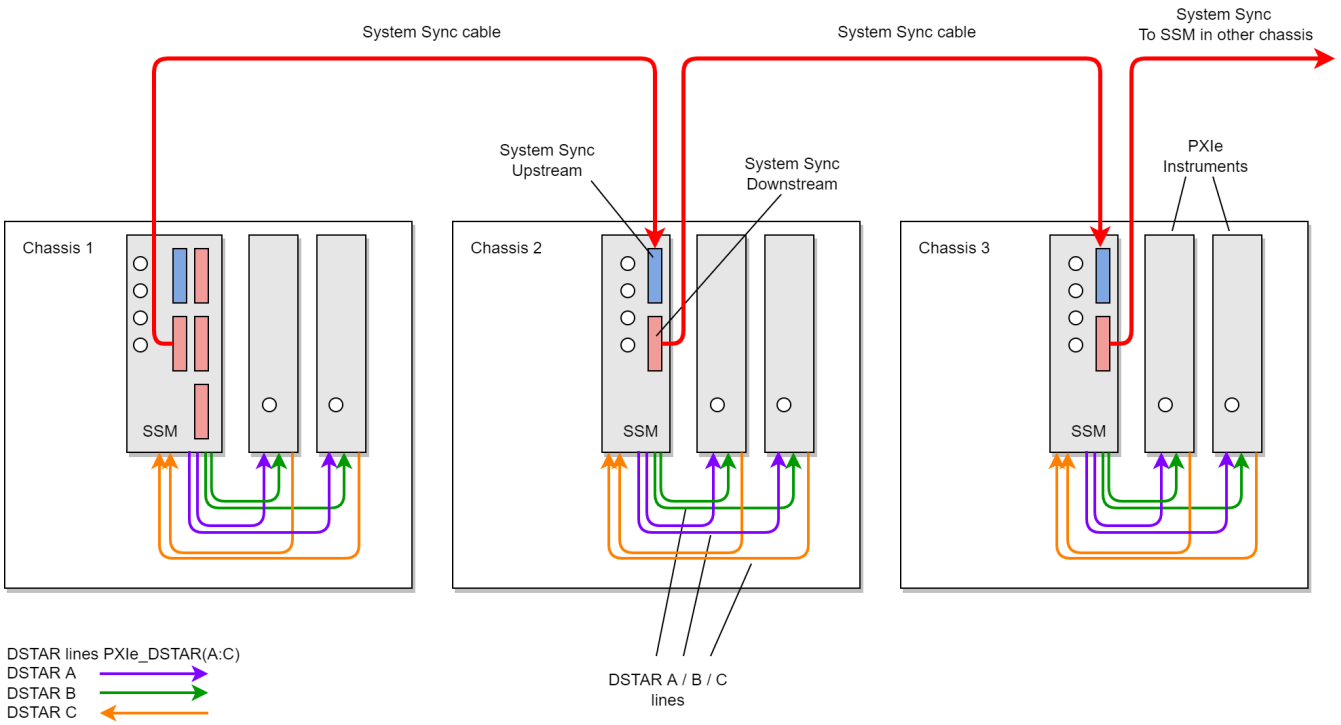
Inter/Intra-chassis connectivity, Synchronization and Data-Sharing Functionality

An SSM can enable both multi-chassis and multi-instrument interconnections. With these connections, SSMs enable synchronization and data sharing across all the instruments in a multi-chassis system.

- Multi-chassis interconnections are made with System Sync connections using their capability to interconnect two SSMs together through their System Sync Downstream/Upstream ports.
- Intra-chassis, multi-instrument interconnections are made with PXIe DSTARA/B/C connections. The SSM can share the precise reference clock over the DSTARA signal.

The following diagram shows a 3 chassis system connected with System Sync cables and DSTARA/B/C signals in each chassis:

3 Chassis system with System Sync



Data can be shared across System Sync and DSTAR connections in several different ways:

- The reference clock can be shared between two interconnected SSMs using the System Sync connection between System Sync Downstream/Upstream ports.

- The System Sync connection can share the signals sent over the PXI_TRIG[0:7] trigger buses, from one SSM to the next. This enables the SSMs to share PXI sync resources used by PathWave Test Sync Executive for the *Hard Virtual Instrument* (HVI) across the different chassis.
- System Sync connections can route data shared using *Fast Data Sharing* (FDS) between PXIe instruments.
- The SSM can send the data between two modules located in the same chassis using the DSTARB/C signals.
- Data can be sent through the System Sync connections to route it to instruments located in a different chassis.

Configuring a System with SSMs and System Sync Connectivity

This section describes how you use System Synchronization Modules to synchronize a Multi-Chassis System.

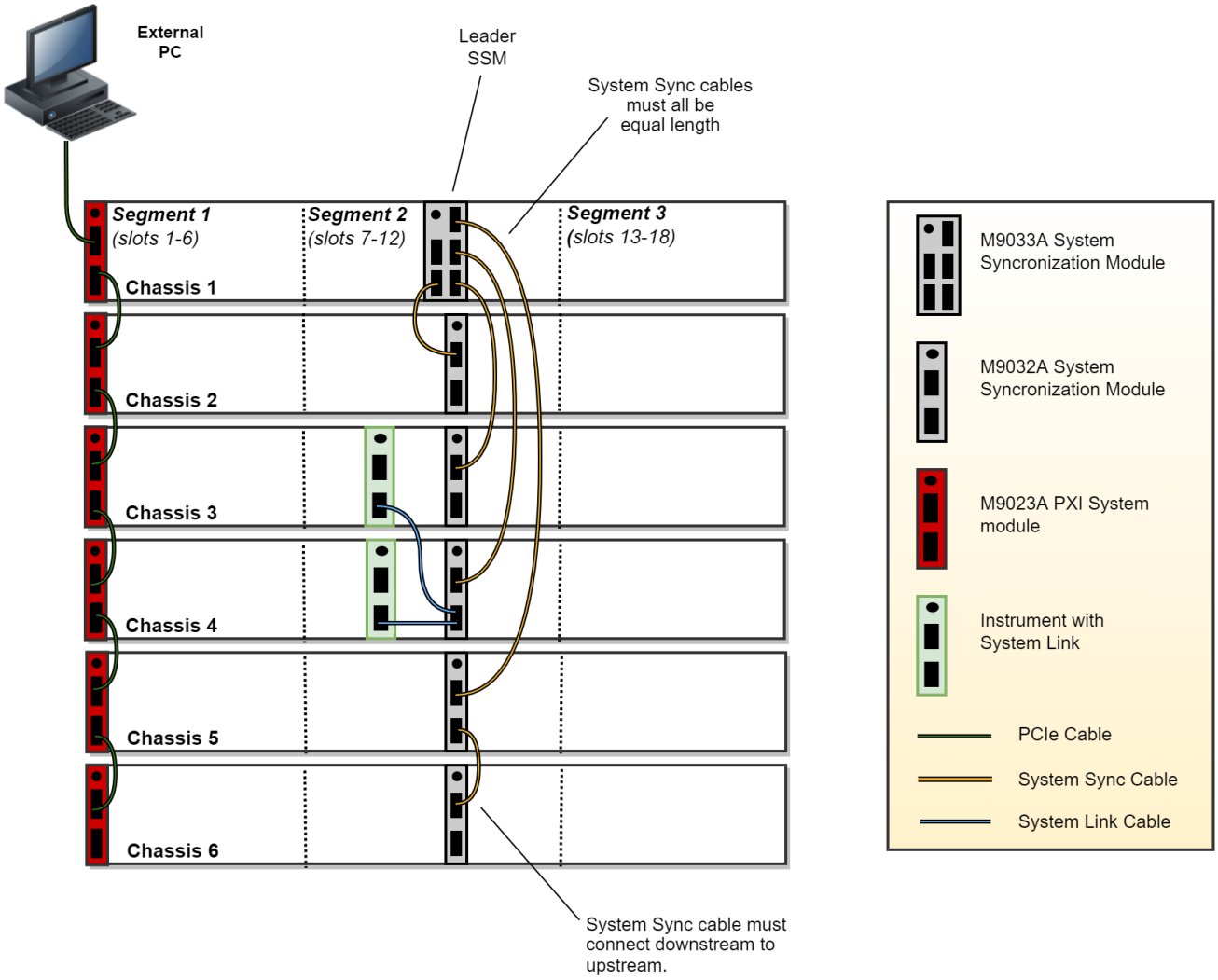
In a multi-chassis system connected with Keysight PXIe System Synchronization Modules, you must include one SSM in each chassis that is part of the system. Each SSM must be inserted in the **timing slot** of your chassis. This is typically slot 10 in Keysight 18-slot chassis, but it can be a different 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 new multi-chassis topology delivered by PathWave Test Sync Executive 2022, 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.



This following code shows how to use the HVI Python API to define and use the SSMs in the multi-chassis system shown in the diagram. Each System Sync Downstream port connects to the System Sync Upstream port of another System Sync Module in a different chassis.

The first step is to define the SSMs placed in each of the chassis during the system definition phase.

```
# Create system definition object
my_system = kthvi.SystemDefinition("MySystem")
#
# Define System Sync Modules
resource_id_ssm_1 = 'PXI0::CHASSIS1::SLOT10::INSTR'
resource_id_ssm_2 = 'PXI0::CHASSIS2::SLOT10::INSTR'
resource_id_ssm_3 = 'PXI0::CHASSIS3::SLOT10::INSTR'
resource_id_ssm_4 = 'PXI0::CHASSIS4::SLOT10::INSTR'
resource_id_ssm_5 = 'PXI0::CHASSIS5::SLOT10::INSTR'
resource_id_ssm_6 = 'PXI0::CHASSIS6::SLOT10::INSTR'
#
# In the options, SSMs are set to be simulated with Simulate=true and there are a number of
parameters.
# For the hardware SSM instruments, set options to an empty string.
options1 = "Simulate=true,DriverSetup=Model=M9033A"
options2 = "Simulate=true,DriverSetup=Model=M9032A"
options3 = "Simulate=true,DriverSetup=Model=M9032A"
options4 = "Simulate=true,DriverSetup=Model=M9032A"
options5 = "Simulate=true,DriverSetup=Model=M9032A"
options6 = "Simulate=true,DriverSetup=Model=M9032A"
#
sync_module_1 = my_system.interconnects.add_sync_module(resource_id_ssm_1, options1)
sync_module_2 = my_system.interconnects.add_sync_module(resource_id_ssm_2, options2)
sync_module_3 = my_system.interconnects.add_sync_module(resource_id_ssm_3, options3)
sync_module_4 = my_system.interconnects.add_sync_module(resource_id_ssm_4, options4)
sync_module_5 = my_system.interconnects.add_sync_module(resource_id_ssm_5, options5)
sync_module_6 = my_system.interconnects.add_sync_module(resource_id_ssm_6, options6)
```

NOTE In the HVI System Definition phase, the SSMs are added to the interconnects collection by using their resource ID and options. Same as for the chassis, it is not necessary to open objects representing the System Sync Modules (SSMs) that are included in the multi-chassis system.

The next step is to define the interconnections among the System Sync Downstream/Upstream ports of each pair of SSMs. The SSM System Sync ports can only be connected Downstream to Upstream.

```
# Define connections among System Sync connectors of the SSMs
#
# Connect SSM 1 to SSM 2
ssm1_downstream_sync1 = sync_module_1.connectivity.systemsync_downstream[0]
ssm2_upstream_sync    = sync_module_2.connectivity.systemsync_upstream[0]
ssm1_downstream_sync1.set_connection(ssm2_upstream_sync)
#
# Connect SSM 1 to SSM 3
ssm1_downstream_sync2 = sync_module_1.connectivity.systemsync_downstream[1]
ssm3_upstream_sync    = sync_module_3.connectivity.systemsync_upstream[0]
ssm1_downstream_sync2.set_connection(ssm3_upstream_sync)
#
# Connect SSM 1 to SSM 4
ssm1_downstream_sync3 = sync_module_1.connectivity.systemsync_downstream[2]
ssm4_upstream_sync    = sync_module_4.connectivity.systemsync_upstream[0]
ssm1_downstream_sync3.set_connection(ssm4_upstream_sync)
#
# Connect SSM 1 to SSM 5
ssm1_downstream_sync4 = sync_module_1.connectivity.systemsync_downstream[3]
ssm5_upstream_sync    = sync_module_5.connectivity.systemsync_upstream[0]
ssm1_downstream_sync4.set_connection(ssm5_upstream_sync)
#
# Connect SSM 5 to SSM 6
ssm5_downstream_sync = sync_module_5.connectivity.systemsync_downstream[0]
ssm6_upstream_sync    = sync_module_6.connectivity.systemsync_upstream[0]
ssm5_downstream_sync.set_connection(ssm6_upstream_sync)
```

Chassis Supported for Multi-Chassis Systems

The following Keysight chassis models are supported:

- M9018B
- M9019A
- M9046A

Software and firmware version requirements are listed on-line here: Chassis Software and Firmware Requirements for KS2201A .

NOTE If you mix different chassis models in your multi-chassis setup, you may observe some skew across the different chassis and different performance depending on the different chassis characteristics.

Non Keysight chassis are not supported for multi-chassis systems.

Clocking

This section describes Clocking:

Types of Clock

Clock Types

In a single or multi-chassis system there are 4 types of clocks used for synchronization and instrument-related tasks:

- Reference clock.
- System clocks.
- Analog clocks.
- Sample clocks.

All these clocks are synchronous with one another, but are used for different purposes and can be configured in different ways trading off performance and complexity/cost.

Reference Clock

The Reference clock determines the absolute frequency and lowest-frequency offset phase noise performance of the analog instrumentation's inputs and outputs. That is because all of the other clocks are phase-locked to the Reference Clock. A PXIe system can either use its own internal reference clock or phase-lock to an external reference clock. It can also provide external reference clock outputs for other instrumentation to phase-lock to.

System Clocks

The System clocks synchronize all the digital operation of all instruments and the PXIe platform. These clocks are derived from the Reference Clock and are used by, for example, the PathWave FPGAs Sandbox logic, the HVI Engine core clock, Fast Data Sharing and other digital capabilities in the instruments. Basically, a system clock is clock that is neither the reference clock nor an analog clock.

Analog Clocks

The Analog Clocks are intermediate frequency clocks from which the instrument's Sample Clocks are derived. Like the Sample clocks, the Analog Clocks affect the overall phase noise performance and skew drift of the instrument analog inputs and outputs. In the simplest clock configurations, each peripheral module generates it's own independent Analog Clock. In the highest fidelity clock configuration, a single common Analog clock is generated by the *High Performance Reference Clock Source* (HPRCS) and is distributed to all the individual peripheral modules though external cables and power dividers.

Sample Clocks

The instrument's ADCs and DACs that digitize analog input signals and generate analog output signals are clocked by their own internal Sample Clocks. The various types of peripheral modules use different sample clock frequencies even though they are ultimately derived from the same Reference clock. These sample clocks determine the overall phase noise performance and skew drift of the analog inputs and outputs because they directly clock the instrument's ADCs and DACs.

System Clock Distribution using SSM and System Sync connectivity

In a multi-chassis system based on the Keysight PXIe SSMs and chassis, the SSM with no other SSM connected to its System Sync Upstream port acts as the leader. This leader SSM forwards a copy of the system clock to other SSMs using System Sync cables. In turn, each SSM shares the forwarded system clock with the instruments located in their respective chassis using the PXIe DSTARA backplane signal.

NOTE You are not required to set the the leader in the HVI API. The leader SSM is determined by the hardware connections. That is, the leader role is automatically taken by the SSM that has no System Sync cable connected to its System Sync Upstream port.

Overview of Supported Clocking Schemes

There are several possible different clocking configurations, the one you should use depends on the hardware and the application requirements. Some of the key aspects to consider when selecting a clocking scheme are:

1. System and Analog clock sources. The source for the System and intermediate-frequency analog clocks is a critical element that determines the system synchronization, phase noise and drift performance. The clock sources covered in this section include:
   a. PXIe chassis.
   b. System Sync Module.
   c. PXIe Chassis with *High Performance Reference Clock Source* (HPRCS). This is only available on Keysight PXIe chassis models M904xA.

2. Internal/external Reference clock. The clock that serves as reference for the System/Analog clocks can be generated internally by the selected source, or externally provided by the user, generated by a clock source external to the PXIe system. In systems that include the *High Performance Reference Clock Source* (HPRCS), and other external instrumentation that you wish to share a common Reference Clock, the best overall jitter performance will usually be achieved by phase-locking the other external instrumentation to the HPRCS Reference Clock instead of the other way around. If the overall system needs to be phase-locked to a GPS or atomic standard reference, you should phase-lock the HPRCS to the GPS or atomic standard and phase-lock all the other instrumentation to the HPRCS Reference Clock.
3. Instruments internal/external Analog Clock. Most instruments can either use an external Analog Clock or generate their own Analog Clock internally for convenience, however, using a common external Analog

Clock will always provide the best performance because all peripheral module sample clocks will jitter and drift together.

The following table shows the different supported/recommended clocking schemes:

Clocking Scheme	Reference Clock Source	Reference Clock Mode	Description	Performance
A.1 Single-chassis, no SSM.	Chassis	Internal 10MHz	An OCXO inside the chassis generates a 10 MHz reference clock. Independent Analog clocks are generated in each peripheral module.	See the chassis datasheet for exact phase noise performance. See the M5xxx PXIe instrument documentation for exact performance of channel to channel skew, jitter, and drift.
	External	External 10MHz	The external reference clock must have a frequency of 10 MHz. As an example, it can come from a *Device Under Test* (DUT), another instrument that is part of the setup, etc. Independent Analog clocks are generated in each peripheral module.	-
A.2 Single/multiple chassis with at least one M904x, Analog clocks, SSMs.	Chassis	Internal 10MHz	An OCXO inside the chassis generates a 10 MHz reference clock. A common Analog clock is externally distributed to each peripheral module.	See the chassis datasheet for exact phase noise performance. See the M5xxx PXIe instrument documentation for exact performance of channel to channel skew, jitter, and drift.
	External	External 10MHz	The external reference clock must have a frequency of 10 MHz. As an example, it can come from a DUT, another instrument that is part of the setup, etc. A common Analog clock is externally distributed to each peripheral module.	-

Clocking Scheme	Reference Clock Source	Reference Clock Mode	Description	Performance
B No external Analog clocks, SSMs.	SSM	Internal 10MHz	An OCXO inside the SSM generates a 10 MHz reference clock. Independent Analog clocks are generated in each peripheral module.	See the SSM datasheet for exact phase noise performance. See the M5xxx PXIe instrument documentation for exact performance of channel to channel skew, jitter, and drift.
	External	External 10/100MHz	The external reference clock can have a 10 or 100 MHz frequency. As an example, it can come from a DUT, from another instrument that is part of the setup, etc. Independent Analog clocks are generated in each peripheral module.	-
C external Analog clocks, SSMs, HPRCS.	HPRCS	Internal 10MHz	The HPRCS generates a 2.4 GHz sine wave that gets divided in frequency to generate a 100 MHz reference clock signal. A common Analog clock is externally distributed to each peripheral module.	This option provides the best performance in terms of phase noise. For more information, see the *Keysight PXIe Chassis M9046A Datasheet*, available at Keysight PXI chassis.
	External	External 10/100MHz	The external reference clock for the HPRCS can have a 10 or 100 MHz frequency. As an example, it can come from a DUT or another instrument that is part of the setup, etc. A common Analog clock is externally distributed to each peripheral module.	-

Depending on the chosen clocking scheme, additional connections to the SSM may be required:

- A.1 or B Internal: No extra connections are required.
- B External: Attach the external reference clock source to the SSM **SClk / Ref In** input.
- A.2 or C Internal: The chassis Ref1 output must be connected to the SSM **SClk / Ref In** input.
- A.2 or C External: The chassis Ref1 clock output must be connected to the SSM **SClk / Ref In** input. The external reference clock must be connected to the chassis external **Ref In** input.

The precise connections are described in the configuration sections later in this guide.

You may also phase-lock external instrumentation to the PXI system using either 10 MHz or 100 MHz external reference clock outputs.

Some instruments also require an analog clock. For more information see the later section titled *Configuring Analog Clock Source for Instruments*.

Configuring the Reference Clock

This section describes configuring the reference clock:

Configuring Clocking Scheme A.1

(Single-chassis, no SSM, no external analog clocks)

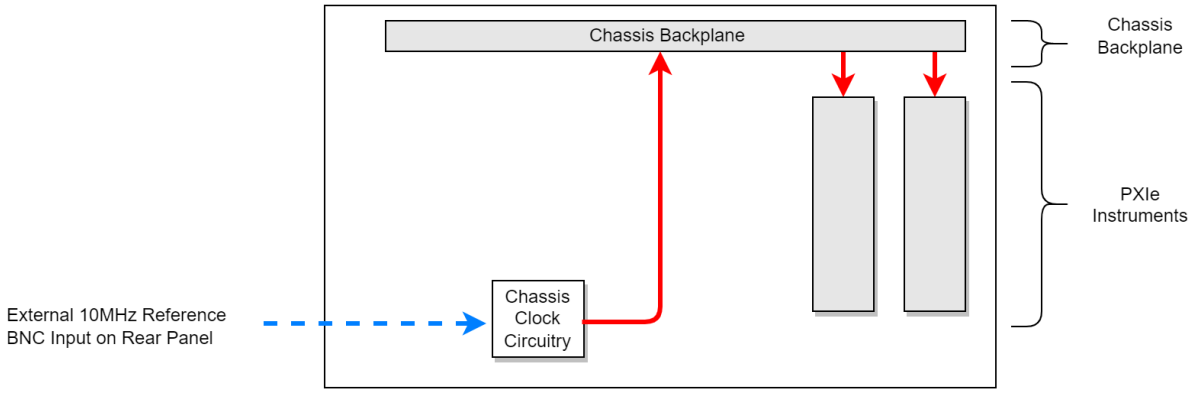
This is the simplest configuration and is the default if you have not specified another.

The chassis is the clock source. For the reference clock, there are two options:

1. **Internal (default):** This is the 10MHz clock built into the chassis (VCXO for the M9019A or OCXO for the M904xA).
2. **External:** A 10MHz signal connected to the 10MHz Ref BNC input located on the chassis rear panel.

The following diagram shows a chassis with an internal clock source (chassis clock) or an external clock source (blue):

Single chassis, no SSM
Internal or External Reference clock



All chassis have on their rear panel a 10MHz reference BNC input and a 10MHz reference BNC output. In the case of the M904x chassis, there are two Reference clock SMA outputs on the front panel.

This clocking scheme is rather constrained in terms of features because it only allows for a single chassis and, given that there is no SSM, advanced features like Fast Data Sharing are not available.

The following snippet shows how to configure the chassis as the clock source:

```
# Create system definition object
my_system = kthvi.SystemDefinition("MySystem")
#
# Define chassis
chassis = my_system.add_chassis(1)
#
# Select the chassis as ref. clock source
clockSource = chassis.clock_source
#
# Set the chassis as clock source
systemDefinition.clocking.reference_source = clockSource
#
# Explicitly set the clock source to use the internal OCXO as the reference clock (this is the
default)
clockSource.set_mode(keysight_hvi.ClockingReferenceMode.INTERNAL)
#
# Alternatively you can configure the chassis to use the external clock reference with the 10Mhz
frequency value in Hz
clockSource.set_mode(keysight_hvi.ClockingReferenceMode.EXTERNAL, 10e6)
```

Configuring Clocking Scheme B

(Single/multi-chassis system with SSM as clock source and no external analog clocks)

Each SSM is equipped with an onboard high-quality 10MHz Oven Controlled Crystal Oscillator (OCXO) that can be used as the reference clock.

Alternatively, the chassis backplane reference clock output (Scheme A.2) or the optional *High Performance Reference Clock Source* (HPRCS) output (Scheme C) can be used as reference clock. The HPRCS option requires a Keysight PXIe Chassis model M9046A. Clocking scheme B assumes the external reference clock is neither output by the chassis nor by the HPRCS because otherwise further configurations would be required for proper operation.

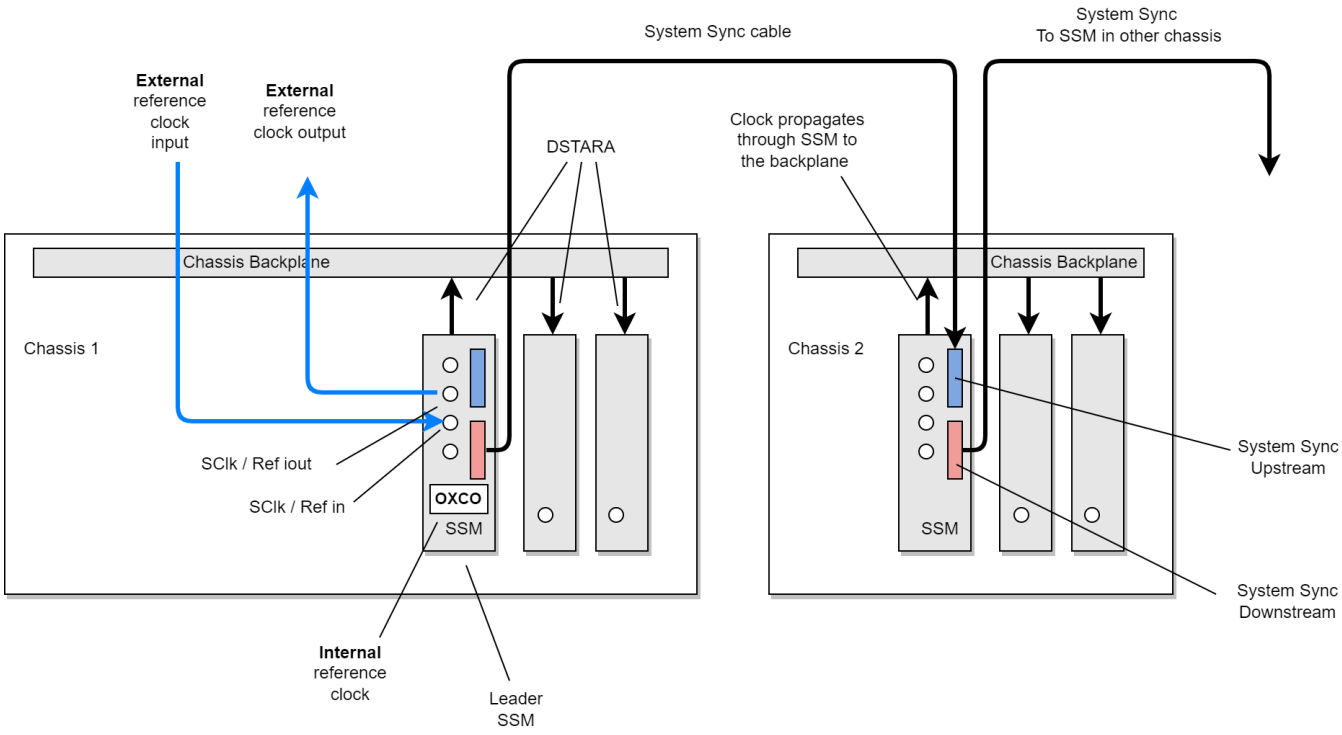
The reference clock can be chosen from two options:

1. **Internal:** This is the default mode. The internal OCXO of the leader SSM is used as the reference clock.
2. **External:** An 10 MHz or 100 MHz external reference clock is connected to the SSM's front-panel **SClk/Ref In** SMP input.

The reference clock gets propagated to all the PXIe instruments within the same chassis through the DSTARA signal path. It gets propagated to the next SSM through the System Sync cable from the downstream connection on leader SSM to the upstream connection on the follower.

The following diagram shows the operation of the Clocking Scheme B. The clock is generated in the SSM in chassis 1 and is passed to the other instruments in the chassis via the DSTARA signal path in the backplane (red arrows). It is also passed to the next chassis via the System Sync cable (in black) where it propagates via the SSM in that chassis. The internal reference is the SSM's OCXO, and the external reference is shown in blue:

Multiple chassis with SSMs
SSM as clock source with internal or external reference



Configuring the SSM as the System Clock source

By default, if you do not specify anything, PathWave Test Sync Executive configures the leader SSM as the reference clock source using its internal OCXO clock. The leader SSM is defined by the hardware connections. In the HVI API, no additional definition other than the connections between SSM is required to identify the leader SSM. You must ensure the connections you define in software match the physical hardware connections between SSMs.

The following code shows how to configure a pair of chassis with SSMs where the OCXO clock is the reference clock source, `options` is set to an empty string:

```
# Create system definition object
my_system = kthvi.SystemDefinition("MySystem")
#
# Define all necessary follower SSMs depending the number of chassis
leader_ssm = my_system.interconnects.add_sync_module(SSM_1, options)
my_system.interconnects.add_sync_module(SSM_2, options)
#
# Define chassis
my_system.add_chassis(1)
my_system.add_chassis(2)
#
#  Select the leader SSM as ref. clock source
clockSource = interconnects[0].clock_source
#
# Set the SSM clock source
systemDefinition.clocking.reference_source = clockSource
#
# Explicitly set the clock source to use the internal OCXO as the reference clock (this is the
default)
clockSource.set_mode(keysight_hvi.ClockingReferenceMode.INTERNAL)
```

Configuring the SSM to explicitly use internal OCXO or external reference clock

The SSM leading the synchronization by default with its internal reference clock, can optionally be connected to an external reference clock. The external reference can come from, for example, a DUT or another source such as a PXIe frequency reference.

To use an external reference clock, you must:

- Connect the external reference source to the SSM's SClk / Ref in port.
- In the HVI API you must set the SSM to synchronize to an external reference clock. To do this, set the mode to `EXTERNAL` and set the frequency in Hz.

To use the external reference, change the final line in the previous code snippet to:

```
# Set clock mode to EXTERNAL and set frequency to 10MHz
clockSource.set_mode(keysight_hvi.ClockingReferenceMode.EXTERNAL, 10e6)
```

Configuring Clocking Schemes A.2 and C

(Single/multi-chassis with Keysight M904xA chassis, external Analog clocks and chassis or HPRCS as clock source)

Some instruments such as the analog ones in the Keysight M5xxx PXIe family derive their Sample Clocks from an Analog Clock source and perform best when configured to use an externally distributed Analog Clock. This section explains how to distribute the analog clocks to these and similar instruments.

The analog clock can be generated from either the M9546A HPRCS or from the M9046A chassis backplane board. The preferred choice for the analog clock source is the M9546A HPRCS inside a Keysight M9046A PXIe chassis because of it's superior phase noise. The HPRCS can generate a sine wave with frequencies of 2.4, 4.8, 9.6, or 19.2 GHz. In this section we assume the analog clock source being set to generate 2.4 GHz, because this is the frequency required by the Keysight M5xxx PXIe family. The frequency can be configured at purchase by choosing the corresponding option for the Keysight M9046A PXIe chassis. For more information, see the *Keysight PXIe Chassis M9046A User Manual* available at Keysight PXI chassis.

Analog clock configuration options

the following table lists the options available:

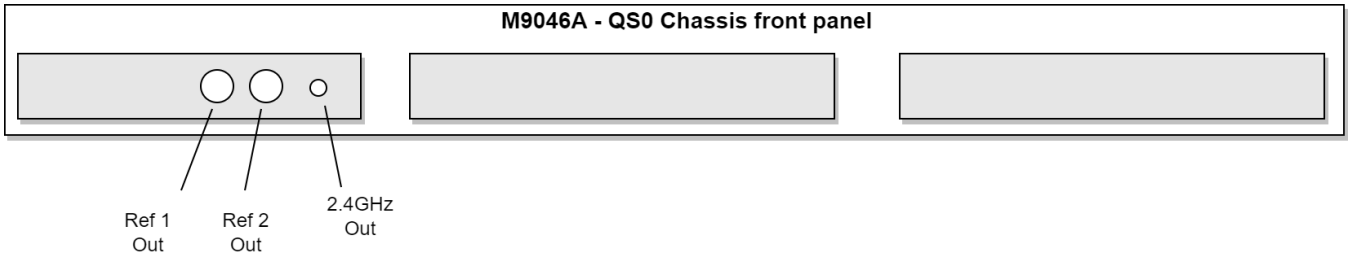
Source	Analog clock, locked to the reference clock	Performance
M9046A chassis with M9546A HPRCS	2.4, 4.8, 9.6, or 19.2 GHz	Best
M9046A chassis without M9546A HPRCS	2.4 GHz	Medium

M9046A Front Panel Clocking IO overview

The following diagrams show the M9046A chassis front panels and how they are connected in different configurations. The type and number of front panel connectors depend on the purchased hardware option for splitters and HPRCS: (-QS0, -QS1/3, -QS2). In the diagrams a frequency of 2.4 GHz is assumed to have been selected for the analog clock. The analog clock frequency is also chosen as hardware option at purchase time. More info in the *Keysight PXIe Chassis M9046A User Manual*, available at Keysight PXI chassis.

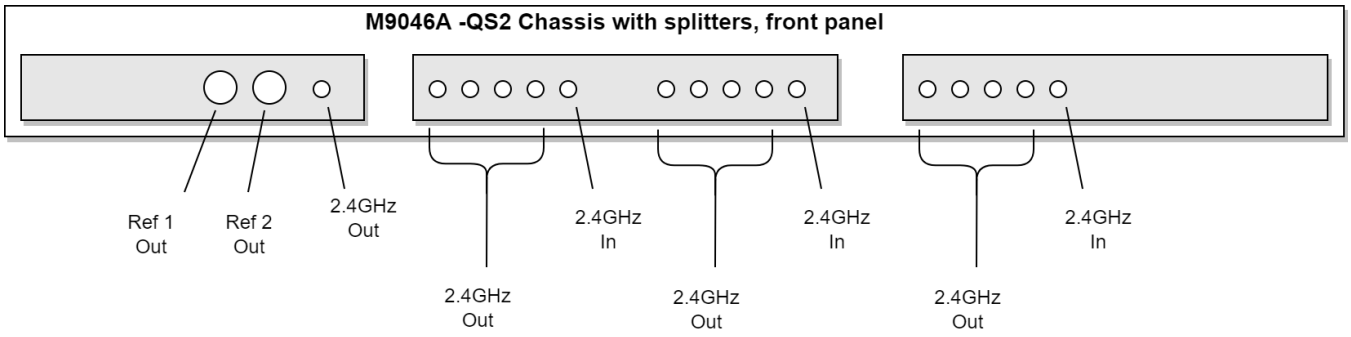
M9046A -QS0 Chassis (no HPRCS)

The following diagram shows the front panel of an M9046A -QS0 chassis.



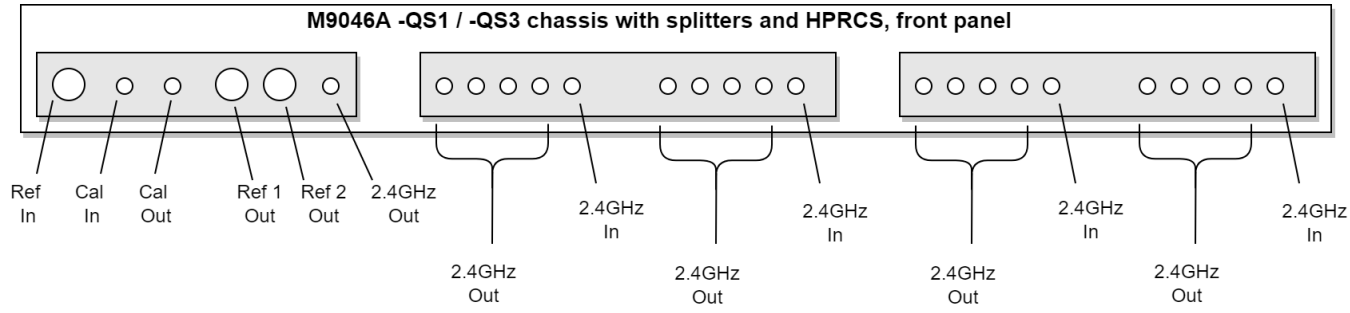
M9046A -QS2 Chassis with Analog clock splitters

The following diagram shows the front panel of a M9046A chassis with -QS2 option including the front panel analog clock splitters to ease the distribution of the analog clocks to all modules.



M9046A -QS1/3 Chassis with Analog clock splitters and HPRCS

The following diagram shows the front panel of an M9046A -QS1/3 chassis with analog clock splitters and Ref In, Cal In and Cal Out for the M9546A HPRCS.



Using the M9046A analog clock source

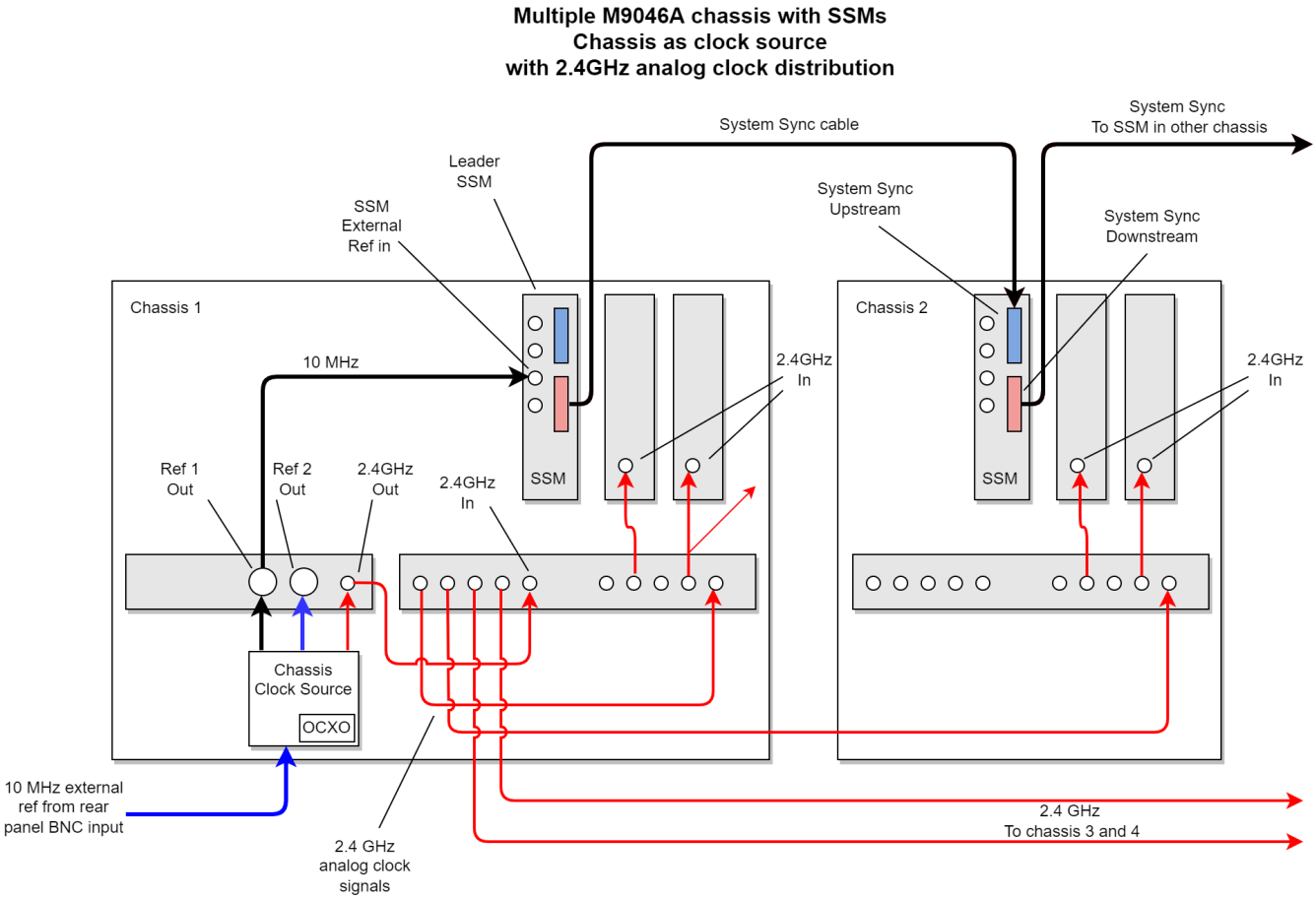
(Clocking Scheme A.2)

This configuration is for single or multiple chassis with SSMs. This configuration is compatible with PXIe Chassis model M9046A with hardware options -QS0, -QS1/3 or -QS2. The internal chassis clock is used as the clock source and this configuration must be defined in the HVI API. The chassis clock must be taken out from the **Ref 1 Out** port on the PXIe M9046 Chassis front panel and must be connected to the **SClk / Ref** In port of the PXIe SSM (see diagram below).

You can use the chassis as the reference clock source with its reference clock set to:

1. **Internal**: Use the chassis internal OCXO.
2. **External**: Using an external reference clock connected to the chassis rear panel 10MHz Ref BNC input.

The following diagram depicts the SSM using the chassis clock (indicated in red) as the clock source. The chassis external reference is indicated by the dotted blue arrow:



Configuring the M9046A as the system and analog clock source

To use the internal chassis clock, you must:

- Connect the Chassis Ref 1 Out output to the SSM's SClk / Ref In located in the same chassis.
- In the HVI API you must instruct the SSM to use the chassis clock.

By default, and if it is not specified otherwise, the chassis clock circuitry uses its internal OCXO as the reference clock.

The following code shows how to configure a pair of chassis with SSMs using the chassis clock as the reference clock, options is set to an empty string:

```
# Create system definition object
ktHvi.SystemDefinition definition("Name")
#
# You must add all necessary follower SSMs depending on the number of chassis
syncModuleLeader = definition.interconnects.add_sync_module(SSM_1, options)
syncModuleFollower = definition.interconnects.add_sync_module(SSM_2, options)
#
# Add chassis
chassis1 = definition.add_chassis(1)
definition.add_chassis(2)
#
# Get the chassis clock
clock_source = chassis1.clock_source
#
# Set as reference
definition.clocking.reference_source = clockSource
#
# Enable the chassis analog clock
clock_output_2_4GHz = ref_chassis.clock_outputs["FP2.4GHzOut"]
clock_output_2_4GHz.set_enabled(True)
```

Configuring the M9046A to use the external reference clock

To use the chassis clock with an external reference clock, you must:

- Connect the external reference clock to the Chassis rear panel's 10 MHz Ref BNC input.
- Connect the Chassis Ref 1 Out to the SSM SClk / Ref In of the SSM in the same chassis.
- In the HVI API you must instruct the chassis to use the external reference clock and set the frequency in Hz.

The following code shows how to set the external reference:

```
# Set the reference mode to use an external reference
# Set clock mode to EXTERNAL and set frequency to 10MHz
clockSource.set_mode(keysight_hvi.ClockingReferenceMode.EXTERNAL, 10e6)
```

Using the M9046A's M9546A High Performance Reference Clock Source

(Clocking Scheme C)

This configuration is for single or multiple chassis with SSMs. For this clocking scheme to be used, the first SSM must be in a Keysight M9046A chassis containing an M9546A *High Performance Reference Clock Source* (HPRCS).

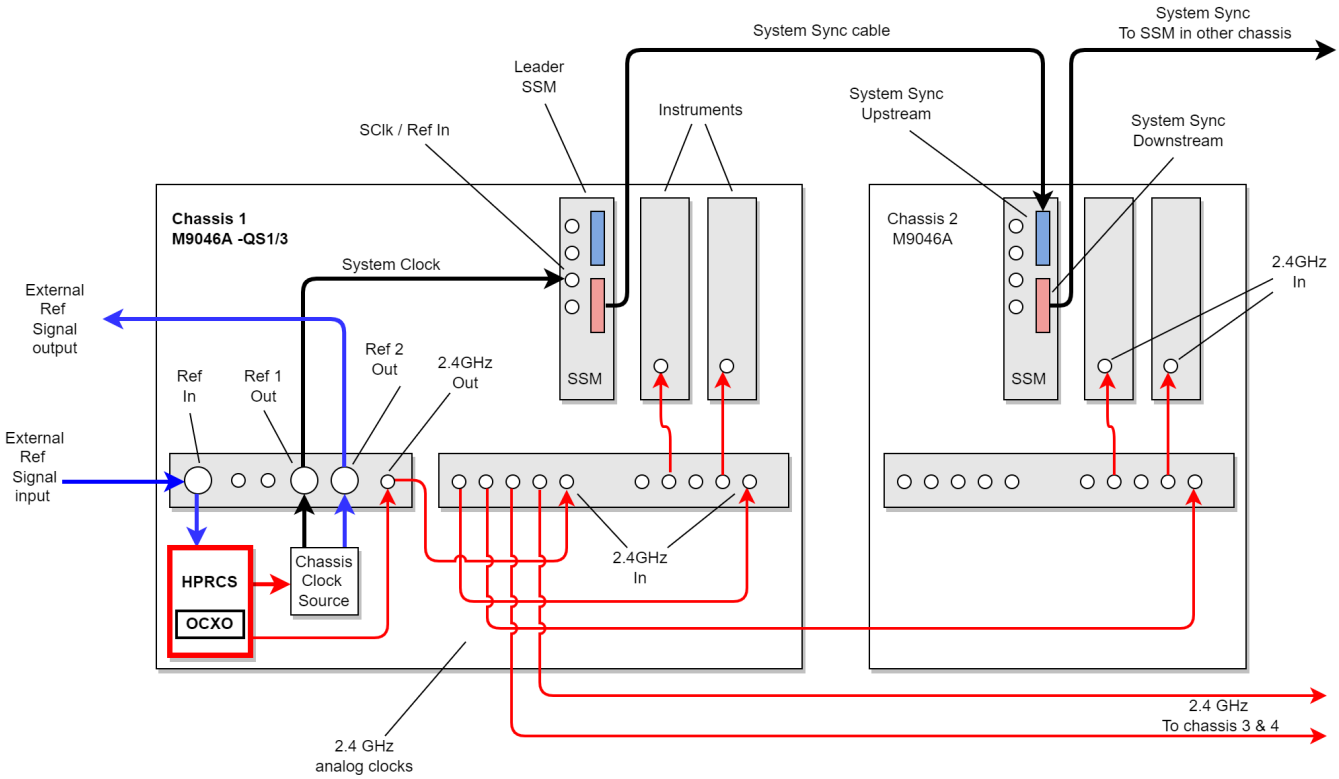
The HPRCS is used as the clock source and this configuration must be specified in the HVI API.

We can use the HPRCS as a clock source with its reference clock set to:

- Internal: Use the HPRCS internal OCXO.
- External: Use an external reference clock connected to the chassis front panel Ref In input.

The following diagram shows the leader SSM using the M9546A HPRCS (indicated in red) as the clock source in chassis 1 M9046A -QS1/3. The HPRCS external reference is indicated by the blue arrow. The distribution of the 2.4 GHz analog clock to up to 4 chassis is also shown. The connection topology and cables used are critical to achieving the optimal channel skew drift performance. For information about how to connect the analog clock to more than 4 chassis, see *Keysight PXIe Chassis M9046A User Manual* available at Keysight PXI chassis.

Multiple M9046A chassis with SSMs,
M9546A HPRCS as clock source in an M9046A -QS1/3 chassis
with external reference clock and 2.4GHz analog clock distribution



Configuring the M9046A + HPRCS as the system and analog clock source

To use the HPRCS as clock source, you must:

- Connect the chassis Ref 1 Out output to the SClk / Ref In of the SSM located in this chassis
- In the HVI API you must:
  - Add the M9046A -QS1/3 chassis with HPRCS to the system definition.
  - Set the HPRCS to be the clock source.

When no external reference clock for the HPRCS is specified, its internal OCXO is used.

The following code shows how to configure a pair of chassis with SSMs using the HPRCS as clock source, options is set to an empty string:

```
# Create system definition object
my_system = kthvi.SystemDefinition("MySystem")
#
# Define all necessary SSMs depending on the number of chassis
my_system.interconnects.add_sync_module(SSM_1, options)
my_system.interconnects.add_sync_module(SSM_2, options)
#
# Define chassis
hprcs_chassis = my_system.add_chassis(1)
my_system.add_chassis(2)
#
# Create HPRCS object
clockSource = hprcs_chassis.high_performance_clock_source
#
# Set the HPRCS as the reference clock
my_system.clocking.reference_source = clockSource
#
# Enable the chassis analog clock
clock_output_2_4GHz = ref_chassis.clock_outputs["FP2.4GHzOut"]
clock_output_2_4GHz.set_enabled(True)
```

Configuring the M9046A + HPRCS to use an external reference clock

To use the HPRCS with an external reference clock, you must:

- Connect the external reference clock to the chassis' front panel Ref In input.
- Connect the chassis' front panel Ref 1 Out output to the SClk / Ref In input of the SSM located in this chassis.
- In the HVI API you must:
  - Add the M9046A -QS1/3 chassis to the system definition.
  - Set the HPRCS to be the clock source.
  - Instruct the HPRCS to use an external reference clock and the desired frequency in Hz.

To use the external reference, set the reference clock mode in the previous code snippet to (defaults to internal):

```
# Set the reference clock mode. Set the HPRCS to use an external reference @10Mhz
clockSource.set_mode(keysight_hvi.ClockingReferenceMode.EXTERNAL, 10e6 )
```

Enabling Chassis Clock Outputs

If you are using a clock output from a chassis you can enable it in the HVI API.

The chassis clock outputs are available in the chassis and you can access them by their name as follows:

```
# Get the Clock configuration for the Rear Panel 10MHz output port from the Chassis
ktHvi.SystemDefinition definition("Name")
#
chassis = definition.add_chassis(1)
#
clockOutputRp10Mhz = chassis.clock_outputs["RP10MHzOut"]
clockOutputRP10Mhz.set_enabled(true/false)
```

Some clock outputs support one single frequency and others support multiple frequencies. For the outputs supporting only one frequency, no frequency must be provided when enabling/disabling them. If the clock outputs do support multiple frequencies, you must specify what frequency (in Hz) you want to enable.

When you disable the clock, the frequency argument is ignored.

The following code shows some examples and error cases:

```
clockOutputRp10Mhz = chassis.clock_outputs["RP10MHzOut"]
clockOutputRP10Mhz.set_enabled(true) # Ok
#
clockOutputFpRef2Out = chassis.clock_outputs["FPRef2Out"]
clockOutputFpRef2Out.set_enabled(true, 10e6) # Ok
```

Enabling the chassis Analog clock

If you are using an analog clock output from a chassis you must enable it.

The following code shows how to enable a 2.4GHz analog clock output from an M9046A chassis.

```
clock_output_2_4GHz = ref_chassis.clock_outputs["FP2.4GHzOut"]
clock_output_2_4GHz.set_enabled(True)
```

Configuring the Analog Clock Source in Instruments

For instruments that require an analog clock, you must set the source and frequency of the analog clock in your system definition.

You can set parameters for the analog clock:

- The source as internal or external.
- The frequencies of the sources, in Hz.

For external sources, the source selected depends on the analog clock frequencies that the instrument supports.

- If you indicate multiple frequencies, the first external frequency supported by the instrument is selected.
- If none of the external frequencies are supported, and the instrument has an internal clock, the internal clock is selected.
- If none of the external frequencies are supported, and the instrument does not have an internal clock, an error is generated.

The code is:

```
my_system.clocking.enable_external_analog_clocks(frequencies)
```

For example, if you are using a M9046A chassis with a M9546A HPRCS with the analog clock set to 2.4GHz (Clocking Scheme C), add the following line:

```
my_system.clocking.enable_external_analog_clocks([2400e6])
```

Instruments that support an external analog clock are set to use this clock. Instruments that do not support this external frequency are set to use an internal clock. If the instrument does not support the frequency and does not have an internal clock, an error is generated.

Selecting the best analog clock source for instruments

While it is often convenient for instruments to use their own internally generated analog clocks, the best jitter and drift performance is achieved by using a single common analog clock source generated within the Leader chassis (with or without the HPRCS) and distributing it using the chassis amplified power splitters in a balanced star configuration. This ensures that any low-frequency jitter skew drift is common across the system, minimizing the inter-channel jitter and drift.

In some cases with high channel count configurations, there may not be enough individual copies of the the Analog Clock available from a full balanced star distribution to connect to every instrument. In those cases, a single daisy-chain connection of the Analog Clock between instrument pairs can be used. Noting that the downstream instrument of the daisy-chained pair will have slightly higher skew drift than the non-daisy-chained instrument. Daisy-chained instruments shall have slightly higher skew drift, so these instruments should be the ones in the system which have the lowest bandwidth. For example, in systems which employ the M5201A Downconverter and the M5200 Digitizer, which are typically used in pairs, it is best practice to route the Analog Clock to the downconverter first and then daisy-chain the downconverter's Analog Clock output to the digitizer's Analog Clock input. This is because the 2 GHz digitizer is less sensitive to the same amount of channel skew than the 16 GHz downconverter.

The Keysight MCX cables are made of a special material that minimizes their propagation delay change with temperature. Matching the total propagation delay from their common clock source to each instrument causes the propagation delay drifts of the clocks to cancel out between instruments.

The balanced-star configuration of external Analog clocks uses custom 4:1 amplified power dividers built into some chassis. These power dividers are designed specifically for minimizing phase noise, temperature drift, and to maintain the Analog clocks amplitude as it is divided many times. Substituting other power dividers to distribute the Analog clocks will degrade jitter and drift performance, so this is not recommended.

Small spurious oscillations can occur within the amplified power divider when any of the outputs are loaded with certain reflective loads. For this reason, terminating unused outputs with 50 ohm loads is recommended. It is only necessary to terminate unused outputs of power dividers that are currently being used to distribute the Analog clocks.

NOTE If you are using a single M9046A chassis, and you are using an instrument as a synchronization source, ensure this instrument is in the middle segment.

Chapter 7: The HVI API

This chapter describes the HVI API. It describes the main classes required to understand the key programming concepts you must understand when you define your own HVI implementation.

The HVI API is a class-based API. It is a combination of the HVI-native API and the HVI instrument add-on API:

- The HVI-native API is the common API used by all instruments that support HVI.
- The HVI Instrument add-on API is an instrument-specific API that complements the HVI-native API.

NOTE The HVI-native API functions alone are not sufficient to fully execute HVI sequences on an instrument. To successfully run an HVI, you must use both APIs.

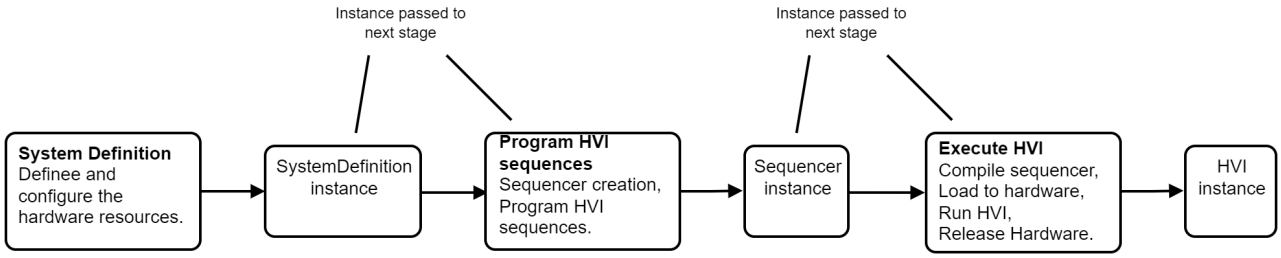
This chapter contains the following sections:

HVI API Main Classes and Use Model

PathWave Test Sync Executive has three main classes. You use them in this order:

1. SystemDefinition.
2. Sequencer.
3. Hvi.

Each of the stages in creating an HVI creates an object instance which is then passed to the next stage. The following diagram shows the stages:



NOTE Once an instance of SystemDefinition, Sequencer, or Hvi classes is created, you cannot modify it in the next HVI step. If you attempt to modify one of these instances at a later stage, the modifications will not apply. That is:

- You cannot modify the **SystemDefinition** instance at the "Program HVI Sequences" or "Execute HVI" stage.
- You also cannot modify the **SystemDefinition** or **Sequencer** instances at the "Execute HVI" stage.

SystemDefinition

You first define the hardware resources in the `SystemDefinition` class. This is the first step of building an HVI. You use the `SystemDefinition` class to define the hardware components, configuration and the resources available in your system. You do this by adding each of the resources to the relevant collection.

`SystemDefinition` contains classes for:

- Chassis.
- Interconnects.
- HVI system clocks.
- Non-HVI core clocks.
- Engines.
- FpgaSandboxes.
- Sync resources.

Once you have added the resources, you can initialize the system. Ensure you initialize the system after adding the resources.

NOTE The default initialization that happens when the Sequencer object is created, initializes all the HVI Engines included in the SystemDefinition object. If you initialize the system using the `initialize()` API method, ensure that all the HVI Engines are added to the SystemDefinition instance before you call `initialize()`.

Sequencer

Once the `SystemDefinition` object is defined and configured, you define and program HVI Sequences with a `Sequencer` object.

In the `Sequencer` object, the hardware collections you defined for the SystemDefinition are available as view collections. View collections enable you to use the hardware resources for Sequence programming, but you cannot modify them.

The `Sequencer` object contains classes for:

- SyncSequences and Sequences.
- Compilation.

The `SyncSequences` object in turn contains collections of Scope and Register objects. Local sequence can be programmed using the `InstructionSet` class.

After you have programmed your sequences, you use the compilation classes to compile the Hvi object.

Hvi

The `Hvi` object is the actual HVI instance that you load to hardware and execute.

`Hvi` contains runtime versions of the objects that you set up with the `SystemDefinition` and `Sequencer` classes. You use the runtime objects for executing the sequences on the hardware, but you cannot modify them.

`Hvi` contains the classes:

- `SyncSequenceRuntime`
- `EngineRuntimeCollection`
- `ScopesRuntimeCollection`

Further Explanations

Detailed explanations of all the main classes and their functions are provided in the help file provided with the KS2201A PathWave Test Sync Executive installer. This is located at:

`C:\Program Files\Keysight\PathWave Test Sync Executive 2022\api\python\Help`

HVI API Functionality

This section describes the functionality that is common across the HVI API. It contains the following sections:

- HVI API capabilities.
- HVI Collections.
- HVI API Error Management.

HVI API Capabilities

The HVI API provides many capabilities, including:

- Chassis/PXI backplane resource configuration.
- Interconnect configuration, for example, with *System Synchronization Modules* (SSMs).
- Access to HVI memory resources in the FPGA user Sandbox.
- Real-time sequencing:
  - Synchronized flow-control statements such as While loops.
  - Synchronized multi-sequence block statements that provide access to local instructions and flow control.
  - Local Instructions and operations. These include HVI-native and instrument-specific instructions.
  - Local flow-control such as While loops and If statements.

HVI Collections

Resources in HVI are grouped into Collections. Collections contain items of the same type, such as:

- Engines.
- Triggers.
- Actions.
- Events.
- Registers.
- FpgaSandboxes.

The concept of collections is fundamental in the HVI API use model because every component used within the HVI must be registered with a collection.

When you are defining an HVI instance, you define resources and add them to the corresponding collections. To register a component, add it to the corresponding collection of items of that type, for example, you must add a trigger to a trigger collection. Once registered, you can then use them inside HVI sequences.

Collections are particularly useful because the member instances can be accessed by index or string. Collections are located within the sequence hierarchy with their corresponding Sync or Local functions.

NOTE If a component is not registered with a collection, it cannot be used. You cannot use the engines, actions, triggers, events, or registers before they are defined and added to their corresponding collections.

Enhanced access properties of collections

Collections have additional access properties beyond those of vectors or lists.

Adding a new collection item

For example, you add new collection items by calling the `add()` method. This takes a name as its first parameter and returns the new item. The following code declares a new register, adds it to a registers collection, and returns the new register with the name `my_register_A`:

```
regA = instrument.registers.add('my_register_A', RegisterSize.SHORT)
```

NOTE Each name in a specific collection must be unique in that collection.

Random access by string or by numerical index

You access collection items with the `[]` operator. You can index items with their name, or by a number that indicates their location inside the collection.

You define the name when you add the item to the collection. For example, the following code returns an `Engine` object named `myEngine`:

```
instrument.engines["myEngine"]
```

To find the number of items in a collection, use either `count` or the built-in `len()` function. For example, the following code returns the number of Engines the instrument has:

```
len(instrument.engines)
```

Managing objects in a collection

The collection is a grouping of members, but it has no knowledge of the parameters or attributes of its members.

Definition and management of the instances within a collection are managed in their own classes, not in the collection class. For instance, you manage an Engine with the `Engine` class, not the `EngineCollection` class. Once an instance is defined, you then add it to the collection using the methods shown previously.

HVI API Error Management

Error handling in the HVI API is based on exceptions. If an error occurs during an HVI execution, the code execution is stopped, and a message is returned that includes an error code and a relevant error message. Error management is done through the Error class that is part of the HVI API.

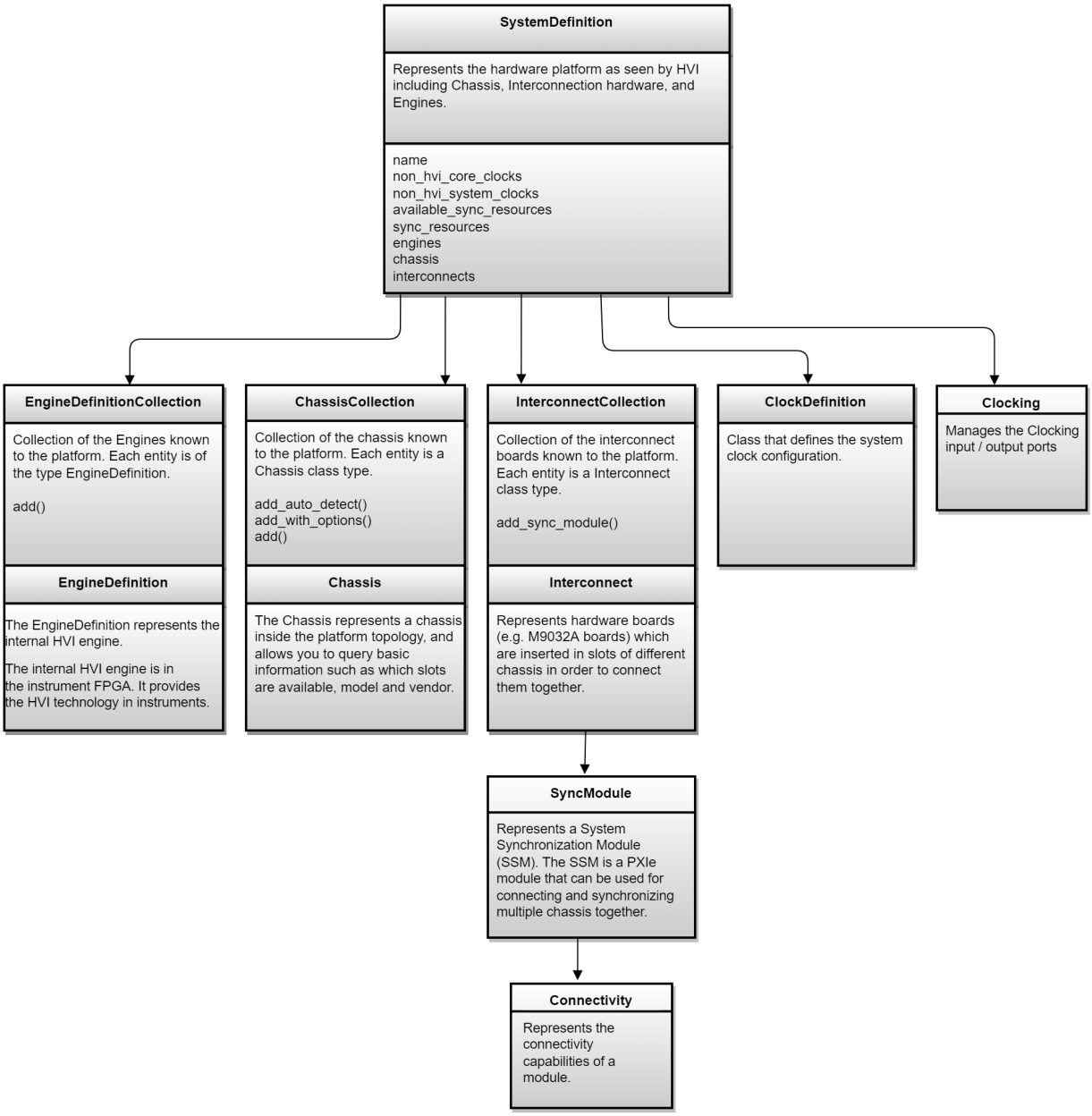
SystemDefinition

This section describes the `SystemDefinition` class, it contains the following sections:

- HVI Engines and their Resources
- Chassis, Interconnects and SyncModules Classes
- Synchronization Resources and Clocks
- User-defined trigger routing
- Clocking API
- Multi-process support
- System Initialization

You use `SystemDefinition` to configure the physical hardware resources available to the HVI. This class has interfaces to the Engines, Chassis, interconnects, and SyncModules.

The following diagram shows the classes:



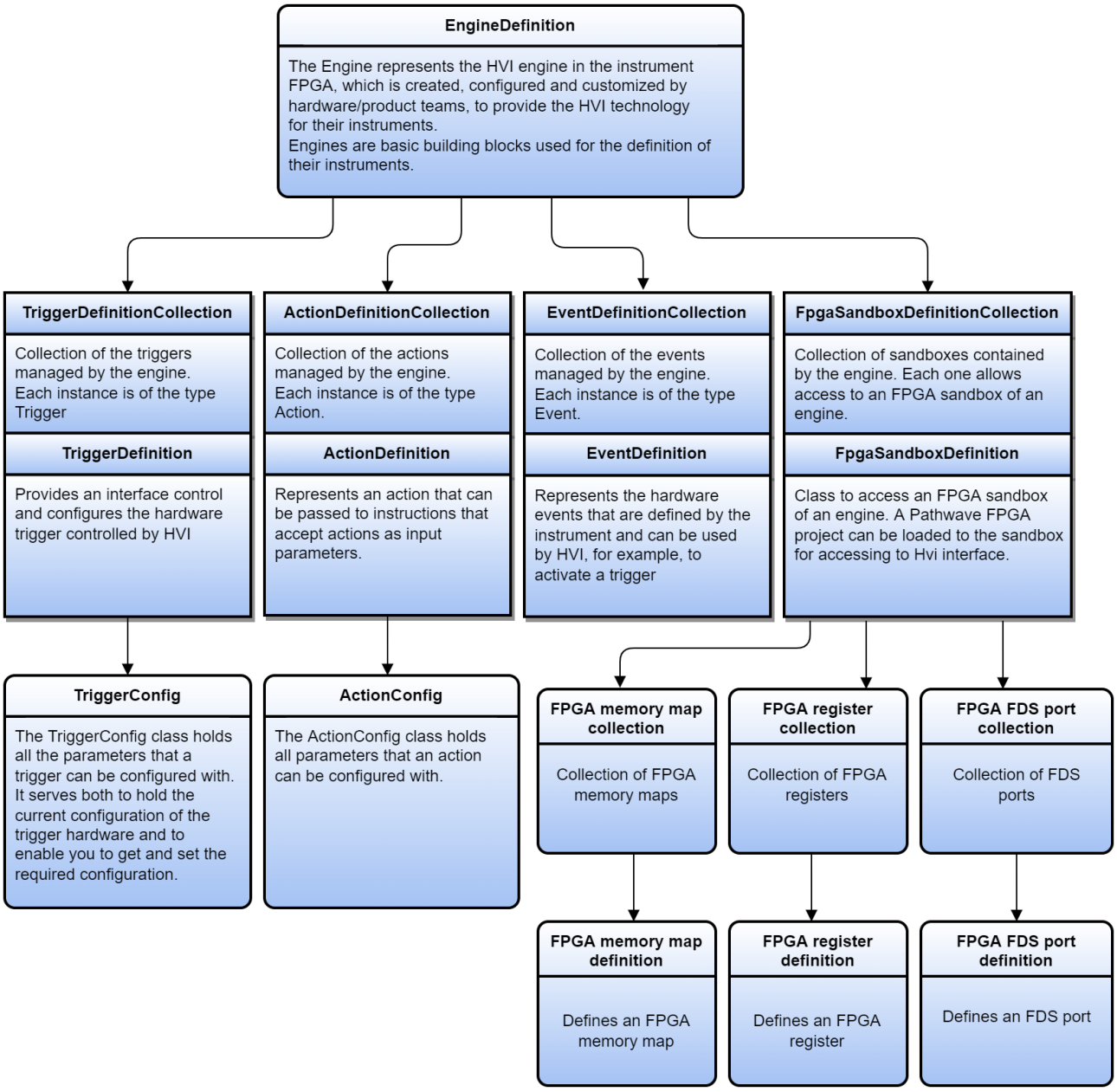
HVI Engines and their Resources

The Engine class provides access to the HVI Engines in the instruments.

You create instrument objects, where each object represents a physical PXIe instrument placed into a specific chassis and slot. You can the obtain the Engine object from the instrument object using the instrument-specific API and then add it to the list of HVI engines in the HVI engine collection. This collection is managed by the `SystemDefinition` object.

When a `SystemDefinition` object instance is created, an HVI engine collection is automatically created as well. This is managed through the `EngineCollection` class. You add HVI Engines to the collection by using the API method `add()` that is common to all collection classes. Each HVI engine manages its own Trigger, Action, Event, and FpgaSandbox collections.

The following diagram shows the classes:



Trigger definition

The `TriggerDefinitionCollection` class is used to add and manage all the trigger signal lines that are used by each HVI engine for triggering. When a trigger is added or queried from the TriggerDefinitionCollection, a TriggerDefinition object is returned. There are multiple types of triggers depending on their physical representation, for example, front panel triggers (usually a SMA connector on the module's front panel), PXIe triggers (connected to the PXIe backplane of the chassis), general purpose digital IO (LVDS connector in the module's front panel), and any other trigger lines enabled within the instrument.

The `TriggerDefinition` provides an interface to query trigger properties like Id, Name (user name), Hardware Name and type, and the TriggerConfig interface to configure the behavior of the trigger.

The `TriggerConfig` holds all the parameters to configure the trigger behavior. It holds the current configuration of the trigger hardware and enables you to query and define the required trigger behavior. The default configuration is included in the table below. The trigger configuration includes the following parameters:

Parameter	Description	Possible values	Default value
`direction`	Get or set the direction of the trigger	`Direction enum: INPUT, OUTPUT`	`INPUT`
`polarity`	Get or set the polarity of the output trigger	`TriggerPolarity enum: ACTIVE_HIGH, ACTIVE_LOW`	`ACTIVE_HIGH`
`trigger_mode`	Get or set the trigger mode	`TriggerMode enum: LEVEL, PULSE`	`LEVEL`
`sync_mode`	Get or set the synchronization mode of the trigger	`SyncMode enum: IMMEDIATE, SYNC, SYNC_BASE`	`IMMEDIATE`
`hw_routing_delay`	Get or set the delay of the trigger in nanoseconds	`Int`	0
`pulse_length`	Get or set the pulse length of the trigger in nanoseconds	`Int`	100ns

Action definition

Use the `ActionDefinition` class to define Actions in the HVI API. Before an action can be used you must register it to the `ActionDefinitionCollection` class that is within the Engine class. The registration locks the resource to the HVI instance for its use, when it is loaded to hardware.

Actions are used in sequences with action-execute instructions.

Event definition

The `EventDefinition` class is used to define Events in the HVI API. Before an event can be set up or used, it must be registered in the `EventDefinitionCollection` class within the Engine class that shall use this event. Registration locks the resource to the HVI instance for its use, when it is loaded to hardware.

FPGA sandbox definition

An FPGA sandbox is a user-configurable region in the FPGA. An HVI interface is provided to the sandbox for the instruments that support it. Through this interface, HVI can access read/write HVI registers and memory inside the sandbox.

To take configure the FPGA, you must use PathWave-FPGA to create your design in the sandbox. When the design is completed and built, PathWave FPGA generates a k7z file. This file is then used by HVI to get all the information needed about the names, addresses, ranges of the registers and memory-mapped locations that are connected to the HVI interface.

FPGA sandbox definition class

For the instruments that support user-configurable sandboxes, the sandboxes can be found in the engine's collection property `fpga_sandboxes,` where each sandbox can be accessed by its name. This returns an FPGA Sandbox Definition object that you then use to load the k7z file that was exported from PathWave FPGA. The HVI uses the k7z file to load the information related to this sandbox. Once the sandbox project is loaded, you can access the contents of the FPGA sandbox, that is, the register and memory map definitions.

```
SANDBOX_0_NAME = "sandbox0"
sandbox = engine.fpga_sandboxes[SANDBOX_0_NAME]
project_file = "c:/fpga/Hvi2SandboxTest.k7z"
sandbox.load_from_k7z(project_file)
```

FPGA register definition class

Using an FPGA sandbox definition object that has already loaded a k7z file, you can access the list of HVI registers (`FpgaRegisterDefinition` objects) defined in the sandbox. The `FpgaRegisterDefinition` objects have one property, the `name` of the register.

`FpgaRegisterDefinition` can be set as a parameter in `InstructionFpgaRegisterRead.fpga_register` and `InstructionFpgaRegisterWrite.fpga_register`.

```
fpga_register = engine.fpga_sandboxes[SANDBOX_0_NAME].fpga_registers[0]
fpga_register.name
```

FPGA memory map definition class

Using an FPGA Sandbox Definition object that has already loaded a k7z file, you can access the list of memory-mapped locations (`FpgaMemoryMapDefinition` objects) defined in the sandbox. The `FpgaMemoryMapDefinition` objects has two properties, the `name` and the `size` of the memory-mapped location.

`FpgaMemoryMapDefinition` can be set as a parameter in `InstructionFpgaArrayRead.fpga_memory_map` and `InstructionFpgaArrayWrite.fpga_memory_map`.

```
fpga_memory_map = engine.fpga_sandboxes[SANDBOX_0_NAME].fpga_memory_maps[0]
```

FPGA FDS Port definition class

Using an FPGA Sandbox Definition object that has already loaded a k7z file, you can access the list of FDS Port locations (`FdsPort` objects) defined in the sandbox.

When a PathWave FPGA project is loaded to an engine's sandbox, the memory maps, registers, and *Fast Data Sharing* (FDS) ports are populated under the sandbox object.

The `FdsPort` class enables you to use the FDS port instances placed in the sandbox of a loaded PathWave FPGA project. An `FdsPort` has one property which is the name of the port.

`FdsPort` can be set as a parameter in `SyncFpgaDataSharingStatement.`

The following code shows how to get FDS ports from the FDS port collections in engines.

```
# get FDS Ports for each engine
source_port_name = 'fds_tx_output_1'
dst_port_name = 'fds_rx_input_1'
#
module_1_fds_ports = sequencer.sync_sequence.engines["Module_1"].fpga_sandboxes[0].fds_ports
module_2_fds_ports = sequencer.sync_sequence.engines["Module_2"].fpga_sandboxes[0].fds_ports
#
source_address = 10
source_port = module_1_fds_ports[source_port_name]
source = keysight_hvi.FdsPortAddress(source_port, source_address)
#
dst1_address = 20
dst1_port = module_2_fds_ports[dst_port_name]
dst1 = keysight_hvi.FdsPortAddress(dst1_port, dst1_address)
```

Chassis, Interconnects and SyncModules Classes

This section describes the Chassis, Interconnect and SyncModule classes, and how to use them. It contains the following sections:

- Classes
- Opening Real or Simulated Devices

Classes

The following classes are supported:

Chassis class

The Chassis class represents a chassis inside your hardware platform topology, it enables you to query basic information such as which slots are available, the chassis model, and chassis vendor.

A Chassis has the following properties:

Property	Description
`number`	The chassis number
`first_slot`	The first slot number in the chassis
`last_slot`	The last slot number in the chassis
`model`	The chassis model
`vendor`	The chassis vendor
`triggering`	The triggering object of the chassis

Interconnects class

This class represents physical hardware boards that are inserted in slots of different chassis to connect them together.

The Interconnects class has the following properties:

Property	Description
`chassis`	The chassis number where the interconnect is located
`slot`	The slot number where the interconnect is located

SyncModule class

Class representing a *System Synchronization Module* (SSM). The SSM is a PXIe instrument that can be used for connecting multiple chassis together, synchronizing the multiple chassis and the instruments within, sharing an high performance clock reference across the multi-chassis system, and managing fast data sharing between PXIe instruments.

Property	Description
`chassis`	The chassis number where the SSM is located
`connectivity`	This describes the connectivity capabilities of the SSM and must be used to specify connections in software that reflect what is connected in your hardware setup.
`slot`	The slot number where the SSM is located

SyncModule sub-classes

SyncConnectivity

This class describes the connectivity capabilities of an SSM.

Opening Real or Simulated Devices

You can use PathWave Test Sync Executive with real or simulated hardware. The simulation mode enables you to test your sequences before running them on real hardware.

When you are opening a device such as a SSM or a Chassis, you can specify an options string. This is a string that contains a list of comma separated options. The options you specify are specific to the device you are opening and change depending on if you are opening real device or using a simulation.

NOTE In some cases a generic simulation built-in to HVI is provided, this is to enable you to get things up and running. A driver based simulation provides a more accurate simulation of the real hardware, so it is better for testing.

Options for opening SSMs

Real SSM

If you are using real SSM hardware, the options sting is typically empty. If you want to specify hardware options when using the SSM, see the SSM user manual for available options.

```
# Add SSM to Interconnects Collection

interconnects.add_sync_module(resource_id,"")
```

Simulated SSM

You can simulate a specific SSM with the driver for that SSM.

When simulating several options should be specified:

Set `Simulate=True`.

The following option must go after `DriverSetup=`

- `Model` specifies the model of SSM you want to simulate.

You can add a simulated SSM in the following way:

```
interconnects.add_sync_module(resource_id, 'Simulate=true,DriverSetup=Model=M9033A')
```

Options for opening a Chassis

Real Chassis

To add a real chassis do the following:

```
# Add chassis with number

my_system.chassis.add(chassis_number)
```

You can also use `add_with_options(chassis_number, options)`, but currently, the only options supported are the ones described in Simulated Chassis part below. Any other options you provide are ignored.

Simulated Chassis

You can simulate a chassis using a generic chassis simulation that is built in to HVI.

To enable chassis simulation, use the method: `add_with_options(chassis_number, options)` .

- `chassis_number` is the number of the chassis you want to simulate.
- `options` is a string that contains a list of comma separated options. You use these options to enable simulation mode and the chassis simulation, any other options are ignored.

The following code shows how to add a chassis in simulation mode using the built-in generic chassis simulation:

```
sys_def.chassis.add_with_options(chassis_
number, 'Simulate=True,DriverSetup=Model=GenericPxieChassis')
```

The *GenericPxieChassis* also simulates the the *High Performance Reference Cock Source* (HPRCS) if required.

Synchronization Resources and Clocks

HVI provides transparent multi-instrument synchronization and synchronized conditional execution, for example, the Sync while statement does synchronized conditional execution. To use these capabilities, for a *Device Under Test* (DUT) or instruments that do not integrate HVI technology, you must assign HVI synchronization resources and specify clock frequencies.

HVI synchronization resources

When you set up your system, you must allocate sufficient synchronization resources for your system and sequences to work correctly. Sync resources in the PXIe platform consist of the PXI Trigger lines. These are a limited resource, so you must be careful when you are allocating them.

The sync resources are used internally by the HVI to implement the following cross-instrument operations, transparently to the user:

- Alignment and Synchronization initialization.
- Real-time Sequencing multi-instrument operations, such as:
  - Sync while.
  - Sync register-sharing.
  - Triggered synchronization in a SyncMultiSequenceBlock.

The HVI optimizes the use of sync resources as much as possible and reuses the same sync resources when possible for different operations, providing they are executed with sufficient time separation. You can estimate the number of sync resources you require by working out how many are required at the different stages of your application.

The sync resources consist of PXI triggers and are defined by the enumeration `keysight_hvi.TriggerResourceId`. The resources must be specified in the `SyncResources` property of the `SystemDefintion` object. For example:

```
# Add sync resources
sys_def.sync_resources = [keysight_hvi.TriggerResourceId.PXI_TRIGGER0,
                          keysight_hvi.TriggerResourceId.PXI_TRIGGER1,
                          keysight_hvi.TriggerResourceId.PXI_TRIGGER2]
```

Where Sync Resources are used

There are 3 areas where you require Sync resources, these are the same 3 stages you set up your HVI in:

System Initialization

Initialization (`SystemDefinition.initialize()` call) requires one Sync resource for instrument synchronization. At this step the Sync resources is configured in HW and used to synchronize all hardware in the `SystemDefinition`, it is important that at this point the Sync resource is available and not in use by any other HVI instance or application. This Sync resource is reused later for the sequence execution, for example, if you use PXI Trigger 0 for synchronization, it will be reused later for the sequence execution.

Sequence Compilation

The HVI sequence requires Sync resources to execute specific multi-instrument real-time operations. Some operations that require Sync resources include:

- Sync while.
- Sync register-sharing.
- Triggered synchronization in a SyncMultiSequenceBlock.

During the sequence compilation, HVI allocates the Sync resources assigned in the `SystemDefinition` as required. So it is important that sufficient Sync resources are assigned for the sequence to compile, if this is not the case, a compilation error will be generated. At the compile stage, Sync resources are not used in hardware, they are just allocated to specific real-time operations in the code resulting of the sequence compilation. These resources will be configured and used in hardware when the HVI instance is loaded to hardware.

Sequencer Creation

Your sequence shall require sync resources to operate, but it can reuse the sync resources previously used in the SystemDefinition for initialization.

If you have not called initialize or it is otherwise required, the initialization still occurs at the beginning of the sequence. Sync resources are required for this, however these resource are reused by the sequence.

The numbers of sync resources required in a sequence depends on:

- The use of certain sync statements such as SyncWhile require 1 sync resource.
- The use of Sync register-sharing statements requires 1 sync resource per bit.
- Triggered synchronization requires 1 additional sync resource.
- The arrangement of your system also affects the number of sync resources required.

HVI Load to Hardware

The Sync resources required to initialize the system (synchronize all hardware) and those allocated to the HVI sequence during compilation, are configured into hardware at this step (`Hvi.load_to_hw()` call). The same Sync resources used to initialize the system are also used to run the HVI sequence. It is important that at the time of the `Hvi.load_to_hw()` call, to ensure the allocated Sync resources are not already in use in hardware by any other HVI instance or application.

Calculating the number of Sync Resources required

Different functionalities require different amounts of Sync resources, this can also depend on the system configuration, in particular if it is a small setup such as a single PXIe-chassis & single segment, or a large system with multiple chassis.

Sync resource usage per functionality

The following table summarizes the Sync resources required by the different functionalities.

Functionality		Sync resources required (for recommended Keysight chassis)	
#	Description	Single PXIe chassis & Segment	Others
1	SystemDefinition::Initialize() and sequence start in Hvi::Run()	1	
2	Sync Flow-Control While statement	1	
3	Sync Multi-Sequence blocks with Triggered-Sync (those with unknown execution time during compilation)	1	2
4	Sync Register sharing of N bits	N	

For information about recommended chassis, see Configuring a System with SSMs and System Sync Connectivity.

Sync resource reuse across functionalities

HVI reuses the same Sync resources for different functionalities and also for the same functionality if executed multiple times. The criteria to reuse Sync resources is:

- Functionalities #1, #2 and #3 reuse the Sync resources.
- Functionality #4 (Sync Register Sharing) reuse Sync resources ONLY when sender module are in the same Chassis and Segment

Calculating the total Sync resources required

To calculate the total amount of Sync Resources required, use the following formula:

- Total Sync Resources = Max(#1, #2, #3) + Sum(Max(#4 for each segment)).
- If a functionality is not used, use 0 in the equation above.

The following table shows examples with the number of sync resources required:

Scenario Description	Functionality				Sync Resource Total
	#1	#2	#3	#4	
System initialization only, SystemDefinition::Initialize() (any number of chassis)	1	-	-	-	1
SyncSequence (1x chassis, 1x segment) No Triggered-Sync SyncMultiSequenceBlocks + Sync-While	1	1	-	-	1
SyncSequence (1x chassis, 1x segment) + Triggered-Sync SyncMultiSequenceBlocks No Sync-While + RegSharing (chassis1, segment 1) (n bits)	1	-	1	n	n + 1
SyncSequence (2+ chassis) + Triggered-Sync SyncMultiSequenceBlocks No Sync-While + RegSharing (chassis1, segment 1) (n bits)	1	-	2	n	n + 2
SyncSequence (2+ chassis) + Triggered-Sync SyncMultiSequenceBlocks + Sync-While + RegSharing (chassis 1, segment 2) (n bits) + RegSharing (chassis 1, segment 2) (m bits)	1	1	2	Max (n,m)	Max(n,m) + 2
SyncSequence (2+ chassis) + Triggered-Sync SyncMultiSequenceBlocks + Sync-While + RegSharing (chassis 1, segment 1) (n bits) + RegSharing (chassis 2, segment 3) (m bits)	1	1	2	n + m	n + m + 2

HVI synchronization signals and modes

HVI uses different periodic digital signals for synchronization purposes. The definition of those digital signals depends on platform and instruments signals. Platform signals are the CLK100 and CLK10 signals in a PXI platform such as a PXI chassis. Instruments have different clock signals inside that are classified as core clocks or system clocks. Platform and instrument clock signals contribute to define the following HVI synchronization signals:

- Sync
- Sync_Base

Synchronization modes

You can configure the synchronization mode. This is used, for example, for generating a trigger value or waiting for an event.

The following modes are supported:

IMMEDIATE

The trigger or action is issued immediately, with no need to wait for any common synchronization clock. For the Wait-For-Event, the HVI execution continues immediately, as soon as the event is received.

SYNC

The trigger or action is issued at the first edge of the SYNC signal. For the Wait-For-Event, the HVI execution continues at the first edge of the SYNC signal, following the event arrival time.

SYNC_BASE

The trigger or action is issued at the first edge of the SYNC_BASE signal. For the Wait-For-Event, the HVI execution continues at the first edge of the SYNC_BASE signal, following the event arrival time.

For more information about synchronization, see Synchronization and Timing.

User-defined trigger routing

About Triggering

HVI uses triggers to communicate between engines in different slots of a PXI chassis, and in different PXI chassis through a SystemSync connection using SSM. HVI includes API methods that enable you to configure custom trigger routings across different instruments and chassis.

The following types of routing are supported:

- One *source* trigger, this must be a PXI trigger.
- One or more *destination* triggers, these must have the same PXI trigger number as the source, but in different slots and chassis.

For example, you can route:

From:

PXI trigger 3 in slot 2 of chassis 1

To:

PXI trigger 3 in slot 10 of chassis 1

and

PXI trigger 3 in slot 2 of chassis 2.

Trigger routing in the HVI API

The HVI API classes that you can use to get triggers and configure trigger routings are described in this section.

Types of trigger

The HVI API supports the following types of trigger:

Trigger

This is a standard HVI trigger, you use these for instruments with an HVI Engine. You get triggers from HVI Engines when you setup your system definition, see HVI Engines and their Resources.

You cannot use triggers that have already been added to sync resources.

PlatformTrigger

This type of trigger is used with instruments that does not contain an HVI engine. This can be any PXI instrument that uses triggers.

The triggering property of `Chassis` returns a triggering object of type `ChassisTriggering`. `ChassisTriggering` provides an interface to get a `PlatformTrigger` from a chassis.

Routing triggers

You configure the trigger routing through the triggering property of `SystemDefinition`.

A `trigger` can be a routing source or destination.

A `PlatformTrigger` can be a routing source or destination.

`SystemDefinition` provides a triggering property that returns a triggering object of type `SystemTriggering`. You use this to configure trigger routings.

System Triggering

`SystemTriggering` provides an interface to configure the routings.

Routing collection

Represents a collection of Routings.

Routing

Represents a route from one or more source TriggeringSignal to one or more destination TriggeringSignals.

TriggerSignal

Represents the signal that is routed between instruments.

Example of using the API to configure trigger routing

The following example shows how to get a `PlatformTrigger`, a `trigger` and then configure a route from one to the other:

```
# Create PlatformTrigger through the Chassis object
slot = 11
platform_trigger = system_definition.chassis[1].triggering.get_platform_trigger(keysight_
hvi.TriggerResourceId.PXI_TRIGGER0, slot)
#
# Create Trigger object through the Engine object
engine = system_definition.engines[0]
hvi_trigger = engine.triggers.add(moduleInSlot2.hvi.triggers.PxiTrigger0, "Destination")
#
# Add a user routing with the Platform trigger as the source, and the Trigger as the destination
routing = system_definition.triggering.routings.add(platform_trigger, [hvi_trigger])
```

Clocking API

In a hardware system, there are a number of different options for the system wide clock reference.

A clocking interface in the `SystemDefinition` class enables you to define the source of the system wide clocking reference along with a mode and frequency.

Setting the Source of the Reference Clock

You can set a *System Sync Module* (SSM) as a reference clock source.

```
#  Select the SSM as the source
clockSource = interconnects[0].clock_source
#
# Set the SSM clock source
systemDefinition.clocking.reference_source = clockSource
```

Alternatively, you set the chassis as a reference clock source with the following code:

```
# Select the chassis as the source
clockSource = chassis.clock_source
#
# Set the clock reference source
systemDefinition.clocking.reference_source = clockSource
```

Setting the Mode and Frequency

You can set the mode as `INTERNAL` or `EXTERNAL`.

`INTERNAL`

The reference clock source is internal. This is the default value.

Do not set the frequency, this raises an error.

`EXTERNAL`

The reference clock source is synchronized to an external clock.

You must set the frequency (in Hz) of the external sources.

To set the *High Performance Reference Clock Source* (HPRCS) as the reference clock do the following:

```
# This will be added only in the main chassis with the leader SSM.
ktHvi.SystemDefinition definition("Name")
#
chassis = definition.add_chassis(1)
clockSource = chassis.high_performance_clock_source
#
# Configuring HPRCS to use its internal clock
clockSource.set_mode(keysight_hvi.ClockingReferenceMode.INTERNAL)
#
# Configuring HPRCS to use an external reference @10Mhz
clockSource.set_mode(keysight_hvi.ClockingReferenceMode.EXTERNAL, 10e6)
#
# As SSM is required because the HPRCS output is connected to REF_IN input of the SSM.
syncModule = definition.interconnects.add_sync_module(resourceIdStm1, options)
#
definition.clocking.reference_source = clockSource
```

In some cases you may want to use the clock source device (Chassis, SSM or HPRCS) internal clock. This also enables you to use an external clock source to drive it, such as an atomic clock or a device under test. The following code shows how to configure the chassis as the clock source and take the clock reference from an external 10MHz source:

```
clockSource = chassis.clock_source
#
# Set clock mode to EXTERNAL and set frequency to 10MHz
clockSource.set_mode(keysight_hvi.ClockingReferenceMode.EXTERNAL, 10e6)
#
systemDefinition.clocking.reference_source = clockSource
```

However, if you want to explicitly configure the clock source to use the chassis internal OCXO clock source:

```
clockSource = chassis.clock_source
#
# Set clock mode to INTERNAL
clockSource.set_mode(keysight_hvi.ClockingReferenceMode.INTERNAL)
#
systemDefinition.clocking.reference_source = clockSource
```

Getting the Mode and Frequency

If you did not set the mode or frequency, you can get the mode and frequency of the clocking reference with the following code:

```
# Get mode and frequency (in Hz)
#
mode = clockSource.mode
frequency = clockSource.frequency
```

Chassis Clock Outputs

The chassis have internal clocks and outputs for them. For instance, the clock output on the rear panel of an M9019A or the clock outputs on the front panel of an M9046A. These clock outputs can be use as a reference clock for instruments in the system and for devices external to the system. The HVI API enables you to enable or disable the chassis clocks.

For the clock options and outputs available on your chassis, see your chassis documentation.

NOTE Chassis Clock Outputs do not have any default behavior. If the user does not specify any configuration for a clock output (see below) the clock output is left untouched.

Enabling chassis clock outputs

The chassis clock outputs are available in the chassis and you can access them by their name as follows:

```
# Get the Clock configuration for the Rear Panel 10MHz output port from the Chassis
ktHvi.SystemDefinition definition("Name")
#
chassis = definition.add_chassis(1)
#
clockOutputRp10Mhz = chassis.clock_outputs["RP10MHzOut"]
clockOutputRP10Mhz.set_enabled(true/false)
```

Some clock outputs support one single frequency and others support multiple frequencies. For the outputs supporting only one frequency, no frequency must be provided when enabling/disabling them. If the clock outputs do support multiple frequencies, you must specify what frequency (in Hz) you want to enable.

When you disable the clock, the frequency argument is ignored.

The following code shows some examples and error cases:

```
clockOutputRp10Mhz = chassis.clock_outputs["RP10MHzOut"]
clockOutputRP10Mhz.set_enabled(true) # Ok
clockOutputRP10Mhz.set_enabled(true, 10e6) # Throws error, no frequency expected
clockOutputRP10Mhz.set_enabled(false, 10e6) # Ok, frequency is ignored
#
clockOutputFpRef2Out = chassis.clock_outputs["FPRef2Out"]
clockOutputFpRef2Out.set_enabled(true) # Throws error, frequency expected
clockOutputFpRef2Out.set_enabled(true, 10e6) # Ok
clockOutputFpRef2Out.set_enabled(false, 10e6) # Ok, frequency is ignored
```

Enabling the chassis Analog Clock Output

If you are using an analog clock output from a chassis you must enable it manually.

The following code shows how to enable a 2.4GHz analog clock output from an M9046A chassis.

```
clock_output_2_4GHz = ref_chassis.clock_outputs["FP2.4GHzOut"]
clock_output_2_4GHz.set_enabled(True)
```

Automatic clock output enable

If the you defines the HPRCS clock from the M9046A chassis as the clock source, you must connect the Front panel **Ref 1 Out** port (FPRef1Out) from the leader M9046A to the REF_IN of the leader SSM. HVI automatically enables the **Ref 1 Out** clock output port from the leader chassis to let the leader SSM take the clock reference from it.

The following code shows an example:

```
# This is only added in the main chassis with the leader SSM.
ktHvi.SystemDefinition definition("Name")
#
chassis = definition.add_chassis(1)
clockSource = chassis.high_performance_clock_source
#
# Configure the HPRCS to use its internal clock
clockSource.set_mode(keysight_hvi.ClockingReferenceMode.INTERNAL)
#
# As SSM is required because the HPRCS output is connected to REF_IN input of the SSM.
syncModule = definition.interconnects.add_sync_module(resourceIdStm1, options)
#
definition.clocking.reference_source = clockSource
#
# The following lines are not required, HVI does this automatically
#clockOutputFpRef1Out = chassis.clock_outputs["FPRef1Out"]
#clockOutputFpRef1Out.set_enabled(true)
```

Enabling the Analog Clock Source in Instruments

For instruments that require an analog clock, you must set the source and frequency of the analog clock in your system definition.

You can set parameters for the analog clock:

- The source as internal or external.
- The frequencies of the sources, in Hz.

For external sources, the source selected depends on the analog clock frequencies that the instrument supports.

- If you indicate multiple frequencies, the first external frequency supported by the instrument is selected.
- If none of the external frequencies are supported, and the instrument has an internal clock, the internal clock is selected.
- If none of the external frequencies are supported, and the instrument does not have an internal clock, an error is generated.

```
my_system.clocking.enable_external_analog_clocks(frequencies)
```

If the instrument does not support the frequency and does not have an internal clock, an error is generated.

Multi-process support

About multi-process

The multi-process feature is a communications infrastructure that enables the future expansion of HVI operation to larger systems. *Keysight Distributed Infrastructure* (KDI) is the underlying service that supports this feature. On top of KDI, PathWave Test Sync Executive will use the same existing programming paradigm, enabling an easy transition for existing users.

In this release the multi-process feature enables you to open multiple instruments in a process or connect to a single instrument from multiple processes.

How to use multi-process

The programming model of a multi-process system is much the same as for a single system. You describe the whole HVI environment (engines, sync resources, sequences, etc.) in a single script as before. Under the hood, the *real* software objects are created where they have direct access to the hardware resources they control. You do not have to deal with the internals of this mechanism, but internally the *remote object* concept is introduced to manage a different execution process.

To keep the existing programming interface for multi-process operations, the creation of a remote object is defined the same way as it was for an existing one, except for adding the location to its description as a URL address.

For example:

```
#MyHviEngine@PXI0::CHASSIS1::SLOT2::INSTR
```

is transformed to:

```
#MyHviEngine@PXI0::CHASSIS1::SLOT2::INSTR<<HVITCP:[::1]:34960>>
```

This creates a remote engine on the local host using a connection through port number 34960. After this, the engine name `MyHviEngine` is used in the script the same way as always.

HVIServer port options

In multi-process operation, you connect to individual instruments remotely via an HVIServer port. Since each instrument is run in its own process, a different HVIServer port must be defined for each instrument.

These are specified with for example, `HVITCP:127.0.0.1::2000` where `2000` is the port number, or `HVITCP:[::1]:0` where 0 specifies a dynamic port.

NOTE Keysight recommends you use dynamic ports whenever possible as it avoids port collisions in a multi-instrument environment.

The following table provides some examples:

Configuration	Option Name	Option Value	Option Value Example	Description
KSF-IPC	HviServer	HVITCP:<ipv4>:<port> HVITCP:<ipv6>:<port>	HVITCP:127.0.0.1::2000 HVITCP:[::1]:2000	Address for opening a server with a port number.
			HVITCP:[::1] HVITCP:[::1]:0 (dynamic port)	If no port is set, or the number 0 is specified, a dynamic port number is used.

Connections to multiple instruments

Multiple instruments can be opened and used remotely within a process.

The following python code shows how to instantiate 2 remote instruments through KDI, that you can later use in a sequence.

For ports, Keysight recommends you use dynamic ports. You do this by setting port number 0 in the URL portion of the options and leave the operating system to find out an available port. It is also possible to manually set the TCP ports but this is not recommended because you have to set a unique port manually for each instrument, and this process can be error prone.

```
leader_module = KtM5300x("kdi://localhost/PXI0::CHASSIS1::SLOT2::INDEX0::INSTR", False, False,
'Simulate=1, DriverSetup=, HviServer=HVITCP:[::1]:0')
follower_module = KtM5300x("kdi://localhost/PXI0::CHASSIS1::SLOT3::INDEX0::INSTR", False, False,
'Simulate=1, DriverSetup=, HviServer=HVITCP:[::1]:0')
#
# The engines are added to the system definition the same way:
my_leader_module_main_engine = my_system.engines.add(leader_module.hvi.engines.main_engine,
"LeaderModuleEngine")
my_follower_module_main_engine = my_system.engines.add(follower_module.hvi.engines.main_engine,
"FollowerModuleEngine")
```

The following python code shows how to instantiate 2 remote instruments by manually setting the TCP ports.

In this case since each instrument is run in its own process, a different HVIServer port must be defined for each instrument, these are specified with `HviServer=HVITCP:[::1]:7890` and `HviServer=HVITCP:[::1]:7891`. Other ports can be used as long they're not being used by another process. Manually managing ports is error prone as it is up to the user to assign an unique TCP port for every instrument.

```
leader_module = KtM5300x("kdi://localhost/PXI0::CHASSIS1::SLOT2::INDEX0::INSTR", False, False,
'Simulate=1, DriverSetup=, HviServer=HVITCP:[::1]:7890')
follower_module = KtM5300x("kdi://localhost/PXI0::CHASSIS1::SLOT3::INDEX0::INSTR", False, False,
'Simulate=1, DriverSetup=, HviServer=HVITCP:[::1]:7891')
#
# The engines are added to the system definition the same way:
my_leader_module_main_engine = my_system.engines.add(leader_module.hvi.engines.main_engine,
"LeaderModuleEngine")
my_follower_module_main_engine = my_system.engines.add(follower_module.hvi.engines.main_engine,
"FollowerModuleEngine")
```

Multiple connections to a single instrument

You can make multiple connections to a single instrument if all connections have declared with `AllowMultipleClientAttach=1`.

Process 1

The following code for process 1 shows how to instantiate the remote instrument through KDI, the remote instrument uses `HviServer=HVITCP:[::1]:0` :

```
leader_module = KtM5300x("kdi://localhost/PXI0::CHASSIS1::SLOT2::INDEX0::INSTR", False, False,
'Simulate=1, DriverSetup=, AllowMultipleClientAttach=1,HviServer=HVITCP:[::1]:0')
#
unique_id = leader_module.hvi.engines.main_engine
#
print(unique_id) ### The following is printed in the console ->
#M5300xHviEngine@PXI0::CHASSIS1::SLOT2::INSTR<<HVITCP:[::1]:60248>>
#
# The engine is added to the system definition the same way.
my_remote_instrument_main_engine = my_system.engines.add(leader_module.hvi.engines.main_engine,
"RemoteInstrumentEngine")
```

Process 2

In this case, since the instrument previously created in a different process is being accessed, it is optional to specify the HviServer option because the instrument is already open.

Keysight recommends you always specify the HviServer option in case of a scenario where the order of launching the processes is unknown and the HVIServer might not be opened.

Process 2 assigned to the same port number because it takes the settings of the first process:

```
leader_module_in_a_different_process = KtM5300x
("kdi://localhost/PXI0::CHASSIS1::SLOT2::INDEX0::INSTR", False, False, 'Simulate=1,
DriverSetup=, AllowMultipleClientAttach=1')
#
unique_id = leader_module_in_a_different_process.hvi.engines.main_engine
#
```

```
print(unique_id)  ### The following is printed in the console ->
#M5300xHviEngine@PXI0::CHASSIS1::SLOT2::INSTR<<HVITCP:[::1]:60248>>
#
# The engine is added to the system definition the same way.
my_remote_instrument_main_engine = my_system.engines.add(leader_module_in_a_different_
process.hvi.engines.main_engine, "RemoteInstrumentEngine")
```

System Initialization

In HVI technology, System Initialization is a process that includes the configuration and alignment of the different systems clocks, including the clocking for each instrument specified as part of the HVI System Definition. This procedure includes the generation and alignment of the instrument internal *Sync* and *SyncBase* signals described in Chapter 9: HVI Time Management and Latency. Initialization also includes the alignment of clocks controlling the *Fast Data Sharing* (FDS) functionality for cases where your HVI definition includes instruments that support this functionality, such as instruments from the Keysight PXIe M5xxx instrument family.

A complete system initialization can be a complex procedure that can take some time. It must be performed when the system is first assembled and powered on, but you are not required to perform this level of initialization every time. However, a basic level of initialization is required every time to ensure that the system is synchronized and the clocks are in alignment.

A number of initialization options are provided to ensure correct setup and minimize initialization time when you want to run operations. You can use the basic options after the system has been fully initialized for the first time, however if you change the hardware setup (instruments, cables, etc.) or clocks (reference clock source, clock connection cables, etc.) you must perform a complete initialization again.

The SystemDefintion class includes an `initialize()` method that initializes the hardware included in the System Definition, and performs synchronization and clock alignment. There are 3 cases where system initialization and clock alignment can occur:

- Manually calling `initialize()` in the System Definition.
- When the Sequencer object is created.
- When calling Load to Hardware.

NOTE The initialization process requires access and control of all of the hardware resources, so it is important that these resources are not already in use by another application or HVI instance already loaded to hardware. An exception is thrown if any of the hardware resources are already in use.

The SystemDefinition initialize()

The `systemDefinition.initialize(...)` method enables you to explicitly trigger a system initialization and alignment. You might want to explicitly control when the initialization process is executed because the initialization process can take some considerable time depending on the parameters and system state.

You can also specific specific alignment modes when calling the initialize() method:

Mode	Description
Default	Calling initialize() without parameters performs the default, or minimal-possible initialization. **This is the mode intended to be used in normal system operation.**
`keysight_hvi.AlignmentModes.Full`	Forces a full system, complete system initialization and alignment.
`keysight_hvi.AlignmentModes.ResetCalibration`	Performs system initialization and alignment resetting and regenerating the stored calibration data.
`keysight_hvi.AlignmentModes.PreCalibration`	Performs system initialization and alignment, ignoring any missing calibration data. This mode is intended for system warm-up or other instances when the use of precise alignment calibration data is not available yet or not required.

You can combine modes using a bitwise-OR operator, for example:

```
systemDefinition.initialize(AlignmentModes::Full | AlignmentModes::PreCalibration);
```

The API call above will combine the *Full* and *PreCalibration* alignment modes as follows:

The software forces a complete system initialization and clock alignment (*Full* mode). While doing this, if the software detects that any of the instruments included in the HVI SystemDefinition object requires calibration data and this data is missing, it shall allow the system initialization procedure to continue and finalize, instead of throwing an error (*PreCalibration* mode).

```
systemDefinition.initialize(AlignmentModes::PreCalibration);
```

The API call above will perform a default system initialization and clock alignment. While doing this, if the software detects that any of the instruments included in the HVI SystemDefinition object requires calibration data and this is missing, it will allow to the system initialization procedure to continue and finalize, instead of throwing an error.

NOTE When using the default mode, in order to minimize the initialization time, PathWave Test Sync Executive relies on storing the initialization state of each instrument to decide what initialization steps are required. If hardware or cabling changes have been made in your system, you must make sure the correct initialization modes, *Full / PreCalibration / ResetCalibration*, is executed as required by your setup. For information about the initialization requirements for cabling or hardware changes, see your instrument documentation.

Default Initialization

This is the mode intended to be used in normal system operation. When calling `systemDefinition.initialize()`without parameters, it performs the default, or minimal-possible initialization. The default initialization tries to minimize the necessary operations to obtain the fastest initialization and synchronization time. The default initialization is automatically executed when a sequencer object is created from a System Definition, and also when the `LoadToHw()` method is called on the HVI instance.

If you have a system that has been power-cycled, the first call to `systemDefinition.initialize()` with no arguments will actually execute a full initialization and complete clock alignment because the system has not been aligned yet. The full initialization procedure can take several minutes depending on on the size and structure of your system, including the number of chassis and instruments, see the *Full Mode* description. Subsequent calls to `systemDefinition .initialize()` after the first call, are very fast.

Some instruments require stored calibration data to initialize correctly, in these cases this calibration data must be available for all instruments in the System Definition for the default initialization to work. If the calibration data is not available, an error is generated. See *ResetCalibration* and *PreCalibration* modes to understand the details on how to manage calibration data. For example, the Keysight PXIe M3xxx family does not require any calibration data, whereas in the Keysight PXIe M5xxx family, the M5300 RF AWG and M5201 Frequency Converter do require calibration data. For information about the calibration requirements, see your instrument documentation.

Full Mode

You can force a full clock alignment by calling: `systemDefinition .initialize(keysight_hvi.AlignmentModes.Full)`. The full initialization procedure can take several minutes depending on on the size and structure of your system, including the number of chassis and instruments.

NOTE This mode in general is only needed when some cabling change is done on the system without shutting it down or in cases when we want to force from software a complete reset of the initialization or alignment to recover from an undesired state.

ResetCalibration Mode

You can force the update of alignment calibration data by calling: `systemDefinition .initialize (keysight_hvi.AlignmentModes.ResetCalibration)`

This mode erases any existing system calibration data and forces the instruments to re-calculate and store new calibration data. Use this mode if there is no system calibration data available for the current hardware configuration and operating temperature, or if the existing calibration requires recalculation.

This operation must be performed when one or more of the following occur:

- A system setup containing any instrument that requires calibration data is used for the 1st time.
- The hardware in the system has been changed. This includes adding any instrument that requires calibration data, changing the cable connections between System Sync Modules, or changing any of the cables connections (clock, in/out, etc.) of any instrument requiring calibration data.
- The clock configuration has been changed. This includes changing the reference clock source, or any of the cable connections from it to any of the instruments that require calibration data.

NOTE Ensure you perform a full initialization with ResetCalibration when your system is fully warmed up. For information about the warm-up and temperature stabilization requirements for best performance, see your instrument documentation.

PreCalibration Mode

Calling `systemDefinition.initialize(keysight_hvi.AlignmentModes.PreCalibration)`configures the system without the need of having calibration data already stored for the specific temperature condition. Typical use of this mode is for system warm-up to prepare the system for a default initialization or a *ResetCalibration* initialization. Other instances include when calibration data is not available but precise calibration is not required.

NOTE When using instruments that require precise calibration data, always us this mode to warm-up the system before use. For information about the warm-up and temperature stabilization requirements for best performance, see your instrument documentation.

Initialization during Sequencer Creation

Behind the code used to create a Sequencer object, a system initialization is implicitly executed only if the System Definition contains instruments that include FDS functionality. If the instruments do not include FDS, these operations are skipped. For example, the `initialize()` of a System Definition that only includes M3xxx instruments is not performed at the Sequencer creation because these instruments do not support FDS. If an M5xxx instrument is included in a System Definition, system initialization and clock alignment is performed when creating the Sequencer, unless the initialize() had already been explicitly called.

At the Sequencer creation, if the system initialization is performed, it corresponds to a call of `sys_def.initialize()` without parameters. This only performs a minimal update to the initialization and clock alignment.

Initialization during Load To Hardware

The Load To Hardware operation also contains an implicit default system initialization. In Load to Hardware such system initialization is always performed and it is equivalent to the explicit call `sys_def.initialize()`executed without parameters. This only performs a minimal update to the initialization and clock alignment.

Example of System Initialization

To use the HVI API to initialize and run real-time operation in your system, there are two main procedures that you must follow:

1. System Warm-up and Calibration
2. Normal Operation

There are also a number of use cases that are variations on these main procedures. The following text describes these procedures along with the use case variations.

NOTE The initialization process requires access and control of all of the hardware resources, so it is important that these resources are not already in use by another application or HVI instance already loaded to hardware. An exception is thrown if any of the hardware resources are already in use.

System Warm-up and Calibration

The system warm-up must be performed every time the system is turned on or the hardware configuration is changed. This is to enable all of the components to reach a stable and repeatable operating temperature. Once the system is warmed-up, the system can be initialized using the stored System Calibration data.

The System Calibration must performed in these cases:

1. The very first time that the system is put together and powered-on.
2. When relevant hardware changes are made that require a new system calibration. These hardware changes include:
   a. Adding/removing a chassis in your SystemDefinition object.
   b. Adding/removing any instrument that requires clock alignment calibration data, such as an M5300 or M5201, or changes the operating temperature of the system.
   c. Changing the cable connections between System Synchronization Modules, even replacing a cable with a similar one with a different serial number.
   d. Changing any of the external System Clock or Analog Clock cable connections, even replacing a cable with a similar one with a different serial number.
   e. Making any change to the clock configuration, even if it is only from the HVI API. This is because this triggers the usage of different clock sources or signal paths.
3. Other situations where the system calibration should be updated.
4. On rare occasions, a component in the system can move into an invalid state and a reset of the calibration might be required. For more information, see *System Troubleshooting* in the ***System Setup Guide*** .

NOTE Warning : Resetting the system calibration shall in turn require you to recalculate the User Calibration for some instruments. Observe extreme caution when doing this to avoid costly time-consuming recalibration.

Procedure steps:

Procedure steps:

1. Power-on the system
   a. Power-on all of the chassis. After this is complete, if you are using an external chassis controller, power it on.

2. Connect to all the instruments
   a. For example: `instrument = ktm5300.KtM5300x(resource_id, query, reset, options)`

3. Create a System Definition using the HVI API and the instrument drivers:
   a. Create a SystemDefinition object that we refer to here as *my_system*. Use the *my_system* object to define all the hardware resources in your system: chassis, SSMs, instruments, clocking configuration, reference clock source, etc.
      For example: `my_system.chassis.add(1)`, `my_system.clocking.reference_source = chassis.clock_source`
   b. Add the HVI Engines of each instrument to the SystemDefinition object.
      For example: `my_system.engines.add(instrument.hvi.engines.main_engine, "MyEngine")`

4. System Initialization for Warm-Up
   a. Execute `my_system .initialize(keysight_hvi.AlignmentModes.FULL | keysight_hvi.AlignmentModes.PRE_CALIBRATION)`. The `PRE_CALIBRATION` flag indicates there is no need to apply any previously stored system calibration values because the system is warming-up. This enables the system to execute code without calibration related errors. After this step, instruments may present channel skew errors which are compensated by the next steps.

5. Wait for System Warm-Up
   a. Wait for the required warm-up time, this can range from a few minutes to about 30 minutes. The actual time typically depends on the type and number of instruments in the system, clocking configuration, etc.
   b. For detailed warm-up time information, see your instrument documentation, for example: *M5300 RF AWG User's Manual* .

6. System initialization to perform System Calibration
   a. Using the SystemDefintion created in step 3, run `my_system.initialize(keysight_hvi.AlignmentModes.FULL | keysight_hvi.AlignmentModes.RESET_CALIBRATION)` to generate internal system calibration data. At first system turn-on, no previous calibration data is expected to be available.

7. Calculate User Calibration or channel deskew (Optional)
   a. This operation is optional and consists of correcting analog channel skews introduced by cable and signal path delays. Note that in some instruments, the User Calibration must be re-calculated when a System Calibration is executed. For information about how to do this, see your instrument documentation.

8. Ready for Normal Operation

Use Cases:

Use Case Scenario	Description
First system start-up and calibration	The very first time that the system is put together and powered-on, you must execute a full warm-up and calibration procedure to achieve the best system performance and repeatability: • Execute all steps #1 to #7 above.
System start-up using existing calibration	If the system has already been calibrated for the current hardware configuration, then, to reuse the existing calibration to configure the system, wait for the system temperature to stabilize then apply the existing calibration: • Execute steps #1 to #5 above. • Skip steps #6 and #7 *System initialization to perform System Calibration* and *Calculate user calibration or channel deskew* , and run `my_system.initialize (keysight_hvi.AlignmentModes.FULL)`.
Simplified uncalibrated system start-up	If you want to use the system for test development, or you can tolerate analog channel drift of up to 50ps across reboots/power-cycles: • Execute steps #1 to #4 above. • Skip steps #5 to #7 *Wait for System Warm-Up* , *System initialization to perform System Calibration* and *Calculate user calibration or channel deskew* .

NOTE **System hot boot-up:** If the system is already warmed-up to the calibration operating conditions, for example after a system restart, you can skip the steps #4 and #5 *System Initialization for Warm Up* and *Wait for System Warm-Up* .

Normal Operation

Once the system is warmed-up and the system calibration has been done, users can use the the HVI API to execute real-time operations:

NOTE Note that if it is the first system start-up or you have introduced any of the HW changes that require new System/User Calibration you must execute the *First system start-up and calibration* use case described in the *System Warm-up and Calibration* procedure.

Procedure steps:

1. Connect to all the instruments, if not already connected.
   a. For example: `instrument = ktm5300.KtM5300x(resource_id, query, reset, options)`

2. Apply user calibration to instruments, You only need to do this if it is required, the user calibration data is available, and it has not been applied already.
   a. The user calibration is calculated during the *System Warm-up and Calibration* process. For information about how to apply existing calibration, see your instrument documentation, for example: *M5300 RF AWG User's Manual* .

3. Create a SystemDefinition object, or reuse an existing one.
4. Initialize the SystemDefinition object (Optional)
   a. Run `my_system.initialize()`. This call executes the minimal or default initialization, provided a Full Initialization has been executed already as described in the *System Warm-up and Calibration* procedure. If the full initialization has not been executed, this step requires calibration data. If the calibration data is not available this operation will fail. To run the system initialization without calibration you can specify the `PRE_CALIBRATION` flag: `my_system.initialize(keysight_hvi.AlignmentModes.PRE_CALIBRATION)`
   b. Note that you can skip the call to `my_system.initialize()` because the minimal or default initialization happens implicitly in steps #5 and #7 described below.

5. Create a Sequencer object
   a. For example: `sequencer = keysight_hviy.Sequencer("MySequencer", my_system)`
   b. Note that the sequencer creation operation implicitly executes a default initialization, this is equivalent to calling `SystemDefinition:Initialize()`.

6. Create an HVI object
   a. For example: `hvi = sequencer.compile()`
   b. The HVI object is created by compiling the Sequencer object after all the HVI sequences have been programmed.

7. Load HVI to HW
   a. For example: `hvi.load_to_hw()`
   b. Note that the load_to_hw() operation implicitly executes a default initialization, this is equivalent to calling `SystemDefinition:Initialize()`.

8. Run HVI
   a. For example: `hvi.run(hvi.no_timeout)`

9. Release HW
   a. For example: `hvi.release_hw()`

NOTE **Forcing a full initialization.** You can optionally force a full initialization. Forcing the full initialization can be useful to unblock a system if it is in a bad state, when some temporary hardware changes in the system are done such as reconnecting cabling using the same cables, or in general when it is useful to ensure the system is fully initialized to discard any previous state. To force the full initialization run:

1. `my_system.initialize(keysight_hvi.AlignmentModes.FULL)`.
2. Or if you are using the system without calibration, add the `PRE_CALIBRATION` flag: `my_system.initialize(keysight_hvi.AlignmentModes.FULL | keysight_hvi.AlignmentModes.PRE_CALIBRATION)`

NOTE **User Calibration not required or already applied:** If user calibration is not required or has already been applied to the instruments, you can skip step #2 *Apply user calibration to instruments.* For more information on how to handle User Calibration in instruments, see your instrument documentation.

Sequencer

This section describes the Sequencer class, it contains the following sections:

About the Sequencer Class

You use the Sequencer class to program and compile your HVI sequences.

The Sequencer object defines a top level Global Sync Sequence. Within this:

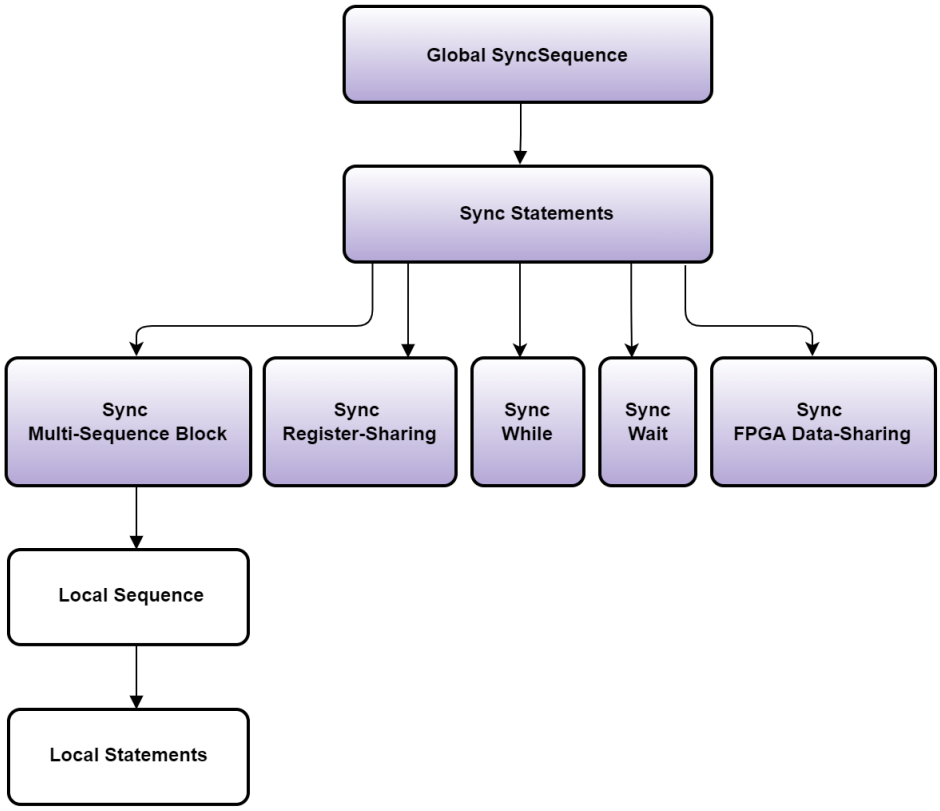
- You add Sync Statements to Sync Sequences with the `SyncSequence` class.
- You can add Sync Sequences within the Global Sync Sequence.
- Within the `SyncMultiSequenceBlockStatement` you add Local Sequences for individual engines using the `Sequence` class.
- You add Local Statements to Local Sequences with the `Sequence` class.

The sequences and statements you add can access the resources you previously added via the `EngineCollectionView` class. The view classes enable you to see the definitions you have set up, but you cannot modify them.

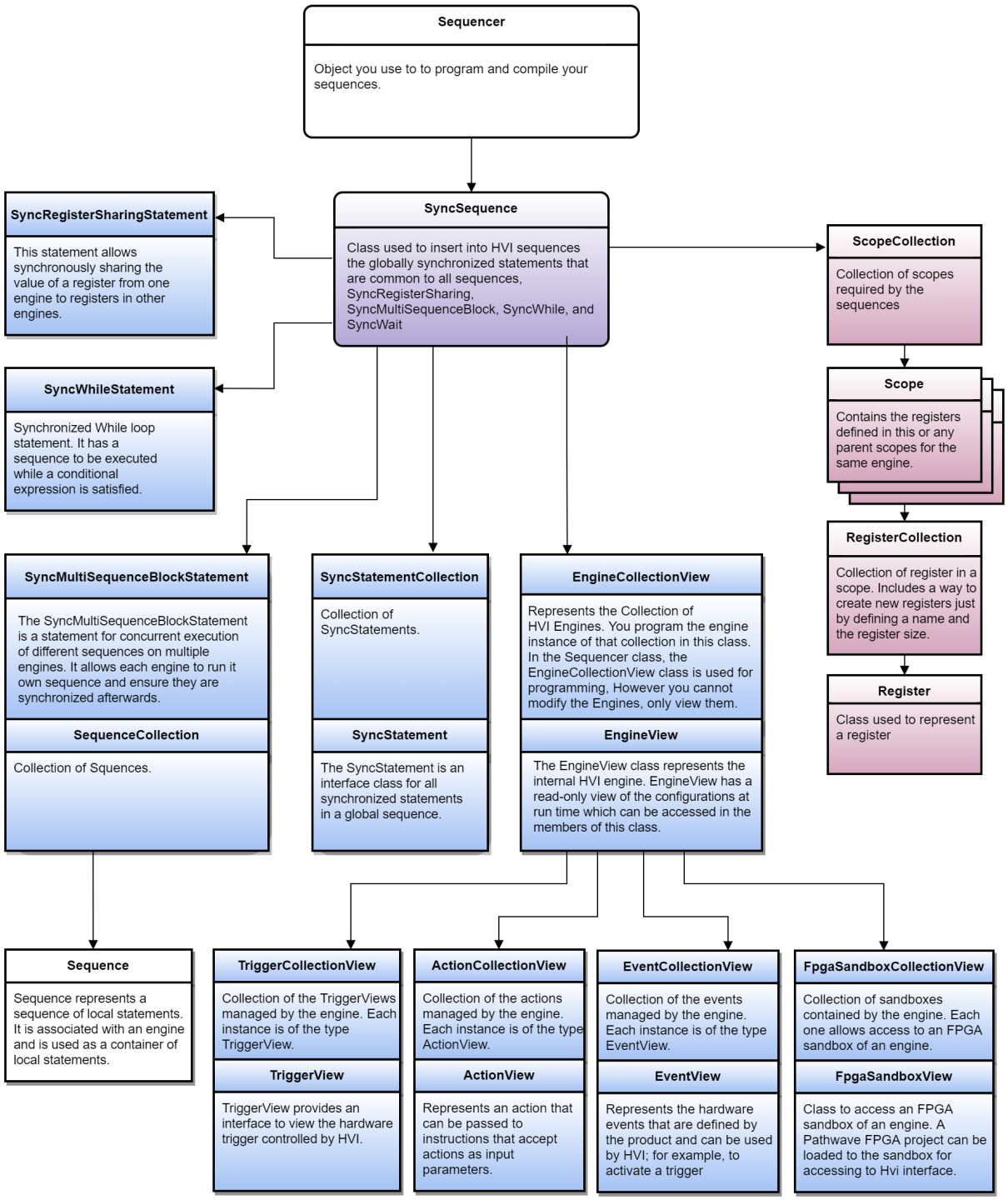
NOTE WARNING: Once a Sequencer instance is created, any change to the System Definition will not affect the Sequencer object or any objects or definitions inside the Sequencer (engines, triggers, actions, events,...). All API calls on the Sequencer object *must* use the objects in the Sequencer. Do not use objects from the System Definition or other Sequencer instances under any circumstances.

Once you have defined all the Sequences that define your HVI, you must compile it. The HVI instance `Hvi`, is generated when you compile the sequencer object.

The following diagram shows the hierarchy of sequences and statements:



The following diagram shows the Sequencer classes:



HVI SyncSequence and Sequence

There are two types of HVI sequence classes that enable HVI sequence programming and usage:

- `SyncSequence`.
- `Sequence`.

HVI uses the `SyncStatement` class to manage all of the engine sequences simultaneously. The class exposes the add statement methods such as `SyncSequence.add_sync_while()`. All of the statements added are collected in the `SyncStatement` class.

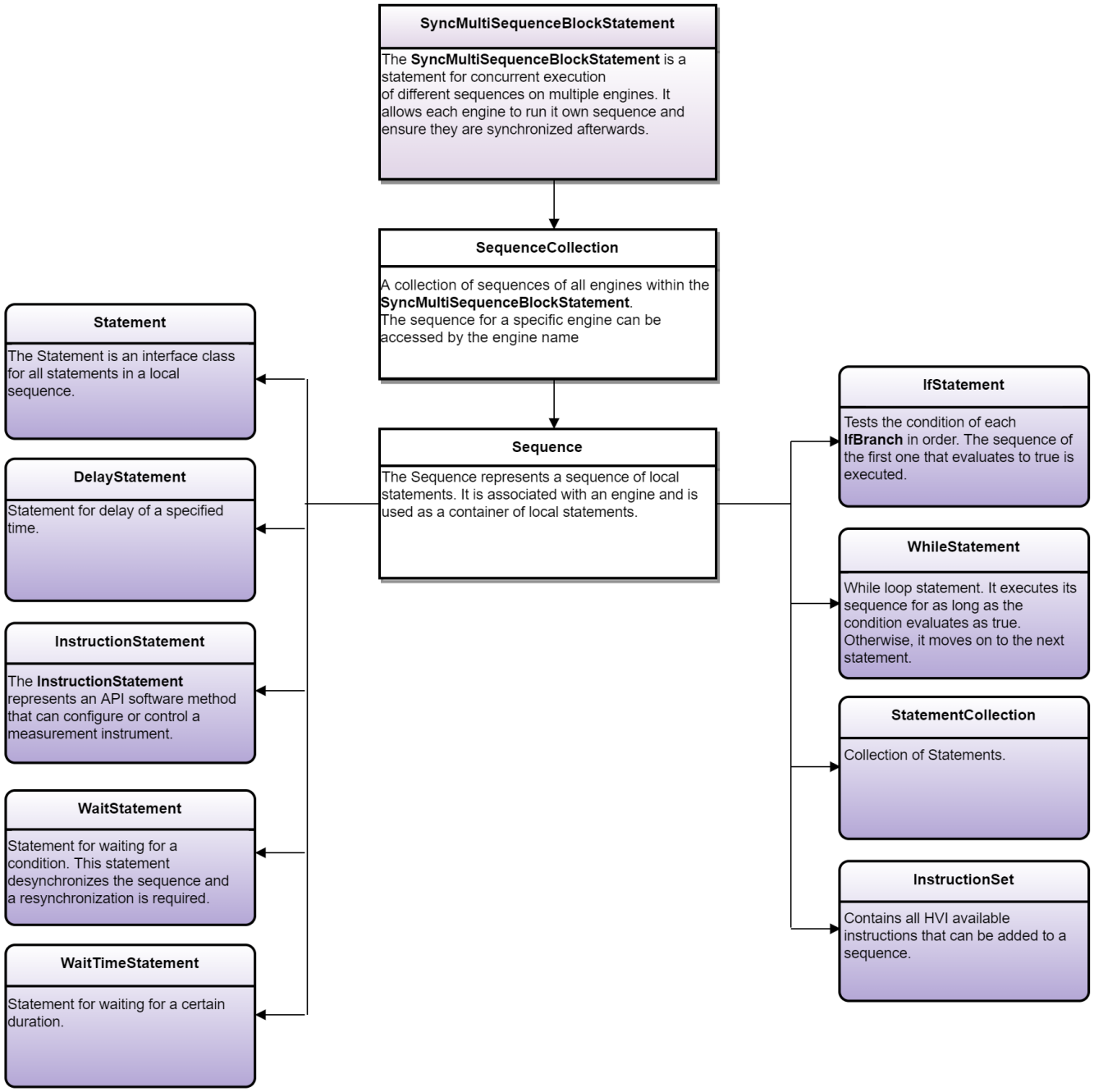
Synchronization and timing information are added within each Sync statement so that all sequences across the HVI are coordinated precisely. The `SyncMultiSequenceBlockStatement` exposes local flow control and instruction statements that are sent by the `Sequence` object. The other Sync statements are all synchronized across all the sequences in the HVI.

An HVI sequence contains the list of HVI Local statements and instructions to be executed by the HVI engine.

The `Sequence` class exposes the add statement methods such as `add_while()`. You add Local flow control statements such as If or While directly into the sequence. All Local instructions are added using `add_instruction()`. The list of available statements for the `add_instruction()` statement is shown in HVI API Local Statements.

The sequence stores a collection of all the statements added to it, along with the scope Variables and registers needed for this sequence. These are sent to a `SyncMultiSequenceBlockStatement`. This class exposes access and execution of Local statements.

The following diagram shows the `SyncMultiSequenceBlockStatement` class:



HVI API Statements

HVI API statements are divided into two types:

Sync statements

Sync statements are the building blocks used to program Sync sequences. The following types of Sync statement are available:

- Sync while.
- Sync multi-sequence block.
- Sync register-sharing.
- Sync FPGA data-sharing.

For a description of each Sync statement with examples and a description of the statement execution, see HVI API Sync Statements.

Local statements

Local statements are programmed on engines in individual instruments. They are always programmed within a Sync statement.

Local statements are in the form of Instrument-specific HVI instructions or HVI-native instructions. See your instrument documentation for instrument-specific HVI instructions. The following types of HVI-native instructions are available:

- Action Execute: AWG trigger, DAQ trigger.
- FPGA register read.
- FPGA register write.
- FPGA memory map write.
- FPGA memory map read.
- Register increment.
- Front panel trigger ON/OFF.
- Register assign.
- Local if statement.
- Local while statement.
- Local wait-for-event statement.
- Local wait-for-time statement.
- Local delay statement.

For a description of each HVI-native instruction with examples and a description of the statement execution, see HVI API Local Statements.

For instrument-specific HVI instructions, see your instrument documentation

InstructionSet

HVI instructions can be one of two types, HVI-native instructions or instrument-specific instructions:

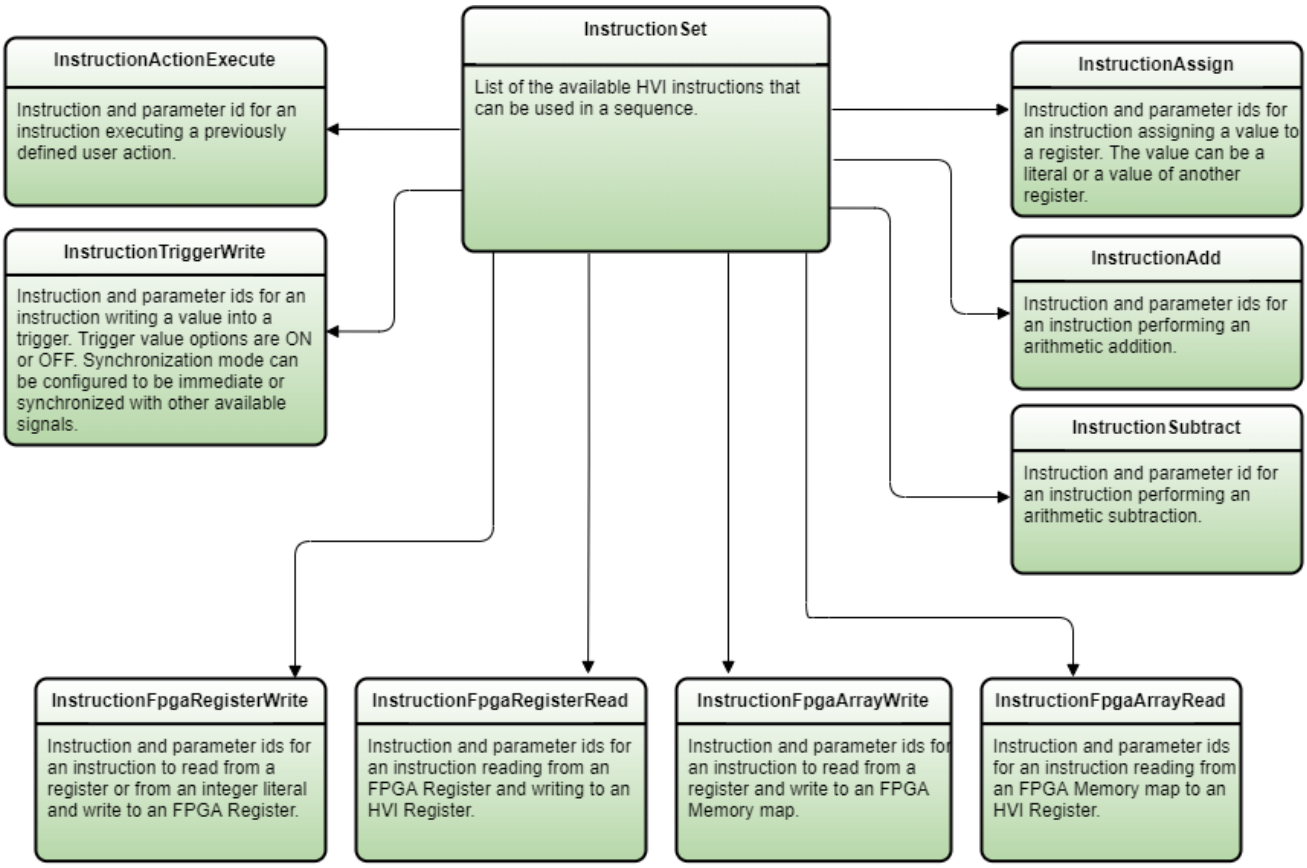
- HVI-native instructions are part of the `InstructionSet` class.
- Instrument-specific instructions are documented in instrument user guides.

The `InstructionSet` class contains the set of available HVI-native instructions that can be executed within an HVI statement. These include instructions for:

- Register arithmetic.
- Reading and writing I/O trigger ports.
- Executing actions.
- Communicating with the instrument sandbox using an HVI Host Interface, previously called an HVI Port.

HVI-native instructions are executed within an instruction execute statement, this is, the same way the instrument-specific HVI Instructions are executed.

The following diagram shows the `InstructionSet` classes:



Using the instruction set

You program HVI instructions into local sequences with the `add_instruction()` API method. You can set instruction parameters with the `set_parameter()` API method and set each parameter with its `parameter.id` property. Some instruction parameters must be set to literal values or to an HVI register, for example, the source and destination parameters in the `InstructionAssign` from the native `InstructionSet`.

You can set other instruction parameters such as the `SyncMode`and `TriggerValue` of the `TriggerWrite` instruction to one value of a pre-defined set of possible values. In this case, the possible values available are stored in properties contained within the parameter object.

```
# Pseudo-code explaining the HVI instruction programming concept
hvi_instr = sequence.instruction_set.hvi_instruction_X
instr = sequence.add_instruction("My HVI Instruction", 10, hvi_instr.id)
instr.set_parameter(hvi_instr.parameter_A.id, hvi_instr.parameter_A.VALUE_1)
instr.set_parameter(hvi_instr.parameter_B.id, hvi_instr.parameter_B.VALUE_XY)
```

Trigger write instruction example

The following example shows an example of the HVI-native instruction `trigger_write`. For the meaning of each parameter value, see the HVI API help that is installed with PathWave Test Sync Executive. It is located at:

`C:\Program Files\Keysight\PathWave Test Sync Executive 2022\api\python\Help`

`C:\Program Files\Keysight\PathWave Test Sync Executive 2022\api\dotNet\Help`

The following table show the parameters for the HVI-native instruction: `trigger_write`

Parameter ID	Parameter Values
trigger.id	`Trigger` object taken from the `TriggerCollection` class from the engine in the sequence.
sync_ mode.id	`sync_mode.immediate`
	`sync_mode.sync`
value.id	`value.on`
	`value.off`

The following example code shows a `trigger_write` instruction.:

```
# Write FP Trigger to ON value
fp_trigger = sequence.engine.triggers["FP Trigger"]
trigger_write_instr = sequence.instruction_set.trigger_write
instr_trigger_ON = sequence.add_instruction("FP Trigger ON", 10, trigger_write_instr.id)
instr_trigger_ON.set_parameter(trigger_write_instr.trigger.id, fp_trigger)
instr_trigger_ON.set_parameter(trigger_write_instr.sync_mode.id, trigger_write_instr.sync_
mode.IMMEDIATE)
instr_trigger_ON.set_parameter(trigger_write_instr.value.id, trigger_write_instr.value.ON)
```

NOTE

Warning: You must take the Trigger from the engine inside the Sequence. Taking the Trigger from the engine via System Definition raises an error when *set_parameter(trigger_write_instr.trigger.id, fp_trigger)* is called.

This also applies to Actions.

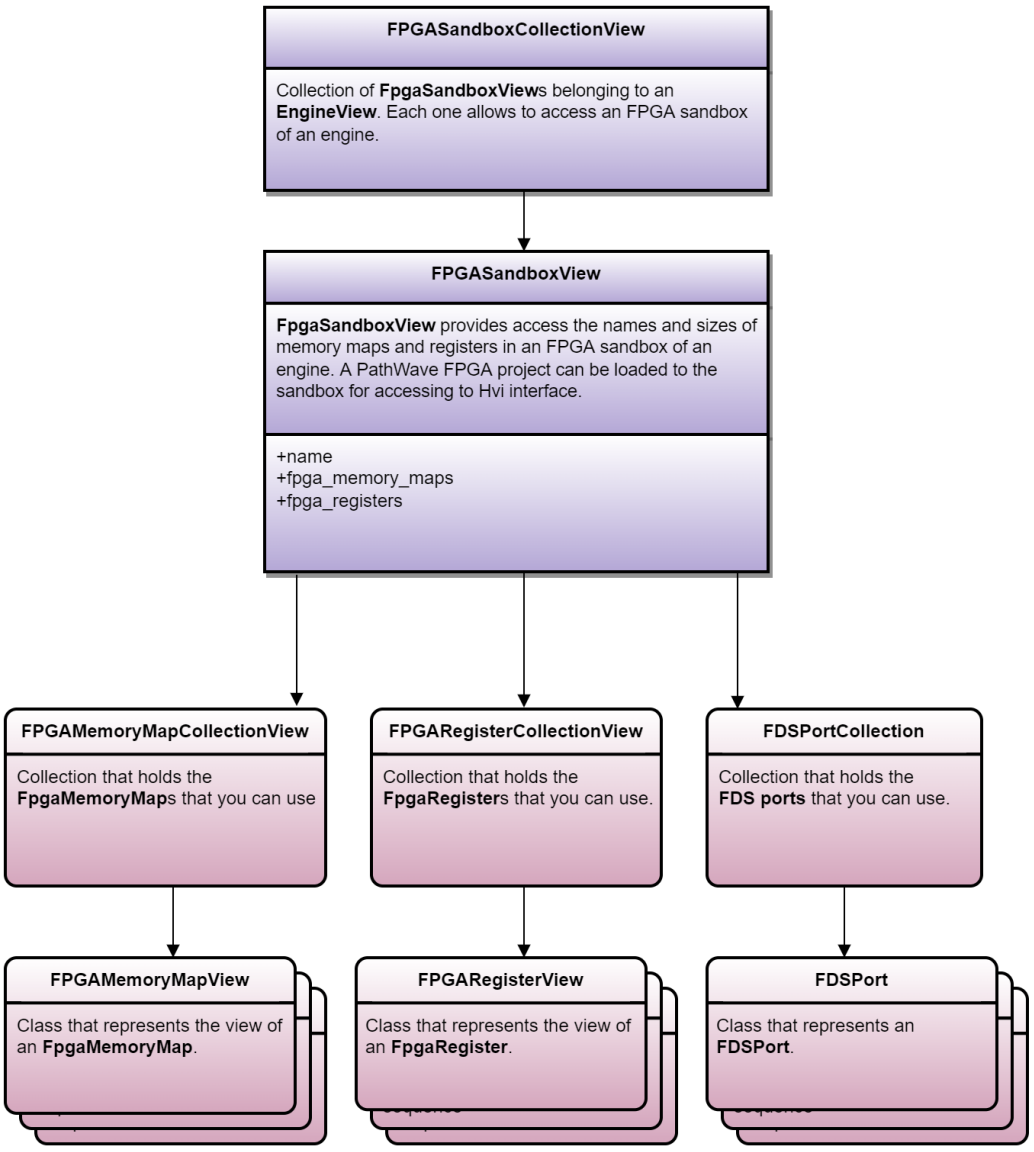
Instrument-specific HVI instructions

You program instrument-specific instructions into your HVI sequences using the same methods as HVI-native instructions, that is, you add Instrument-specific instructions to local sequences with the `add_instruction()` API method. Parameters of instrument-specific instructions are also set with the `set_parameter()` API method. For documentation on instrument-specific instructions and their parameters, see your instrument documentation. For M3xxxA PXI instruments, the information is located in the *SD1 3.x Software for M320xA / M330xA Arbitrary Waveform Generators User's Guide* available at M3201A PXIe Arbitrary Waveform Generator.

FPGA Sandbox View

This section describes the FPGA Sandbox View.

The following diagram shows the `FPGASandboxCollectionView` classes:



FPGA sandbox and memory maps

The `FpgaSandboxView` object provides access to FPGA memory maps by providing handles to FPGA registers and memory maps that are defined in the FPGA memory. You can use `FpgaRegisterView` and `FpgaMemoryMapView` as parameters for HVI instructions for reading or writing FPGA memory. You must load the PathWaveFPGA project as part of the system definition and then you can use the `FpgaSandboxView` object in the sequencer.

FpgaRegisterView

Once the sandbox project is loaded, you can access the contents of the FPGA sandbox and use them as parameters for HVI instructions. The FPGA write operation can accept registers and literal values as parameters. The following example shows writing FPGA registers:

```
# Retrieve FPGA register object from FPGA registers collection
# All sandbox object collections are populated when loading a bit file generated by PathWave
FPGA
fpga_register_view = sequencer.sync_sequence.engines[0].fpga_sandboxes[SANDBOX_0_NAME].fpga_
registers[FGPA_REGISTER_NAME]
# Write FPGA register
fpga_regw_instruction = sequence.instruction_set.fpga_register_write
fpga_register_write = sequence.add_instruction("my_fpga_register_write", 10, fpga_regw_
instruction.id)
fpga_register_write.set_parameter(fpga_regw_instruction.fpga_register.id, fpga_register_view )
fpga_register_write.set_parameter(fpga_regw_instruction.value.id, value_register)
```

FpgaMemoryMapView

Like FPGA registers, the FpgaMemoryMapView can be used after the PathWaveFPGA project has been loaded. The destination of FPGA read operation must be a register. The following example shows how you use it to read from an FPGA memory map:

```
# Retrieve memory map object from memory maps collection
# All sandbox object collections are populated when loading a bit file generated by PathWave
FPGA
memory_map = sequencer.sync_sequence.engines[0].fpga_sandboxes[SANDBOX_0_NAME].fpga_memory_maps
[FGPA_MEMORY_MAP_NAME]
# Read Memory Map
fpga_arrayr_instr = sequence.instruction_set.fpga_array_read
fpga_array_read = sequence.add_instruction("my_fpga_array_read", time_ns, fpga_arrayr_instr.id)
fpga_array_read.set_parameter(fpga_arrayr_instr.fpga_memory_map.id, memory_map)
fpga_array_read.set_parameter(fpga_arrayr_instr.fpga_memory_map_offset.id, 1)
fpga_array_read.set_parameter(fpga_arrayr_instr.value.id, value_register))
```

FdsPort

FdsPort Enables you to access the FDS ports you defined in the System Definition. FdsPort is a class which provides the name of a sandbox FDS port, it enables you to use an FDS port instance that you placed in the sandbox of a loaded PathWave FPGA project.

As with the other FPGA software definitions, FdsPort can only be used after the PathWaveFPGA project has been loaded.

HVI Registers and Scopes

HVI registers

HVI registers are similar to Variables in a programming language. They hold values that can be modified at runtime and can be used as parameters for instructions and statements. Physically, HVI registers are small hardware memories located in HVI engines. The number of registers available depends on the instrument (see the HVI engine settings HviRegCount).

Registers are specific to individual HVI engines and cannot be accessed by other HVI engines. To transfer data between registers you must use register sharing instructions.

You define HVI registers by adding them to the HVI register collection that is bound to an HVI scope.

HVI scope

HVI Sync sequences and HVI Local sequences both include the concept of *the scope of registers*, this is similar to the concept of *the scope of Variables* in programming languages. The scope defines what registers or memory resources can be used within each HVI Sequence, and when they can be used.

Each scope is associated with a specific sequence and HVI engine. Registers can only be defined within the Global Sync sequence scope, but they can be retrieved from any child sequence scope providing it is on the same engine. Registers are always defined with a clear connection to a specific engine and their visibility only propagates to child sequences that execute on the same engine. HVI engines do not have visibility of, and cannot access registers that are in the scopes of other engines.

NOTE Registers can only be added to the HVI top Sync sequence scopes. This means that you can only add global registers that are visible in all child sequences.

NOTE Registers are created using the sequencer class, but to read/write registers during HVI execution, you must use the `RegisterRunTime` class within the `Hvi` class. For more information, see The Hvi Object.

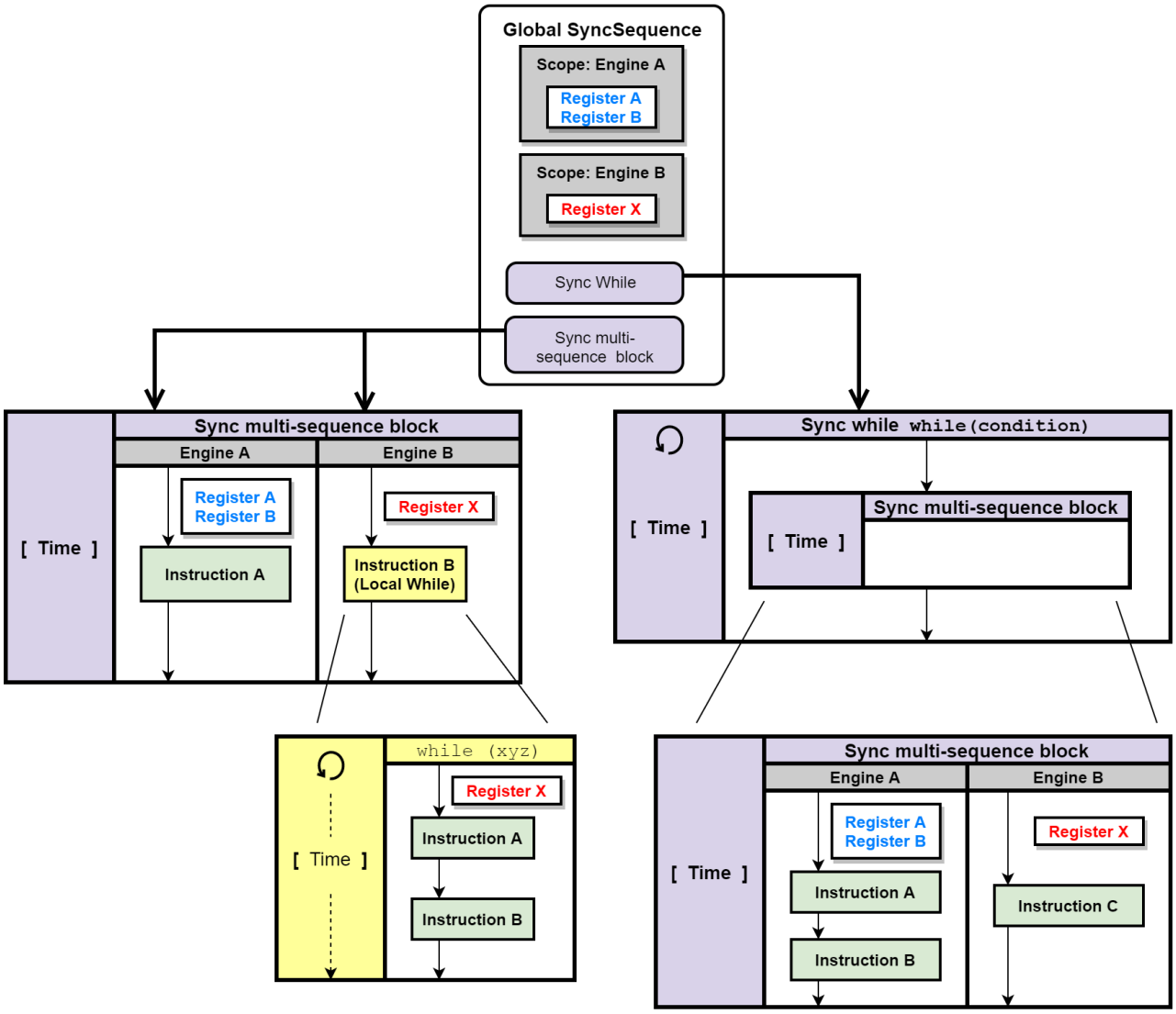
The following diagram shows the scope concepts. The registers available are shown in the sequences and child sequences.

In the Global Sync sequence a scope is defined for each of Engine A and Engine B.

- Engine A scope contains Register A and Register B.
- Engine B scope contains Register X.

The Global Sync sequence contains Sync statements including a Sync while and a Sync multi-sequence block. These are expanded as HVI diagrams. The Sync multi-sequence block contains sequences for both engines. These are shown with the registers available in blue for Engine A, red for Engine B.. The sequence for engine B contains a Local while. This is expanded below with the available Register X shown in red.

The Sync while in the Global Sync sequence is also shown, it contains another Sync multi-sequence block which is shown expanded.



Scope class and ScopeCollection

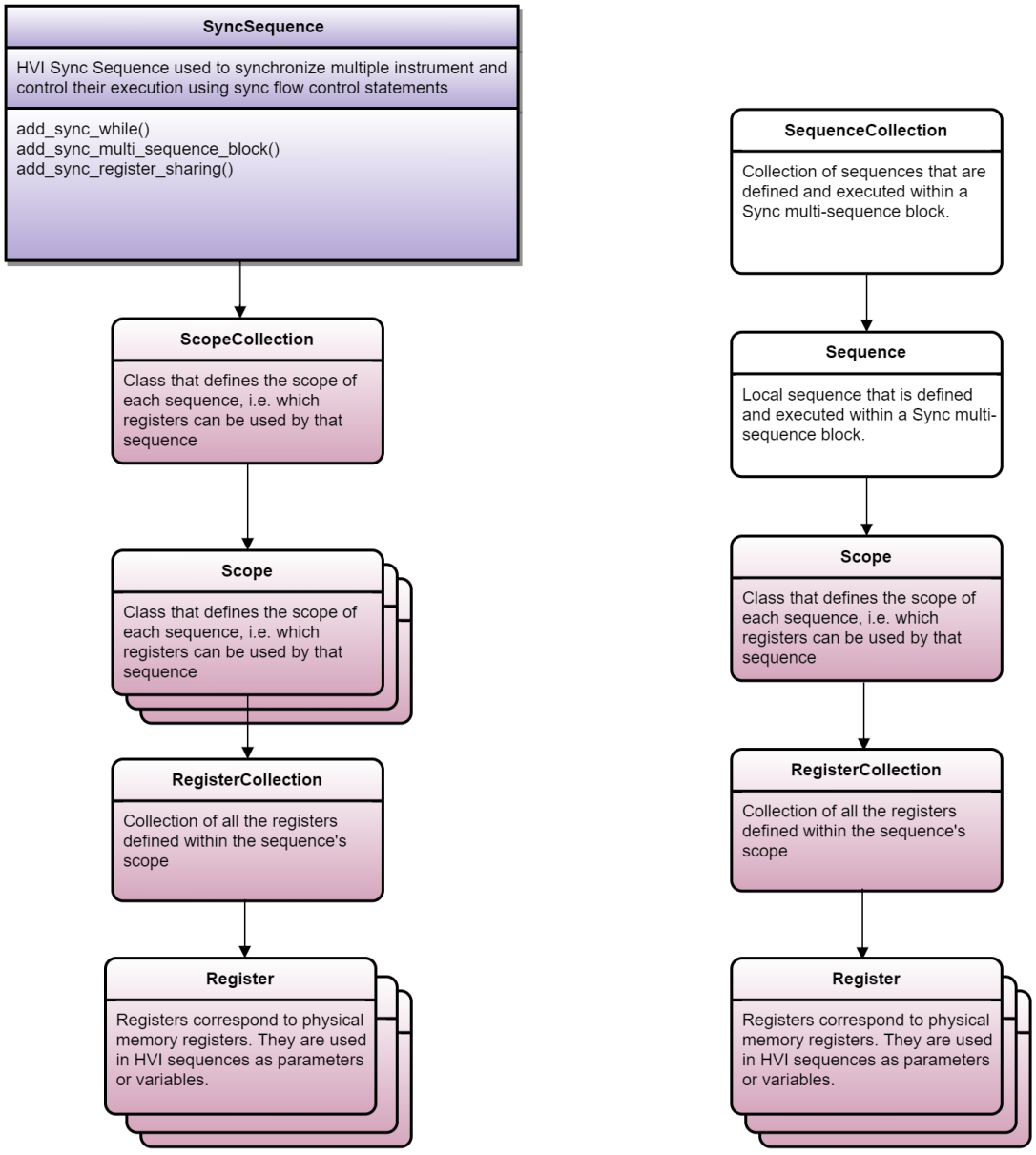
The scopes of HVI sequences are managed through the Scope class. Each Local sequence is an instance of the Sequence class, it is associated to a specific HVI engine and has its own Scope object. SyncSequences are associated to multiple HVI engines and consequently have an HVI Scope collection that contains a Scope object for each associated HVI Engine.

The HVI Scope collection is an instance of the ScopeCollection class, it contains objects that are instances of the Scope class. There is one Scope object for each HVI Engine.

Each HVI Scope object can be accessed from the Scope collection using the same name as the corresponding HVI engine. HVI Scope objects are used to define the registers within a sequence.

To use registers in HVI sequences, you must define them beforehand in the register collection within the scope of the corresponding HVI sequence. You can do this using the RegisterCollection class that is within the Scope object corresponding to each sequence.

The following diagram shows the Scope classes and their relationship to the Sequence and SyncSequence classes:



HVI Time API

This section describes the API related to the Time inside HVI.

The main time class is the `Duration` which is located in the namespace `Time`. The Duration class represents a time interval.

You can create a Duration object in one of these ways:

- By only providing a single floating point value. In this case, the value is treated as time in nanoseconds.
- By providing a floating point value and the unit of time you want the value to represent.

The signature of the class is:

```
Duration(double valueInNanoseconds);
Duration(double value, Time::Unit unit);
```

This class is also the base for a subclass called the `Minimum`. The minimum represents the minimum time interval possible.

The signature for this class is:

```
Minimum();
```

The class to define the unit of the duration is called the `Unit`. The supported units are the following:

```
enum class Unit
{
       Seconds,
       Milliseconds,
       Microseconds,
       Nanoseconds,
       Picoseconds
};
```

The following is an example of usage:

```
from keysight_hvi import time
a_duration = time.Duration(35.0)
assert a_duration.type == time.Type.FIXED_DURATION
assert a_duration.value == 35.0
assert a_duration.unit == time.Unit.NANOSECONDS
another_duration = time.Duration(35.78, time.Unit.MICROSECONDS)
assert another_duration.type == time.Type.FIXED_DURATION
assert another_duration.value == 35.78
assert another_duration.unit == time.Unit.MICROSECONDS
a_minimum_duration = time.Minimum()
assert a_minimum_duration.type == time.Type.MINIMUM_DURATION
assert a_minimum_duration.value == 0.0
assert a_minimum_duration.unit == time.Unit.NANOSECONDS
```

HVI Compilation

Once you have programmed all of your HVI Sequences, the next step is to compile them. The compilation process returns the `Hvi` object that is used to run the created sequences on hardware.

Call the `compile()` method in the `Sequencer` object to perform the compilation operation. If successful, this method returns an `Hvi` object, if the compilation fails, it throws an exception.

The compilation process translates the programmed sequence into binary instructions to be loaded into the hardware. During this process, the compiler applies the compilation rules, evaluates the specified constraints, and determines if the number of resources required (PXI triggers, actions, events, HVI registers) is available in hardware and can be acquired. The compiler returns an error if any of the HVI statements was not programmed properly inside the HVI sequence or if any of the HVI resources are missing or not registered properly.

NOTE **At this point you can no longer modify sequences, actions, events or triggers.**

Information returned

The value returned from the compilation procedure is an `Hvi` object. This object can be used to:

- Load and execute the binary instructions by each engine.
- Retrieve the `CompileStatus` object.

Errors returned

If the compilation fails, the object `keysight_hvi.CompilationFailed` is thrown. This contains the `CompileStatus` object.

In the following Python snippet, the `CompileStatus` object is retrieved from the exception object thrown:

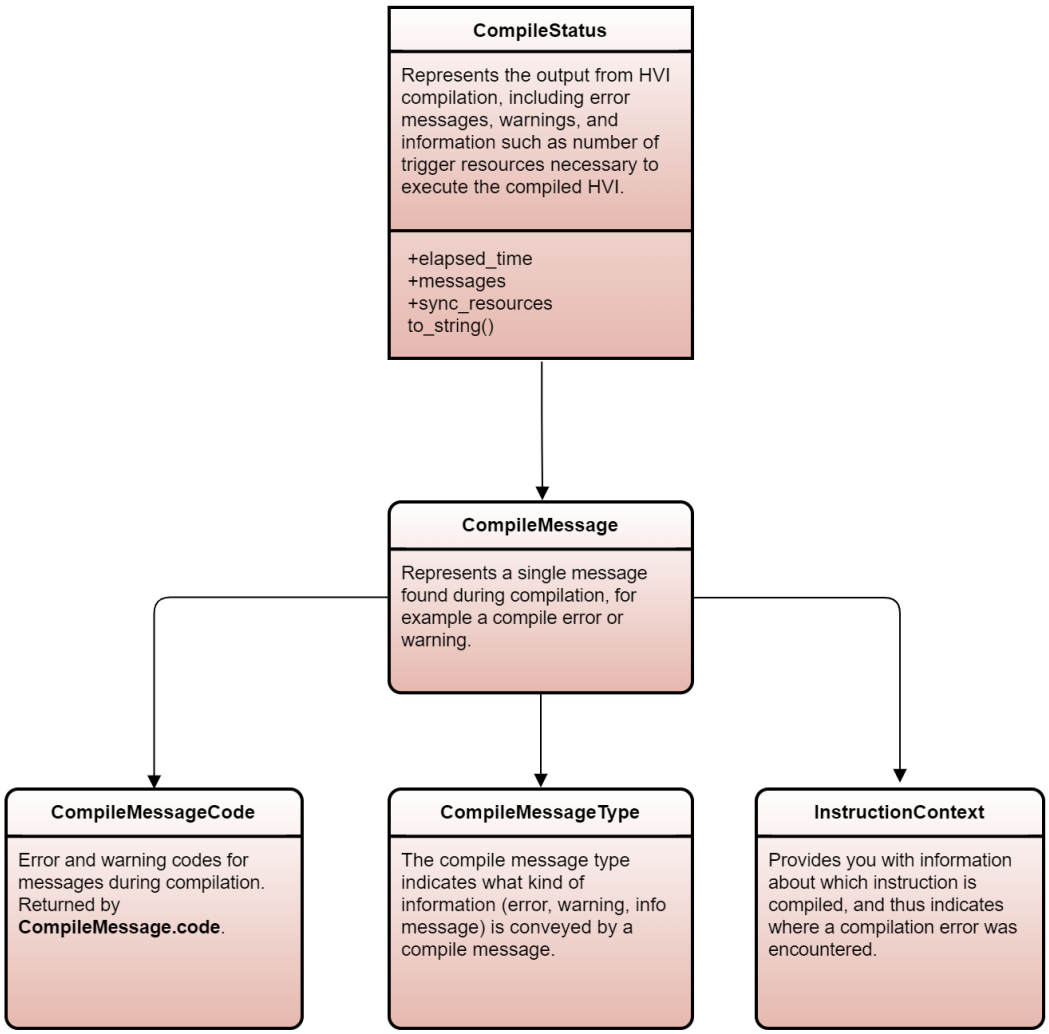
```
try:
   hvi = sequencer.compile()
   print('Compilation completed successfully!')
except kthvi.CompilationFailed as err:
   print('Compilation failed!')
   compile_status = err.compile_status
   print(compile_status.to_string()) # This line will print all the errors and warnings
collected during compilation raise err
```

Compile status

The `CompileStatus` object contains the following information:

- The warning and error messages generated by the compilation.
- Information about the PXI sync resources that must be reserved.
- The elapsed time of the compilation process.

The following diagram shows the `CompileStatus` classes and the information they contain:



Sequence Visualization

PathWave Test Sync Executive enables you to troubleshoot your sequences with sequence visualization.

The sequence visualization displays statements, timing values, and statement parameters. The output is designed so you can read it and see what your sequences are doing.

NOTE This is only available for Sync sequences in this release.

Using sequence visualization

To activate the output, In Python use the sequence method `to_string()`:

```
output = sequencer.sync_sequence.to_string(kthvi.OutputFormat.DEBUG)
print(output)
```

If you are programming with C#, use the method `ToString`:

```
var output = GlobalSequence.ToString(OutputFormat.Debug);
System.Console.WriteLine(output);
```

Format of the sequence visualization output

Sequence visualization has a basic structure with variations for different types of statements.

The visualization out format has one statement per line and uses curly braces to begin and end any inner or Local statements.

The basic format is:

```
Time-related information => "User-assigned Label" : Statement Name(Parameter List) {

Optional statements

}
```

For example:

```
+10ns => "FP Trigger ON": TriggerWrite(Trigger = [trigger"TriggerIO"], Mode = IMMEDIATE, Value =
ON)
```

For Arithmetic-like and FPGA statements the format is:

```
Time-related information => "User-assigned Name" : ASSIGNEE = EXPRESSION
```

where:

- `ASSIGNEE` is a named reference, such as `event`, `trigger`, `action`, `reg`, or `fpgaReg` followed by the label in quotes.
- `EXPRESSION` is a mathematical expression with binary operators, such as addition, subtraction, and assignment.

For example:

```
+10ns => "Increment counter register": reg"LeaderEngine.Loop Counter" = reg"LeaderEngine.Loop
Counter" + 1
```

Time related information

The time information section of the visualization output is in the following format:

```
+Start_delay <Duration> Absolute_time =>
```

NOTE There are a number of limitations in this release:

- Duration is shown as `Min` or `?`.
- Absolute time is not shown in this release.

Indicators

The visualization output uses the following characters to indicate different pieces of information:

Category	Indicators	Description
Timing-related information	+	Appears at the start, the number with this indicates the Start delay.
	<>	Encloses a Duration if it is set. If the Duration is not set, this defaults to `min`, which is the minimum time possible.
		Absolute time (not supported in this release).
Separator	=>	Separator. The time information for the statement is on the left of this and information about the statement is on the right.
Command label and name	" "	Encloses labels
	:	Divides the label and the command description.
Blocks and parameters	{ ... }	Encloses blocks of statements: • Sync multi-sequence block. • Engine instructions. • Sync flow-control. • Local flow-control.
	(...)	Enclose parameters. These can be optional.
	[...]	Enclose lists. For example [element, ...], or for named element lists [name"username", ...]
Register indicators	reg	Indicates a register.
	fpgaReg	Indicates an FPGA register

Code blocks

Code blocks are indented and shown within curly braces. Code blocks include code in Sync multi-sequence blocks, Engines, and flow-control statements.

The following example from *Programming Example 1* shows a Sync multi-sequence block `TriggerAWGs` that contains a pair of engines `AwgEngine0` and `AwgEngine1`.

```
+30ns<Min> => "TriggerAWGs": SyncMultiSequenceBlock {
    Engine "AwgEngine0" {
        +10ns => "FP Trigger ON": TriggerWrite(Trigger = [trigger"TriggerIO"], Mode = IMMEDIATE,
Value = ON)
        +100ns => "FP Trigger OFF": TriggerWrite(Trigger = [trigger"TriggerIO"], Mode =
IMMEDIATE, Value = OFF)
        +10ns => "AWG trigger": ActionExecute([action"GenMarker1", action"GenMarker2"])
    }
    Engine "AwgEngine1" {
        +10ns => "FP Trigger ON": TriggerWrite(Trigger = [trigger"TriggerIO"], Mode = IMMEDIATE,
Value = ON)
        +100ns => "FP Trigger OFF": TriggerWrite(Trigger = [trigger"TriggerIO"], Mode =
IMMEDIATE, Value = OFF)
        +10ns => "AWG trigger": ActionExecute([action"GenMarker1", action"GenMarker2"])
    }
}
```

If an engine does not execute any statements, the engine is shown with empty braces. For example, in the previous example, if the EngineAwgEngine1 didn't have any instructions, it would be shown as:

```
Engine "AwgEngine1" {}
```

Format variations

There are variations of the sequence visualization output format for different types of statement.

Sync statements

The following example shows a Sync register-sharing command that copies the contents of the `Steps` register in the Digitizer Engine to the `Wavefrom ID` register in the AWG Engine:

```
+190ns<Min> => "Share steps->wfm_id": SyncRegisterSharing {
    reg"Digitizer Engine.Steps"[1:0] => [reg"AWG Engine.Waveform ID"]
}
```

Sync multi-sequence blocks

The output for a Sync multi-sequence block indicates any engines it contains. The sequences and the statements they contain are shown within each engine.

The following example shows the output for a Sync multi-sequence block that contains 2 engines. The first engine is labelled Digitizer Engine and contains a sequence with a pair of local statements. A second engine labelled AWG Engine does not contain any sequences. This is indicated with empty braces.

Visualization output for a Sync multi-sequence block:

```
+260ns<Min> => "Loop Delay": SyncMultiSequenceBlock {
    Engine "Digitizer Engine" {
        +10ns => "loops++": reg"Digitizer Engine.Loops" = reg"Digitizer Engine.Loops" + 1
        +30ns<?> => "WaitTime: loop_delay": WaitTime(reg"Digitizer Engine.Loop Delay")
    }
    Engine "AWG Engine" {}
}
```

Sync flow-control and Local flow-control statements

Flow control statements show the flow-control condition and the statements executed if the condition is met.

The following example shows a Local If. The condition is indicated along with the matching branches parameter and the statement executed is also shown inside braces.

Visualization output for a Local If statement:

```
+70ns<?> => "Check wfm_id": If(condition = (reg"AWG Engine.Waveform ID" > = 1), MatchingBranches
= TRUE) {
    +30ns => "wfm_id = 0": reg"AWG Engine.Waveform ID" = 0
}
```

If a flow control instruction contains multiple branches, these are also listed.

The following example contains a Local If with a condition and an Else branch that is executed when the If condition is not met.

```
+70ns<?> => "Queue Wfm AWG": If(condition = (reg"AWG Engine0.Queue Reg" == 0), MatchingBranches
= TRUE) {
    +100ns => "Queue Waveform A CH1": M30xxA.AwgQueue(Channel = 1, WaveformId = reg"AWG
Engine0.Wfm A", Cycles = 3, StartDelay = 0, Prescaler = 0, TriggerMode = AUTOTRIG)
}
Else {
    +100ns => "Queue Waveform B CH1": M30xxA.AwgQueue(Channel = 1, WaveformId = reg"AWG
Engine0.Wfm B", Cycles = 2, StartDelay = 0, Prescaler = 0, TriggerMode = AUTOTRIG)
}
```

Local instructions

Local Instruction statements show the Start delay, the label, instruction and any parameters. For example:

```
+10ns => "Increment counter register": reg"LeaderEngine.Loop Counter" = reg"LeaderEngine.Loop
Counter" + 1
```

Custom instructions

Custom instructions indicate the product family before the instruction in the form:

```
ProductFamily.CustomInstructionName
```

In the following example, the product family `KtM30xxA` is indicated before the custom instruction `QueueWaveform`:

```
+100ns => "QueueWaveform(CH1, wfm_id)": M30xxA.AwgQueue(Channel = 1, WaveformId = reg"AWG
Engine.Waveform ID", Cycles = 1, StartDelay = 0, Prescaler = 0, TriggerMode = AUTOTRIG)
```

Examples

The following example is an excerpt from Programming Example 1. It shows the Python code for setting up the TriggerWrite and ActionExecute instructions and the resulting sequence visualization output that is generated.

Python Code:

```
# Write FP Trigger ON to all instruments
fp_trigger = sequence.engine.triggers[config.fp_trigger_name]
trigger_write = sequence.instruction_set.trigger_write
instr_trigger_ON = sequence.add_instruction("FP Trigger ON", 10, trigger_write.id)
instr_trigger_ON.set_parameter(trigger_write.trigger.id, fp_trigger)
instr_trigger_ON.set_parameter(trigger_write.sync_mode.id, trigger_write.sync_mode.immediate)
instr_trigger_ON.set_parameter(trigger_write.value.id, trigger_write.value.on)
# Write FP Trigger OFF to all instruments
instr_trigger_OFF = sequence.add_instruction("FP Trigger OFF", 100, trigger_write.id)
instr_trigger_OFF.set_parameter(trigger_write.trigger.id, fp_trigger)
instr_trigger_OFF.set_parameter(trigger_write.sync_mode.id, trigger_write.sync_mode.immediate)
instr_trigger_OFF.set_parameter(trigger_write.value.id, trigger_write.value.Off)
# Execute AWG trigger from the HVI sequence of each module
# "Action Execute" instruction executes the AWG trigger from HVI
action_list = sequence.engine.actions
instruction1 = sequence.add_instruction("AWG trigger", 10, sequence.instruction_set.action_
execute.id)
instruction1.set_parameter(sequence.instruction_set.action_execute.action.id, action_list)
```

The Sequence visualization output from the previous code:

```
+10ns => "FP Trigger ON": TriggerWrite(Trigger = [trigger"TriggerIO"], Mode = IMMEDIATE, Value =
ON)
+100ns => "FP Trigger OFF": TriggerWrite(Trigger = [trigger"TriggerIO"], Mode = IMMEDIATE, Value
= OFF)
+10ns => "AWG trigger": ActionExecute([action"GenMarker1", action"GenMarker2"])
```

Sequence Visualization Error Messages

The sequence visualization system can detect and report errors.

Errors can be part of an assignment expression, destination, or parameter value.

For example, if a parameter has not been set in an instruction. In the case of register parameters this can result in values completely missing from the output or exceptions being thrown.

Error formats

Errors are indicated in the following formats:

Errors with a message

An error in indicated in the following manner, the messages provided do not contain @ symbols:

```
@ERROR: <message>@
```

Errors with no message

In some cases, an error is be indicated without a message:

```
@ERROR@
```

The following example output shows some example errors:

```
+90ns<Min> => "Sync MIMO Trigger": SyncWhile(reg"AwgEngine0.Loops" < 3) {
       +250ns<Min> => "TriggerAWGs":SyncMultiSequenceBlock {
               Engine "AwgEngine0" {
                       +10ns => "assign":@ERROR: register is not set@ = @ERROR@
            +100ns => "QueueWaveform(CH1, wfm_id)": M30xxA.AwgQueue(Channel = @ERROR@,
WaveformId = @ERROR: invalid id@, Cycles = 1, StartDelay = 0, Prescaler = 0, TriggerMode =
AUTOTRIG)
               }
       }
}
```

HVI Component Versions

You can obtain the following HVI component versions:

- HVI Core Version
- HVI Software Version
- HVI Firmware Version

Python	Class	Description
`keysight_hvi.SystemDefinition.hvi_core_version`	`SystemDefinition`	The version of the HVI core component that gets installed by PathWave Test Sync Executive software to deliver the HVI API.

`Engine.software_version`
`EngineView.software_version`
`EngineRuntime.software_version`
	`EngineDefinition`
	`EngineRuntime`
	`EngineView`
		The version of the HVI software component used by the instrument associated with this engine object.

`Engine.firmware_version`
`EngineView.firmware_version`
`EngineRuntime.firmware_version`
	`EngineDefinition`
	`EngineRuntime`
	`EngineView`
		The version of the HVI Engine FPGA IP that is programmed into the FPGA of the instrument associated with this engine object.

hvi_core_version

This version is a property of the SystemDefinition.

This is the version of the HVI core component that gets installed by PathWave Test Sync Executive software. This provides the HVI API.

software_version

This is a property of each Engine. It can be obtained from the `EngineDefintion`, `EngineRuntime` and `EngineView` classss.

This is the version of the HVI core component consumed by the instrument corresponding to this engine object. This version does not need to be the same as the HVI core installed by PathWave Test Sync Executive software for the system to work, but a matching version is necessary to be able to deploy all the latest features. This software version depends on the version of the instrument drivers provided by the instrument Software Front Panel software.

firmware_version

This is a property of each Engine. It can be obtained from the `EngineDefintion`, `EngineRuntime` and `EngineView` classss.

This is the version of the HVI Engine FPGA IP that is programmed in the FPGA of the instrument corresponding to this engine object. The HVI Engine IP version changes with each version of the instrument FPGA firmware. You can program the firmware into the FPGA using the instrument Software Front Panel software.

NOTE If the `software_version`and `hvi_core_version`are different, the HVI core component that gets installed in your system is the newest of the one provided by the instrument and the one delivered by PathWave Test Sync Executive software, regardless of the installation order.

Major, minor and revisions

The version has the sub-versions: `major`, `minor` and `revision`.

The following code shows an example of how to get the versions:

HVI Software and Firmware Versions

```
logging.info("HVI Core : {}".format(sys_def.hvi_core_version.to_string()))
for engine in sys_def.engines:
       logging.info("Firmware Version : {}.{}.{}".format(engine.firmware_version.major,
engine.firmware_version.minor, engine.firmware_version.revision))
       logging.info("Software Version : {}.{}.{}".format(engine.software_version.major,
engine.software_version.minor, engine.software_version.revision))
```

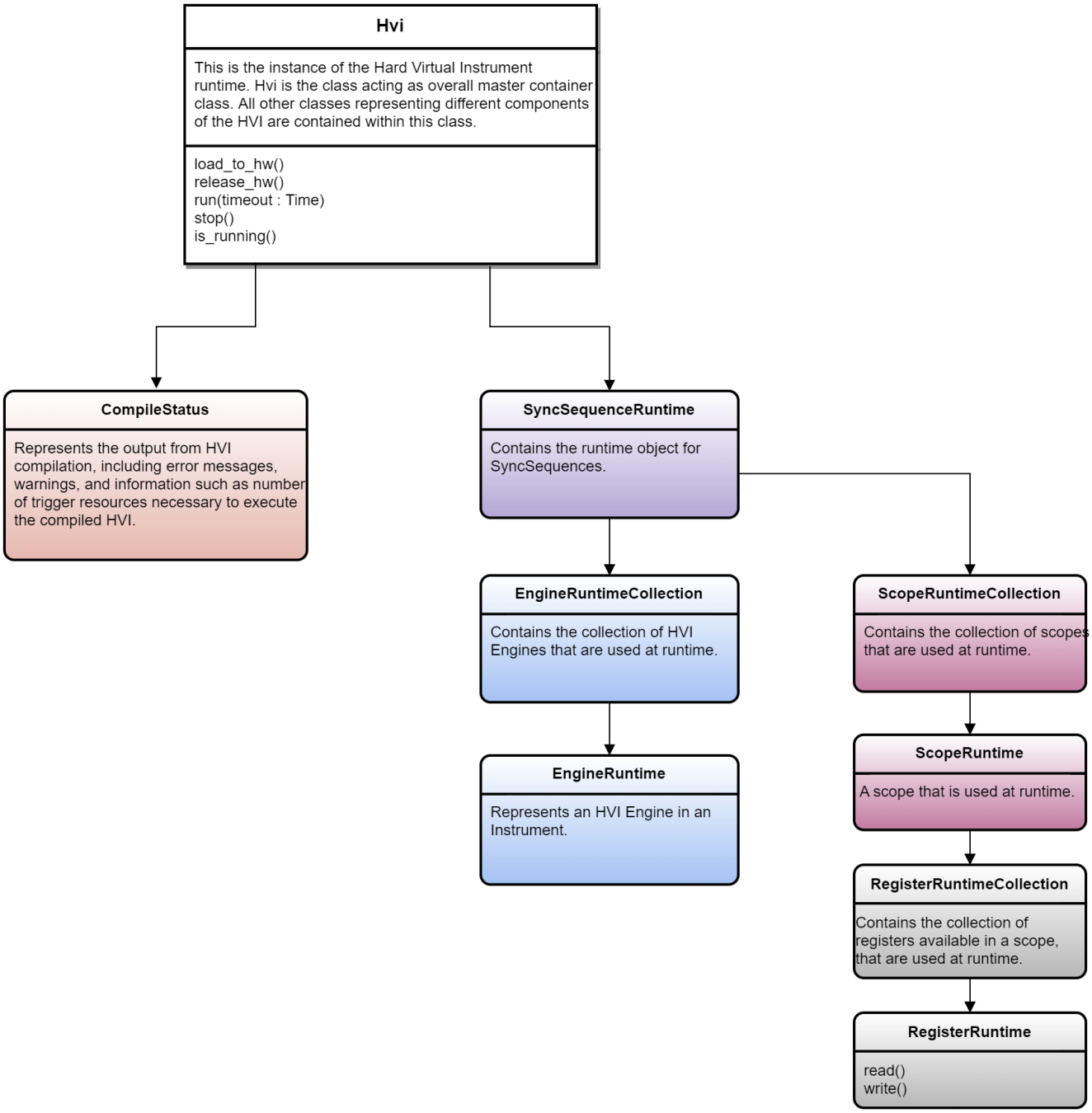
The Hvi Object

This section describes the Hvi object, it contains the following sections:

The Hvi object is the actual HVI instance. This is ready to be loaded to hardware and executed. It contains the runtime versions of the objects you set up with the `SystemDefinition` and `Sequencer` classes. The runtime objects are the instances of the objects that operate while the HVI is running. You cannot modify these objects at runtime, but you can access resources such as HVI registers or an FPGA memory map.

NOTE The Hvi object is the runtime object. once you have compiled it, you can no longer change resources or sequences.

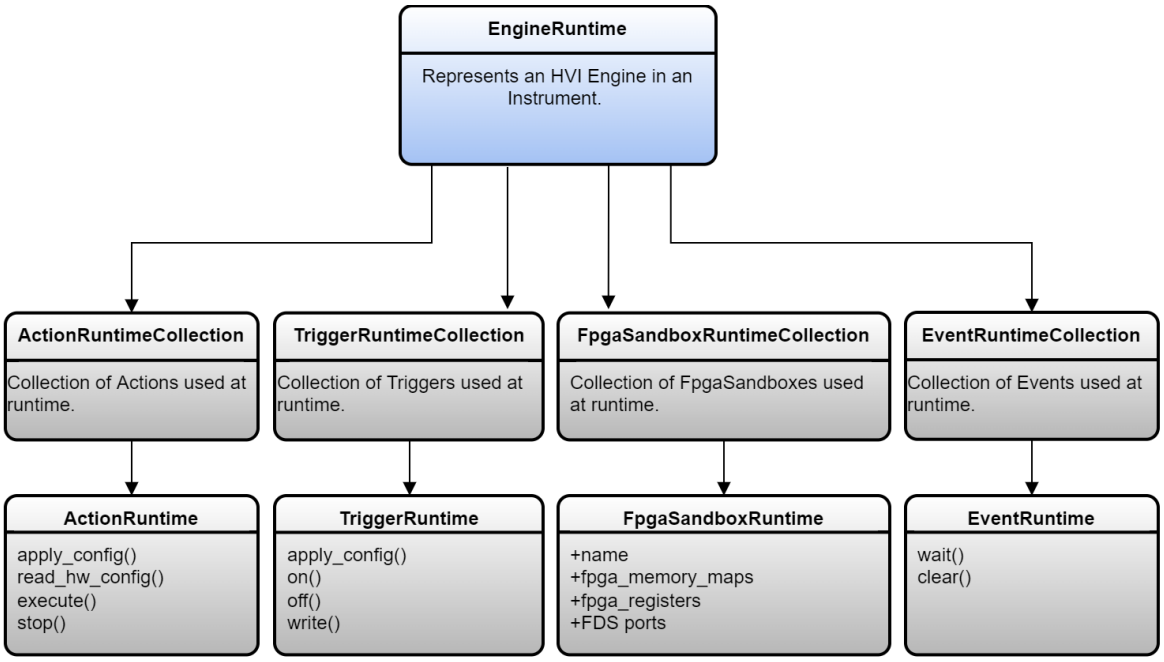
The following diagram shows the classes:



EngineRuntime Components

A number of runtime components are under the EngineRuntime.

The following diagram shows the EngineRuntime and classes:



ActionRuntime

Represents an action which can be passed to `InstructionStatement.set_parameter` as an input parameter.

TriggerRuntime

Trigger provides an interface control and configure the hardware trigger controlled by HVI. This Instance can be passed to `InstructionStatement.set_parameter` as input.

EventRuntime

The `EventRuntime` class is used to represent hardware events which are defined by an instrument and can be used by HVI, for example, to activate `TriggerRuntime`.

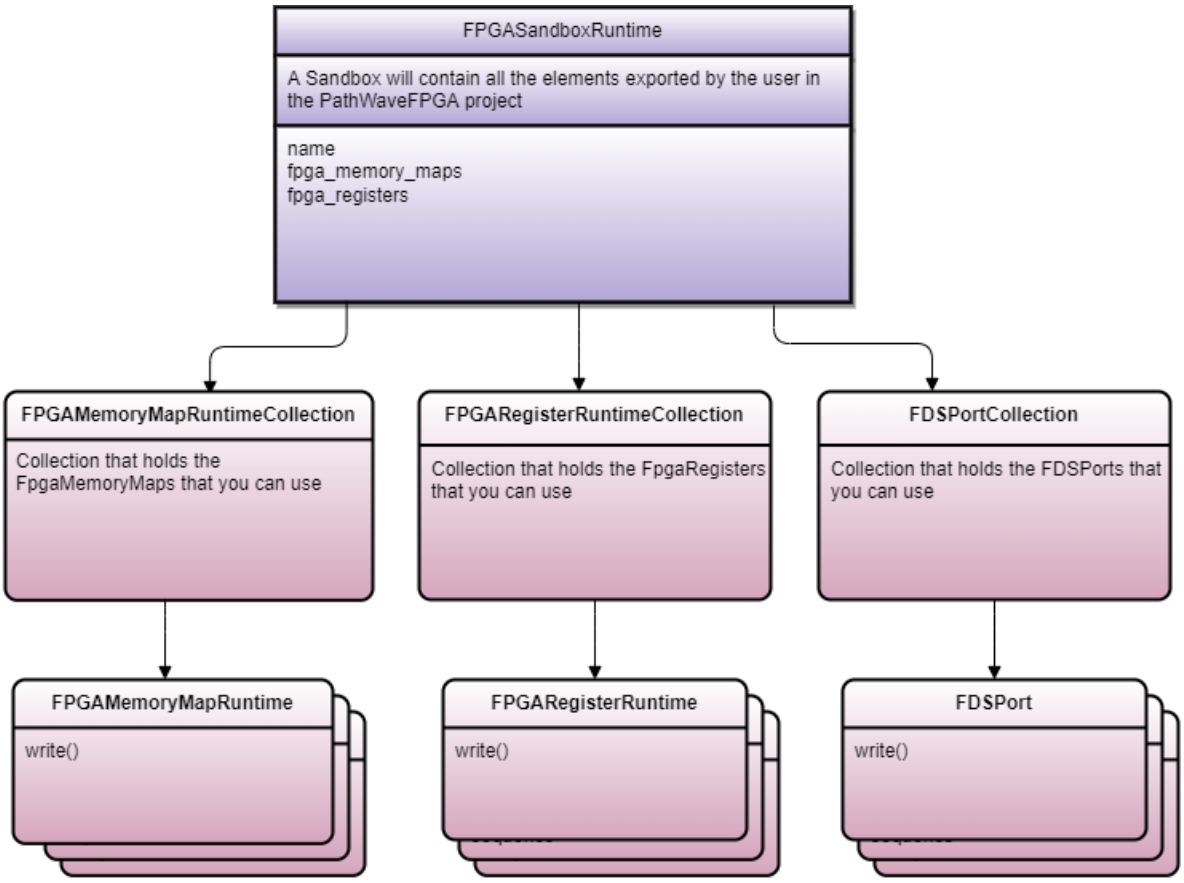
RegisterRuntime

Represents instrument-defined hardware registers that can be used like Variables in HVI sequences as parameters for statements.
These registers can be accessed and modified by both HVI instructions in real-time during the sequence execution and HVI software calls.
Registers can be treated as signed or unsigned.
The range of the value of a register depends on the register size and must be within the signed or unsigned range.

FpgaSandboxRuntime

This section describes the FPGA sandbox runtime.

`FPGASandboxRuntime` contains all the FPGA registers and memory maps available at runtime. The following diagram shows the classes:



The `FPGASandboxRuntime` object can be obtained from the `Hvi` object:

```
SANDBOX_0_NAME = "sandbox0" sandbox = hvi.sync_sequence.engines[0].fpga_sandboxes[SANDBOX_0_
NAME]
```

NOTE Hvi resources can only be read or written loaded, that is, between the `load_to_hw()` and `release_hw()` calls. Any attempt to read or write resources without having the instrument loaded to hardware results in an exception being thrown.

FPGA registers

Once the Sequencer has been compiled and the HVI has been loaded to hardware, the register can be read and written. If the HVI is not loaded, an exception is thrown.

```
fpga_register = hvi.sync_sequence.engines[0].fpga_sandboxes[SANDBOX_0_NAME].fpga_registers[0]
hvi.load_to_hw()
fpga_register.write(1) # ok
hvi.release_hw()
fpga_register.write(1) # exception is thrown
```

FPGA memory maps

As with registers, FPGA Memory maps can be used after HVI has been loaded to hardware. They can only be accessed, read, or written while the HVI is loaded to hardware.

```
fpga_memory_map = hvi.sync_sequence.engines[0].fpga_sandboxes[SANDBOX_0_NAME].fpga_memory_maps
[0]
hvi.load_to_hw()
fpga_memory_map.write(0x10, 0x1245) # ok
hvi.release_hw()
fpga_memory_map.write(0x10, 0x1245) # exception is thrown
```

FDS Ports

As with registers, FPGA Memory maps can only be accessed after HVI has been loaded to hardware.

Load to Hardware and Run

After the Hvi object is compiled, you retrieve it from the compilation output. To execute it, you must load it to hardware and run it.

These operations are performed using the following API methods that are within the Hvi API object.

To load the HVI to hardware call the method `hvi.load_to_hw()`.

The `hvi.load_to_hw()` method deploys HVI to hardware and does all of the resource configuration including:

- Synchronization resources.
- Trigger resources.
- Clocks.

The `hvi.load_to_hw()` method also loads the binaries containing information to execute the HVI sequences, to the relevant HVI engines.

Once the HVI has been loaded to hardware, you can execute your sequences by calling `hvi.run()`. The HVI execution in Hardware finishes when the HVI sequence reaches the end. The `Stop()`method can be used to stop or cancel the HVI execution.

When the HVI has finished execution and it is not needed to run the HVI again, call the method `ReleaseHw()` to release or free all resources used by the HVI.

Real-time Hardware Execution Error Handling

This section describes real-time hardware error handling during the HVI execution.

If a hardware error is detected while a sequence is running, it is possible the execution results are invalid or unreliable. HVI captures some critical hardware errors and reports them to the user. Errors are indicated in a different way, depending on if you are running your HVI sequence in blocking mode or non-blocking mode:

Errors in an HVI Sequence Running in Blocking Mode

If an error occurs when the HVI is executed in blocking mode, the sequence execution is halted, an exception is thrown, and a list of the errors can be queried.

The error reporting sequence goes in the following order:

1. Call `loadtoHw()`
2. Call `run()`. When an error event occurs while the HVI is running in hardware:
   a. The HVI sequence is halted.
   b. The error list is updated.
   c. An exception is thrown.
3. Call `getExecutionErrors()` to obtain details of the errors that has occurred during the execution.

Errors in an HVI Sequence Running in Non-Blocking Mode

If an error occurs when the HVI is in non-blocking mode, sequence execution is not halted automatically and no exception is thrown, since in non-blocking mode the `run()` method returns immediately as soon as the sequence execution starts. In order to query the status of the HVI execution you must call `getExecutionStatus()`. This method returns the enum `ExecutionStatus` with values for:

- Not started.
- Running.
- Sequence completed successfully.
- Sequence stopped due to timeout.
- Sequence stopped due to error.
- Sequence stopped by user.

Retrieving the Errors List

If a sequence stops because of an error, a list is populated with all the errors detected during execution.

To retrieve the errors call the method `getExecutionErrors()`, it returns a list of any errors that occurred during the last run.

If you call `getExecutionErrors()`more than once, it returns the same list. Calling `run()`, `loadToHw()` or `ReleaseHw()` clears the list.

The `ExecutionError` class provides the following information:

- A complete string-based representation for easy printing.
- The name of the HVI engine that reported the error.
- The product code, this is an integer defined by the instrument that identifies the specific error.
- The product message, this is a string provided by the instrument with any relevant details.

For instrument specific errors, see your instrument documentation for a description of the errors a specific instrument returns.

The HVI Logger

PathWave Test Sync Executive comes with an integrated logger that you can use for troubleshooting.

The logger has the following features:

- The level of logging is configurable.
- You can force flush messages.
- The output can be configured to go to the console or to an output file.
- You can configure the logger from environment Variables or in a **`.env`** configuration file.
- You can instruct some instruments to produce logs.

The logger can produce the following levels of logging information, where each level also includes all the information in the levels below it:

Logger level	Description
Trace	Produces trace information that is useful to support engineers
Debug	Produces debug information that is useful to support engineers
Info	Produces generally useful information
Warning	Logs anything that can potentially cause application oddities, but are automatically recovered.
Error	Logs any errors that are fatal to an operation, but not the service or application
Fatal	Logs any errors that forces a shutdown of the application.
Off	Does not log anything

Logger Configuration

The logger is configured with environment Variables. The following table describes the Variables:

Environment Variable	Values	Description
`HVI_LOGGER_LEVEL`	• Trace • Debug • Info • Warning • Error • Fatal • Off	This value indicates the level of information printed to the log. The information printed out contains the information for the level specified and all of the levels below it. For example, if the level is set to Debug, all messages except Trace are printed to the log. By default, the level is set to Error, so only Error and Fatal are printed.
`HVI_LOGGER_OUTPUT_PATH`	Any existing valid path in your system, For example: `C:\tmp`	This Variable disables console output and tells the logger to save the log to a file at the specified location. The file with the log messages is called: **`HVILog_[num].txt`**
`HVI_LOGGER_FORCE_FLUSH`	`1` or `0`	This Variable forces the log messages to be flushed to the output every time a message is logged. Enable this if you want to troubleshoot a program that is crashing, so that all messages before the crash shall be written. Do not enable this option in any other cases, because it impacts the performance of the execution.
`HVI_LOGGER_EXTENDED`	"`*`", "`ALL`", or a comma separated list. For example: `M9032,M9546`	This Variable enables the logging output of instruments managed by HVI. An output file for each instrument is generated in the path specified with **`HVI_LOGGER_OUTPUT_PATH`** . The file is saved as: `{MODEL}_{Chassis Slot for M903x}_{date}.log` See the section *Logger Extended mode Supported Instruments* for a list of supported instruments.

NOTE By default the configuration for the logger is:

- Logging level: Error.
- Output: console.
- Force flush: disabled.
- Logger Extended: disabled.

.env Configuration File

The logger configuration can be also configured from a **.env** file. The configuration values are stored in the file as KEY=VALUE pairs and you can use **#** for comments.

The **.env** file must be located in the same folder as the HVI script to be executed. HVI parses the **.env** file and sets all the environment Variables found for that script.

The following shows an example **.env** file:

.env

```
# The hvi logger level: Trace, Debug, Info, Warning, Error, Fatal, Off.
HVI_LOGGER_LEVEL=Fatal
#
# Set this parameter to write the logs to a file instead of being printed to the console
HVI_LOGGER_OUTPUT_PATH=C:\tmp\hviLogs
#
# Set this parameter to force flush the log every new line instead of doing it at the end.
# This helps you to identify the line of code before a crash.
HVI_LOGGER_FORCE_FLUSH=0
#
# Activates the Logger for all HVI controlled instruments. The supported models are the
# System Synchronization Modules (M9032,M9033), the High Performance Reference Clock Source,
(M9546)
# or "ALL"HVI_LOGGER_EXTENDED=ALL
```

Logger Extended mode Supported Instruments

PathWave Test Sync Executive can control a number of different instruments. The environment Variable **HVI_LOGGER_EXTENDED** activates logging output from the instruments that support it. The way the logs are produced depends on the instruments, some instruments produce individual log files whereas other instruments combine log files together into a single file.

The supported models for release 2022 are:

Model	Description
M9546x	High Performance Reference Clock Source
M9032, M9033	System Synchronization Modules

Recommended Logger settings for contacting support

If you require support for PathWave Test Sync Executive, a log file will help the support team to rapidly diagnose any problems.

If you want to contact support, first generate a log with the following settings:

- `HVI_LOGGER_LEVEL=Trace`
- `HVI_LOGGER_FORCE_FLUSH=1`
- `HVI_LOGGER_OUTPUT_PATH=C:\Logs` or another path[1]

1 The path must be an existing valid path.

HVI API Sync Statements

This section describes the HVI Statements in the HVI API that you use to program HVI Sequences. The functions of each statement are explained in detail along with a corresponding HVI diagram. Python code examples are provided showing how to program the statements with the HVI Python API.

Sync statements

Sync statements are the building blocks used to program Sync sequences. The following types of Sync statement are available:

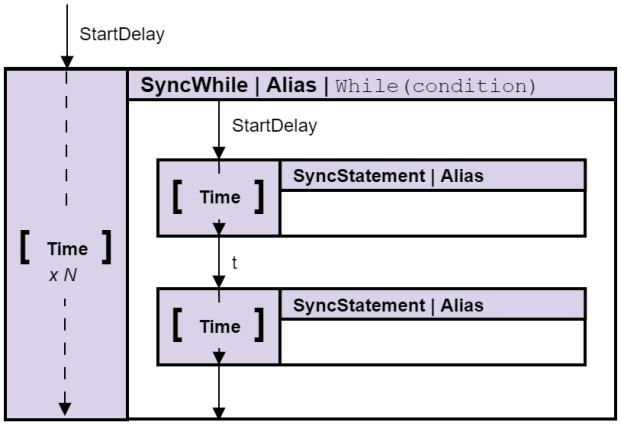
- Sync while.
- Sync multi-sequence block.
- Sync register-sharing.
- Sync FPGA data-sharing.

Sync while

The Sync while statement is a type of Sync statement that is defined by the API class `SyncWhileStatement`. A Sync while enables you to synchronously execute multiple local sequences while a condition you specify is met. The Sync while condition is evaluated each time at the beginning of the statement execution. If the condition is true, an iteration of the Sync while statement is executed. If the condition is false, the HVI execution jumps to the statement following the Sync while.

You can add other Sync statements inside a Sync while. To define local sequences within the Sync while, you must use a Sync multi-sequence block.

A Sync while that contains a pair of Sync statements is shown in the following diagram:



If you are using a Sync while statement across multiple engines, during its execution, one of the engines is set to the role of *Leader* and the remaining engines have the role of *Follower*:

Leader

The condition of the Sync while statement is evaluated in this engine and the result is propagated to the other engines through hardware resources, for example, PXI triggers in a PXI platform.

Follower

A *Follower* engine monitors the result of the condition and acts on it, following the *Leader*.

The condition expression assigned to the Sync while must use resources that belong to the same HVI engine. The *Leader* engine of the Sync while is selected automatically by the HVI compiler from the condition expression.

The following code example shows how to add a Sync while statement and access the Sync sequence in the Sync while.

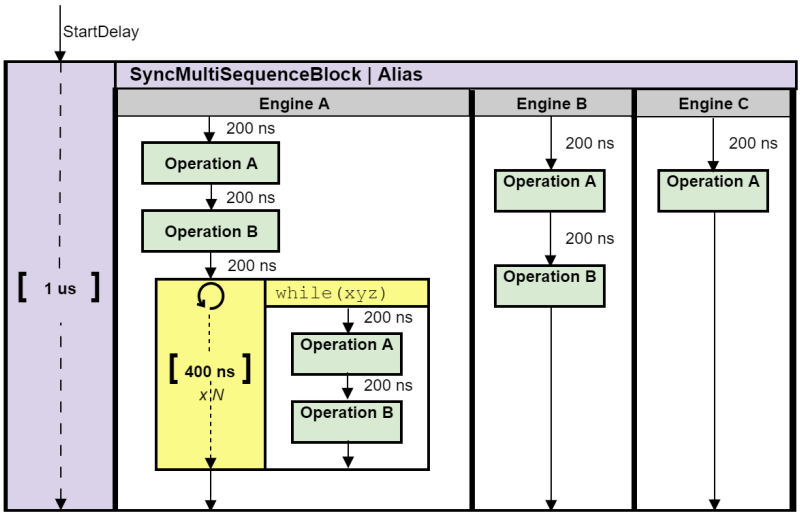
```
# Configure Sync While Condition
sync_while_condition = keysight_hvi.Condition.register_comparison(reg, keysight_
hvi.ComparisonOperator.GREATER_THAN, 10)
#
# Add Sync While to a sync-sequence
sync_while = my_sync_seq.add_sync_while("sync_while", 10, sync_while_condition)
#
# Access the sync sequence in the Sync-While and add Sync-Statements inside
sync_block = sync_while.sync_sequence.add_sync_multi_sequence_block("exec_block",10)
```

Sync multi-sequence block

Sync Multi-Sequence Blocks are a type of Sync statement that contains a set of local sequences. It serves as a container and boundary between sections, where each local sequence executes on an individual engine within a specific instrument.

The Sync multi-sequence block enables you to program each engine to do specific operations and run them on each engine concurrently. The Sync multi-sequence block synchronizes all the engines so that all of the contained Local sequences start at exactly the same time, and the sync sequence remains synchronous afterwards. You can define which Local statements each engine is going to execute, and the exact time each Local statement starts to execute compared to the previous one.

The following diagram shows a Sync multi-sequence block that contains three Local sequences:



The following code snippet shows a Sync multi-sequence block being added with the call `add_sync_multi_sequence_block()`, a Local sequence is then obtained and an instruction added to it:

```
# Add Sync Multi-Sequence Block
sync_block = keysight_hvi.sync_sequence.add_sync_multi_sequence_block("TriggerAWGs")
#
# Add instruction to a local sequence in the block
sequence = sync_block.sequences["Main Engine"]
inst = sequence.add_instruction("Add Instruction", 10, seq.instruction_set.add_instruction.id)
```

Sync register-sharing

Sync register-sharing enables you to share data from a source register to a destination register in any engine in your HVI. Specifically, you share the contents of `N` adjacent bits from a source register to a destination register.

Sync register-sharing is defined in and programmed using the class `SyncRegisterSharingStatement`.

In the following code example, Sync register-sharing is used to share the content of the digitizer register `feedback` and write into the AWG register `wfm_id`:

```
# Digitizer registers
feedback = keysight_hvi.sync_sequence.scopes["Dig Engine"].registers.add("Feedback Reg",
keysight_hvi.RegisterSize.SHORT)
feedback.initial_value = 0
#
# AWG registers
wfm_id = keysight_hvi.sync_sequence.scopes["AWG Engine"].registers.add("Wfm ID", keysight_
hvi.RegisterSize.SHORT)
wfm_id.initial_value = 0
#
# Add sync register sharing
bits_to_share = 3
sync_while_2.sync_sequence.add_sync_register_sharing("Share feedback->wfm_id", 10, steps, wfm_
id, bits_to_share)
```

Sync FPGA data-sharing

This statement enables you to share data between FPGA sandboxes in different instruments. It uses Fast Data Sharing technology to share data.

The transfer is preformed with an `FpgaDataSharingTransaction.` An FPGA data sharing transaction enables sharing of a specified number of bits. The number of bits to be shared must be a multiple of 4.

The start and the end of the statement are time synchronized. The `SyncFPGADataSharing` statement blocks the execution in all instruments until the last instrument has received the last 4 bits of data.

Related classes:

New Classes	Description
`FpgaDataSharingTransaction`	An FPGA data sharing transaction enables you to share a specified number of bits from one FPGA sandbox to one or more FPGA sandboxes in different instruments. You add a transaction to a `SyncFpgaDataSharingStatement` using one of the `FpgaDataSharingTransactionCollection.add()` methods.
`FpgaDataSharingTransactionCollection`	A list of FPGA data sharing transactions.
`FdsPortAddress`	Represents the source or destination address of a Fast Data Sharing transaction. It consists of the FDS port of an instrument that shall be used to transmit or receive data, as well as the address of the IP connected to that FDS port, where the data will be read from / written to.

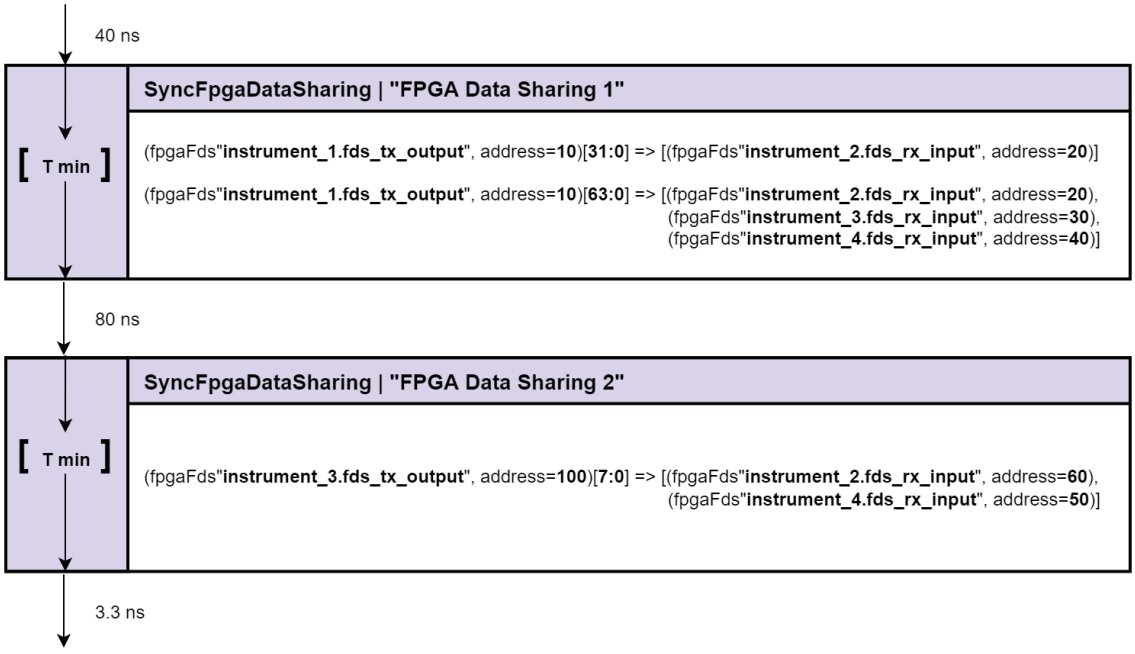
Use the following method to add a `SyncFpgaDataSharingStatement` statement into a sequence:

New Method	Description
`SyncSequence.add_sync_fpga_data_sharing`	Adds a `SyncFpgaDataSharingStatement`

The following code snippet shows examples of `SyncFpgaDataSharingStatement` statements:

```
# The name for each sandbox provided below should match exactly the name that each instrument
has given to each sandbox
# For simplicity, we assume that all instrument use the same name for their sandboxes
instrument_1_sandbox_name = "SampleSandbox"instrument_2_sandbox_name =
"SampleSandbox"instrument_3_sandbox_name = "SampleSandbox"instrument_4_sandbox_name =
"SampleSandbox"# The name for each FDS port provided below should match exactly the name that
was given by each user to the FDS port in the sandbox
# For simplicity, we assume that all Tx ports share a common name
source_port_name = 'fds_tx_output'
# For simplicity, we assume that all Rx ports share a common name
dst_port_name = 'fds_rx_input'
instrument_1_fds_ports = sequencer.sync_sequence.engines["instrument_1"].fpga_sandboxes
[instrument_1_sandbox_name].fds_ports
instrument_2_fds_ports = sequencer.sync_sequence.engines["instrument_2"].fpga_sandboxes
[instrument_2_sandbox_name].fds_ports
instrument_3_fds_ports = sequencer.sync_sequence.engines["instrument_3"].fpga_sandboxes
[instrument_3_sandbox_name].fds_ports
instrument_4_fds_ports = sequencer.sync_sequence.engines["instrument_4"].fpga_sandboxes
[instrument_4_sandbox_name].fds_ports
# Add an FPGA Data Sharing Statement
name = 'FPGA Data Sharing 1'
start_delay = 40
fpga_data_sharing_st = sequencer.sync_sequence.add_sync_fpga_data_sharing(name, start_delay)
# Add transaction 1: single destination
source_address = 10
source_port = instrument_1_fds_ports[source_port_name]
source = keysight_hvi.FdsPortAddress(source_port, source_address)
dst1_address = 20
dst1_port = instrument_2_fds_ports[dst_port_name]
dst1 = keysight_hvi.FdsPortAddress(dst1_port, dst1_address)
num_bits_to_share = 32
fpga_data_sharing_st.transactions.add(source, dst1, num_bits_to_share)
# Add transaction 2: multiple destinations
dst2 = keysight_hvi.FdsPortAddress(instrument_3_fds_ports[dst_port_name], 30)
dst3 = keysight_hvi.FdsPortAddress(instrument_4_fds_ports[dst_port_name], 40)
dests = [dst1, dst2, dst3]
num_bits_to_share = 64
fpga_data_sharing_st.transactions.add(source, dests, num_bits_to_share)
# Add another FPGA Data Sharing Statement
name = 'FPGA Data Sharing 2'
start_delay = 80
fpga_data_sharing_st = sequencer.sync_sequence.add_sync_fpga_data_sharing(name, start_delay)
source = keysight_hvi.FdsPortAddress(instrument_3_fds_ports[source_port_name], 100)
dst1 = keysight_hvi.FdsPortAddress(instrument_2_fds_ports[dst_port_name], 60)
dst2 = keysight_hvi.FdsPortAddress(instrument_4_fds_ports[dst_port_name], 50)
dests = [dst1, dst2]
num_bits_to_share = 8
fpga_data_sharing_st.transactions.add(source, dests, num_bits_to_share)
```

The following diagram shows `SyncFpgaDataSharing` statements as a result of the previous script:



Sync FPGA data-sharing Errors

The `SyncFpgaDataSharing` statement uses FDS for transfers. It has the following error conditions:

1. The number of bits to share is not a positive multiple of 4.
2. A transaction has multiple FDS ports from the same engine.
3. A transaction contains a singular FDS port more than once. The same FDS port cannot be a source and a destination in the same transaction.

HVI API Local Statements

This section describes the HVI Local Statements in the HVI API that you use to program HVI Sequences.

The functions of each statement are explained in detail together with Python code examples showing how to program the statements with the HVI Python API.

Programming local sequences

You program local sequences within a Sync multi-sequence block or a Local flow-control statement (Local while or Local If). The following code shows an example of a Local sequence programmed within a Sync multi-sequence block.

```
# Add statements to each local sequence within the Sync multi-sequence block
# HVI Local sequence collection is automatically created form the
# user-defined HVI Engine Collection
# Each HVI Local sequence can be retrieved using the name of the corresponding HVI Engine
sequence = sync_block.sequences[engine_name]
#
# Add FP Trigger ON to all instruments
instr_trigger_write = sequence.instruction_set.trigger_write
instr_trigger_ON = sequence.add_instruction("FP Trigger ON", 10, instr_trigger_write.id)
instr_trigger_ON.set_parameter(instr_trigger_write.trigger, fp_trigger)
instr_trigger_ON.set_parameter(instr_trigger_write.sync_mode.id, instr_trigger_write.sync_
mode.immediate)
instr_trigger_ON.set_parameter(instr_trigger_write.value.id, instr_trigger_write.value.on)
```

Instruction statements

Instruction statements are operations that can be executed by the instrument hardware within an HVI sequence. There are two types of instruction statements:

- Instrument-specific HVI instructions.
- HVI-native instructions.

Instrument-specific HVI instructions

Instrument-specific HVI instructions are specific to individual instruments. They are defined by the instrument add-on API and exposed in each instrument driver as instrument specific HVI definitions. Instrument-specific HVI instructions can change instrument settings such as amplitude, frequency, or trigger an instrument function such as output a waveform or trigger a data acquisition. For example, the M3xxxA documentation describes all the HVI instructions available for each of the M3xxxA PXI instruments.

The following code is an example of using the `awgQueueWaveform` custom instruction that is part of the HVI instruction set of the Keysight M320xA AWG instrument. This example shows how to add an instrument specific HVI instruction to a Local sequence using the `add_instruction()` API method and also how to set the instruction parameters using the `set_parameter()` API method:

```
# Retrieve engine sequence:
seq = sync_block.sequences["engine_0"]
#
# Add and program AWG Queue Waveform instruction:
instr_queue_wfm = instrument.hvi.instruction_set.queue_waveform
instruction0 = seq.add_instruction("awgQueueWaveform", 10, .id)
#
# Set instruction parameters:
instruction0.set_parameter(instr_queue_wfm.waveform_number.id, seq.registers
[waveformNumberRegisterName])
instruction0.set_parameter(instr_queue_wfm.channel.id, nAWG)
instruction0.set_parameter(instr_queue_wfm.trigger_mode.id, keysightSD1.SD_
TriggerModes.SWHVITRIG)
instruction0.set_parameter(instr_queue_wfm.start_delay.id, startDelay)
instruction0.set_parameter(instr_queue_wfm.cycles.id, nCycles)
instruction0.set_parameter(instr_queue_wfm.prescaler.id, prescaler)
```

HVI-native instructions

HVI-native instructions are available on all Keysight instruments. They are general purpose and instrument independent. They include Local instructions and Local flow-control statements. The HVI-native instructions and parameters are defined in the interface `hvi.instruction_set`.

The set of HVI-native instructions include:

- Action Execute: AWG trigger, DAQ trigger.
- FPGA register read.
- FPGA register write.
- FPGA memory map write.
- FPGA memory map read.
- Register increment.
- Front panel trigger ON/OFF.
- Register assign.

Action Execute: AWG trigger, DAQ trigger

To add actions to an HVI sequence, you must add them to the instrument's HVI engine with the API `add()` method of the `ActionCollection` class.

Once the required actions are added to the list of the HVI engine actions for the instruments, an instruction to execute them can be added to the instrument's sequence using the HVI API class `InstructionsActionExecute`. One or multiple actions can be executed at the same time within the same Action Execute instruction.

The following code example shows an Action Execute instruction being used to initiate an AWG trigger:

```
# Previously defined actions to be executed within the experiment
awg_trigger_12 = [hvi.sync_sequence.engines["engine_name"].actions["previously_defined_action_
1"], hvi.sync_sequence.engines["engine_name"].actions["previously_defined_action_2"]]
#
# AWG trigger CH1, CH2 - Generates first pulse
sequence = sync_block_2.sequences["engine_name"]
inst_awg_trigger = sequence.add_instruction("AwgTrigger(CH1, CH2)", 10, sequence.instruction_
set.action_execute.id)
inst_awg_trigger.set_parameter(hvi.instruction_set.action_execute.action.id, awg_trigger_12)
```

Register increment

You can implement a register increment within a sequence with the class `InstructionsAdd`. The same instruction can be used to add registers and constant values (operands) and put the result in another register (result). To increment the register, it must have been added previously to the scope of the corresponding HVI Engine.

The following code shows an example of a register increment:

```
# Previously defined
counter = sync_sequence.scopes["AWG Engine"].registers.add("Counter Reg", keysight_
hvi.RegisterSize.SHORT)
#
# Increment counter register
instruction = awg_sequence.add_instruction("Increment counter", 10, awg_sequence.instruction_
set.add.id)
instruction.set_parameter(awg_sequence.instruction_set.add.destination.id, counter)
instruction.set_parameter(awg_sequence.instruction_set.add.left_operand.id, counter)
instruction.set_parameter(awg_sequence.instruction_set.add.right_operand.id, 1)
```

Front panel trigger ON/OFF

The following code example shows a front panel trigger ON/OFF instruction. The instruction is added to the sequence with the method `add_instruction()`. Instruction parameters are set using the API method `set_parameter()`. All HVI-native instructions and parameters are defined in the `hvi.InstructionSet` interface.

```
# Add FP Trigger ON to all instruments
sequence = sync_block.sequences[engine_name]
instr_trigger_write = sequence.instruction_set.trigger_write
instr_trigger_ON = sequence.add_instruction("FP Trigger ON", 10, instr_trigger_write.id)
instr_trigger_ON.set_parameter(instr_trigger_write.trigger, fp_trigger)
instr_trigger_ON.set_parameter(instr_trigger_write.sync_mode.id, instr_trigger_write.sync_
mode.immediate)
instr_trigger_ON.set_parameter(instr_trigger_write.value.id, instr_trigger_write.value.on)
```

Register assign

A register assign statement can be used to initialize a register to an initial value using the instruction class `InstructionsAssign` from the 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, before its execution, by using the API property `Register.initial_value`.

The following code shows an example of Register Assign:

```
# Previously defined registers
wfm_id = hvi.sync_sequence.scopes["AWG Engine"].registers.add("Wfm ID", keysight_
hvi.RegisterSize.SHORT)
#
# Initialize Waveform ID
seq = sync_block_1.sequences["AWG Engine"]
instruction = seq.add_instruction("Initialize Wfm ID", 10, seq.instruction_set.assign.id)
instruction.set_parameter(seq.instruction_set.assign.destination.id, wfm_id)
instruction.set_parameter(seq.instruction_set.assign.source.id, 0)
```

HVI-native FPGA instructions

FPGA register read

The instruction `fpga_register_read` is an HVI-native instruction that enables you read from an HVI FPGA register. The value read from the HVI FPGA register is written to a destination HVI register.

The following code example shows an FPGA register read instruction:

```
# Read FPGA Register into an HVI Register used in the HVI sequence
sequence = sync_block_1.sequences["engine_name"]
hvi_register = hvi.sync_sequence.scopes["engine_name"].registers["my_register"]
fpga_register = hvi.sync_sequence.engines["engine_name"].fpga_sandboxes["sandbox_name"].hvi_
registers["sandbox_register"]
readFpgaReg0 = sequence.add_instruction("Read FPGA Register_Bank_HviAction4Cnt", 10,
sequence.instruction_set.fpga_register_read.id)
readFpgaReg0.set_parameter(sequence.instruction_set.fpga_register_read.destination.id, hvi_
register)
readFpgaReg0.set_parameter(sequence.instruction_set.fpga_register_read.fpga_register.id, fpga_
register)
```

FPGA register write

The instruction `fpga_register_write` is an HVI-native instruction that enables you to write an HVI FPGA register placed in an FPGA sandbox. The value to be written to the HVI FPGA register is taken from an HVI register or from a literal.

The following code example shows an FPGA register write instruction:

```
# Write FPGA Register from an an HVI Register used in the HVI sequence
hvi_register = hvi.sync_sequence.scopes["engine_name"].registers["my_register"]
fpga_register = hvi.sync_sequence.engines["engine_name"].fpga_sandboxes["sandbox_name"].hvi_
registers["sandbox_register"]
seq = sync_block_1.sequences["engine_name"]
writeFpgaReg0 = seq.add_instruction("Write FPGA Register_Bank_HviPxiTrigOut", 50,
hvi.instruction_set.fpga_register_write.id)
writeFpgaReg0.set_parameter(seq.instruction_set.fpga_register_write.fpga_register.id, fpga_
register)
writeFpgaReg0.set_parameter(seq.instruction_set.fpga_register_write.value.id, hvi_register)
```

FPGA memory map read

The instruction `fpga_array_read` is an HVI-native instruction that enables you to read from an HVI FPGA memory map. The value read from the HVI FPGA memory map is written to a destination HVI register.

The following code example shows an FPGA memory map read instruction:

```
# Register, Memory map objects
register = sync_sequence.scopes["engine_name"].registers["my_register"]
hvi_memory_map = sync_sequence.engines["engine_name"].fpga_sandboxes["sandbox_name"].hvi_memory_
maps["memory_map_name"]
# Read Memory Map
seq = sync_block_1.sequences["engine_name"]
readMemoryMap = sync_block_1.sequences["engine_name"].add_instruction("Read FPGA Memory Map",
20, hvi.instruction_set.fpga_array_read.id)
readMemoryMap.set_parameter(seq.instruction_set.fpga_array_read.fpga_memory_map.id, hvi_memory_
map)
readMemoryMap.set_parameter(seq.instruction_set.fpga_array_read.destination.id, register)
readMemoryMap.set_parameter(seq.instruction_set.fpga_array_read.fpga_memory_map_offset.id, 0)
```

FPGA memory map write

The instruction `fpga_array_write` is an HVI-native instruction that enables you to write to an HVI FPGA memory map that is located in an FPGA sandbox. The value to be written to the HVI FPGA memory map is taken from an HVI register or from a literal.

The following code example shows an FPGA memory map write instruction:

```
# Register, Memory map objects
register = sync_sequence.scopes["engine_name"].registers["my_register"]
hvi_memory_map = sync_sequence.engines["engine_name"].fpga_sandboxes["sandbox_name"].hvi_memory_
maps["memory_map_name"]
# Write Memory Map
seq = sync_block_1.sequences["engine_name"]
writeMemoryMap = sync_block_1.sequences["engine_name"].add_instruction("Write FPGA Memory Map",
10, seq.instruction_set.fpga_array_write.id)
writeMemoryMap.set_parameter(seq.instruction_set.fpga_array_write.fpga_memory_map.id, hvi_
memory_map)
writeMemoryMap.set_parameter(seq.instruction_set.fpga_array_write.value.id, register)
writeMemoryMap.set_parameter(seq.instruction_set.fpga_array_write.fpga_memory_map_offset.id, 0)
```

FPGA-instruction statement

The `FpgaInstruction` statement enables you to issue commands to custom FPGA Sandbox logic from within HVI sequences.

You can add the `FpgaInstruction` statement in an HVI sequence. When the HVI sequence is running, the HVI Engine executes the `FpgaInstruction` statement and sends the command and data parameters to a parser and your logic in the FPGA sandbox. Your logic reads the parameter data and executes the command as indicated.

This instruction can only be used successfully on instruments that support it. For more information, see your instrument documentation.

FPGA-instruction Statement Parameters

The `FpgaInstruction` statement has the following parameters:

Parameter	Description	Size	Notes
Port Number	Selects the port in the FPGA sandbox	-	Number of available ports defined by the instrument
Command ID	An identifier for a command you have implemented in custom logic	16 bits	
Data A	The data to send to the IP in the sandbox	40 bits	Supports registers • If the source register is a `short` (32 bits), the top 8 (most significant) bits are set to 0. • If the source register is a `long` (48 bits), the top 8 (most significant) bits are truncated.
Apply [1]	A 1 bit field which determines if the command is applied immediately or is set up for later execution	1 bit	• 0 = Set up now, apply the instruction later. • 1 = Apply instruction immediately (this is the default).

[1] To apply the instruction after setup, you initiate it with the next instruction with Apply set to 1. You can set a number of instructions each with Apply = 0, then the next instruction with Apply = 1 shall trigger all of them.

The Apply=0 followed by an Apply=1 provides a set up now, and apply later option. This enables you to set up the command and then delay the execution so it happens at a specifically timed interval. This also enables you to set up a number of commands and then have them execute simultaneously.

The following code shows an example of an FPGA-instruction:

```
# Set up local sequence
fpga_inst = local_sequence.instruction_set.fpga_instruction
instruction = local_sequence.add_instruction('fpgaInstruction', 10, fpga_inst.id)
#
port_number = 2
data_a = 1234
command_id = 5
apply = 1
#
instruction.set_parameter(fpga_inst.port_number.id, port_number)
instruction.set_parameter(fpga_inst.data_a.id, data_a)
instruction.set_parameter(fpga_inst.command_id.id, command_id)
instruction.set_parameter(fpga_inst.apply.id, apply)
```

Local flow-control statements

Local flow-control statements execute within Local sequences. These include wait statements, loops such as while, and conditional execution like If. Local flow-control statements are depicted with a yellow box in the HVI diagrams in this User Manual.

Local flow-control statements include:

Local wait-for-time

Causes the sequence to wait for a certain time specified in an HVI register. Once the time has elapsed, the sequence will continue.

Local wait-for-event

Causes the sequence to stop and wait for a condition to evaluate true. Once the condition is true, for example, when the selected event occurs, the next instruction is executed. In future releases, this will be extended to more complex conditions.

Local while

Executes the same sequence in a loop while the condition is met.

Local delay

Delays the sequence for a time you specify. The delay is specified in nanoseconds.

Local if (if-elseif-else)

Selects and executes a different possible Local sequence according to the value of a defined condition.

All Local flow-control statements except wait statements, include one or more Local sequences. For instance, Local while statements have a single sequence and the Local If statement can have multiple sequences. These statements have the following common characteristics:

- Sequences in Local flow-control statements can contain any Local statement including Local flow-control statements.
- Only Local statements can be added inside Local sequences and consequently inside Local flow-control statements. You cannot add Sync statements inside Local flow-control statements.

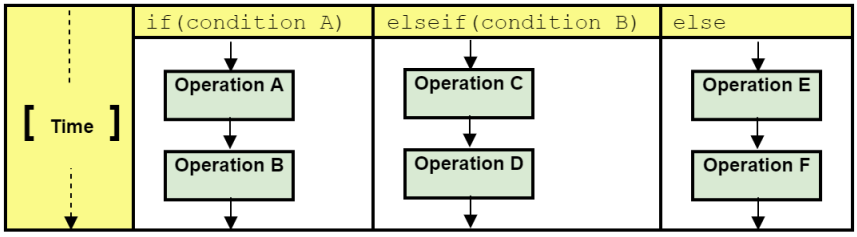
Local if statement

The Local flow-control statement `If` conditionally executes one of a set of different possible Local sequences (if/ elseif / else) depending on the value of predefined conditions.

The conditions are evaluated in the order they are inserted. The possible sequences are:

- At least one sequence that is conditionally executed. This is the main If branch.
- Optional conditional sequences where their conditions are evaluated in order. The first sequence with a true condition is executed if the conditions in previous branches evaluated false. These are the Elseif branches.
- If more then one Elseif condition evaluates to true, only the first is executed.
- One optional Else sequence, which is executed if all above previous conditions evaluate to false. This is the Else branch.

The following diagram shows a Local If flow-control statement:



The class `IfStatement` enables you to add an If-Elseif-Else construct within the main HVI sequence of any HVI engine. The Local If statement contains one If branch, zero or more Elseif branches and one Else branch. The instructions and statements contained in each If or Else branch are executed if the condition of each branch is met.

You can program the branch sequences with the same API methods and classes used to program the main HVI sequence, using the classes `IfBranch` and `ElseBranch, and` you define the condition of each branch with the API class `ConditionalExpression`. The conditions are stored in registers.

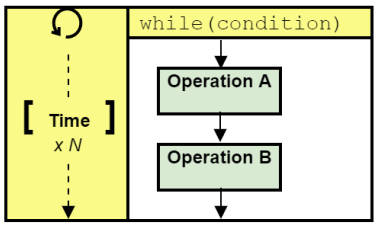
The following code is an example of a Local If statement:

```
# Configure IF condition
if_condition = keysight_hvi.Condition.register_comparison(reg, keysight_
hvi.ComparisonOperator.SMALLER_THAN, 10);
#
# Set flag that enables to match the execution time of all the IF branches
enable_matching_branches = True
if_statement = my_sync_multi_seq_block.add_if("MyIfBlock", 10, if_condition, enable_matching_
branches)
#
# Program IF branch
if_sequence = if_statement.if_branch.sequence
#
# Add statements in if-sequence
instruction = ifSequence.add_instruction("ExecuteAction0", 10, if_sequence.instruction_
set.action_execute.id)
instruction.set_parameter(...) ...
#
# Program Else-If branches
# Else-If Condition
else_if_condition_1 = keysight_hvi.Condition.register_comparison(reg, keysight_
hvi.ComparisonOperator.SMALLER_THAN, 15)
else_if_branch_1 = if_statement.else_if_branches.add(else_if_condition_1)
#
# Program Else-If branch
else_if_sequence_1 = else_if_branch_1.sequence
#
# Add statements in Else-If-sequence
instruction = else_if_sequence_1.add_instruction("SetFrequency", 10, module.HVI.instruction_
set.set_frequency.id)
instruction.set_parameter(...) ...
#
# Eventually add more Else-If-branches
else_if_condition_2 = ... else_if_branch_1 = ... ...
#
# Else-branch
# Program Else branch
else_sequence = else_branch.sequence
#
# Add statements in Else-sequence
instruction = else_sequence.add_instruction(...) ...
```

Local while statement

The Local while flow-control statement executes a same sequence in a loop while a condition is met. The value for the condition is stored in a register.

The following diagram shows a Local while:



The following code is an example of a Local while statement:

```
# Configure while condition
while_condition = keysight_hvi.Condition.register_comparison(reg, keysight_
hvi.ComparisonOperator.NOT_EQUAL, 1)
#
# Add WHILE sequence within the sequence of "engine_0"seq = sync_block.sequences["engine_0"]
while_loop = seq.add_while("While Loop", 10, while_condition)
#
# Program local while sequence
instruction = while_loop.sequence.add_instruction("Initialize Pulse Counter", 10,
seq.instruction_set.assign.id)
instruction.set_parameter(seq.instruction_set.add.destination.id, pulse_counter)
instruction.set_parameter(seq.instruction_set.add.source.id, 0)
```

Local wait-for-event statement

The Local wait-for-event statement causes the HVI sequence to stop and wait for a condition to evaluate true. Once the condition is true, for example the selected event occurs, the next instruction is executed.

The Local wait statement is implemented with the API class `WaitStatement`. This sequence block statement sets an instrument to wait for a condition. The condition can be defined by a trigger, an event, or a combination of them using logical operators. You can only use one event in the condition.

In the following example, the Local wait is used to set a digitizer instrument to wait for an external front panel trigger. The wait statement is set to wait for a trigger falling edge using the `.wait` mode `keysight_hvi.WaitMode.TRANSITION` combined with a trigger configuration as `ACTIVE_LOW`. The sync mode `keysight_hvi.SyncMode.IMMEDIATE` sets the wait event to let the execution continue immediately, that is, as soon as the trigger event is received:

```
# Trigger resource to be used as a wait condition
fp_trigger_id = module_list[0].hvi.triggers.front_panel_1
fp_trigger = sync_sequence.engines[digitizer_engine_name].triggers.add(fp_trigger_id, "FP
Trigger")
#
# Trigger configuration
# NOTE: Trigger to be used as WaitEvent conditions must be configured
# as keysight_hvi.Direction.INPUT
fp_trigger.configuration.direction = keysight_hvi.Direction.INPUT
fp_trigger.configuration.drive_mode = keysight_hvi.DriveMode.PUSH_PULL
fp_trigger.configuration.polarity = keysight_hvi.TriggerPolarity.ACTIVE_LOW
fp_trigger.configuration.hw_routing_delay = 0
fp_trigger.configuration.trigger_mode = keysight_hvi.TriggerMode.LEVEL
#
# Define the condition for the wait statement
wait_condition = keysight_hvi.Condition.trigger(hvi.sync_sequence.engines[digitizer_engine_
name].triggers["FP Trigger"])
#
# Add a Wait For Event
wait_event = sync_block_1.sequences[digitizer_engine_name].add_wait("Wait for FP Trigger", 100,
wait_condition)
wait_event.set_mode(keysight_hvi.WaitMode.TRANSITION, keysight_hvi.SyncMode.IMMEDIATE)
```

For information about timing implications for wait for event statements, *Synchronization Points and Sync Sequence Start* in Chapter 9: HVI Time Management and Latency.

Local wait-for-time statement

The wait-for-time statement causes the sequence to wait for a certain time specified in an HVI register. Once the time is elapsed, the sequence continues.

The following code is an example of a wait-for-time statement:

```
# Wait Time makes the HVI sequence wait for an amount of time specified by
# a register (register 'tau' in this example)
#
waitTau = sync_block.sequences["digitizer_engine"].add_wait_time("WaitTau", 10, tau)
```

Local delay statement

The Local delay statement delays the execution of a local sequence for a time you specify. The default unit is nanoseconds but the delay is specified in any unit of seconds. The delay is fixed and cannot be changed during HVI execution, so the delay value must be known at the time of creating the HVI sequence.

The delay statement works in a similar way as the start delay statement parameter. The difference is that the start delay can only be specified before the other statements in a sequence. The delay statement enables you to place a fixed delay at the end of Sync multi-sequence block or a flow control statement.

If you require a Variable delay that can be changed during HVI execution, use the Local wait-for-time statement.

The following code shows an example of a Local delay statement:

```
# Delay makes the HVI sequence wait for an amount of time specified by a constant
#
wait = sync_block.sequences["digitizer_engine"].add_delay("Delay", 30)
```

Chapter 8: Building an Application with the HVI API

This chapter describes the steps you must follow to use the HVI API. If you do not follow these steps your application shall not work correctly.

HVI uses *program-within-a-program* model. That is, the HVI enables you to define a program that runs on the instrument's hardware while the software programs run in parallel and can interact with the instruments. HVI is also responsible for all the setup, compilation, and hardware execution management. When you run your application, it generates an HVI instance and the sequences within it are executed on the instruments.

This chapter contains the following sections:

NOTE The code examples provided in this chapter are in both Python and C#.

Planning an HVI with the HVI Use Model

Programming and executing an HVI requires you to follow a precise use model. You must write your code in the correct order and be aware of the requirements of each stage, or your application shall not work correctly.

The HVI Use Model consists of 3 broad stages:

1. System Definition

You must define hardware resources before you can use them in HVI. The resources you can use depends on your hardware set up, what instruments you have, what capabilities they have, and how they are arranged. You set these up first and then you can use their functionalities in your HVI sequences. This operation is called *System Definition* and it can be done by using an instance of the SystemDefinition class.

The initialization of the system you have defined is also important to understand. By default, the defined system is initialized at the code line that is creating the `Sequencer` object from the `SystemDefinition` object. If you use the default initialization, this ensures that the complete system is correctly initialized.

There are some use cases when you might need to use the `initialize()` API method to perform a custom initialization, for example, a full realignment of the HVI Engine clocks. For more information, see the description of *AlignmentMode* list in System Initialization and the Python API Help. In this case it is important to make sure that the `SystemDefinition` object is not modified after calling the `initialize()` API method.

NOTE Ensure you initialize your system after all the resources have been added and defined. If you call the `initialize()` API method before the system is fully defined, the system shall not be initialized properly. Consequences of an improper initialization might be that some instruments included in the HVI might be out of sync or that their HVI sequence execution will misbehave by for example, missing a trigger or not playing a waveform.

For example, in the following code `initialize()` is called incorrectly before all the engines are added to the `SystemDefinition`.

```
# call initialize()
#
sys_def.initialize()
#
# Incorrect usage, Engines added after the initialize() call are not initialized.
sys_def.engines.add(...)
```

2. Program HVI Sequences

As a next step, the `SystemDefinition` object is passed into the `Sequencer` object at the sequencer creation.

Once the `Sequencer` object is created, the `SystemDefinition` instance is fixed. All resources added and defined using the `SystemDefinition` object must be modified before this step. You cannot make any changes to the `SystemDefinition` instance after this. Any changes made in the `SystemDefinition` after this point are not passed into the `Sequencer` object and therefore are not included in the HVI.

Once the hardware is set up and resources assigned, you can write your sequences and set initialization values. You create Sync sequences for globally synchronized operations, and you create Local sequences for operations in the HVI engines in individual instruments.

3. Execute the HVI

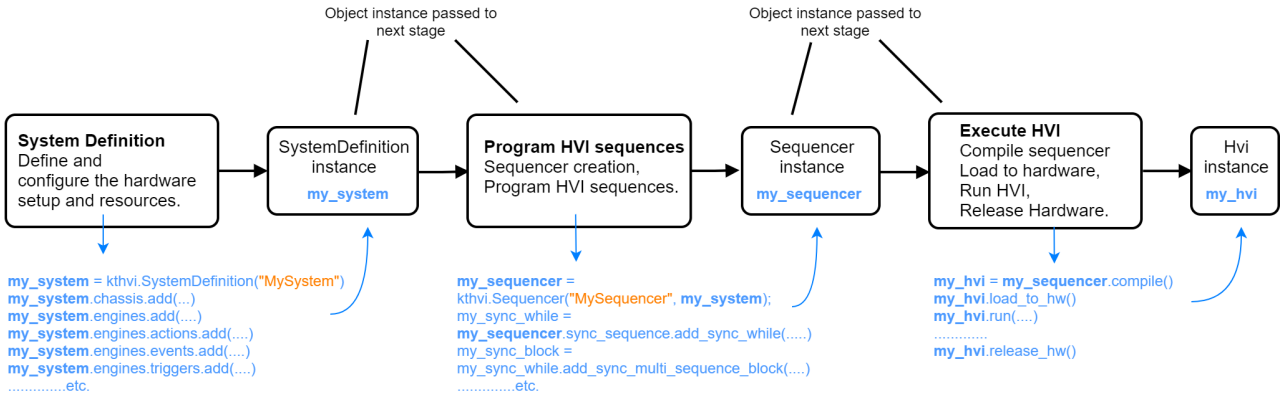
When you have programmed your sequences, you call the compile software method to create an instance of the `Hvi` class from the `Sequencer` object instance containing the information about the programmed sequences.

After a successful sequencer compilation, t he sequencer configuration is passed into the Hvi object when it is created. Once the Hvi object is created, the Sequencer instance is fixed. You cannot make any changes to the Sequencer or SystemDefinition instances after this point.

The compilation generates binary files that can be loaded to hardware and execute your HVI. Before running the HVI, you can redefine the initial values of some of the resources that are included in the HVI, such as HVI registers for different engines.

Object Instances in the HVI Use Model

The following diagram shows the stages and highlights how each of the stages in the HVI use model creates and uses an object instance which is then passed to the next stage:



NOTE Once an instance of SystemDefinition, Sequencer, or Hvi classes is created and configured, you cannot modify it in the next HVI step. If you attempt to modify one of these instances at a later stage, the modifications will not apply to your HVI. That is:

- You shall not modify the SystemDefinition object at the "Program HVI Sequences" or "Execute HVI" stage.
- You also shall not modify the SystemDefinition or Sequencer instances at the "Execute HVI" stage.

Correct Example

In the following example the value `non_hvi_core_clocks` in `SystemDefintion` is set.

This is set before the Sequencer is created so this is the correct place to do this.

```
# Define System Definition
my_system = kthvi.SystemDefinition("MySystem");
#
# Set value of non_hvi_core_clocks (in Hz)
sys_def.non_hvi_core_clocks = [10e6]
#
# Create the sequencer
sequencer = kthvi.Sequencer("MySequencer", my_system); .......
#
# Get the Hvi
hvi = sequencer.compile().
```

Incorrect Example

In the following example the value `non_hvi_core_clocks` in `SystemDefintion` is set. In this case the value is set *after* the Sequencer is defined.

This example will not work because you cannot change a value in `SystemDefintion` after you have created the sequencer.

```
# Define System Definition
my_system = kthvi.SystemDefinition("MySystem");
#
# Create the sequencer
sequencer = kthvi.Sequencer("MySequencer", my_system); .......
#
# Set value of non_hvi_core_clocks (in Hz)
sys_def.non_hvi_core_clocks =[10e6] # THIS FAILS
#
# Get the Hvi
hvi = sequencer.compile().
```

NOTE If you need to make a change to SystemDefintion object after creating the sequencer, you must create a new Sequencer for the change to have an effect.

1 Create the System Definition

Setting up the HVI requires several steps:

- Include the HVI library in your application.
- Define the hardware in your HVI.
- Define and configure HVI resources.
- Define FPGA sandbox resources.

Include the HVI Library in your Application

Include the HVI library in your application:

Python code:

```
import keysight_hvi as kthvi
```

C# code:

```
using Keysight.Hvi;
```

You must first create an instance of a `SystemDefinition` object.

Python code:

```
# Create SystemDefinition instance
my_system = keysight_hvi.SystemDefinition("Multi-chassis Setup")
```

C# code:

```
// Create SystemDefinition instance
var sysDef = new SystemDefinition("My System");
```

When you have done this, specify the hardware and hardware resources that you require in your HVI:

- Define the hardware in your HVI.
- Define the HVI resources.
- Register the resources with relevant collections.
- Initialize HVI hardware resources for the HVI.

Define the Hardware in your HVI

Add the hardware resources in your system to the `SystemDefinition` object, including:

- Chassis.
- Chassis interconnections.
- PXI trigger synchronization resources.
- Synchronization clocks.

Define the chassis

Python code:

```
# Add chassis with number or options
my_system.chassis.add(chassis_number)
my_system.chassis.add_with_options(chassis_number, "DriverSetup=model=GenericPxieChassis")
```

C# code:

```
// Add chassis with number or options
sysDef.Chassis.Add(1);
sysDef.Chassis.AddWithOptions(1, "Simulate=True,DriverSetup=model=GenericPxieChassis");
```

Define the chassis interconnects

You must first define the SystemSync modules. The options specify a number of parameters about each module:

Python code:

```
# Define SystemSync Modules
ssm_m9032_resource_id_ssm_1 = 'PXI0::CHASSIS1::SLOT10::INSTR'
ssm_m9033_resource_id_ssm_2 = 'PXI0::CHASSIS2::SLOT10::INSTR'
ssm_m9032_options = "Simulate=true,DriverSetup=Model=M9032A, HviEngineIpVersion=1.1.0,
HviGatewareFeatureVersion=2,model=M9032"ssm_m9033_
options = "Simulate=true,DriverSetup=Model=M9033A, HviEngineIpVersion=1.1.0,
HviGatewareFeatureVersion=2,model=M9033"system_sync_modules_descriptors =
[SystemSyncModuleDescriptor('PXI0::CHASSIS1::SLOT10::INDEX0::INSTR', ssm_options)]
```

C# code:

```
// Define SystemSync Modules
public static string Ssm9032Options { get; set; } = "Simulate=true,DriverSetup=Model=M9032A,
HviEngineIpVersion=1.1.0, HviGatewareFeatureVersion=2,model=M9032"public static string
Ssm9033Options { get; set; } = "Simulate=true,DriverSetup=Model=M9033A,
HviEngineIpVersion=1.1.0, HviGatewareFeatureVersion=2,model=M9033"public
List<SystemSyncModuleDescriptor> SystemSyncModulesDescriptors { get; set; } = new
List<SystemSyncModuleDescriptor>
{
    new SystemSyncModuleDescriptor("PXI0::CHASSIS1::SLOT10::INDEX0::INSTR", Ssm9032Options),
    new SystemSyncModuleDescriptor("PXI0::CHASSIS2::SLOT10::INDEX0::INSTR", Ssm9033Options),
};
```

You must add the modules to the interconnects collection within the system definition:

Python code:

```
# Add SystemSync Modules to chassis
ssm_m9032 = my_system.interconnects.add_sync_module(ssm_m9032_resource_id, ssm_m9032_options)
ssm_m9033 = my_system.interconnects.add_sync_module(ssm_m9033_resource_id, ssm_m9033_options)
```

C# code:

```
// Add SystemSync Modules to chassis
ssmList.Add(interconnects.AddSyncModule(descriptor.ResourceId, descriptor.Options));
```

Once done, get the interface objects for each of the SSM connectors:

NOTE The items in the collection `systemsync_downstream` are indexed from 0.

Python code:

```
# Get the 8x SystemSync downstream connector on first SSM
ssm_m9032_down = ssm_m9032.connectivity.systemsync_downstream[0]

# Get the 8x SystemSync upstream connector on second SSM
ssm_m9033_up = ssm_m9033.connectivity.systemsync_upstream[0]
```

C# code:

```
// Get the 8x SystemSync downstream connector on first SSM
ssm1Down = ssm1.Connectivity.SystemsyncDownstream[0]

// Get the 8x SystemSync upstream connector on second SSM
ssm2Up = ssm2.Connectivity.SystemsyncUpstream[0]
```

Set the connection between the connectors. This tells the HVI that these connections are connected together.

Python code:

```
# Set the connection
ssm_m9032_down.set_connection(ssm_m9033_up)
```

C# code:

```
// Set the connection.
ssm1.Connectivity.SystemSyncDownstream[0].SetConnection(ssm2.Connectivity.SystemSyncUpstream
[0]);
```

Define the synchronization resources

Python code:

```
# Define sync resources
my_system.sync_resources = [keysight_hvi.TriggerResourceId.PXI_TRIGGER0,
                            keysight_hvi.TriggerResourceId.PXI_TRIGGER1,
                            keysight_hvi.TriggerResourceId.PXI_TRIGGER2]
```

C# code:

```
// Define sync resources
TriggerResourceId[] resources = {
                           TriggerResourceId.PxiTrigger0,
                           TriggerResourceId.PxiTrigger1,
                           TriggerResourceId.PxiTrigger2};
```

Define the clocks

In simple setups this is only required when dealing with instruments that do not support HVI technology, or *Devices Under Test* that have specific clocking requirements.

For more complex setups see the *System Setup Guide*.

Python code:

```
# clocks configuration
my_system.non_hvi_core_clocks = [100MHz]
my_system.non_hvi_system_clocks = [500MHz]
```

C# code:

```
// clocks configuration
sysDef.NonHviCoreClocks = {100};
sysDef.NonHviCoreClocks = {500};
```

Define and Configure HVI Resources

Triggers, Actions, and Events are all HVI resources that can be used by the HVI engine and the HVI sequence to interact with the outside world, that is, with other instruments, the instrument sandbox, or any other external entities.

You must define the resources you are going to use and register them with collections for the engines you want to use them with. You must do this for the following types of resources:

- HVI Engines.
- Actions.
- Events.
- Triggers.
- FPGA Sandbox resources.

Define HVI Engines

First, you must define the engines you want to use and add them to an engine collection. The method `add_engine()` returns an engine.

Python code:

```
# Add engines
engine0 = my_system.engines.add(module.hvi.engines.main_engine, "Receiver")
engine1 = my_system.engines.add(module.hvi.engines.main_engine, "Transmitter")
```

C# code:

```
// Add Engines
sysDef.Engines.Add(module.Hvi.Engines.MainEngine, "Receiver");
sysDef.Engines.Add(module.Hvi.Engines.MainEngine, "Transmitter");
```

The procedure for defining and registering the other HVI resources follows the same pattern. As a first step, the resource must be added to the corresponding collection using the method `add()` within the classes `TriggerCollection` , `ActionColletion` , `EventCollection` , etc.

For example, to define and register an event, do the following:

There is an event collection for each engine. Get the event collection with the property `engine.events` . This returns the `EventCollection` object. Add the events you want to use to the event collection with the `add()` method of `EventCollection` . To add each event you must specify both an `event id` and an `event name:`

Python code:

```
my_event = engine.events.add(module.hvi.events.PXI0, "My Event")
```

C# code:

```
myEvent = Engine.Events.Add(module.Hvi.Events.Pxi0, "My Event")
```

Actions, Triggers, and FpgaSandboxes all require their own collection classes, for example `ActionCollection` is for Actions.

Use the same procedure to get collections and add Actions, Triggers, and FpgaSandboxes to their respective collections. The ID of engines, actions, events, and triggers related to a specific instrument are defined by the instrument API, typically within the `instrument.hvi` interface of an `instrument` object.

Define HVI actions

The following code example defines all HVI actions necessary to perform AWG (*Arbitrary Waveform Generator*) trigger operations. The AWG trigger actions for each AWG channel is defined and registered into the `ActionCollection` of the AWG engine that needs to execute them in its local sequence.

Python code:

```
# Define AWG trigger actions for all AWG channels
for ch_index in range(1, num_channels + 1):
# Actions need to be added to the engine's action list so that they can be executed
action_name = "AWG Trigger CH" + str(ch_index)        # arbitrary user-defined name
instrument_action = "awg{}_trigger".format(ch_index)  # name decided by instrument API
action_id = getattr(instrument.hvi.Actions, instrument_action)
my_system.engines[awg_engine_name].actions.add(action_id, action_name)
```

C# code:

```
// Define AWG trigger actions for 4 AWG channels
// Actions must be added to the engine's action list so that they can be executed
//
mySystem.Engines[engineName].Actions.Add(module.Hvi.Actions.Awg0Trigger, "awg0trigger")
mySystem.Engines[engineName].Actions.Add(module.Hvi.Actions.Awg1Trigger, "awg1trigger")
mySystem.Engines[engineName].Actions.Add(module.Hvi.Actions.Awg2Trigger, "awg2trigger")
mySystem.Engines[engineName].Actions.Add(module.Hvi.Actions.Awg3Trigger, "awg3trigger")
```

Define HVI events

The code example below adds the AWG CH1 Waveform Start event to the event collection of an M320xA AWG's HVI engine object called `awg_engine`. For further information on M320xA events see SD1 3.x Software for M320xA / M330xA Arbitrary Waveform Generators User's Guide available at M3201A PXIe Arbitrary Waveform Generator.

Python code:

```
wfm_start_event = awg_engine.events.add(instrument.hvi.events.awg1_waveform_start, "AWG CH1 Wfm
Start Event")
```

C# code:

```
// adding wait for trigger event
wfmStartEvent = awgEngine.Events.Add(instrument.Hvi.Events.Awg1WaveformStart, "AWG CH1 Wfm Start
Event")
```

Define HVI triggers

The code example below defines a *Front Panel* (FP) trigger to be used by a digitizer instrument. The `TriggerCollection` is accessed through the `dig_engine.triggers` interface, where `dig_engine` is an HVI Engine object.

Python code:

```
# Defines the FP trigger to be used as a wait condition by the digitizer
# Add to the HVI Trigger Collection of each HVI Engine the FP Trigger object of that same
instrument
#
fp_trigger_id = instrument.hvi.triggers.front_panel_1
fp_trigger = dig_engine.triggers.add(fp_trigger_id, "FP Trigger")
#
# Trigger configuration
# NOTE: Trigger to be used as WaitEvent conditions must be configured as kthvi.Direction.INPUT
# DriveMode (e.g. PushPull/OpenDrain) is defined by the instrument and cannot be changed by the
user
fp_trigger.config.direction = kthvi.Direction.INPUT
fp_trigger.config.polarity = kthvi.Polarity.ACTIVE_HIGH
fp_trigger.config.hw_routing_delay = 0
fp_trigger.config.trigger_mode = kthvi.TriggerMode.LEVEL
```

C# code:

```
// Defines the FP trigger to be used as a wait condition by the digitizer
// Add to the HVI Trigger Collection of each HVI Engine the FP Trigger object of that same
instrument
//
fpTriggerId = instrument.Hvi.Triggers.frontPanel1;
fpTrigger = digEngine.Triggers.Add(fpTriggerId, "FP Trigger");
//
// Trigger configuration
// NOTE: Trigger to be used as WaitEvent conditions must be configured as Direction.Input
// DriveMode (e.g. PushPull/OpenDrain) is defined by the instrument and cannot be changed by the
user
fpTrigger.Config.Direction = Direction.Input;
fpTrigger.Config.Polarity = Polarity.ActiveHigh;
fpTrigger.Config.HwRoutingDelay = 0;
fpTrigger.Config.TriggerMode = TriggerMode.Level;
```

Define FPGA sandbox resources

The `SandboxCollection` is accessible through the `engine.fpga_sandboxes` interface of an engine object. Unlike other HVI collections, this collection is already populated by a number of sandboxes where the number of sandboxes depends on the instrument being used. Most instruments have a single sandbox region in their FPGA, but some instruments might have multiple sandbox regions. Sandbox objects do not need to be added to the collection, you only need to access them.

Python code:

```
# 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
sandbox_name = 'sandbox0'
awg_sandbox = awg_engine.fpga_sandboxes[sandbox_name]
```

C# code:

```
// 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
sandboxName = "sandbox0";
awgSandbox = AwgEngine.FpgaSandboxes[sandboxName];
```

2. Program HVI Sequences

Programming HVI sequences requires a number of steps:

- Create a Sequencer object
- Define HVI Registers and initialize register values
- Start with the global SyncSequence
- Adding Sync Statements and Sync Sequences
- Adding Local Statements
- Adding HVI instructions
- Adding Instrument Specific Instructions
- Using Triggers, Actions, and Events
- Using FPGA Sandbox Resources

Create a Sequencer Object

Before you can begin writing sequences, you must create a `Sequencer` object and pass the `SystemDefinition` to the `Sequencer` object:

Python code:

```
sequencer = keysight_hvi.Sequencer("sequencer", my_system)
```

C# code:

```
Sequencer seq = new Sequencer("sequencer", sysDef);
```

Define HVI Registers and Initialize Register Values

Define the HVI registers resource you require in each engine and use the `add()` method to add them to the register collection for that engine. Then define their initial values:

Python code:

```
loop_register = sequencer.sync_sequence.scopes["Engine 1"].registers.add("Loop Register",
keysight_hvi.RegisterSize.SHORT)
loop_register.initial_value = 0
```

C# code:

```
var loopRegister = sequencer.SyncSequence.Scopes["Engine 1"].Registers.Add("Loop Register",
RegisterSize.SHORT);
loopRegister.InitialValue = 0;
```

The registers that you to use in the HVI sequences must be defined beforehand in 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 hardware registers of that specific instrument. Registers from one HVI engine cannot be used by other engines or outside of their scope. Registers can only be added to the HVI top Sync sequence scopes. This means that you can only add global registers that are visible in all child sequences. The number and size of registers is defined by each instrument.

To reserve a register resource:

1. Get the register collection from the engine you want to reserve the register on.
2. Add the registers you require. Use the `add()` method to the register collection for that engine

NOTE Register size is defined by the following:

- SHORT = 32 bit
- LONG = 48 bit

Create Sequences

After you have got the `Sequencer` object and set up the registers you require, you can write the program the HVI executes, this is composed of:

- Sequences.
- Statements.
- Instructions.
- Time restrictions.

To define your program, you must:

- Create sequences.
- Add statements and instructions.

Start with the Global SyncSequence

When HVI starts execution, it starts in a global sequence `SyncSequence`, this is defined by the `Sequencer` object. This is used in the previous example when the registers were reserved:

Python code:

```
engine_1_registers = sequencer.sync_sequence.scopes["Engine 1"].registers
```

C# code:

```
var engine1Registers = seq.SyncSequence.Scopes[engine1Name].Registers;
```

Adding Sync Statements and Sync Sequences

You add Sync statements to the `SyncSequence` class with *add_statement* methods such as `SyncSequence.add_sync_while()`:

Python code:

```
# Create Sync While statement (loop_register < SYNC_WHILE_LOOP_ITERATIONS):
SYNC_WHILE_LOOP_ITERATIONS = 5
sync_while_condition = keysight_hvi.Condition.register_comparison( engine_1_registers["loop_
register"], keysight_hvi.ComparisonOperator.LESS_THAN, SYNC_WHILE_LOOP_ITERATIONS)
sync_while = sequencer.sync_sequence.add_sync_while("sync_while", 100, sync_while_condition)
```

C# code:

```
// create Sync While statement (loop_register < SYNC_WHILE_LOOP_ITERATIONS)
var syncWhileCondition = Condition.RegisterComparison(
engine1Registers["loop_register"], ComparisonOperator.LessThan, SYNC_WHILE_LOOP_ITERATIONS);
var syncWhile = seq.SyncSequence.AddSyncWhile("sync_while", 100, syncWhileCondition);
```

You can also add Sync sequences within the global Sync sequence and add Sync statements within the Sync sequences.

Adding Local Statements

To add local instructions or local flow-control operations, you must add them within a Sync multi-sequence block. You must add this Sync multi-sequence block within a Sync Sequence by using the `add_sync_multi_sequence_block()` method:

Python code:

```
# Add a sync multi-sequence block:
multi_seq_block_1 = sync_while.sync_sequence.add_sync_multi_sequence_block("multi_seq_block_1",
210)
```

C# code:

```
// Add a sync multi-sequence block
var multiSeqBlock1 = syncWhile.SyncSequence.AddSyncMultiSequenceBlock("multiSeqBlock1", 220);
```

To add the local statements, you must get a `Sequence` object for each engine in the Sync multi-sequence block and add them using the corresponding add_*XXX*() method. Local instructions can be added to a Sync multi-sequence block using the `add_instruction()` method. For each instruction parameter, use the `set_parameter()` method to set it.

By adding Local statements to the sequences, you define the Local sequence that each local engine executes in parallel with the other engines.

Adding HVI Instructions

There are two types of HVI instructions:

- HVI-native instructions.
- Instrument specific instructions.

HVI-native instructions

The `InstructionSet` class contains the set of native instructions that can be executed within an HVI statement, including:

- Register arithmetic.
  - Add / Subtract.
  - Assign.
- Read/write I/O trigger ports.
- Communications operations with the instrument sandbox using an HVI Host Interface.
  - FPGA register read/write.
  - FPGA array read/write.
- Action execute.
- Trigger write.

To use the HVI-native instructions, you must use the `InstructionSet` class. You get this from the local `Sequence` class:

Python code:

```
# Initialize loop_register
loop_reg = multi_seq_block.scope.registers["loop_register"]
awg_sequence = multi_seq_block.sequences["AWG Engine"]
instruction_a = multi_seq_block.add_instruction("loop_register = 0", 10, awg_
sequence.instruction_set.assign.id)
instruction_a.set_parameter(awg_sequence.instruction_set.assign.destination.id, loop_reg)
instruction_a.set_parameter(awg_sequence.instruction_set.assign.source.id, 0)
#
# Increment pulse_counter
pulse_counter = multi_seq_block_1.scope.registers["pulse_counter"]
instruction = multi_seq_block_1.add_instruction("Increment Pulse Counter", 10, awg_
sequence.instruction_set.add.id)
instruction.set_parameter(awg_sequence.instruction_set.add.destination.id, pulse_counter)
instruction.set_parameter(awg_sequence.instruction_set.add.left_operand.id, pulse_counter)
instruction.set_parameter(awg_sequence.instruction_set.add.right_operand.id, 1)
```

C# code:

```
// Initialize loop_register
var reg = sequence.Scope.Registers[registerName];
```

```
var instructionA = sequence.AddInstruction(registerName + "_assign", startDelay,
sequence.InstructionSet.Assign.Id);
instructionA.SetParameter(sequence.InstructionSet.Assign.Value.Id, value);
instructionA.SetParameter(sequence.InstructionSet.Assign.Destination.Id, reg);
//
// Increment register by 1
private void incrementRegisterBy1(ISequence sequence, string registerName, int startDelay)
{
   var reg = sequence.Scope.Registers[registerName];
   var instructionA = sequence.AddInstruction("Increment Pulse Counter",
   startDelay, sequence.InstructionSet.Add.Id);
   instructionA.SetParameter(sequence.InstructionSet.Add.LeftOperand.Id, reg);
   instructionA.SetParameter(sequence.InstructionSet.Add.RightOperand.Id, 1);
   instructionA.SetParameter(sequence.InstructionSet.Add.Destination.Id, reg);
}
```

Instrument specific instructions

Instrument specific instructions are described in the documentation for the instrument. For example, the following code shows how to set a channel amplitude value:

Python code:

```
# Set CH1 amplitude to 1.0 V:
instruction = multi_seq_block_1.add_instruction("Set CH1 amplitude to 1.0 V", 10,
instrument.hvi.instruction_set.set_amplitude.id)
instruction.set_parameter(instrument.hvi.instruction_set.set_amplitude.channel.id, ch1)
instruction.set_parameter(instrument.hvi.instruction_set.set_amplitude.value.id, 1.0)
```

C# code:

```
// Set CH1 amplitude to 1.0 V
instruction = multiSeqBlock1.AddInstruction("Set CH1 amplitude to 1.0 V", 10,
instrument.Hvi.InstructionSet.SetAmplitude.id);
instruction.SetParameter(instrument.Hvi.InstructionSet.SetAmplitude.Channel.id, ch1);
instruction.SetParameter(instrument.Hvi.InstructionSet.SetAmplitude.Value.id, 1.0);
```

Using Triggers, Actions, and Events

The examples below provide an overview of how to use triggers, actions and events within an HVI sequence.

Using Triggers

There are two typical use cases of trigger objects (previously defined by the user during system definition). The first one is the usage of the trigger object as a wait condition inside a Wait statement:

Python code:

```
# Add a wait statement that has a FP trigger as a condition
fp_trigger = awg_engine.triggers["fp_trigger"]
wait_condition = keysight_hvi.Condition.trigger(fp_trigger)
wait_event = awg_sequence.add_wait("wait for fp trigger", 10, wait_condition)
wait_event.set_mode(kthvi.WaitMode.TRANSITION, kthvi.SyncMode.IMMEDIATE)
```

C# code:

```
// Add a wait statement that has a FP trigger as a condition
fpTrigger = awgEngine.Triggers["fpTrigger"];
waitCondition = Condition.Trigger(fpTrigger);
waitEvent = awgSequence.AddWait("wait for trigger", 10, waitCondition);
waitEvent.SetMode(WaitMode.Transition, SyncMode.Immediate);
```

The second use case involves the `TriggerWrite` HVI Native instruction, where the trigger object can be used to specify which electrical trigger line can be written from the HVI sequence:

Python code:

```
# Write FP Trigger to ON value
fp_trigger = awg_engine.triggers["fp_trigger"]
trigger_write_instr = sequence.instruction_set.trigger_write
instr_trigger_ON = sequence.add_instruction("FP Trigger ON", 10, trigger_write_instr.id)
instr_trigger_ON.set_parameter(trigger_write_instr.sync_mode.id, trigger_write_instr.sync_
mode.immediate)
instr_trigger_ON.set_parameter(trigger_write_instr.trigger.id, fp_trigger)
instr_trigger_ON.set_parameter(trigger_write_instr.value.id, trigger_write_instr.value.on)
```

C# code:

```
// Write FP Trigger to ON value
var tw = sequence.InstructionSet.TriggerWrite;
var instOn = sequence.AddInstruction("Trigger On", 20, tw.Id);
instOn.SetParameter(tw.Trigger.Id, trigger);
instOn.SetParameter(tw.SyncMode.Id, tw.SyncMode.Immediate);
instOn.SetParameter(tw.Value.Id, tw.Value.On);
```

Using Actions

User-defined actions can be executed using the HVI native instruction `ActionExecute`. A list of actions `action_list`, can be executed simultaneously within the same instruction. The `action_list` object must have been be previously defined.

Python code:

```
# "Action Execute" instruction executes the AWG trigger from HVI
instruction = sequence.add_instruction("AWG trigger", 10, sequence.instruction_set.action_
execute.id)
instruction.set_parameter(sequence.instruction_set.action_execute.action.id, action_list)
```

C# code:

```
//  "ActionExecute" instruction executes the AWG trigger from HVI
var actionArray = sequence.Engine.Actions.ToArray();
instruction = sequence.AddInstruction("AWG trigger", 10,
sequence.InstructionSet.ActionExecute.id);
instruction.SetParameter(sequence.InstructionSet.ActionExecute.Action.id, actionArray);
```

Using Events

The typical use case of events within HVI sequences is as a condition for a Wait Statement:

Python code:

```
# Add a wait statement that waits for AWG CH1 queue to be empty
awg_queue_empty = awg_engine.events["Awg1QueueIsEmpty"]
wait_condition = keysight_hvi.Condition.event(awg_queue_empty)
wait_event = awg_sequence.add_wait("Wait for AWG Queue to be Empty", 10, wait_condition)
wait_event.set_mode(kthvi.WaitMode.TRANSITION, kthvi.SyncMode.IMMEDIATE)
```

C# code:

```
// adding wait for trigger
var waitTrigger = sequence.Engine.Triggers["wait_trigger"];
var waitEvent = sequence.AddWait("wait for trigger", 10, Condition.Trigger(waitTrigger));
waitEvent.SetMode(WaitMode.Transition, SyncMode.Immediate);
```

Using FPGA Sandbox Resources

To use FPGA Resources, the sandbox must be loaded using the `load_from_k7z()` method specifying the path containing the **`.k7z`** file produced compiling a project designed using PathWave FPGA. For more information see the PathWave FPGA User Manual at PathWave FPGA . Once the sandbox is loaded, all the HVI registers and memory maps that were inserted in the specified PathWave FPGA project file can be accessed to be used in the FPGA sequence. Please note that the same names used in the PathWave FPGA project must be used to access the FPGA resources. In the following example, the register name *Register_Bank_MyCounter* is not arbitrary but assumed to be taken from the PathWave FPGA project that generated the file **`MySandboxProject.k7z`** :

Python code:

```
sandbox = engine.fpga_sandboxes["sandbox0"]
sandbox.load_from_k7z("MySandboxProject.k7z")
counter_register = sandbox.fpga_registers["Register_Bank_MyCounter"]
```

C# code:

```
sandbox = Engine.FpgaSandboxes["sandbox0"];
sandbox.LoadFromk7z("MySandboxProject.k7z");
counterRegister = sandbox.FpgaRegisters["registerBankMyCounter"];
```

Write to FPGA resources

The following example shows how to write to an FPGA register and read an FPGA array. The process in both cases is very similar:

Python code:

```
# Write FPGA register
fpga_register = engine.fpga_sandboxes[sandbox_name].fpga_registers[register_name]
fpga_regw_instruction = sequence.instruction_set.fpga_register_write
fpga_register_write = sequence.add_instruction("my_fpga_register_write", 10, fpga_regw_
instruction.id)
fpga_register_write.set_parameter(fpga_regw_instruction.fpga_register.id, fpga_register)
fpga_register_write.set_parameter(fpga_regw_instruction.value.id, value_register)
#
# Read FPGA array
memory_map = engine.fpga_sandboxes[sandbox_name].fpga_memory_maps[0]
fpga_arrayr_instr = sequence.instruction_set.fpga_array_read
fpga_array_read = sequence.add_instruction("my_fpga_array_read", time_ns, fpga_arrayr_instr.id)
fpga_array_read.set_parameter(fpga_arrayr_instr.fpga_memory_map.id, memory_map)
fpga_array_read.set_parameter(fpga_arrayr_instr.fpga_memory_map_offset.id, loop_reg)
fpga_array_read.set_parameter(fpga_arrayr_instr.value.id, value_register))
```

C# code:

```
// Write FPGA register
fpga_register = engine.fpga_sandboxes[sandbox_name].fpga_registers[register_name];
fpga_regw_instruction = sequence.instruction_set.fpga_register_write;
fpga_register_write = sequence.add_instruction("my_fpga_register_write", 10, fpga_regw_
instruction.id);
fpga_register_write.set_parameter(fpga_regw_instruction.fpga_register.id, fpga_register);
fpga_register_write.set_parameter(fpga_regw_instruction.value.id, value_register);
//
// Read FPGA array
memoryMap = Engine.fpgaSandboxes[sandbox_name].fpgaMemoryMaps[0];
fpgaArrayrInstr = sequence.InstructionSet.FpgaArrayRead;
fpgaArrayRead = sequence.AddInstruction("myFpgaArrayRead", timeNs, fpgaArrayrInstr.id);
fpgaArrayRead.SetParameter(fpgaArrayrInstr.FpgaMemoryMap.id, memoryMap);
fpgaArrayRead.SetParameter(fpgaArrayrInstr.FpgaMemoryMapOffset.id, loopReg);
fpgaArrayRead.SetParameter(fpgaArrayrInstr.Value.id, valueRegister));
```

3. Compile Your Sequences

After writing the Sequences, you must add the command that compiles the HVI. Call the `compile()` method in the `Sequencer` object to perform the compilation operation. The `compile()` method returns the HVI instance `Hvi`.

Python code:

```
# Compile HVI sequences:
try:
    hvi = sequencer.compile()
       print('HVI Compiled')
except keysight_hvi.CompilationFailed as err:
       print(err.compile_status.to_string())
       raise err
```

C# code:

```
// Compile HVI sequences:
try
{
     hvi = sequencer.Compile();
     Console.WriteLine("compile DONE");
}
catch (CompilationFailed err)
{
     Console.WriteLine(err.CompileStatus.ToString());
     throw err;
}
```

NOTE At this point you can no longer modify sequences, actions, events or triggers.

The property `hvi.sync_resources` provides information about the PXI sync resources you must reserve.

Python code:

```
print("This needs to reserve {} PXI trigger resources to execute".format(len(hvi.sync_
resources)))
```

C# code:

```
Console.WriteLine("This needs to reserve {} PXI trigger resources to execute".Format(len
(Hvi.SyncResources)));
```

If the compilation fails, the object `keysight_hvi.CompilationFailed` is thrown. This contains compilation error messages that you can print.

4. Load To Hardware

Before your compiled sequences can be executed, they must be uploaded into the HVI engines in the instrument hardware. To upload the compiled sequences, you must use the `Hvi` method `load_to_hw()`.

Python code:

```
# Load HVI to hardware:
Hvi.load_to_hw()
print("HVI Loaded to hardware")
```

C# code:

```
// Load HVI to hardware:
Hvi.LoadToHw();
Console.WriteLine("load DONE");
```

5. Modify Initial Register Values (Optional)

The HVI execution can be parameterized using registers, the initial values of all registers are updated when the `run()` method in `Hvi` is called. To modify the initial value of the registers in the HVI object, use:

Python code:

```
# Modify register initial value
value = 10
register_runtime = hvi.sync_sequence.scopes[0].registers[loop_register.name]
register_runtime.initial_value = value
```

C# code:

```
// Modify register initial value
var value = 10;
registerRuntime = Hvi.SyncSequence.Scopes[0].registers[loopRegister.name];
registerRuntime.initialValue = value;
```

Once the instrument has been loaded to hardware, you can write to the FPGA memory map.

Python code:

```
memory_map.write(0, 1)
memory_map.write(1, 2)
memory_map.write(2, 3)
```

C# code:

```
memoryMap.Write(0, 1);
memoryMap.Write(1, 2);
memoryMap.Write(2, 3);
```

6. Execute Sequences

To execute the binaries, call the `run()` method in `Hvi`. The HVI can be run in a blocking or non-blocking mode:

Blocking mode

In blocking mode, the execution is blocked at the HVI execution code line for a fixed amount of time specified by the timeout input parameter. If `timeout = hvi.no_timeout` is used as an input parameter, the execution can be blocked until the HVI sequences finish their execution.

Python code:

```
hvi.run(hvi.no_timeout)
```

C# code:

```
hvi.Run(System.TimeSpan.FromSeconds(10));
```

Non-blocking mode

In non-blocking mode, the execution is not blocked. This enables you to initiate a second HVI instance to run in parallel.

Python code:

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

C# code:

```
// Execute HVI in non-blocking mode
// This mode allows SW execution to interact with HVI execution:
hvi.Run(IHvi.no_wait);
Console.WriteLine("HVI Running...");
```

While and after execution is finished, you can read or write registers and execute the binaries again.

Python code:

```
# Modify register initial value
value = 20
register_runtime = hvi.sync_sequence.scopes[0].registers[loop_register.name]
register_runtime.initial_value = value
hvi.run(hvi.no_timeout)
```

C# code:

```
// Modify register initial value
ver value = 20;
registerRuntime = hvi.SyncSequence.Scopes[0].Registers[loopRegister.name];
registerRuntime.initialValue = value;
hvi.Run(IHvi.NoTimeout);
```

7. Release All Resources

To release all HVI resources and enable other applications or HVI instances to use the hardware, you must release the hardware. Your application cannot perform any operation with the hardware resources in the HVI after this point.

Python code:

```
# Unlock and release hardware resources:
hvi.release_hw()
print("Releasing Hardware...")
```

C# code:

```
// Unlock and release hardware resources:
hvi.ReleaseHw();
Console.WriteLine("Releasing Hardware...");
```

Chapter 9: HVI Time Management and Latency

This chapter describes HVI time management and latency. It introduces the concepts involved and describes the timing and latencies of statement execution, how they impact the overall execution timing of sequences, and the constraints on the start delay and duration of statements. It also provides latency information for the different statements and instructions.
This chapter contains the following sections:

About Time Management and Latency Concepts

This section introduces the main concepts, and additional parameters and values involved in HVI time management. It includes the following sections:

- Timing Concepts Overview.
- Additional Timing Concepts and Limitations.

Timing Concepts Overview

The following list describes the main concepts that apply to all statement types:

HVI Engine Clock

This is the clock at which an HVI engine is running on.

HVI Engine Cycle

An engine cycle is the timeframe in which the HVI engine can fetch, dispatch or execute instructions. One engine cycle is equal to the period of the engine clock. For example, for an engine that runs at 100 MHz, the duration of an engine cycle will be equal to 10 ns.

HVI Common Clock

This is not a real clock, it is a definition to calculate timing for Sync Statements within HVI sequences. It can be seen as a clock that has its rising edge aligned with all HVI Engines clocks rising edges.

Therefore, its frequency is equal to the GCD of the frequencies of all the HVI Engine clocks:

$HVI\_Common\_Clock_{Frequency} = GCD\{HVI\_Engine\_Clock\_1_{Frequency}, HVI\_Engine\_Clock\_2_{Frequency}, ..., HVI\_Engine\_Clock\_N_{Frequency}\}$, where N is the number of engines added to HVI.

The period can be calculated in two ways:

- as the LCM of HVI Engine Cycles (the periods of all the HVI Engine clocks):

  $HVI\_Common\_Clock_{Period} = LCM\{HVI\_Engine\_Clock\_1_{Period}, HVI\_Engine\_Clock\_2_{Period}, ..., HVI\_Engine\_Clock\_N_{Period}\}$, where N is the number of engines added to HVI

- or, just the inverse of the HVI Common Clock frequency:

  $HVI\_Common\_Clock_{Period} = 1/\ HVI\_Common\_Clock_{Period}$

  For example, if the engines added to HVI have the following HVI Engine Clock frequencies {100MHz, 187.5MHz, 300MHz}, the HVI Common Clock frequency/period will be:

  $HVI\_Common\_Clock_{Frequency} = GCD\{100MHz, 187.5MHz, 300MHz\} = 12.5MHz$,

$$HVI\_Common\_Clock_{Period} = 1/\ HVI\_Common\_Clock_{Frequency} = 80ns$$

When calculating Sync Statement timing, the HVI Common Clock period is used to round some timing magnitudes to the next HVI Common Clock period (below **ncc** stands for **n**ext **c**ommon **c**lock):

$$round_{ncc}(TimeValue) = ceil(TimeValue\ /\ HVI\_Common\_Clock_{Period})\ *\ HVI\_Common\_Clock_{Period}$$

NOTE When working with fractional or periodic time values or periods to avoid problems with the numerical precision it may be better to use the frequency value instead following this simple equation:

$$HVI\_Common\_Clock_{Frequency} = 1/\ HVI\_Common\_Clock_{Period}$$

Start Time of HVI execution

This is the time 0 for the HVI execution. It always matches the rising edge of the Sync signal (in PXIe systems aligned with the PXIe-SYNC100 signal).

Start Time of statement execution

The relative time in nanoseconds from the HVI Execution Start Time to the start of the execution of a statement.

Fetch time

This is the time interval required by the HVI engine to fetch and dispatch a statement for processing. The Fetch time consumes HVI engine execution cycles. A statement may take several HVI engine cycles to complete the fetch before processing can start. The number of cycles a fetch takes depends on the statement or instruction characteristics, for instance, the number of parameters.

Start Delay

This is the user-defined delay value from the Start Time of the previous statement to the Start Time of the current statement. This value can be expressed in seconds or one of its fractions, down to picoseconds. Generally, the valid range is from 0 to +infinity, however the exact range and granularity of this value is defined by the following:

- The acceptable values are multiples either of the `HVI Engine Clock` period (in local statements) or, of the `HVI Common Clock` period (in sync statements).
  - For example, for a local statement for an HVI Engine with Clock frequency of 100MHz, the *clock period* is 10 ns, so the acceptable values are the multiples: 0 ns, 10 ns, 20 ns, etc.

- As another example, for a sync statement, if there are three engines added to HVI with the frequencies {100MHz, 187.5MHz, 300MHz}, the HVI Common Clock frequency will be 12.5MHz and the *period* is 80 ns, so the acceptable values are the multiples: 0 ns, 80 ns, 160 ns, etc.
- The acceptable margin of the value is defined in the Error and Warning Margins section below.

- The minimum possible value is affected by the Start-Latency of the current statement and the End-Latency of the previous statement. Formulas to calculate the minimum values are provided in the Timing Tables.
- The maximum possible value is only limited by the actual representation of the value in hardware and software. While this limit in hardware is instrument-dependent, in software it is defined as: The maximum value that can be represented in a signed 64-bit integer value.

The following sections explain how to calculate the Start Delay. When compiling the sequence, the compiler will report any timing violation and suggest a closer correct value.

Execution Time

This is the time interval from the Start time until the End time of the statement. This interval is determined by constraints and inherent limits of the instrument, such as propagation delays and resource availability. Sync and Flow-control statement execution cannot overlap with other statements, so in these cases the execution time must be added to the timing calculation. The Start delay of the next statement from a flow-control or Sync statement is measured from the end-time of the statement.

Sequence Time

Sequence Time is the sum of all the Start Delay of all the statements in a sequence, plus the Execution time values for any flow-control or Sync statements.

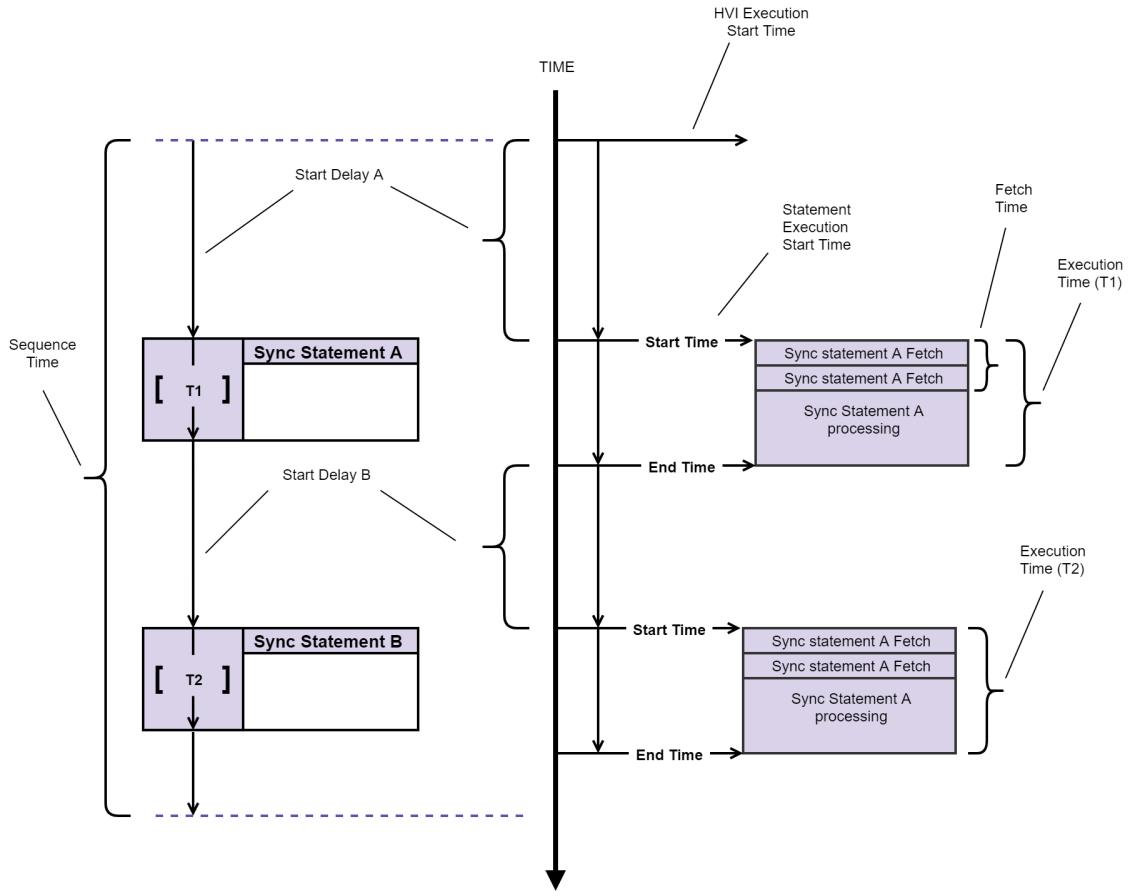
Internal Sequences

Some Sync and all Local flow-control statements are broken into internal sequences for execution in HVI engines.

Duration Property

The Sync statements and Local flow-control statements If and While include a `duration` property that you can set. The `duration` property enables you to specify the time interval that a statement takes to execute.

The following diagram shows these concepts in an HVI diagram:



Additional Timing Concepts and Limitations

There are several additional concepts and parameters you must be aware of to calculate timing, especially for specifying Start delays and the Duration of statements.

Even though the knowledge of these concepts can assist you to understand HVI timing and accurately specify proper values for these timing properties, it is not mandatory to use them at development time. This is because all limitations are checked by HVI at the time of compilation and any violation is reported with information provided about how it can be resolved. This enables you to focus on its sequence creation without worrying about complex timing calculations.

Latency Parameters

The latency parameters are defined for all Sync and flow-control statements. They impose a minimum value to the Start delays of the statements used in a sequence:

Start-Latency

This is the minimum number of clock cycles a Sync or flow-control statement requires to start execution.

Entry-Latency

This is the minimum number of clock cycles a flow-control statement requires to start the execution of the internal sequence. This imposes a minimum value on the Start delay of the first statement of the internal sequence.

End-Latency

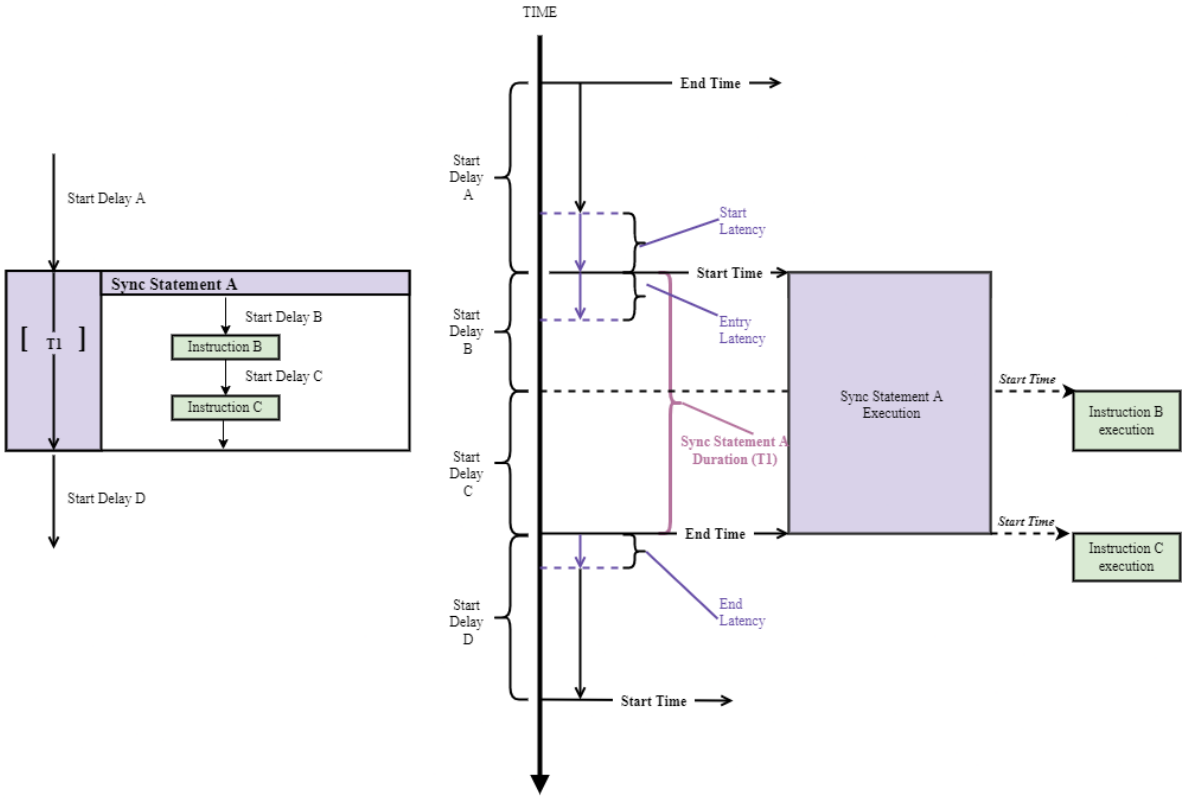
This is the minimum number of clock cycles a statement requires to exit its execution, before another statement can be executed.

Iteration Latency (loop statements)

For loop statements only, this is the minimum number of cycles a loop statement requires to start another execution of the internal sequence after one iteration is completed. This imposes a minimum value on the start delay of the first statement of the internal sequence.

The exact definitions of Start latency, Entry latency and End latency depend on the type of statement. Latency values are used in Sync Statement Timing and Local Instruction Timing . The Latency values are listed in Sync Statement Timing Tables , Local Flow-Control Statement Timing Tables and Local Instruction Statement Timing Tables .

The following diagram shows the Start, Entry and End Latencies and how they relate to Start delays:



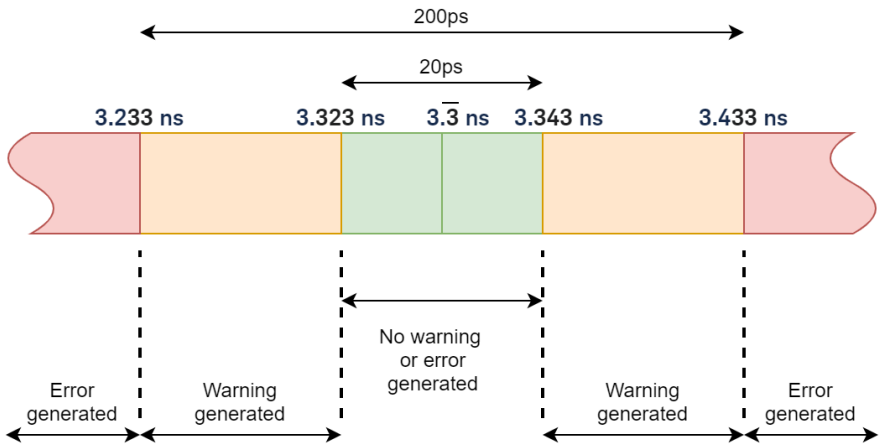
Error and Warning Margins Related to Timing Resolution

PathWave Test Sync Executive implements a policy for error and warning margins when you specify the timing for a Start delay or a duration.

The following table shows example values for an instrument with a 300MHz clock ($3.\overline{3}$ns clock period):

Range Type	Range	Example	Description
No Error or Warning	±10ps	3.323ns to 3.343ns	If you set a value with ±10ps error from the exact clock period multiplier value, no error or warning is generated.
Warning	±100ps	3.233ns to 3.323ns, or 3.343ns to 3.433ns	If you set a value between ±10ps and ±100ps of the exact clock period multiplier value, a warning is generated.
Error	>100ps	0.000ns to 3.233ns, or 3.433ns to 6.566ns	If you set a value with more than ±100ps error from the exact clock period multiplier value, an error is generated.

The following diagram shows an example where the exact clock period multiplier value is $3.\overline{3}$ ns, this is the same as the example in the table.



To calculate the margins for other period multiplier values, warnings are +-10ps from the exact value and errors are +-100ps away from the exact value.

Duration Property of Statements

The Sync statements and Local flow-control statements If and While include a `duration` property that you can set. The `duration` property enables you to specify the time interval that a statement takes to execute.

This value can be expressed in seconds or one of its fractions, down to picoseconds. Generally, the valid range is from 0 to +infinity, however the exact range and granularity of this value is defined by the following:

- The acceptable values are multiples either of the `HVI Engine Clock` period (in local statements) or, of the `HVI Common Clock` period (in sync statements).
  - For example, for a local statement for an HVI Engine with Clock frequency of 100MHz, the *clock period* is 10 ns, so the acceptable values are the multiples: 0 ns, 10 ns, 20 ns, etc.
  - As another example, for a sync statement, if there are three engines added to HVI with the frequencies {100MHz, 187.5MHz, 300MHz}, the HVI Common Clock frequency will be 12.5MHz and the *period* is 80 ns, so the acceptable values are the multiples: 0 ns, 80 ns, 160 ns, etc.
  - The acceptable margin of the value is defined in the Error and Warning Margins section below.

- The minimum possible value is affected by internal operations of the statement. For statements that contain internal sequences, the minimum is affected also by the Start-Delay and the Duration of the internal statements. Formulas to calculate the minimum values are provided in the Timing Tables.
- The maximum possible value is only limited by the actual representation of the value in hardware and software. While this limit in hardware is instrument-dependent, in software it is defined as: The maximum value that can be represented in a signed 64-bit integer value.

NOTE For the loop statements Local while and Sync while, the `duration` property specifies the execution time of 1 iteration. This means that the overall execution time of a while statement depends on the number of iterations that are executed. The total execution time is `duration` multiplied by the number of iterations.

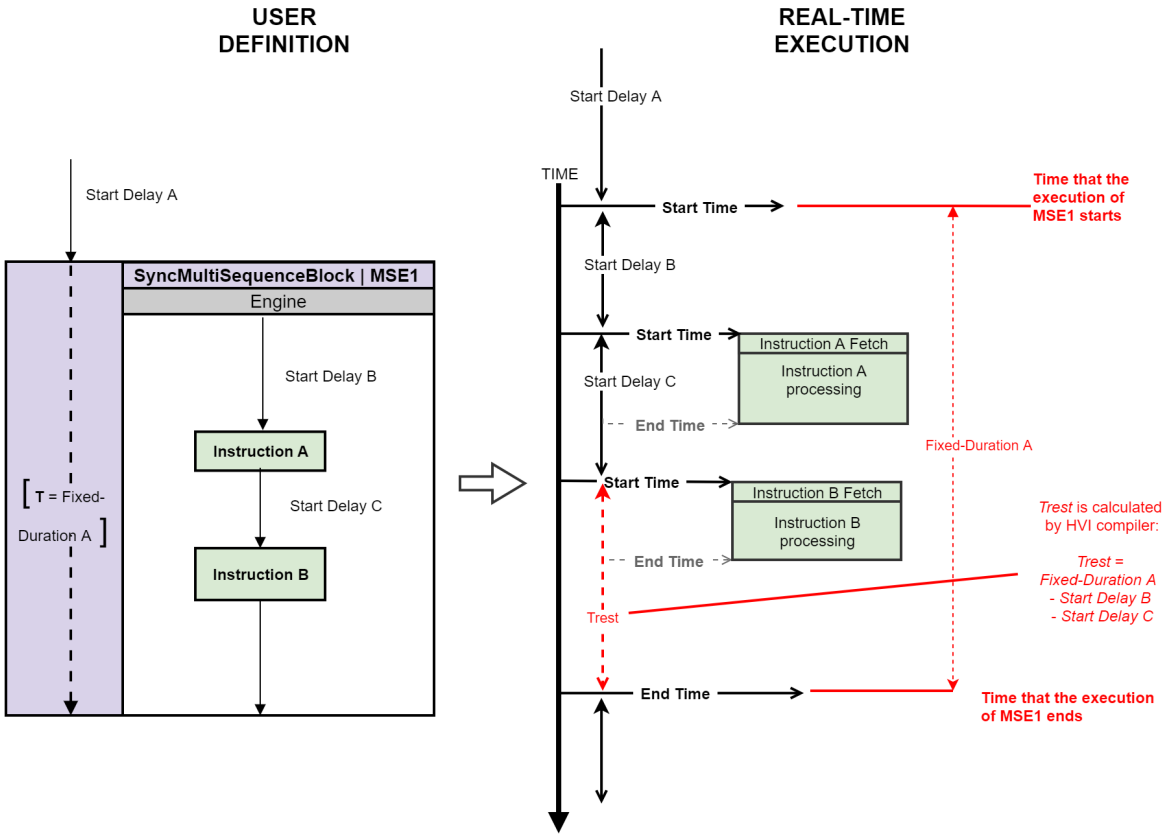
If the `duration` is set to a **fixed-time** interval, then the execution time of the statement shall match the value specified in the `duration` property. If this time cannot be matched an error is generated. For example, this can happen with an if-statement when more time is required to complete the statements inside a branch than the duration specified.

The `duration` property cannot be set to fixed value if there is a flow control statement inside that has an unknown duration.

If the `duration` is set to a **minimum-time** interval, then the execution time of the statement is the minimum possible given by the statements inside.

NOTE By default, if not specified, `duration` property is set to **minimum-time**.

The following diagram shows how the `duration` property is applied to a Sync multi-sequence block:

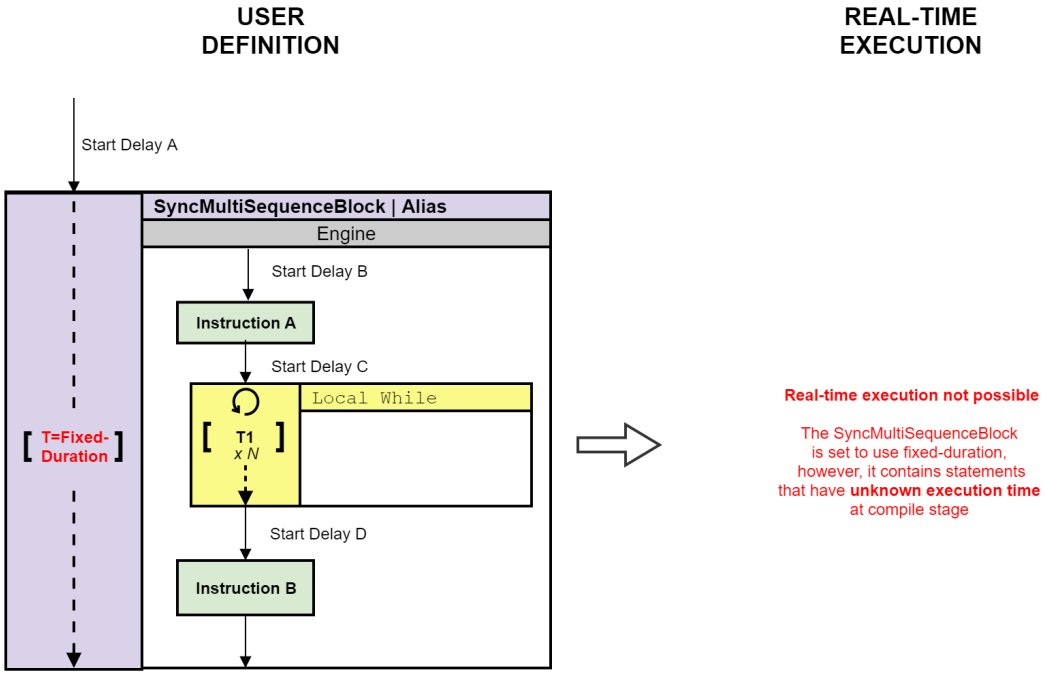


Python code for the preceding diagram:

```
fixed_duration_A = time.Duration(xxx)
mse1 = sequencer.sync_sequence.add_sync_multi_sequence_block('mse1', start_delay_A)
mse1.duration = fixed_duration_A
sequence = mse1.sequences['Engine1']
instructionA = sequence.add_instruction("instructionA", start_delay_B, sequence.instruction_
set.action_execute.id)
instructionB = sequence.add_instruction("instructionB", start_delay_C, sequence.instruction_
set.action_execute.id)
```

NOTE It is not allowed to set the `duration` property of a Statement A to a fixed-time if the Statement A contains a flow control statement with an unknown duration (e.g. WaitEvent, WaitTime, While, etc.). Doing so will result into an error at compilation.

As an example, the following diagram shows a while loop that generates an error if the user would try to set to a fixed-value the duration of the SyncMultiSequenceBlock that contains a local While statement:



Synchronization Clocks, Signals, and Modes

To correctly manage timing without jitter, the HVI needs information about all of the clocks in each instrument. For instruments that support HVI technology and are included in the HVI, the clocking information is already available and handled transparently. For instruments that do not support HVI technology, you must specify the instrument clocking constraints.

HVI Clocks

HVI supports the definition of the following types of clocks:

Non-HVI system clocks

Instrument system clocks are those clocks used by the instrument that do not directly impact the operation of the specific feature that the HVI must trigger. System clocks are used by the HVI to determine the Sync_Base period.

Non-HVI core clocks

Core clocks are instrument clocks that directly impact the operation of the specific feature that the HVI must trigger. Core clocks are used by the HVI to determine both the Sync and the Sync_Base period.

Sync and Sync_Base Signals

HVI uses different periodic digital signals for synchronization purposes. The definition of those digital signals depends on platform and instruments signals. Platform signals are the CLK100 and CLK10 signals in a PXI platform such as a PXI chassis. Instruments have different clock signals inside that are classified as core clocks or system clocks.

Platform and instrument clock signals contribute to define the HVI Sync signals according to the following definitions.

The period (and inversely, also the frequency) of the Sync signal is defined as:

```
Sync_Period = N x LCM(all instrument core clocks), N such that NxLCM(.) ≥ Phys-
icalPropagationDelay
```

The period (and inversely also the frequency) of the Sync_Base signal is defined as:

```
Sync_Base_Period = LCM(CLK10, Sync_Period, all instrument system clocks)
```

where, in the above formulas, LCM(.) stands for the *Least Common Multiple* operation.

Both Sync and Sync_Base periods must be equal to or greater than the *Physical Propagation Delay* value for the relevant multi-chassis topology, these are given in the following table. If the LCM(all instrument core clocks) is smaller than that, then you must take the next multiple after the LCM(.) that is equal to or greater than the Physical Propagation Delay. The next multiple is the actual Sync Period value for your system. Once that Sync Period value is obtained, you can use it in the Sync_

Base LCM formula to estimate the Sync_Base Period, which is automatically also greater than the Physical Propagation Delay.

Physical Propagation Delay

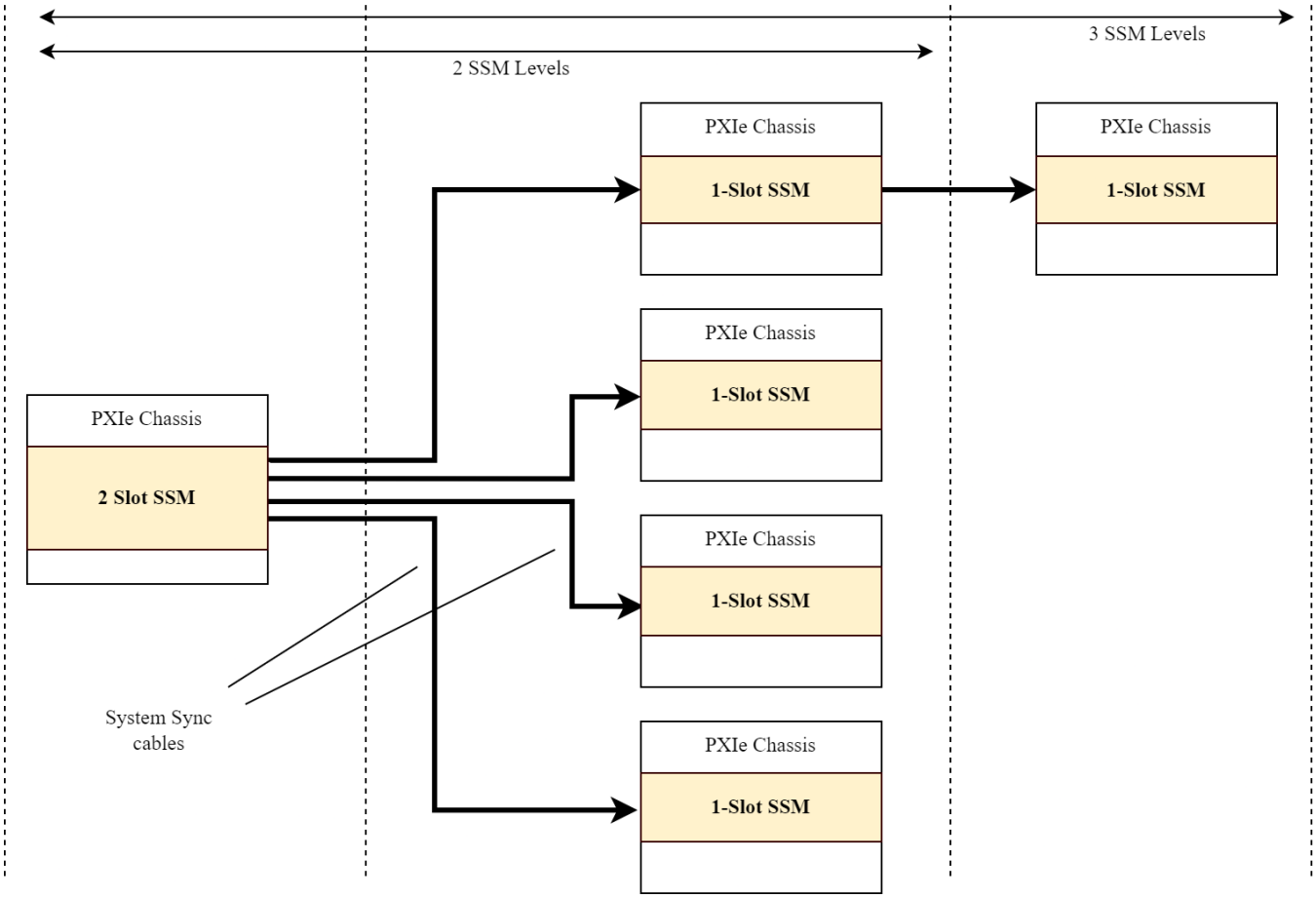
The Physical Propagation Delay corresponds to the amount of time (expressed in nanoseconds) that a PXIe trigger needs to cover the path between any given pair of segments in a topology. This value is used when running Sync statements because it provides information about how long the execution signaling between modules takes.

The topology is defined by the number of chassis in your system and how they are connected to each other with the System Sync cabling.

The System Sync cabling distributes clocks, triggers, and data from the Leader SSM to the followers, possibly going through intermediate followers. The number of System Sync hops between the Leader SSM and each Follower determines what is known as the SSM level. The Leader is SSM level 1, all SSMs connected with 1 hop to the Leader SSM are Level 2 SSMs, those with 3 hops are Level 3 SSMs, and so on.

In the case of the M9033A SSM, there can be up to 4 followers connected to a single SSM, so there can be up to 5 chassis in system with 2 SSM levels. If you connect additional SSMs to the level-2 SSMs, this creates a 3rd level. In this arrangement you can add one additional chassis, this is because the PathWave Test Sync Executive 2022 release supports up to 6 chassis.

The following diagram shows a 6 chassis system with 3 SSM levels:



The following tables shows the Physical Propagation Delay values for different numbers of chassis and SSM levels:

Number of Chassis	Number of SSM levels	Physical Propagation delay [1,2] (ns)	Notes
1 chassis	-	100	
2 chassis	-	200	
3 chassis	-	300	
>3 chassis	2 SSM levels	300	Maximum 5 chassis
>3 chassis	3 SSM levels	400	Maximum 6 chassis with PathWave Test Sync Executive 2022

1 *Upper bound on the time it takes for a PXIe trigger to travel from the furthest most segments*

2 *Ensure your M904x chassis has version 5 or higher firmware revision for the Left and Right Trigger Bridges. See the hardware revision in your chassis Software Front Panel (SFP).*

Sync Period Calculation

The Sync Period must always be greater than or equal to the Physical Propagation Delay. To obtain the actual Sync Period value, you first calculate the *Least Common Multiple* (LCM) of all HVI and non-HVI core clock periods added to the System Definition. Secondly, you compare the LCM with the Physical Propagation Delay and take the next multiple of the LCM that is greater than or equal to the *Physical Propagation Delay*. This is what was also conveyed by the previous Sync Period formula.

The base unit of time measurement on an HVI engine is the period of its own HVI Engine Clock, but the Physical Propagation Delay is expressed in nanoseconds. To be able to use it, each engine must express it in Clock cycles, so a conversion is required:

```
Propagation_delay_cycles = Round(Physical_Propagation_Delay/Hvi_Engine_Clock_period)
```

For example. to calculate the Sync frequency for instruments A and B use the formula:

```
Sync = LCM(all instrument core clocks)
```

Instrument A Core clock = 100 MHz, period = 10 ns

Instrument B Core clock = 300 MHz, period = 3.333 ns.

Since 10ns is a multiple of 3.333 ns, the LCM is 10ns. If your instruments are all in 1 chassis, the *Physical Propagation Delay* constrained by the propagation delay is 100 ns (per the values in the previous table). Therefore, you need to take the next multiple of the LCM = 10 ns which is also equal or greater than 100 ns. This gives the final value of the Sync Period as 100 ns and the Sync signal frequency is 10MHz.

NOTE You can find the instrument System and Core clocks in the documentation of each instrument.

Synchronization Modes

You can configure the synchronization mode. This is used, for example, for generating a trigger value or waiting for an event.

The following modes are supported:

`IMMEDIATE`

The trigger or action is issued immediately, with no need to wait for any common synchronization clock. For the Wait-For-Event, the HVI execution continues immediately, as soon as the event is received.

`SYNC`

The trigger or action is issued at the first edge of the Sync signal. For the Wait-For-Event, the HVI execution continues at the first edge of the Sync signal, following the event arrival time.

`SYNC_BASE`

The trigger or action is issued at the first edge of the Sync_Base signal. For the Wait-For-Event, the HVI execution continues at the first edge of the Sync_Base signal, following the event arrival time.

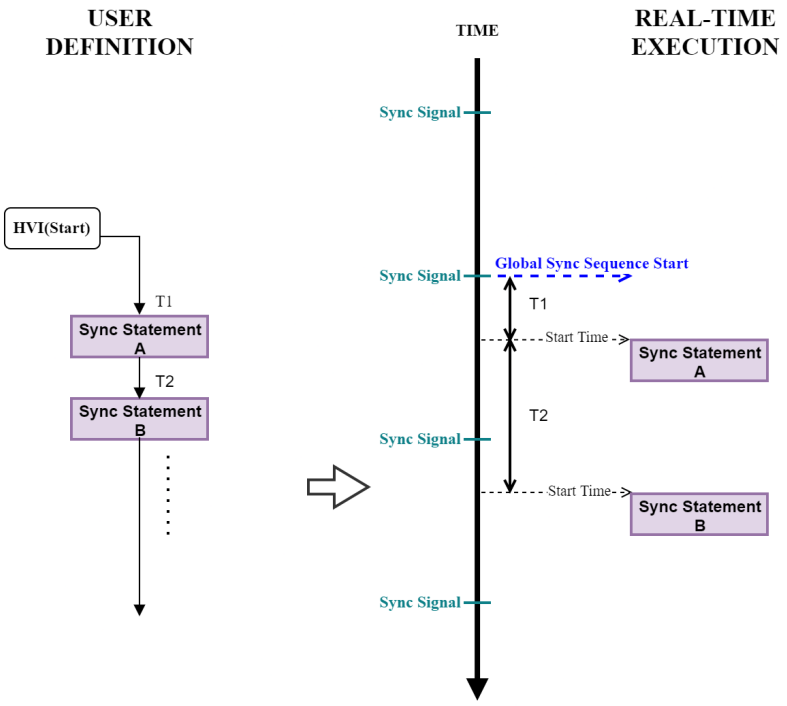
Sync Statement Timing

This section describes Sync statement timing. It contains the following sections:

- Global Sync Sequence Start
- About Sync Statement Timing.
- Sync Multi-Sequence Block Timing.
- Sync While Timing.
- Sync Register-Sharing.
- Sync FPGA Data-Sharing.

Global Sync Sequence Start

The Global Sync Sequence start is the timing point when the HVI sequence will start executing, at the same time, in all the engines added in the SystemDefinition. This timing point is aligned with the arrival of the Sync signal.



About Sync Statement Timing

Sync statements consume HVI engine execution time and cannot overlap their execution with other statements. Their start and end is synchronized and happens at the same HVI Common Clock cycle across all the engines participating in the system. The Start delay of a Sync statement is measured from the end of previous Sync statement to the start of the current one.

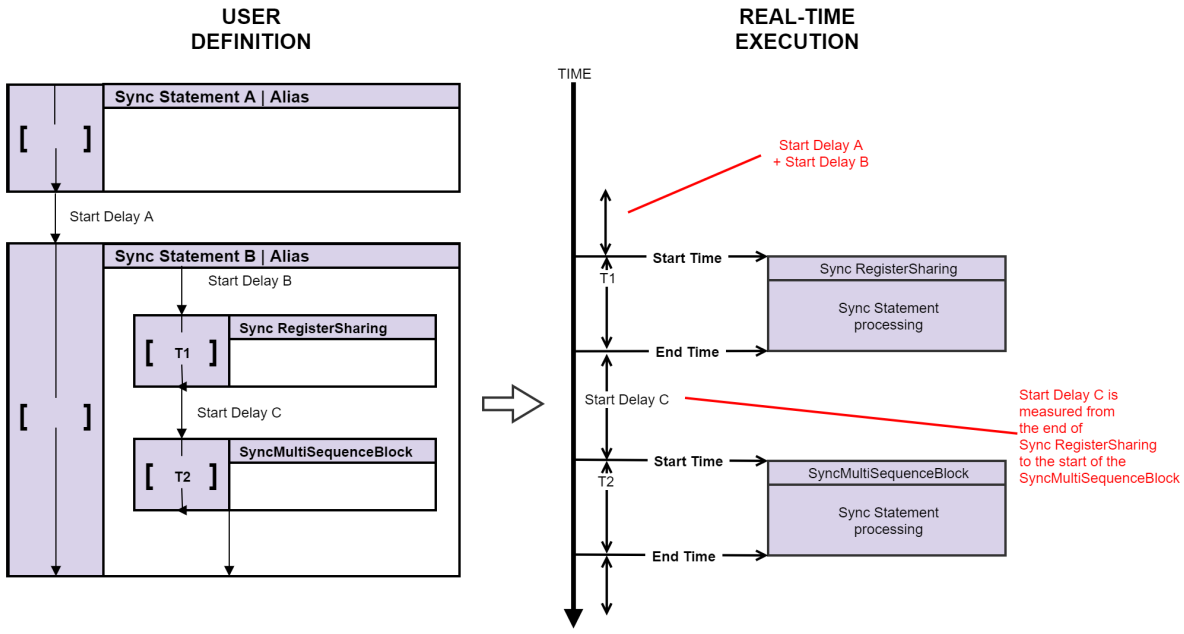
The following diagram shows the timing between a number of Sync Statements including a Sync register-sharing statement and Sync multi-sequence block statement.

The diagram shows two Sync Statements A and B. Sync Statement B is a container for two further Sync Statements: Sync register-sharing and Sync multi-sequence block. The times indicated are Start Delay A, Start Delay B, Start Delay C, T1, and T2.

The time between the end of Sync Statement A and the start of Sync register-sharing is Start Delay A + Start Delay B .

The time between the end of Sync register-sharing and the start of Sync multi-sequence block is Start Delay C.

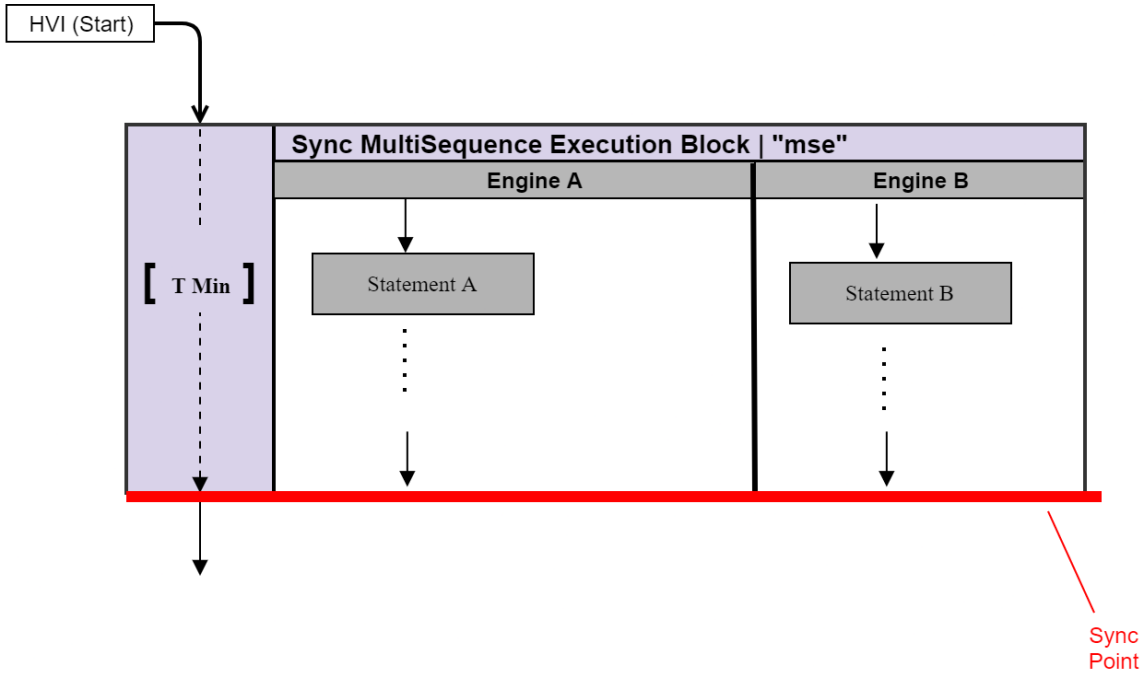
Sync register-sharing and Sync multi-sequence block timing:



Sync Multi-Sequence Block Timing

In a synchronized multi-sequence block, you can define the statements that the HVI engines execute in parallel with other engines.

Local sequences start and end their execution within the Sync multi-sequence block synchronously.



HVI automatically calculates the execution time of each local sequence and adjusts the execution of all local sequences within the Sync multi-sequence block. This is so that all the sequences within the Sync multi-sequence block can end together deterministically. The final time is calculated automatically.

There are two cases for the Sync-Point that are treated in different ways by HVI:

- Timed-Sync: When the execution time is known at HVI compilation time for all Local sequences within the Sync multi-sequence block.
- Triggered-Sync: When the execution time is unknown at HVI compilation time for one or more Local sequences within the Sync multi-sequence block.

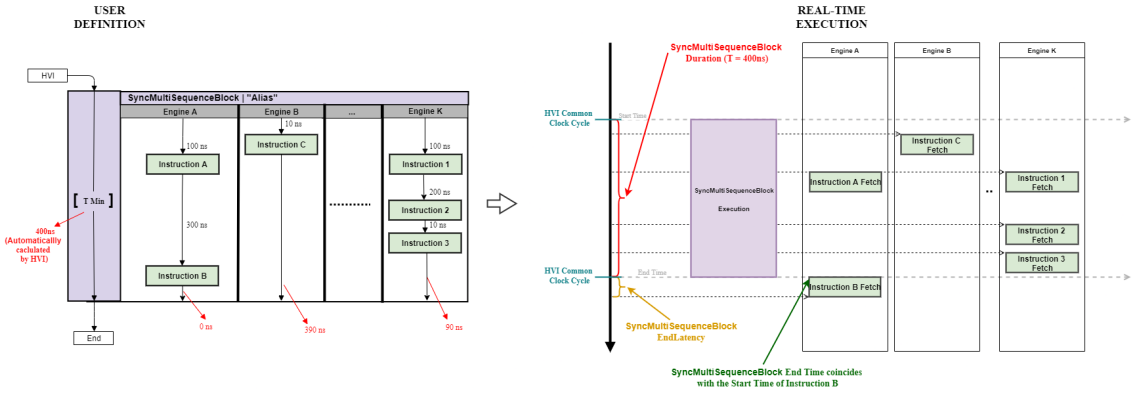
Timed-Sync (Sync multi-sequence block containing Local sequences with known total execution time)

For Sync multi-sequence blocks that contain instructions or flow-control statements with execution times that are known at HVI compilation time, the HVI compiler accounts for the different sequence execution times during compilation and then adjusts the final times. This ensures all of the Local sequences reach the end of the Sync multi-sequence block at the same time.

When the execution time (duration property) of the Sync multi-sequence block is not specified, the compiler adjusts the total execution time to be the minimum possible to allow the execution of the longest Local sequence. Note that in the case that the Engines participating in the system do not

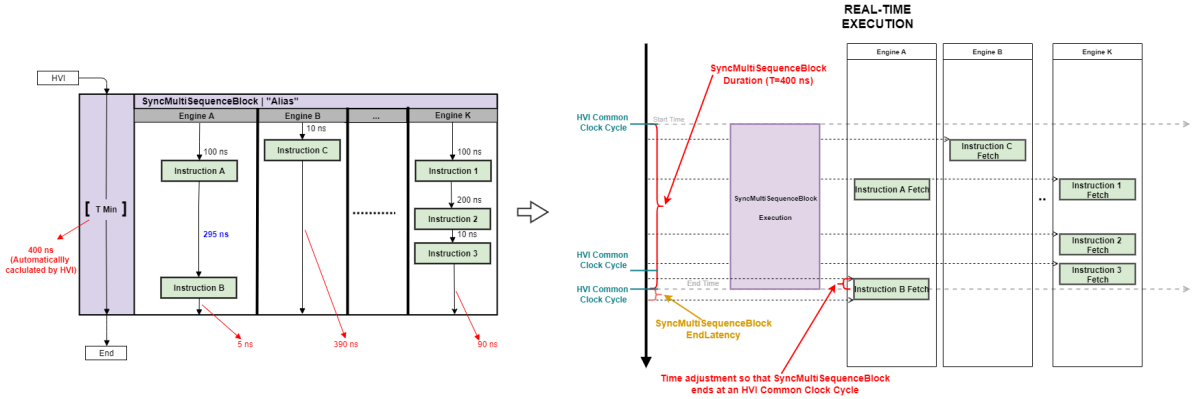
share the same frequency, HVI will automatically adjust the duration (or execution time) of the Sync Multi-Sequence Block statement to a multiple of the HVI Common Clock.

For the example in the diagram below, the total time for Engine A is 400 ns. HVI calculates the times required for the other engines to finish at the same time. For Engine B this is 390 ns, for Engine K this is 90 ns.



Using the previous example but assuming that the Engine A runs at 200 MHz, while the rest engines run at 100 MHz, the common clock cycle will happen at multiples of 10 ns. In the following diagram we can notice two things:

- The duration of the longest sequence (Engine A) is 395 ns, which is not a multiple of a common clock cycle. Therefore, HVI will adjust the end of the Sync Multi-Sequence Block to the next Common Clock Cycle at 400 ns and then make sure the sequences of all the Engines match this time.
- The start of the end-latency of the statement will not start from the start time (395ns from the beginning of the Sync Multi-Sequence Block statement) of the last statement of the longest sequence (Engine A) because it is not at a common clock cycle. Rather, it will start from the next Common Clock Cycle, at 400 ns.

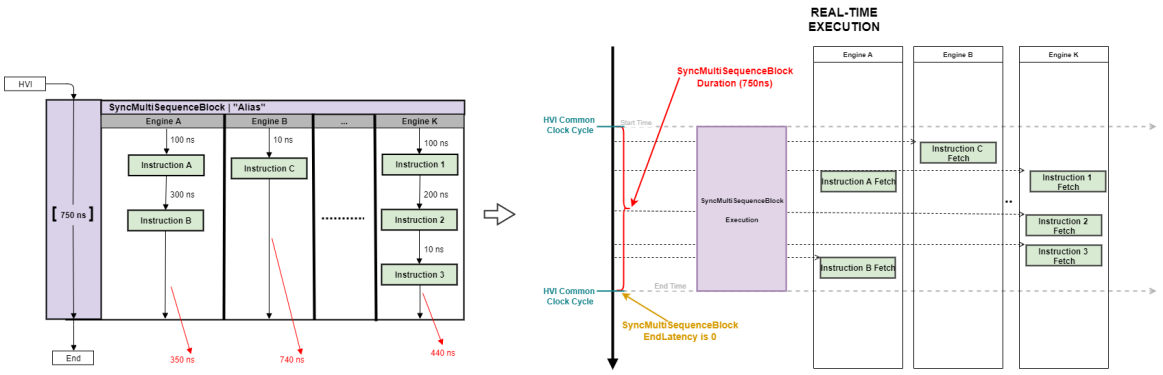


Sync multi-sequence block with a specific execution time (duration property)

When the execution time (duration property) of the Sync multi-sequence block is specified, the compiler verifies that the specified execution time is enough to allow the execution of the longest

Local sequence, if not an error is generated. Note that in the case that the Engines participating in the system do not share the same frequency, the specified execution time (duration property) must be a multiple of the HVI Common Clock.

In the following diagram, the times of the instructions and the delays between them are known, so the timing between them and for the entire block can be calculated. In this case the total time is specified at 750 ns. The HVI calculates the times required for all the other engines to finish at the same time. For Engine A this is 350 ns, for Engine B this is 740 ns, for Engine K this is 440 ns.



Triggered-Sync (Sync multi-sequence block containing Local sequences with unknown execution times)

In some cases, one or more of the local sequences within the Sync multi-sequence block include a local flow-control statement that has an execution time that is unknown at HVI compilation time. At the point in the Local sequence where the unknown execution time is encountered, the Local sequence becomes de-synchronized and **an active triggering process is required** at the end of the Sync multi-sequence block to re-synchronize the execution of all HVI engines. This guarantees that all the HVI engines then continue execution at exactly the same point after the **Triggered-Sync point**. The execution resumes in all HVI engines at the same time, aligned with a sub-sequent Sync pulse, which forces the execution to be aligned to a multiple of the Sync period of the main Sync signal. Triggered-sync points require the use of trigger resources assigned in the `SyncResources` property in the `SystemDefinition` instance and the main Sync signal.

Possible cases of the unknown execution time is when one of the Local sequences contain:

- A Wait-for-time statement with a register defining the wait time at runtime.
- A Wait-for-event statement.
- A While statement.
- An If statement with unmatched branches, that take different execution times.

NOTE Note that specifying the execution time (duration property) in this scenario is not allowed and will lead to a compilation error.

Triggered-Sync delay

A triggered-sync point adds a delay to the sequence timing that has four parts. Two of them are constant and the other two vary depending on the last statement and its position compared to the Sync pulse time. The formula to calculate the delay is:

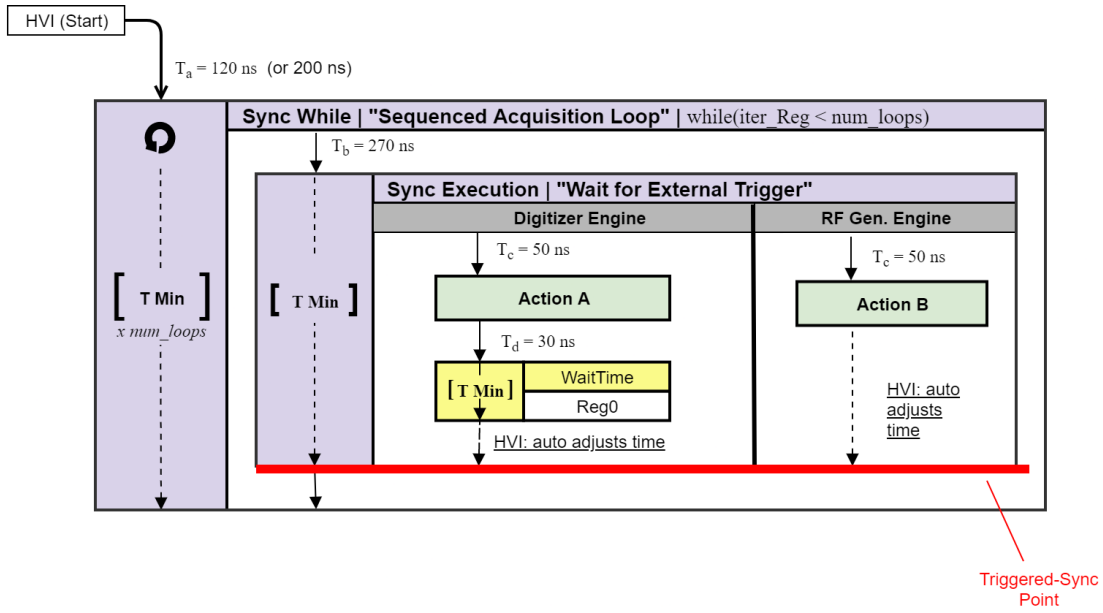
```
triggered_sync_delay = end_latency + sync_overhead + edge_offset + sync_period
```

where:

- `end_latency` is the End-latency of the last statement before the resync. If the last statement is a local instruction, this is equal to its Fetch time.
- `sync_overhead` is constant per instrument. Its value is 3 cycles.
- `edge_offset` is the time interval from the end of the `sync_overhead` to the sync-pulse edge. This time can vary depending on the position of the last statement compared to the Sync pulse time.
- `sync_period` is constant per configuration and is calculated by the equation defined previously.

Example of timing management with Triggered-Sync point

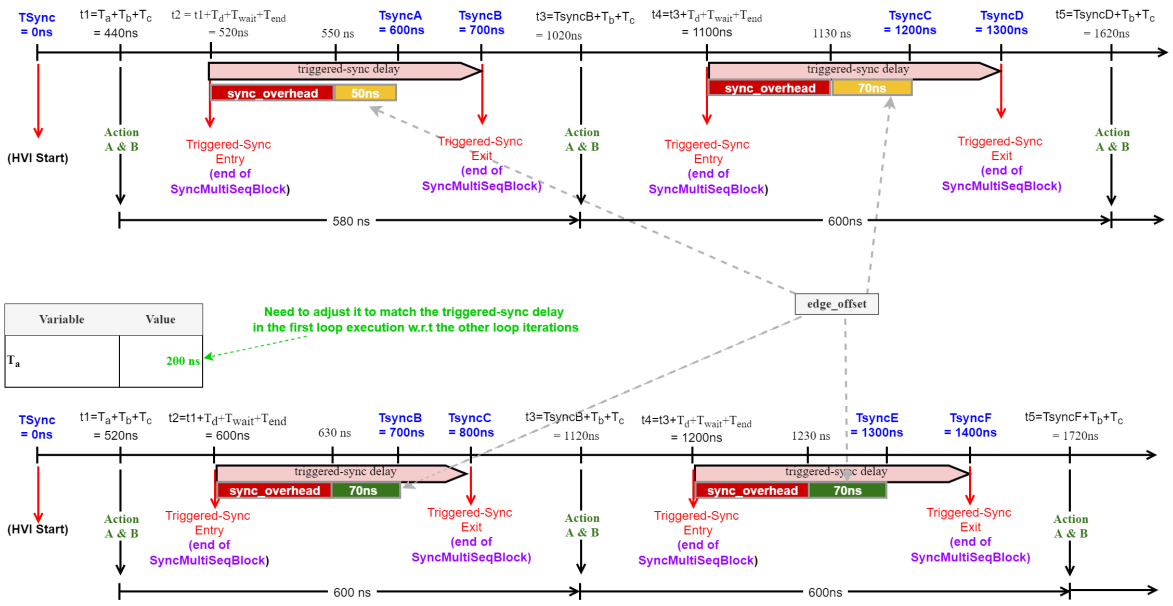
The following diagram shows an example with a simple sequence where the triggered-sync point is marked in red. The triggered-sync point is at the end of the Sync multi-sequence block and it is required because there is a WaitTime statement and the time for this cannot be determined at compile time.



The following table shows the Variables and their execution times:

Variable	Value	Description
T_a	120 ns	Start delay of Sync-while statement
T_b	270 ns	Start delay of Sync multi-sequence block
T_c	50 ns	Start delay of Action A instruction
T_d	30 ns	Start delay of Wait-for-time statement
Reg0	4	The register used for the Wait-for-time
T_{WAIT}	40 ns	The total wait time based on the value on the value of Reg0
T_{END}	10 ns	End-latency of Wait-for-time statement
T_{sync_period}	100 ns	Sync period for 1 chassis
$T_{sync\_overhead}$	30 ns	Sync overhead

The following diagrams shows the execution timeline for the first 3 iterations of the sequence shown in the previous diagram, it is important to note that the first triggered-sync aligns the execution with the Sync pulse and consequently the duration (or execution time) for the following cycles will be different, this effect is in some cases seen as a skew (or jitter) in the 1st cycle. A way to eliminate the first cycle variation is to adjust the sync while start time.



Jitter when waiting for external events or triggers

The triggered-sync is controlled by the Sync signal. This means that repeated executions, for example, inside a Sync While loop of an Sync multi-sequence block that contains a WaitEvent, may show jitter of the Sync multi-sequence block actions with respect to the event that is in the WaitEvent condition. In these cases, the maximum skew variation (or jitter) is the maximum time difference between Trigger events and the Sync Period. The Variable skew (or jitter) value can be:

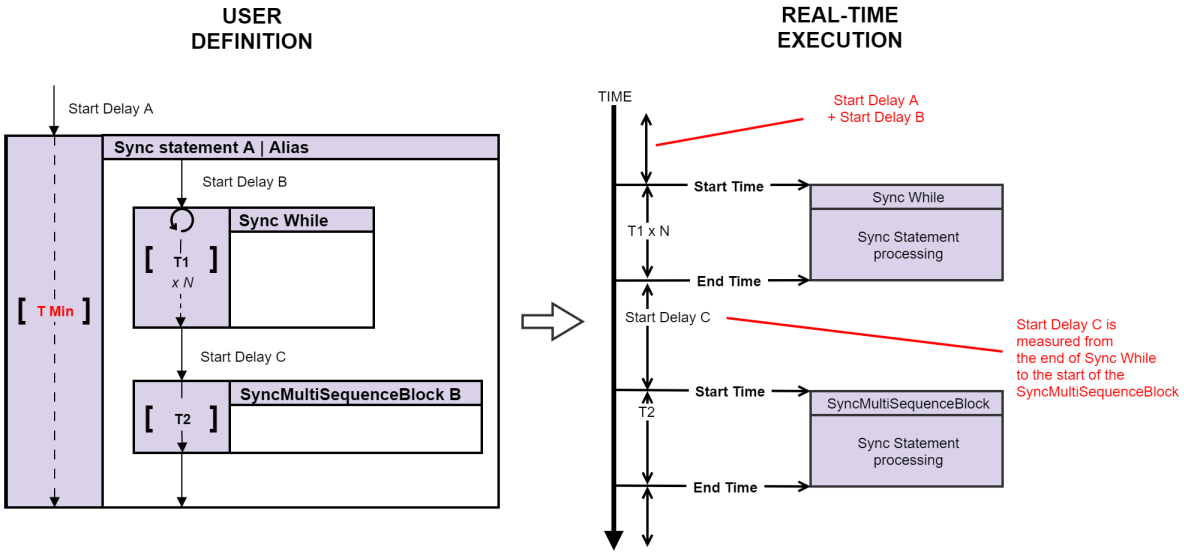
- 0 => when the trigger events have the same time delay with respect to the Sync signal.
- Sync period => When the trigger events are asynchronous and at a rate that is not multiple of the Sync period.
- If more than one synchronization signal is used (Sync, Sync_Base, etc.), the largest will dominate:
  - A Sync multi-sequence block always aligns its start to the Sync signal, so at least the jitter for an asynchronous event will be equal to the Sync period.
  - For example, if you also re-sync the Wait-For-Event with the Sync_Base (by using the SYNC_BASE SyncMode), and the trigger is asynchronous to the Sync_Base, then the jitter will be equal to Sync_Base.
- The Sync and Sync_Base periods depend on any Non-Hvi clocks (core/system) added to the systemDefinition using the HVI API.

Sync While Timing

For the Sync flow-control Statement Sync while, the timing is different compared to other Sync statements. The Sync while statement continues operation while a condition is met. It stops executing when the condition is no longer met.

The following diagram shows a Sync while statement with other Sync statements. The time for an iteration of Sync while is T2 x N, where T2 is the time per iteration and N is the number of iterations. The time cannot be indicated exactly on a diagram or in code because the number of iterations is not known until runtime.

The time for the containing statement Sync statement A cannot be indicated because it contains a flow-control statement. This is indicated by the dotted line and the time indicated as T min.

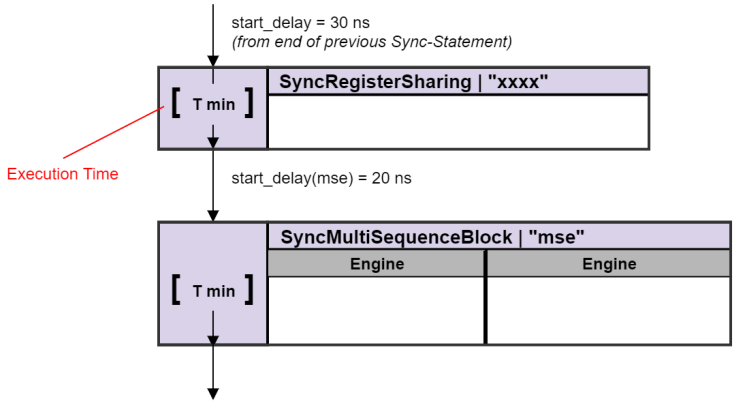


Sync Register-Sharing

The Sync register-sharing statement execution time must be accounted for when calculating the Sync sequence timing.

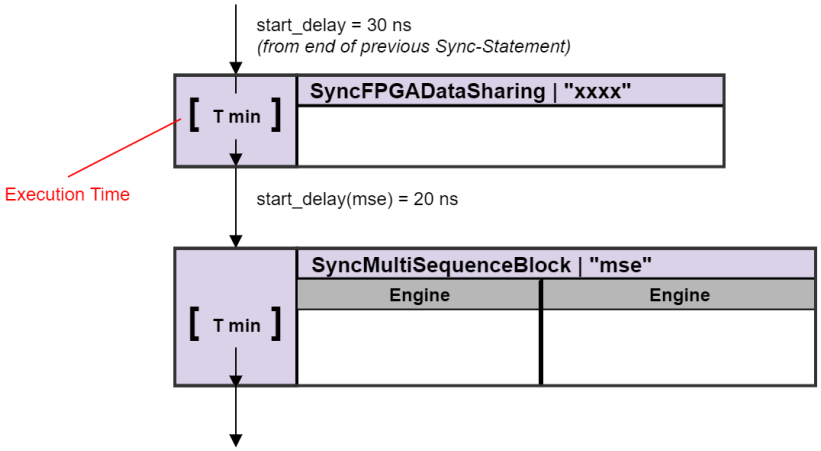
The following diagram shows Sync register-sharing statement followed by a Sync multi-sequence block.

For the execution time see Sync Statement Timing Tables.



Sync FPGA Data-Sharing

The Sync FPGA data-sharing statement enables you to share data between the FPGA Sandboxes on different instruments in the same or different chassis.



Estimating the execution time of a Sync FPGA Data-Sharing Statement

Sync FPGA data-sharing execution time depends on several factors:

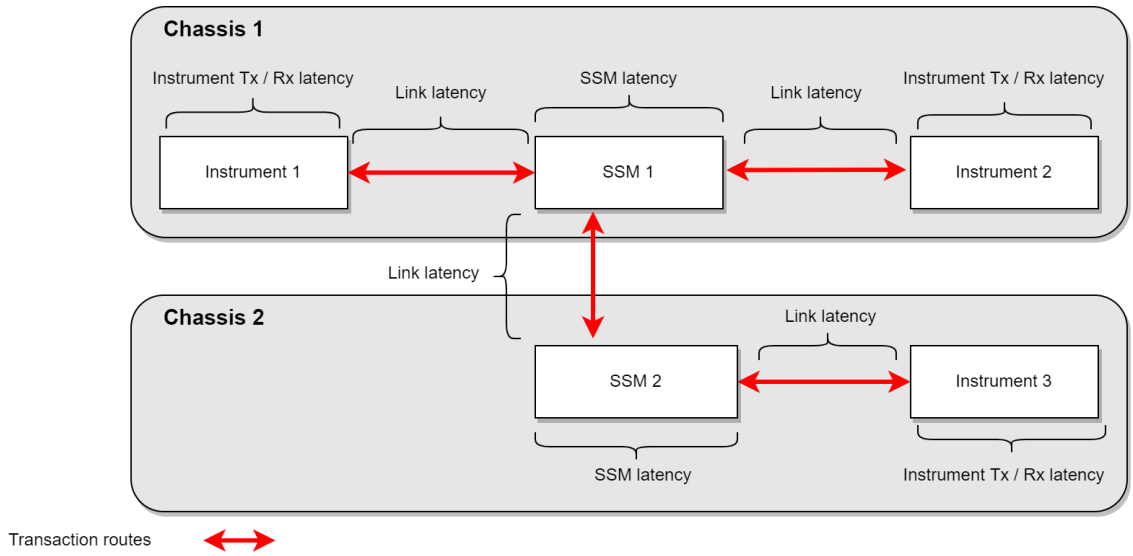
- Instrument specific delay characteristics.
- The topology of the system, for instance, the number of chassis and the System Sync connectivity topology across chassis.
- Location of the source and destination instruments, such as if the data transfer happens in a single chassis or through multiple chassis.
- The amount of data be transferred.
- The scheduling of multiple transactions.

In order to assist you estimate of the execution time, the following examples are provided that abstract some of the complexities, these should enable you to make good estimates of execution time.

Latency Equations

To estimate the Sync FPGA data-sharing statement execution time, a number of equations are provided below.

The following image shows the example topology used in the examples:



The formula to calculate the time it takes to send a single nibble (4-bits) of data from Instrument 1 to Instrument 2 (in the same chassis) is:

$$T_{single\_chassis} = T_{tx\_latency} + T_{link\_latency} + T_{ssm\_latency} + T_{link\_latency} + T_{rx\_latency} = T_{tx\_latency} + 2*T_{link\_latency} + T_{ssm\_latency} + T_{rx\_latency}$$

The formula to calculate the time it takes to send a single nibble of data from Instrument 1 to Instrument 3 (in a different chassis) is:

$$T_{two\_chassis} = T_{tx\_latency} + T_{link\_latency} + T_{ssm\_latency} + T_{link\_latency} + T_{ssm\_latency} + T_{link\_latency} + T_{rx\_latency} = T_{tx\_latency} + 3*T_{link\_latency} + 2*T_{ssm\_latency} + T_{rx\_latency}$$

To generalize this to N number of SSMs hops, the formula is:

$$T_{N\_chassis} = T_{tx\_latency} + (N_{SSM}+1)*T_{link\_latency} + N_{SSM}*T_{ssm\_latency} + T_{rx\_latency}$$

Since the previous equations are for sending only one nibble of data, if you want a transaction with more bits, this will be split into multiple of 4-bit transactions happening one after the other on consecutive clock cycles. For N number of data bits (which must always be multiple of 4-bits), the equation is:

$$T_{transaction\_duration} = T_{tx\_latency} + (N_{SSM}+1)*T_{link\_latency} + N\_SSM*T_{ssm\_latency} + T_{rx\_latency} + N_{num\_bits}/4$$

Constants Estimation

In the formulas above, the constants of the different latency equations are *instrument-specific*. See your instrument or *System Synchronization Module* (SSM) documentation for the exact latency values. The following values are are reference values for the product-specific latencies:

Variable	Reference Value (Clock Cycles)
$T_{tx\_latency}$	4
$T_{rx\_latency}$	3
$T_{SSM\_latency}$	4

In addition to the instrument/SSM specific latency the calculations must account for the link latency that depends on the link characteristic (PXIe backplane or System Sync/Link cable length) and receiving instrument implementation. See your instrument or SSM documentation for the exact latency values. The following is a reference value for the link latencies:

Variable	Reference Value (Clock Cycles)
$T_{link\_latency}$	12

Example Scenarios

As described above, a Sync FPGA data-sharing statement can contain multiple transactions. A transaction can go to one or multiple destinations, that is, deliver the same data to multiple Rx endpoints, but independently of the number of destinations, a transaction has only one transmission point and operation. PathWave Test Sync Executive optimizes the Sync FPGA data-sharing statement timing by parallelizing as much as possible the execution of different transactions. Depending on the number of transactions, the system topology and the overlap in terms of sources and destinations of the different transactions, the level of parallelization can vary. In the following examples we show some typical use cases.

Example 1: Single transaction with multiple destinations

The simplest use case for Sync FPGA data-sharing statement consists of a single data sharing transaction. A transaction can go to one or more destinations, that is, deliver data to multiple Rx endpoints, but independently of the number of destinations, a transaction has only 1 Tx endpoint and operation. The execution time for a single-transaction Sync FPGA data-sharing statement is given by the time required to complete the final Rx operation, where tr1 is the transaction, rx1, rx2 ...rxN are the Rx endpoints:

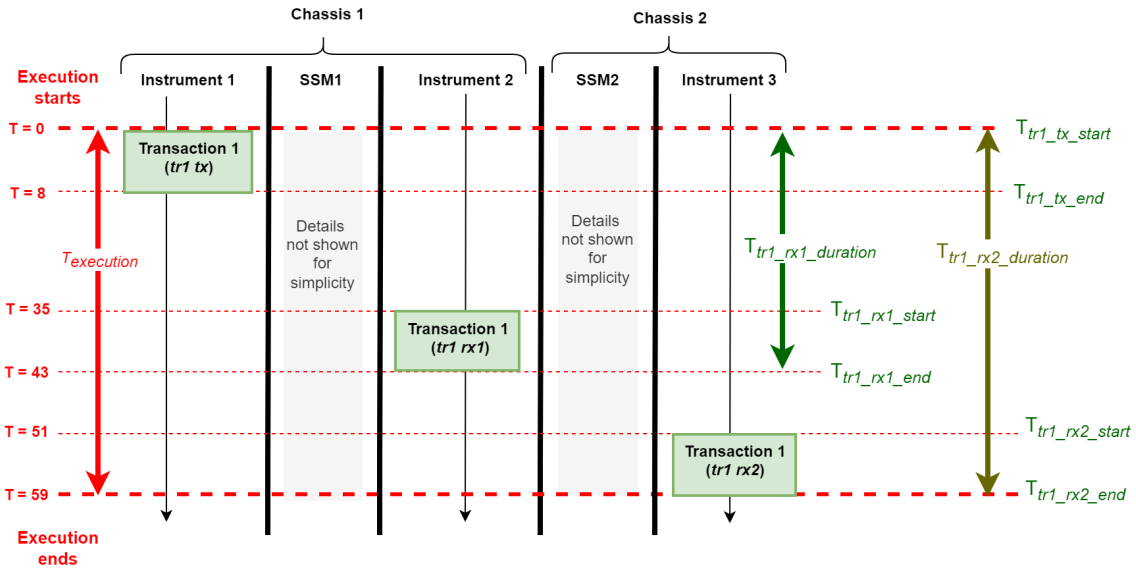
$$T_{execution} = \max(T_{tr1\_rx1\_end}, T_{tr1\_rx2\_end}, \ldots., T_{tr1\_rxn\_end})$$

The following example shows how to calculate the Sync FPGA data-sharing statement execution time for a single-transaction (*tr1*) with 2 destinations (rx*1* and rx*2*):

The following code snippet shows a Sync FPGA data-sharing statement sharing to two destinations:

```
# SyncFpgaDataSharing definition with a single transaction to 2 destinations (Rx)
#
## Transaction 1 (tr1)
instrument1_tx = keysight_hvi.FdsPortAddress(source_port, source_address)
instrument2_rx1 = keysight_hvi.FdsPortAddress(dst1_port, dst1_address)
instrument3_rx2 = keysight_hvi.FdsPortAddress(dst2_port, dst2_address)
fpga_data_sharing_st.transactions.add(instrument1_tx, [instrument2_rx1 , instrument3_rx2], num_
bits_to_share)
```

The following diagram shows the timing (execution starts at end of cycle 0).



tr1_tx refers to the transmission of the data from the transmission port.

tr1_rx1 and tr1_rx2 refer to reception of the data at the two receive ports.

tr1_rx1_duration is the total time from the beginning of transmission tr1_tx to the end of reception of the data at receive point 1 (tr1_rx1_end).

tr1_rx2_duration is the total time from the beginning of transmission tr1_tx to the end of reception of the data at receive point 2 (tr1_rx2_end).

Single Tx operation start:

$$T_{tr1\_tx\_start} = T_{execution\_start} = 0 \text{ cycles}$$

$$T_{tr1\_tx\_end} = T_{tr1\_tx\_start} + N_{num\_bits}/4 = 32/4 = 8 \text{ cycles}$$

Timing for tr1_rx1:

$$T_{tr1\_rx1\_duration} = T_{tx\_latency} + (N_{SSM}+1)*T_{link\_latency} + N_{SSM}*T_{ssm\_latency} + T_{rx\_latency} + N_{num\_bits}/4 = 4 + (1+1)*12+1*4+3+32/4 = 43 \text{ cycles}$$

$$T_{tr1\_rx1\_end} = T_{tr1\_tx\_start} + T_{tr1\_rx1\_duration} = 0 + 43 = 43 \text{ cycles}$$

Timing for tr1_rx2:

$$T_{tr1\_rx2\_duration} = T_{tx\_latency} + (N_{SSM}+1)*T_{link\_latency} + N_{SSM}*T_{ssm\_latency} + T_{rx\_latency} + N_{num\_bits}/4 = 4 + (2+1)*12+2*4+3+32/4 = 59 \text{ cycles}$$

$$T_{tr1\_rx2\_end} = T_{tr1\_tx\_start} + T_{tr1\_rx2\_duration} = 0 + 59 = 59 \text{ cycles}$$

The execution time of the Sync FPGA data-sharing statement is:

$$T_{execution} = \max(T_{tr1\_rx1\_end}, T_{tr1\_rx2\_end}) = \max(43,59) = 59 \text{ cycles}$$

NOTE Note that if only one destination is used, for instance only Instrument 2, then $T_{execution} = T_{tr1\_rx1\_end} = 43$ cycles

Example 2: Multiple simultaneous transactions

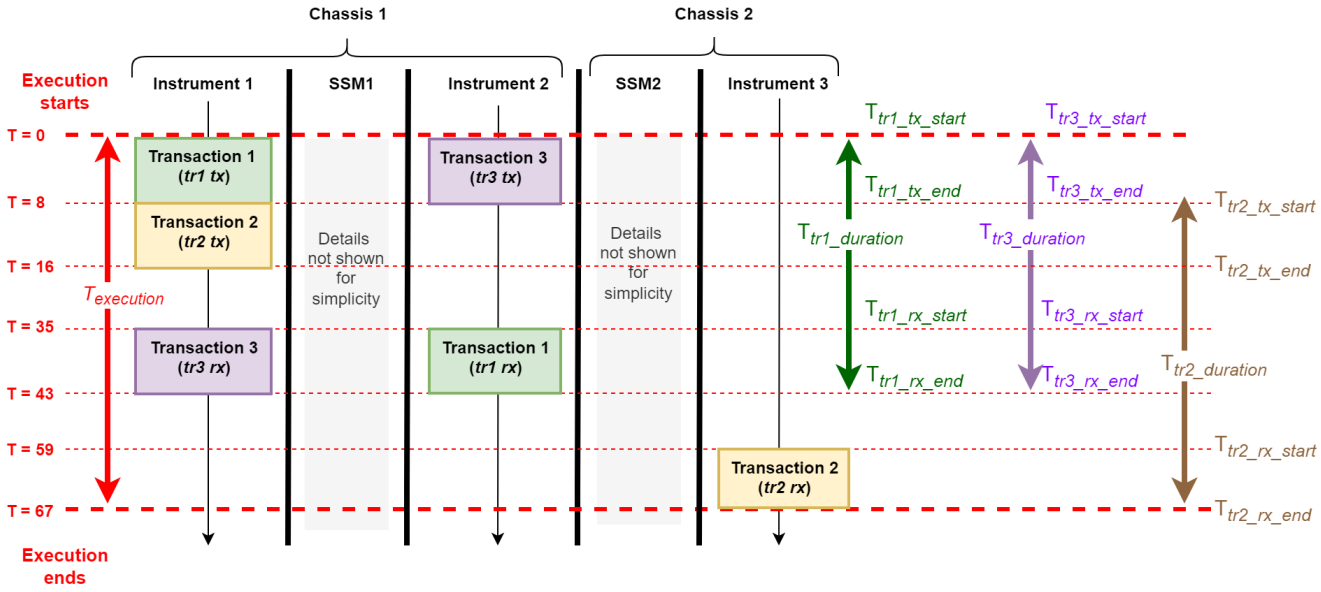
When sending data from the same transmitter, the transactions are executed in series. This means that the next transaction starts transmission as soon as the last nibble of the previous transaction is transmitted. Therefore, to estimate the execution time of the Sync FPGA Data-Sharing statement, you must delay the individual transactions accordingly, then compare their end times and you pick the highest one.

In this example there are two different transactions originating from the same Tx port in Instrument 1. In the first transaction (tr1), 32 bits is sent from Instrument1 to instrument2 and in the second transaction (tr2), a different data packet of 32 bits is sent from Instrument1 to Instrument3. At the same time as the first transaction, Instrument2 does a separate transaction3 (tr3), sending 32 bits of data to Instrument1. Unlike example 1, these are all different transactions that send different data packets.

The following code snippet shows the transactions:

```
# SyncFpgaDataSharing definition with 3 transactions to 3 destinations
#
# Sources
instrument1_tx = keysight_hvi.FdsPortAddress(source1_port, source1_address)
instrument2_tx = keysight_hvi.FdsPortAddress(source2_port, source2_address)
# Destinations
instrument1_rx = keysight_hvi.FdsPortAddress(dst1_port, dst1_address)
instrument2_rx = keysight_hvi.FdsPortAddress(dst2_port, dst2_address)
instrument3_rx = keysight_hvi.FdsPortAddress(dst3_port, dst3_address)
#
# Transaction 1
fpga_data_sharing_st.transactions.add(instrument1_tx, [instrument2_rx], num_bits_to_share)
#
# Transaction 2
fpga_data_sharing_st.transactions.add(instrument1_tx, [instrument3_rx], num_bits_to_share)
#
# Transaction 3
fpga_data_sharing_st.transactions.add(instrument2_tx, [instrument1_rx], num_bits_to_share)
```

The following diagram shows the execution time of the Sync FPGA Data-Sharing statement as well as the timings of each individual transaction:



tr1_tx, tr2_tx, and tr3_tx, refers to the transmission of the data from the transmit ports.

tr1_rx, tr2_rx, and tr3_rx, refer to reception of the data at the receive ports.

tr1_duration is the total time from the beginning of transmission (tr1_tx) to the end of reception of the data at receive point 1 (tr1_rx_end).

tr2_duration and tr3_duration are the total times for tr2 and tr3 respectively.

The timing for Transaction1 (tr1) is the same as tr1_rx1 in example 1:

$T_{tr1\_end} = T_{tr1\_start} + T_{tr1\_duration} = 0 + 43 = 43$ cycles

For Transaction2 (tr2), Instrument1 sends data to Instrument 3.

tr2 can start the data transmission as soon as the transmission of Transaction1 has ended. The duration for t2 is 59 cycles, the same as tr1_rx2 in example 1.

Therefore, the timing for Transaction2 is:

$T_{tr2\_start} = T_{tr2\_tx\_start} = T_{tr1\_tx\_end} = 8$ cycles

$T_{tr2\_end} = T_{tr2\_start} + T_{tr2\_duration} = 8 + 59 = 67$ cycles

Transaction3 (tr3) sends data from Instrument 2 to instrument 1. This is sent and received at the same time as tr1, the duration is the same as tr1_rx1 in example 1.

$T_{tr3\_end} = T_{tr3\_start} + T_{tr3\_duration} = 0 + 43 = 43$ cycles

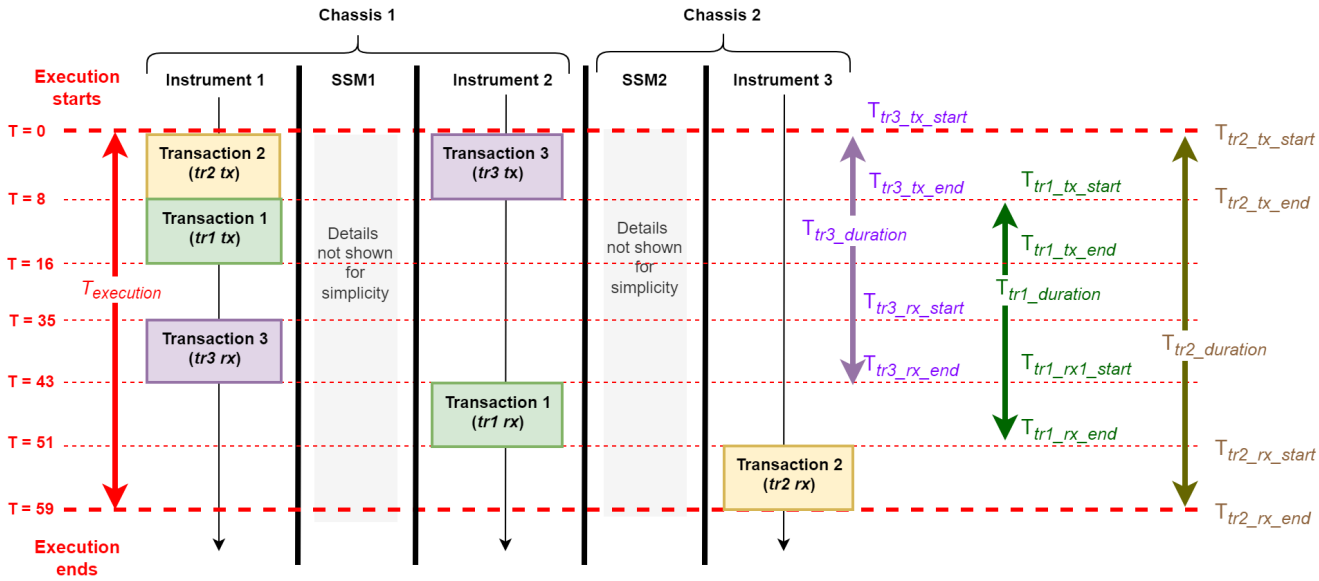
The execution time of the Sync FPGA Data-Sharing statement is:

$T_{execution\_end} = \max(T_{tr1\_end}, T_{tr2\_end}, T_{tr3\_end}) = \max(43,67,43) = 67$ cycles

Example 3: Optimizing the timing of multiple simultaneous transactions by reordering transactions

Changing the order of the transactions can affect the execution time of the statement. If example 2 is modified so tr2 is sent before tr1, some time is saved.

```
# SyncFpgaDataSharing definition with a 3 transactions to 3 destinations
# transaction 1 and 2 are reversed
#
# Sources
instrument1_tx = keysight_hvi.FdsPortAddress(source1_port, source1_address)
instrument2_tx = keysight_hvi.FdsPortAddress(source2_port, source2_address)
# Destinations
instrument1_rx = keysight_hvi.FdsPortAddress(dst1_port, dst1_address)
instrument2_rx = keysight_hvi.FdsPortAddress(dst2_port, dst2_address)
instrument3_rx = keysight_hvi.FdsPortAddress(dst3_port, dst3_address)
#
# Transaction 2
fpga_data_sharing_st.transactions.add(instrument1_tx, [instrument3_rx], num_bits_to_share)
#
# Transaction 1
fpga_data_sharing_st.transactions.add(instrument1_tx, [instrument2_rx], num_bits_to_share)
#
# Transaction 3
fpga_data_sharing_st.transactions.add(instrument2_tx, [instrument1_rx], num_bits_to_share)
```



tr1_tx, tr2_tx, and tr3_tx, refers to the transmission of the data from the transmit ports.

tr1_rx, tr2_rx, and tr3_rx, refer to reception of the data at the receive ports.

tr1_duration is the total time from the beginning of transmission (tr1_tx) to the end of reception of the data at receive point 1 (tr1_rx_end).

tr2_duration and tr3_duration are the total times for tr2 and tr3 respectively.

In this case Transaction2 starts first, The duration for t2 is 59 cycles, same as tr1_rx2 in example 1:

$T_{transaction2\_start} = T_{transaction2\_transmission\_start} = T_{execution\_start} = 0$ cycles

This time transaction2 starts at 0:

$T_{tr2\_tx\_end} = N_{num\_bits}/4 = 8$ cycles

$T_{tr2\_end} = T_{tr2\_start} + T_{tr2\_duration} = 0 + 59 = 59$ cycles

Transaction1 starts as soon as Transaction2 finishes transmission. The duration for tr1 remains at 43 cycles (same duration as tr1_rx1 in example 1), however it starts 8 cycles later and the timing becomes:

$T_{tr1\_start} = T_{tr1\_tx\_start} = T_{tr2\_tx\_end} = 8$ cycles

$T_{tr1\_end} = T_{tr1\_start} + T_{tr1\_duration} = 8 + 43 = 51$ cycles

Transaction 3 (tr3) is the same as tr3 in example 2. The duration is 43 cycles, same as tr1_rx1 in example 1. The timing is:

$T_{tr3\_end} = T_{tr3\_start} + T_{tr3\_duration} = 0 + 43 = 43$ cycles

The execution time of the Sync FPGA data-sharing statement is:

$T_{execution\_end} = \max(T_{tr2\_end}, T_{tr1\_end}, T_{tr3\_end}) = \max(59,51,43) = 59$ cycles

In this case, by starting Transaction2 first, you can save 8 cycles compared to example 2.

$T_{difference} = T_{example3\_end} - T_{example2\_end} = 67 - 59 = 8$ cycles

Example 4: Multiple simultaneous transactions with a resource conflict

Transactions are sent across different types of link between instruments. These can be DSTARB/C links between SSMs and instruments that are in the same chassis or point-to-point SystemSync links between two SSMs located on different chassis. The difference is that DSTARB/C links allow only a single data path, whereas the SystemSync connections have multiple data paths. The multiple data paths in SystemSync connections allow the SSMs to route multiple transactions simultaneously. On the contrary, DSTARB/C links can only send one transaction at a time. Different data packets cannot use the same link simultaneously and must be queued instead. Therefore, DSTARB/C links are much more prone to transaction conflicts.

In principle, when all the transactions being sent are from different Tx ports, HVI tries to execute them in parallel. If this is possible, the execution time of the Sync FPGA data-sharing statement is just the maximum of the duration of the individual transactions. However, this is not always possible because in some case there is a conflict if two or more transactions try to use the same path at the same time, for example, if they both arrive at the same Rx port at the same time. In these cases, the HVI delays some of the transactions to avoid the conflict, where any delay added is kept as small as possible.

The following example is similar to example 3, except for transaction 3 which in this case, goes from instrument 2 to instrument 3.

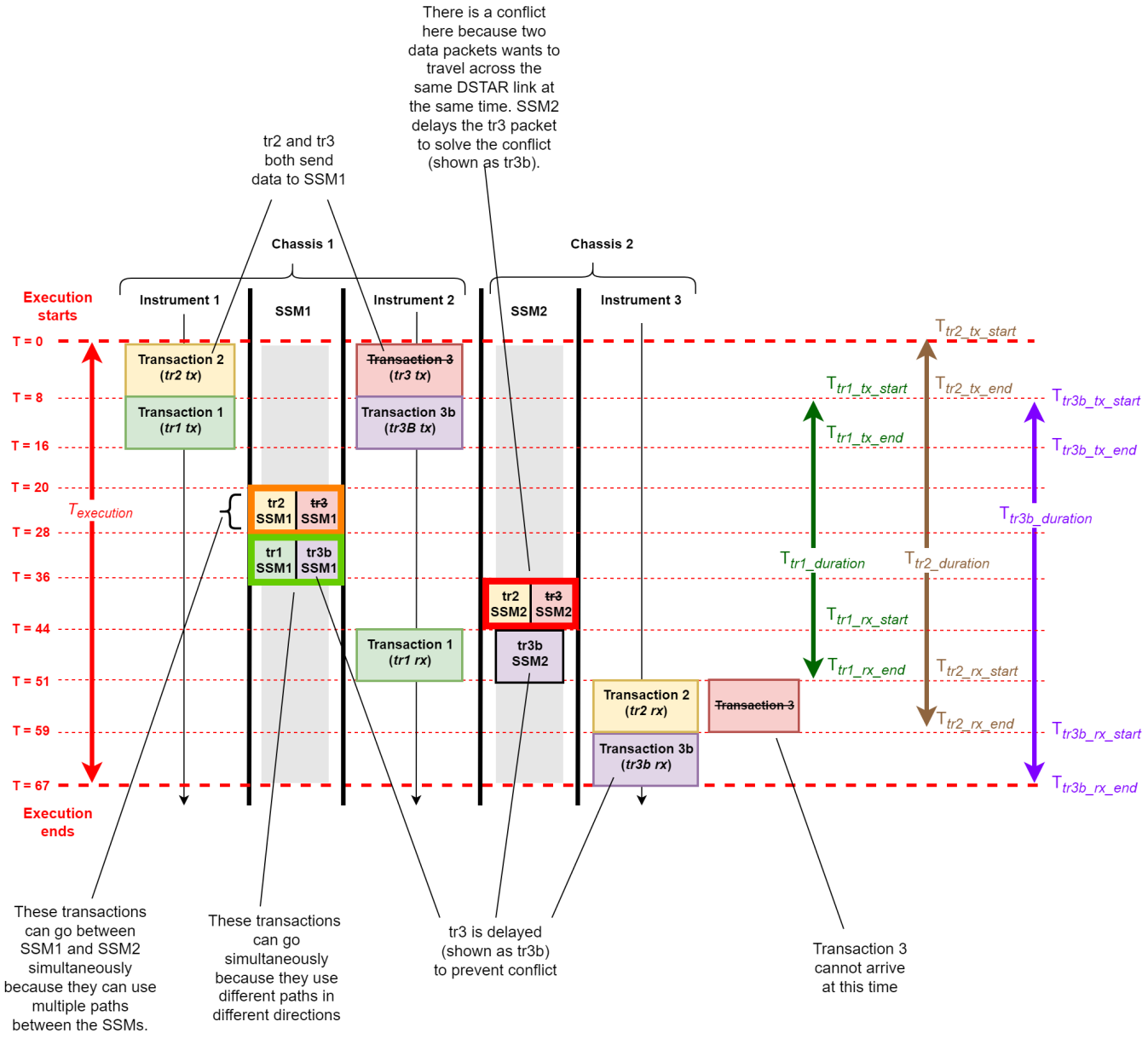
Transaction 2 (tr2) and transaction 3(tr3) both start at the same time and send data to SSM1. This is possible because both transactions are sent to different ports on SSM1.

SSM1 then sends this data to SSM2, the SSMs have multiple paths for data between them so both transactions can be sent at the same time.

SSM2 then sends the data for tr2 and tr3 to instrument 3, but there is a conflict at this point because the data from both transactions cannot be sent at the same time to the same DSTARB port. DSTARB/C links have only one data path and the packets cannot cross it simultaneously, as explained before. To resolve this conflict, the HVI delays transaction3 at compilation time. This is shown as Transaction 3b (tr3b) in the diagram. The delay enables the transaction to be sent to instrument 3 with no conflicts.

```
# SyncFpgaDataSharing definition with a 3 transactions to 2 destinations
# transaction 1 and 2 are reversed
# transaction 3 goes to instrument 3
#
# Sources
instrument1_tx = keysight_hvi.FdsPortAddress(source1_port, source1_address)
instrument2_tx = keysight_hvi.FdsPortAddress(source2_port, source2_address)
# Destinations
instrument1_rx = keysight_hvi.FdsPortAddress(dst1_port, dst1_address)
instrument2_rx = keysight_hvi.FdsPortAddress(dst2_port, dst2_address)
instrument3_rx = keysight_hvi.FdsPortAddress(dst3_port, dst3_address)
```

```
#
# Transaction 2
fpga_data_sharing_st.transactions.add(instrument1_tx, [instrument3_rx], num_bits_to_share)
#
# Transaction 1
fpga_data_sharing_st.transactions.add(instrument1_tx, [instrument2_rx], num_bits_to_share)
#
# Transaction 3 - This will be delayed
fpga_data_sharing_st.transactions.add(instrument2_tx, [instrument3_rx], num_bits_to_share)
```



tr1_tx, tr2_tx, tr3_tx, and tr3b_tx, refers to the transmission of the data from the transmit ports.

tr1_rx, tr2_rx, and tr3b_rx, refer to reception of the data at the receive ports.

tr1_duration is the total time from the beginning of transmission (tr1_tx) to the end of reception of the data at receive point 1 (tr1_rx_end).

tr2_duration and tr3b_duration are the total times for tr2 and tr3b respectively.

SSM1 and SSM2 indicate the System Synchronization Modules. In the boxes, these indicate when data is transmitted from the SSM.

In this example, first 32 bits of data from Instrument1 to Instrument3 (transaction2, tr2), and then 32 bits of data from Instrument2 to Instrument3 (transaction3, tr3). By checking the topology in the system diagram, you can see that both transactions pass through SSM1 and then SSM2 before they reach Instrument3.

To see if there is going to be a collision, calculate the time when each transaction is at the exit of SSM2, if it started at time 0.

For Transaction2, this is:

$$T_{tr2\_at\_SSM2\_start} = T_{tx\_instrument\_latency} + T_{link\_latency} + T_{ssm\_latency} + T_{link\_latency} + T_{ssm\_latency} = 4 + 12 + 4 + 12 + 4 = 36 \text{ cycles}$$

$$T_{tr2\_at\_SSM2\_end} = T_{tr2\_at\_SSM1\_start} + N_{num\_bits}/4 = 36 + 32/4 = 44 \text{ cycles}$$

For Transaction3 the numbers are the same, so there is an overlap from clock cycle 36 to 44. To resolve this, you must delay the second transaction, tr3, to start at SSM2 when tr2 ends. That is, you must delay tr3 by:

$$T_{tr3\_delay} = T_{tr2\_at\_SSM2\_end} - T_{tr3\_at\_SSM2\_start} = 44 - 36 = 8 \text{ cycles}$$

In the diagram the delayed transaction3 is shown as transaction3b (tr3b). Given this delay, you can now calculate the timing for both transactions:

For Transaction2, you don't need to change anything, so the timing is the same as in example 3:

$$T_{tr2\_end} = T_{tr2\_start} + T_{tr2\_duration} = 0 + 59 = 59 \text{ cycles}$$

As soon as Transaction2 finishes transmission, Transaction3b starts.

Transaction3b starts 8 cycles after tr2 and sends its data to SSM1 and then onto SSM2. SSM2 can then pass the data Instrument3. The duration remains at 59 cycles, so the timing is:

$$T_{tr3b\_start} = T_{tr2\_start} + 8 = 0 + 8 = 8 \text{ cycles}$$

$$T_{tr3b\_end} = T_{tr3b\_start} + T_{tr3b\_duration} = 8 + 59 = 67 \text{ cycles}$$

Transaction1 (tr1) goes through SS1, at time=8 which is also when tr3b goes through SSM1. However, these transactions go in different direction and use different ports so there is no conflict, therefore tr1 and tr3b can go ahead.

For Transaction1, the timing is the same as in example 3:

$T_{tr1\_end} = T_{tr1\_start} + T_{tr1\_duration} = 8 + 43 = 51$ cycles

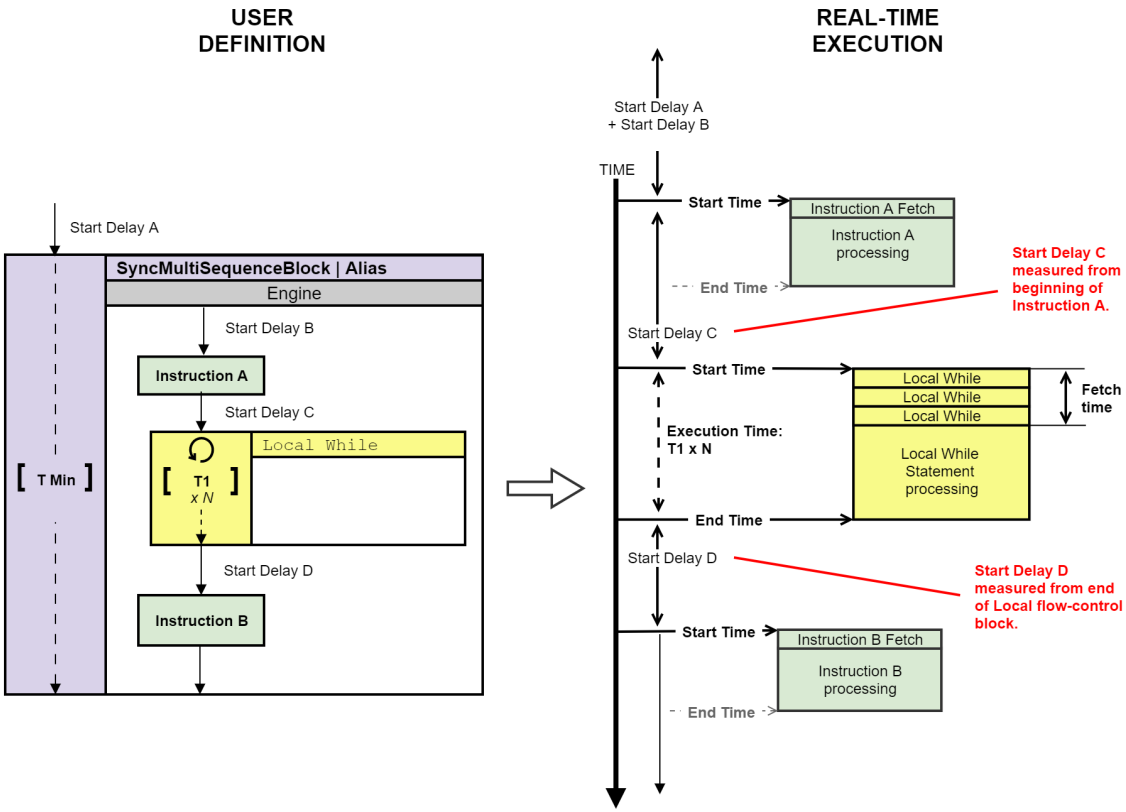
The execution time of the Sync FPGA data-sharing statement is:

$T_{execution\_end} = \max(T_{tr1\_end}, T_{tr2\_end}, T_{tr3b\_end}) = \max(51,59,67) = 67$ cycles

Local Flow-Control Statement Timing

Local flow-control statements and Sync statements consume HVI engine execution time and do not overlap their execution. When you are calculating the timing of a sequence, you must consider the execution time of these statements.

The following diagram shows the timing for a Sync Multi-sequence block that contains a pair of Local instruction statements and a Local while:



Local while

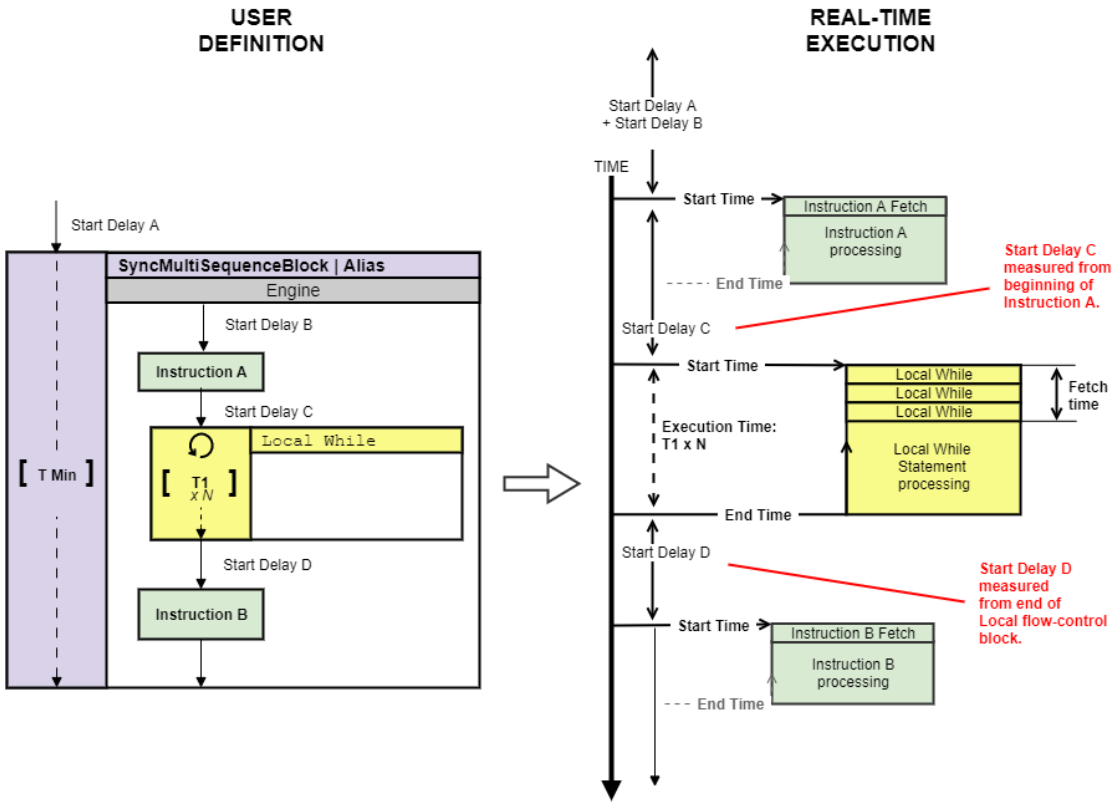
The Local while statement continues execution while a condition is met and finishes the execution when the condition is no longer met. This has the same timing as Sync while statements.

The following diagram shows a Local while statement with other instructions.

The total execution time for a Local while is T1 x N, where T1 is the iteration time and N is the number of times it iterates. The time cannot be indicated exactly on a diagram or in code because the number of iterations is not known until runtime.

For statements coming after a Local while statement, the Start delay is measured from the end of the Local while statement. In the following diagram, Start delay D is measured from the end of the Local while statement.

The dotted line indicates that the execution time of the Local while block T1 is not known at compile time.



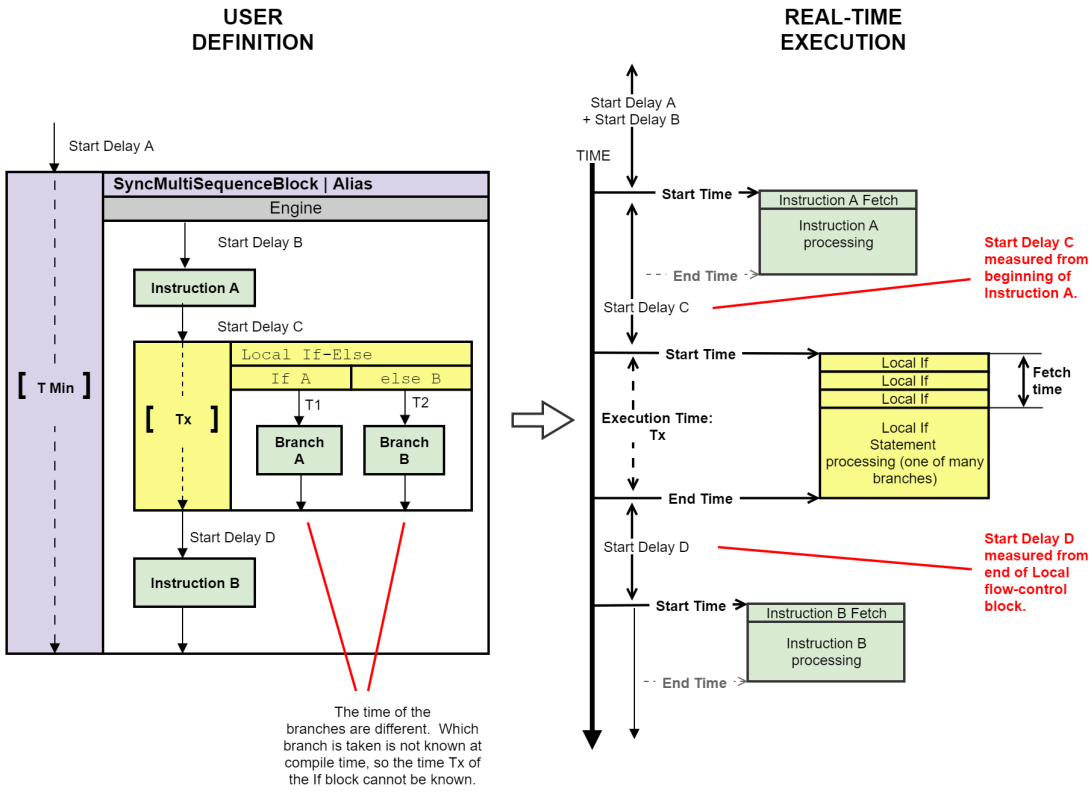
Local if

For Local if statements (if-elseif-else), the following Start delay is measured from the end of the Local if statement. The time taken is only known at runtime, so it is not possible to indicate them on a diagram or in code. This is the same as while statements.

This following diagram shows the timing of Local if statements. The Start delay D is measured from the end of the Local if statement.

The Local If has two branching options with times T1 and T2. These times can be different. Since the choice of branch is not known at compile time, the time for the Local If block cannot be known.

The line for the Local if block is dotted. This indicates that the execution time of the Local If block Tx is unknown. The time of the containing block is also therefore unknown, and it is also dotted. The time of the Sync multi-sequence block is indicated as T min.



Local If with matched branches

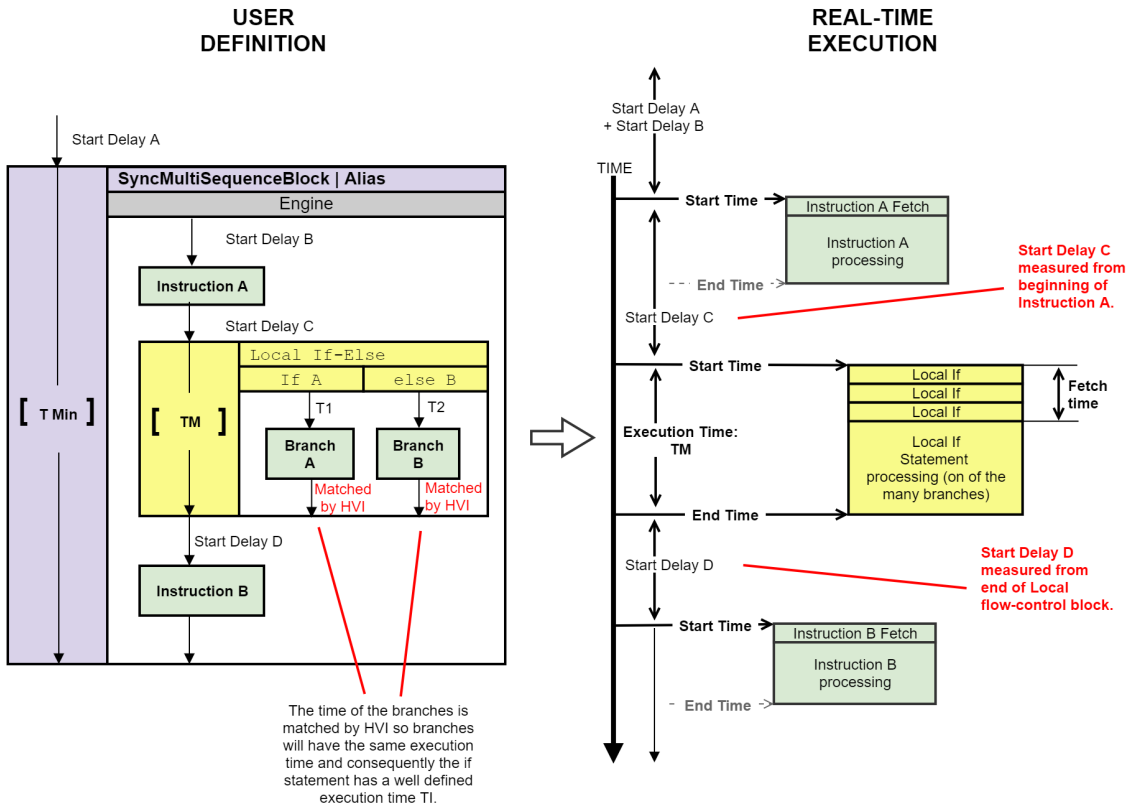
Unlike other flow-control options, the Local if statements can have different execution paths, each with different times. The matched branches option enables you to control how the HVI deals with them.

Enabling matched branches ensures the HVI synchronizes the times of the branches, so they are the same. The shorter branches get an additional delay added when they are finished so that the durations of all the branches are equal. If the matched branches option is not enabled, the branches can end at different times, that is, they are *de-synchronized*.

In the following diagram the branches in the If and else branches are matched. This ensures the Local if ends at the same time irrespective of the branch taken.

The total branch time is marked with the time TM, this represents the matched time. The choice of branch is not known at compile time, but since the times are matched the time TM is known.

The times are known at compile time so the timelines of the local If block and the Sync multi-sequence block that contains it are both solid.



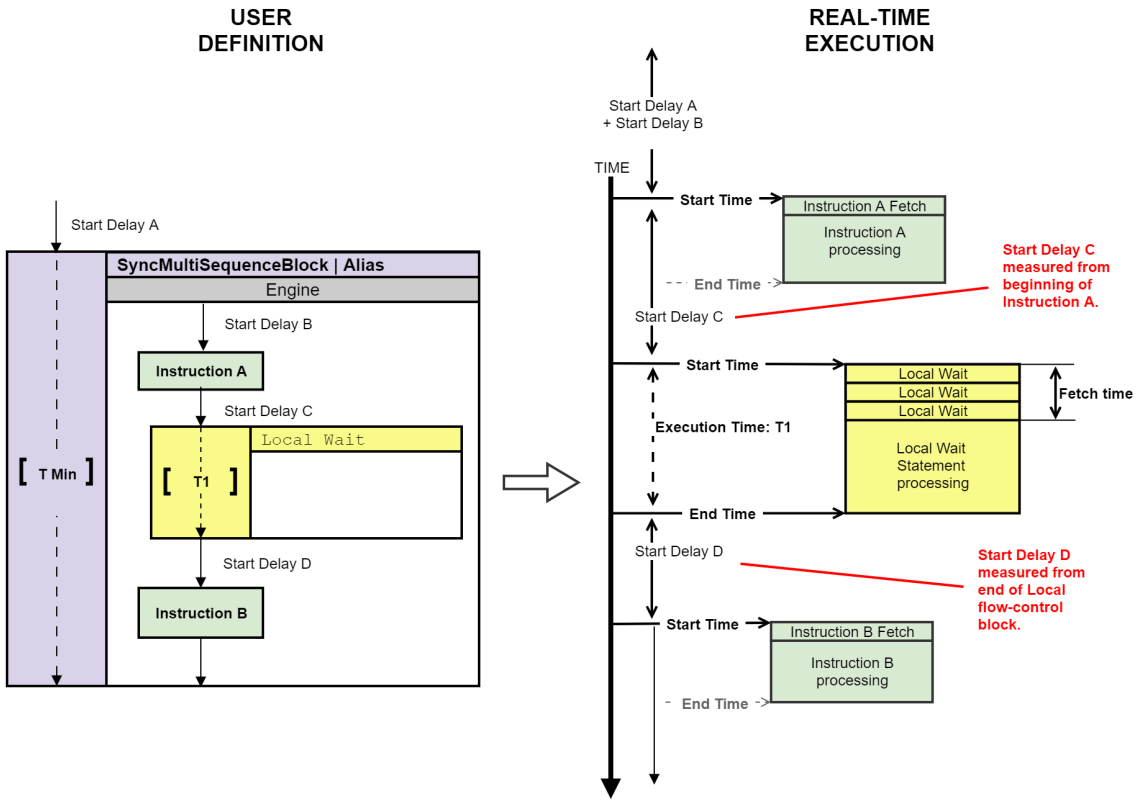
Local wait (event or time in register)

For Local wait statements, the following Start delay is measured from the end of the Local wait statement. As with Sync while statements, the time taken is only known at runtime, so it is not possible to indicate them on a diagram or in code.

The following diagram shows the timing of a Local wait statement. The following Start delay D is measured from the end of the Local wait statement.

The execution time of the Local wait statement T1 is not known at compile time, this is indicated by the dotted line.

The time of the Sync multi-sequence block is indicated as T min. The dotted line indicates an unknown time.



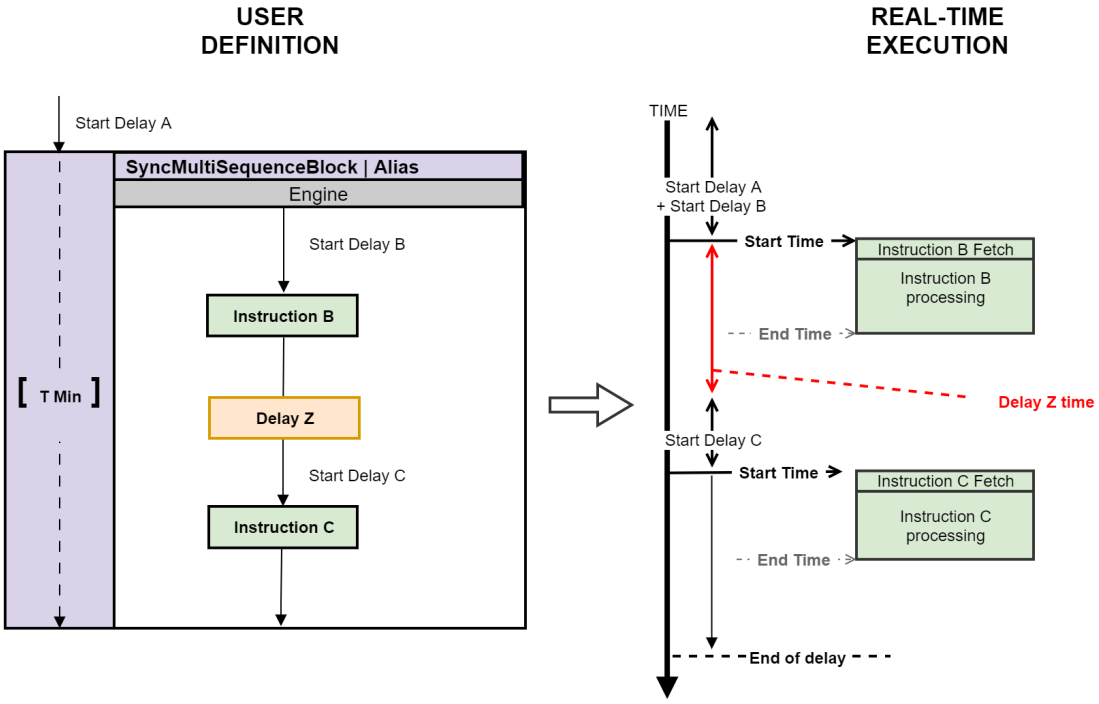
Local delay statement

The Local delay statement delays the execution of a local sequence for a time you specify. The default unit is nanoseconds but the delay is specified in any unit of seconds. The delay is fixed and cannot be changed during HVI execution, so the delay value must be known at the time of creating the HVI sequence.

The delay statement works in a similar way as the start delay statement parameter, however the difference is that the Start delay can only be specified before the other statements in a sequence. The delay statement enables you to place a fixed delay at the end of Sync multi-sequence block or a flow control statement.

Unlike a wait-for-time statement, the delay statement does not introduce a de-synchronization and therefore does not trigger a resynchronization. This therefore avoids the timing overhead introduced by the triggered re-synchronization point.

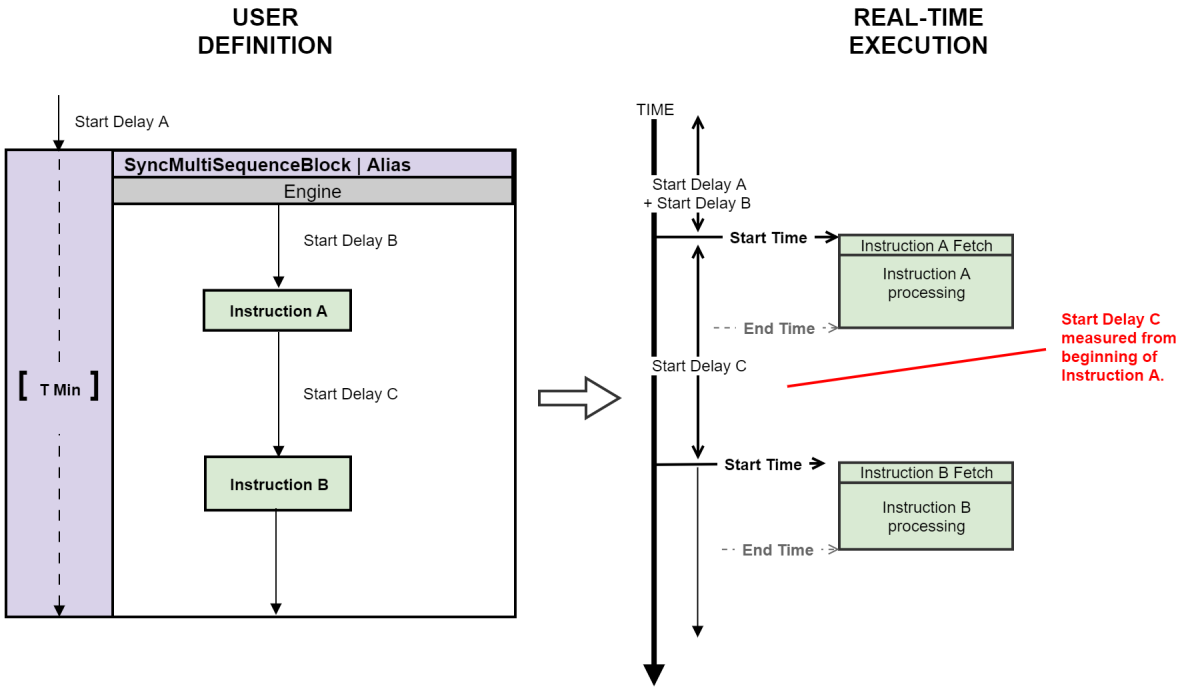
The following diagram shows the timing of a delay statement Delay Z:


USER DEFINITION
REAL-TIME EXECUTION
Start Delay A
SyncMultiSequenceBlock | Alias
Engine
Start Delay B
Instruction B
[T Min]
Delay Z
Start Delay C
Instruction C
TIME
Start Delay A + Start Delay B
Start Time
Instruction B Fetch
Instruction B processing
End Time
Delay Z time
Start Delay C
Start Time
Instruction C Fetch
Instruction C processing
End Time
End of delay

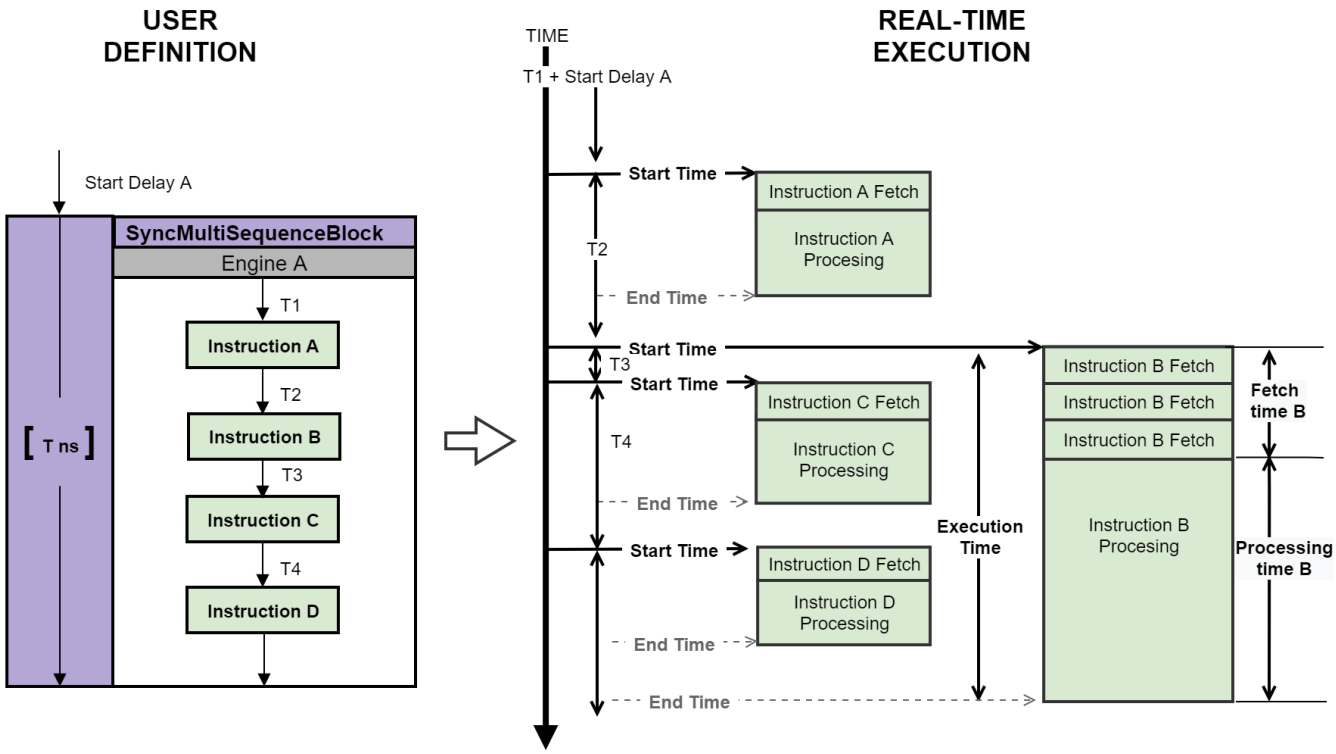
Local Instruction Timing

The following section explains with diagrams Local instruction timing.

For Local instructions, the Start delay of the next instruction is measured from the start of the current instruction. The following diagram shows two instructions and their timing:



The processing of Local instructions can be overlapped, that is, instructions can be processed in parallel inside the HVI Engine. Under specific conditions, instructions can be fetched, dispatched, and executed at the same time. This is because of the intrinsic parallel execution capability of the HVI Engine. This capability offers better performance and increased flexibility for instruction sequencing. The following diagram shows an example of overlapping instructions. Instruction B has an overlap in its fetching cycles with instruction C and an overlap during its processing with instruction D. It also shows that an instruction can start and finish while another instruction is already executing.


USER DEFINITION
Start Delay A
SyncMultiSequenceBlock
Engine A
T1
Instruction A
T2
Instruction B
T3
Instruction C
T4
Instruction D
[T ns]
REAL-TIME EXECUTION
TIME
T1 + Start Delay A
Start Time
Instruction A Fetch
Instruction A Procesing
T2
End Time
Start Time
T3
Start Time
Instruction C Fetch
Instruction C Processing
T4
End Time
Start Time
Instruction D Fetch
Instruction D Processing
End Time
End Time
Execution Time
Instruction B Fetch
Instruction B Fetch
Instruction B Fetch
Instruction B Procesing
Fetch time B
Processing time B

Instruction position

This section provides a high-level description of the concept of *instruction position*.

NOTE This is provided for your information, you are not typically required to program an HVI at this level of performance.

Instructions are broken into internal-instructions that the compiler maps onto the HVI engine hardware. During one HVI engine cycle, the HVI engine can fetch, dispatch and execute multiple instructions in parallel.

Instructions can be scheduled for execution together, however, depending on the Instructions involved, this is not always possible because of the inner structure of the HVI engine. To understand why, you must understand the concept of instruction position.

An HVI engine is a processor with a set of execution pipelines, each of which has a numbered *position*. The individual internal-instructions are mapped across the different pipeline positions for execution.

For parallel instruction fetching to be possible, the internal-instructions must use different positions inside the instruction register of the HVI Engine. If two internal-instructions are using overlapping positions, then they cannot be fetched in parallel. The positions where each internal-instruction is to be mapped depends on the instruction. This means the hardware can only execute certain internal-instruction in specific positions. The internal-instructions are mapped by the compiler. This process is not user programmable.

A table with the per-instruction mapping is provided in the documentation for each instrument. See Local Instruction Statement Timing Tables for the table for HVI-native instructions.

The following diagram provides an example table.

	Position										
Instruction	1	2	3	4	5	6	7	8	9	10	n (n > 10)
A					5 - 7						
B	1 - 4										
C								8 - 9			
D					5 - 7			8 - 10			
E	1 - 7										

From the table, you can see that:

- Instruction A can be mapped, one at a time, to positions 5 to 7.
- Instruction B can be mapped, one at a time, to positions 1 to 4.
- Instruction C can be mapped, one at a time, to positions 8 to 10.
- Instruction D can be mapped, two at a time, to positions 5 to 7, positions 8 to 10, or both.
- Instruction E can be mapped, one at a time, to positions 1 to 7.

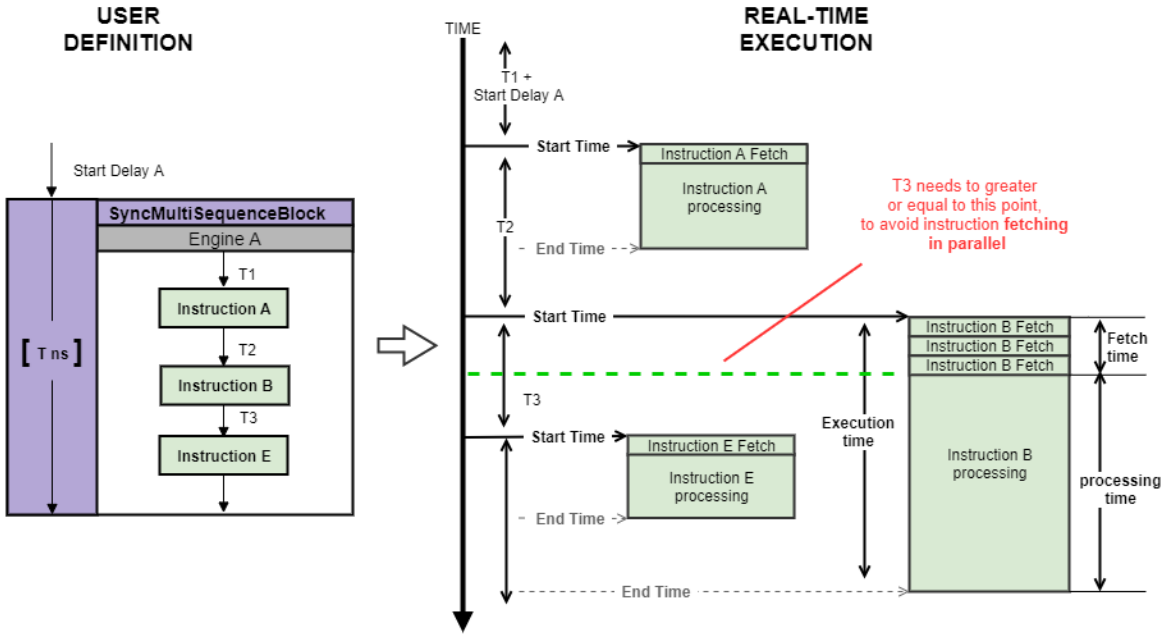
At compile time, HVI maps the instructions to be executed to their respective supported positions. If an instruction cannot be mapped to its supported position because another instruction is already mapped there, HVI generates an error and informs the user. For example:

- If an instruction is A is followed by a second instruction A at the same time, HVI assigns the first instruction A to positions 5-7, but generates an error with the second instruction A because positions 5-7 are already used.
- If an instruction D is followed by another instruction D at the same time, HVI will assign the first instruction D to positions 5-7 and will then assign the second instruction D to positions 8-10. However, if there is a third instruction D to be fetched at the same time, HVI generates an error because neither possible position for D are available for the third instruction.
- If there are instructions A, B and C at the same time, HVI assigns them to positions 5-7, 1-4 and 8-10, respectively, without any issue.
- If there are instructions A, B and D at the same time, HVI assigns them to positions 5-7, 1-4 and 8-10, respectively, without any issue. If, however, the order was B, D and A, then HVI assigns B to positions 1-4, D to positions 5-7 and, then, HVI generates an error because positions 5-7 are not be available for instruction A.
- If there is an instruction E, then if it is fetched at the same time with any of the instructions A, B or E, then HVI generates an error. However, if it is fetched in parallel with C or D, then there will be no issue.

NOTE When there is no fetching in parallel, An HVI engine is capable of executing instructions in parallel irrespective of their instruction position.

Overlapping instruction execution

The following diagram shows Instruction B and Instruction E are executed in parallel, even though they are using the conflicting positions in the instruction register (positions 1-4 are overlapping as seen in the table earlier). This is possible, as long as the Start delay T3 for instruction E is such that its fetch cycle does not coincide with the fetch cycles of instruction B. The green dotted line indicates the minimum extent that T3 should have.



Overlapping instruction fetching

An HVI engine is capable of fetching and executing multiple instructions in parallel, providing their instruction positions are not overlapping. Most instructions have only 1 fetch cycle, but it is possible for instructions to require multiple fetch cycles. Refer to the instructions timing tables for details on the fetch cycles of the different instructions.

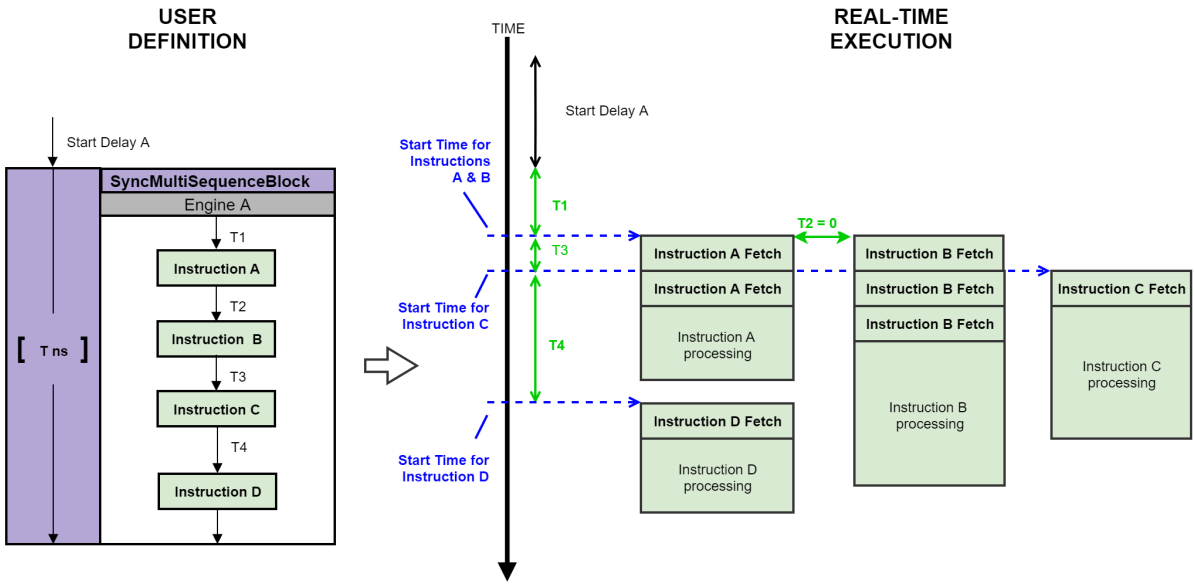
The following figures show examples of instruction fetching in parallel. For the instruction positions that are being used by each instruction, the values are from the previously defined example table.

Example A. In this example it is assumed that:

- Start delay T1 > 0 cycles.
- T2 = 0 cycles.
- T3 = 1 cycle.
- T4 > 3 cycles.

At real-time execution, after the T1 delay has passed, Instruction A and Instruction B are fetched at the same time, since the Start delay T2 for instruction B is equal to 0. Then, after one cycle, that is, the Start delay T3, instruction C is fetched before the fetching of Instructions A and B is completed. Finally, after delay T4 from the beginning of instruction C, instruction D is fetched.

As shown in the diagram, instructions A and B are fetched in parallel for 2 engine cycles and instructions A, B and C are fetched in parallel for 1 engine cycle. Looking at the table, instructions A, B and C can be fetched in parallel as they are not using the same positions. Instruction D is fetched later, so there is no conflict in the available positions.

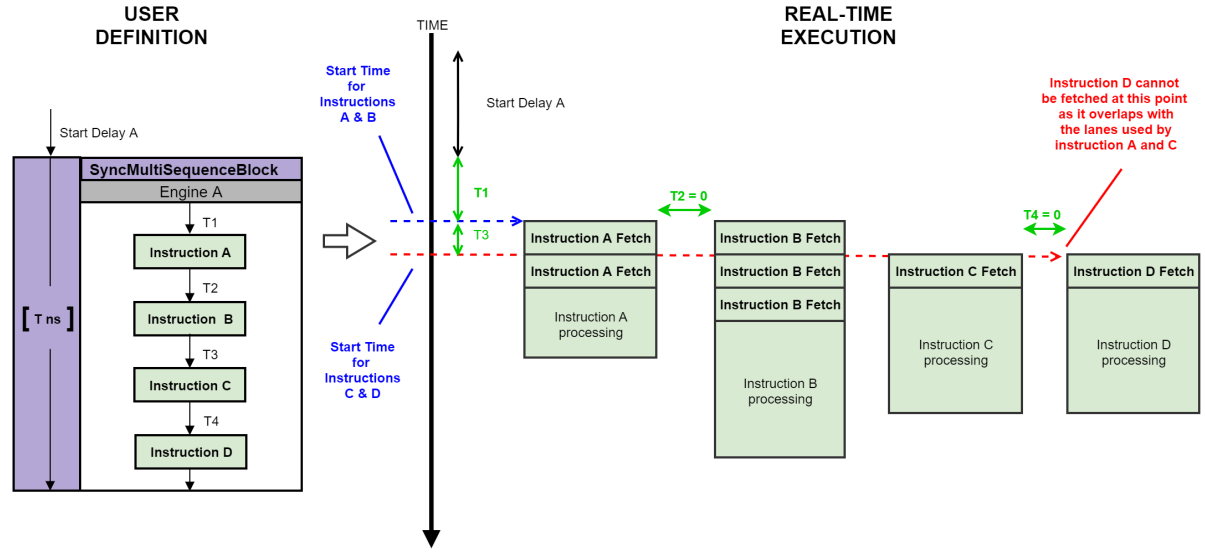


Example B. In this example start delay T1 > 0 cycles, T2 = 0 cycles, T3 = 1 cycle, and T4 = 0 cycles.

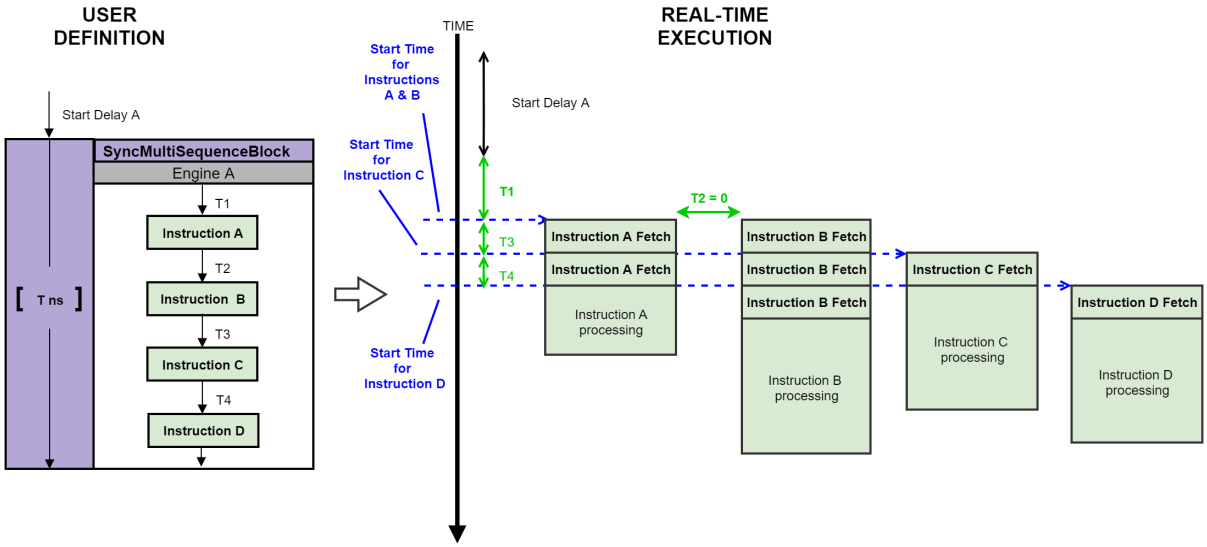
At real-time execution, after the T1 delay has passed, instructions A and Instruction B are fetched at the same time because the Start delay T2 for instruction B is equal to 0.

Then, after one cycle, that is the Start delay T3, instruction C is being fetched and at the same time (T4 = 0) instruction D is fetched.

Compared to the previous example, in this case, instruction D cannot be placed to either positions 5-7 (assigned to instruction A) or positions 8-10 (assigned to instruction C), so it is not possible to fetch instruction D at the same time. as A and C. This example generates an error during the HVI compilation.



One way to fix the issue is to increase the Start delay T4 of instruction D so that it is not fetched at the same time as instruction A and C. This can be done by increasing T4 by at least 1 cycle. This is shown in the following figure:

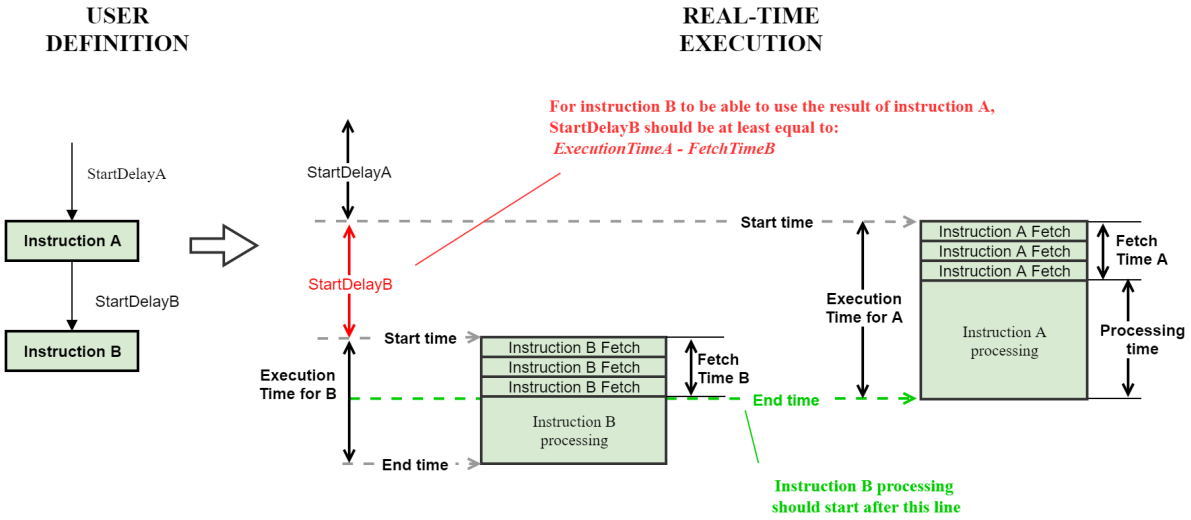


Overlapping instruction execution with result dependencies

HVI is capable of processing instructions in parallel. This is a powerful capability, but it can lead to unexpected results when there are dependencies between the instructions, that is, when one instruction depends on the result of the other. For example, an instruction might update the value of an HVI register and the following instruction might need to use that updated register value. To avoid unexpected results, the user needs to ensure that the delay between the independent and the dependent instructions is big enough so that the processing of the independent (Instr1) is completed before or when the processing of the dependent instruction (Instr2) start. The minimum delay to achieve this can be expressed with the following formula:

```
MinDelay_Instr1_to_Instr2 = Instr1_ExecutionTime - Instr2_FetchTime
```

The following diagram shows an example with two local instruction statements and the timing when executed by the HVI engine. Assuming that instruction B is using the result of instruction A, you must ensure that the value of StartDelayB is greater or equal to the Processing Time of instruction A, minus the Fetch Time of instruction B. This way, the processing of instruction B will start after the end of processing of instruction A.



NOTE It is important to consider the effects of overlapping instructions with dependencies, because HVI does not track dependencies. This is because in some cases it is desirable to implement pipelines of operations and exploit the fact that the next instruction uses the previous value of a register, before the previous operation is completed. It is your responsibility to ensure you have specified sufficient Start delay between instructions with dependencies.

Overlapping instruction execution with Sync or Flow-Control statement with result dependencies

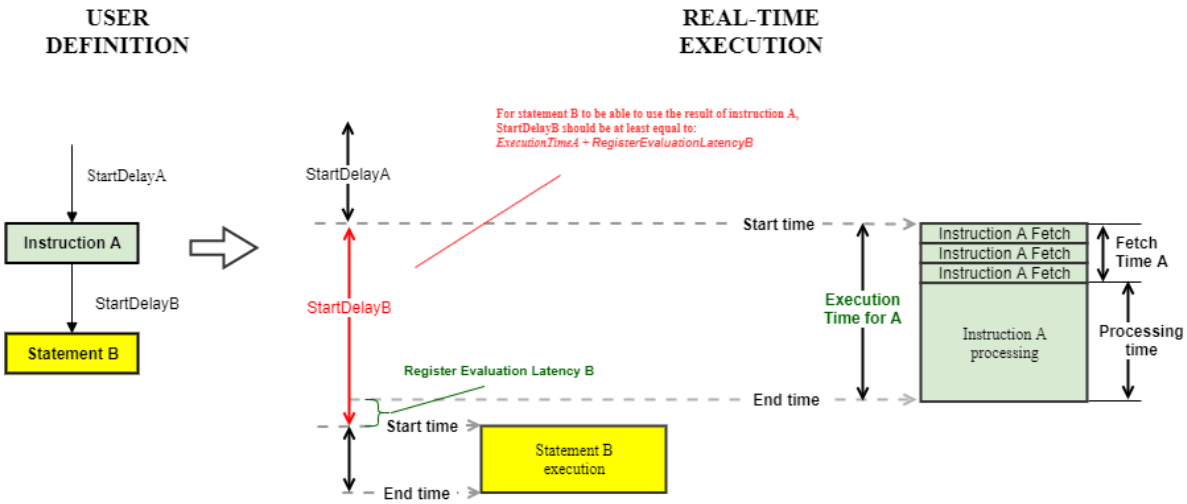
For the case that the result of an instruction is used from a sync or a flow-control statement (e.g. register used in the condition of a While or a Sync While), the RegisterEvaluationLatency of that statement need to be taken into account. Therefore, the formula is updated to:

```
MinDelay_Instr_to_Statement = Instr_ExecutionTime + Statement_RegisterEvaluationLatency
```

NOTE

If the flow-control (or sync) statement comes right after the instruction from which it needs the result, this imposes a minimum value for the StartDelay of that statement. The final StartDelay to be used should be the maximum between the MinDelay calculated with the previous formula and the MinStartDelay applicable (see Minimum Start Delay Calculation for Flow-Control and Sync Statement)

If there are more statements/instructions between the flow-control (or sync) statement and the instruction from which it needs the result, then the MinDelay imposes a minimum to the sum of the StartDelays of all the intermediate statements/instructions and the flow-control (or sync) statement.



Example cases with instruction result dependencies

The following examples show how to calculate the minimum delay required when the result of Local instructions is used by Flow-Control statements. The latency information is provided in the Timing Tables and in the instrument documentation.

Example 1: Instruction "ADD" followed by a Local if statement

In this example an `Add` instruction writes to a register and the new value of the register is used for the `if` condition.

1. `Reg1 = RegN + 10 (`Add).
2. `If(Reg1 > 10)` (the `if` uses the result of the previous `Add` instruction).

In this case, the minimal delay between the `If` and the previous `Add` using the fetch and execution timing is calculated with this equation:

```
MinDelay_If = Add_ExecutionTime + If_RegisterEvaluationLatency = 8 + 3 = 11 cycles
```

Example 2: Instruction "ADD" inside a While Statement

In this example there is an `Add` instruction that writes to a register and the new value of the register is used by the while condition.

1. `Reg1 = 0`
2. `While(Reg1 < 1)` (the While uses the result of the internal `Add` instruction).
3. `Reg1 = Reg1 + 1 (`Add).

In this case, the minimal delay between the `Add` inside the `While` and the condition check for executing one more iteration is calculated with the following equation.

```
MinDelay_While = Add_ExecutionTime + While_RegisterEvaluationLatency = 8 - 3 = 5 cycles
```

To add this extra time at the end of the internal while sequence a Delay statement can be added. The Delay Statement will need at least 4 cycles of delay which with the EndLatency of the Delay statement, will give the total the added delay of 5 cycles.

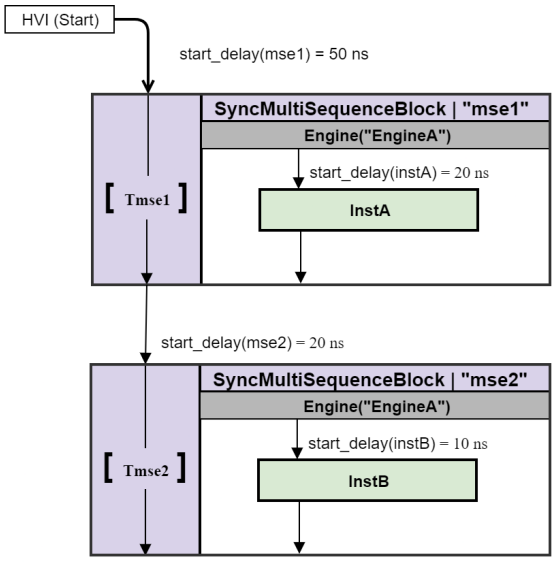
Examples of Local instruction Timing Calculation across Sync and Local Flow-Control Statements

This chapter shows basic examples of Local instruction across within Sync and Local Flow-Control statements and how the timing is calculated.

Local instruction timing across Sync Multi-Sequence Blocks example

This example shows a pair of Sync Multi-sequence blocks each with a Local instruction each. A diagram and the code and timing calculations are shown.

The following is a diagram of the example:



The code for the example:

```
mse1 = sequencer.sync_sequence.add_sync_multi_sequence_block("mse1", 50)
seq = mse1.sequences['EngineA']
instA = seq.add_instruction("instA", 20, seq.instruction_set.trigger_write.id)
#
mse2 = sequencer.sync_sequence.add_sync_multi_sequence_block("mse2", 20)
seq = mse2.sequences['EngineA']
instB = seq.add_instruction("instB", 10, seq.instruction_set.trigger_write.id)
```

The timing calculations for the example:

InstA Execution Start time from HVI-Start (`InstA_start`):

```
InstA_start = start_delay(mse1) + start_delay(instA) = 50ns + 20ns = 70ns
```

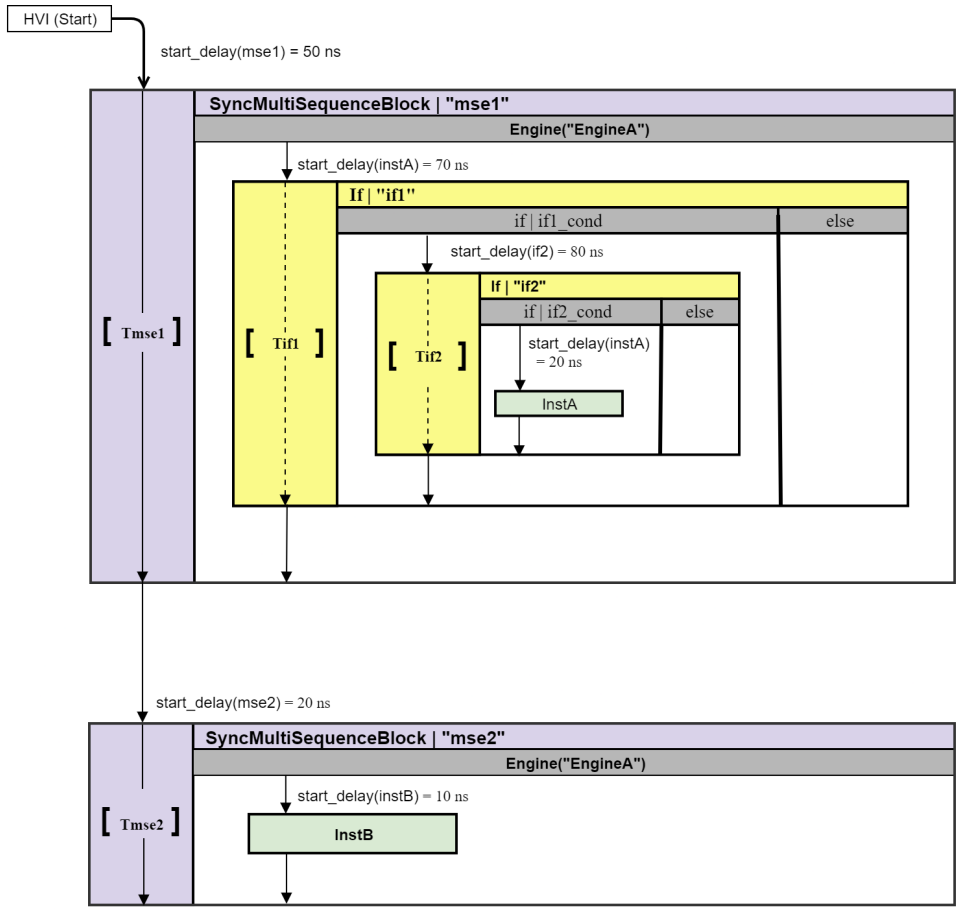
Time from InstA to InstB (`T_InstA_InstB`) :

```
T_InstA_InstB = start_delay(mse2) + start_delay(instB) = 20ns + 10ns = 30ns
```

Local instruction timing across Sync Multi-Sequence Blocks and Local if example

This example shows cascaded Local if statements within a Sync multi-sequence block followed by a Local instruction in a Sync multi-sequence block. The code and timing calculations are also shown:

The following is a diagram of the example:



The code for the example:

```
mse1 = sequencer.sync_sequence.add_sync_multi_sequence_block("mse1", 50)
seq = mse1.sequences['EngineA']
if1 = seq.add_if('if1', 70, if1_cond, True)
if1_branch_seq = if1.if_branch.sequence
if2 = if1_branch_seq.add_if('if2', 80, if2_cond, True)
if2_branch_seq = if2.if_branch.sequence
instA = if2_branch_seq.add_instruction("instA", 20, seq.instruction_set.trigger_write.id)
#
mse2 = sequencer.sync_sequence.add_sync_multi_sequence_block("mse2", 20)
seq = mse2.sequences['EngineA']
instB = seq.add_instruction("instB", 10, seq.instruction_set.trigger_write.id)
```

The timing calculations for the example:

The formula to calculate the `InstA` execution start time from HVI-Start, `InstA_start` is:

```
InstA_start = start_delay(mse1) + start_delay(if1) + start_delay(if2) + start_delay(instA) =
50ns + 70ns + 80ns + 20ns = 220ns
```

The formula to calculate time from InstA to InstB, `T_InstA_InstB` is:

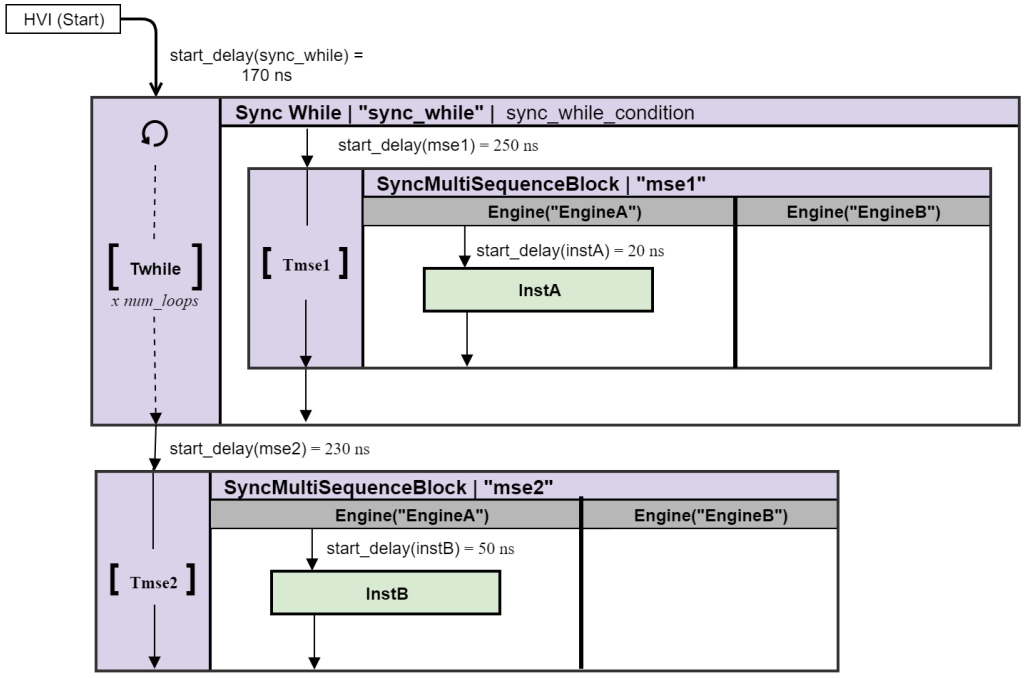
```
T_InstA_InstB = start_delay(mse2) + start_delay(instB) = 20ns + 10ns = 30ns
```

NOTE The `end_latency(mse1)` is accounted for in the `start_delay(mse2)`, this imposes a minimum value.

Local instruction timing across Sync While and Sync Multi-Sequence Blocks example

This example shows how time is calculated for a Sync while statement that contains a Sync multi-sequence block and a single instruction:

The following diagram shows the example:



The following block shows the example code:

```
sync_while = sequencer.sync_sequence.add_sync_while('sync_while', 170, sync_while_condition)
mse1_sequence = sync_while.sync_sequence.add_sync_multi_sequence_block("mse1", 250).sequences
['EngineA']
instA = mse1_sequence.add_instruction("InstA", 20, seq.instruction_set.assign.id)
```

```
#
mse2_sequence = sequencer.sync_sequence.add_sync_multi_sequence_block("mse2", 230).sequences
['EngineA']
instB = mse2_sequence.add_instruction("InstB", 50, seq.instruction_set.assign.id)
```

The following are the equations used to calculate the timing in the example:

InstA Execution Start time from HVI-Start, `InstA_start`:

```
InstA_start = start_delay(sync_while) + start_delay(mse1)  + start_delay(instA) = 170ns +
250ns + 20ns = 440ns
```

Sync multi-sequence block Execution time, `Tmse1`:

```
Tmse1 = SequenceTime =  20ns
```

Sync while Execution time for 1 loop when looping, `Twhile_loop`:

```
Twhile_loop = Twhile  = {start_delay(mse1) + Tmse1}  = {250ns + 20ns}  = 270ns
```

Time from InstA to InstA in consecutive repetitions, `Tloop_InstA`:

```
Tloop_InstA = Twhile_loop
```

Time from InstA to InstB (the last Sync while execution), `T_InstA_InstB`:

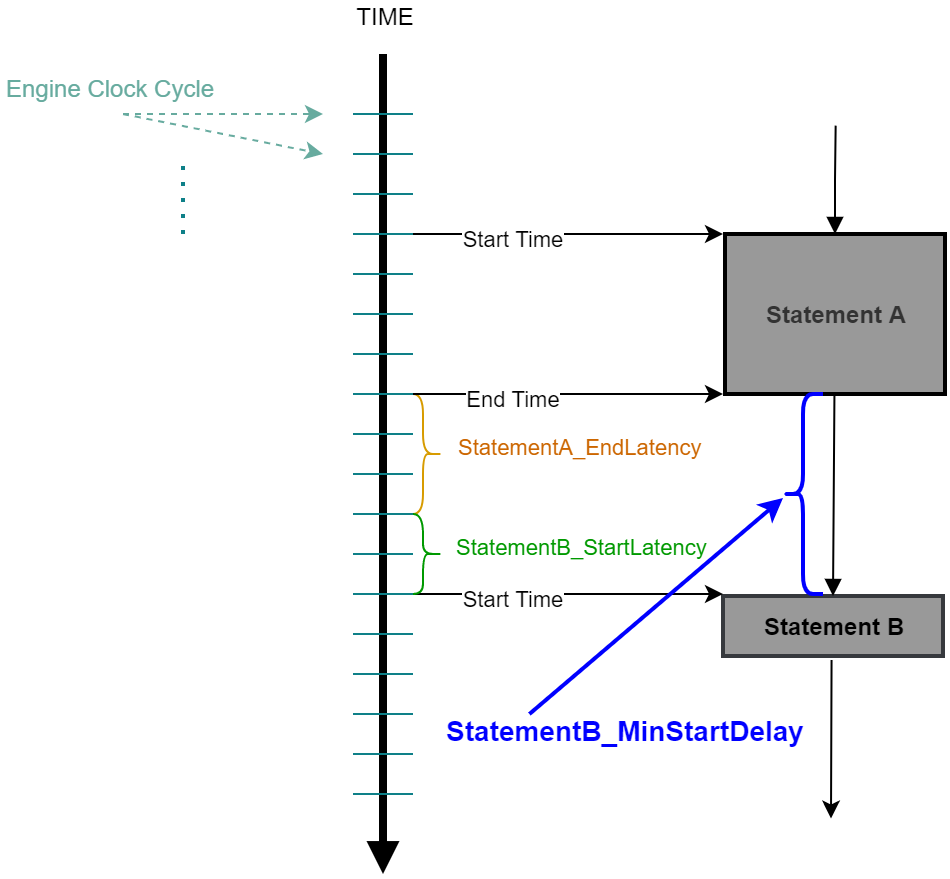
```
T_InstA_InstB = start_delay(mse2) + start_delay(instB) = 230ns + 50ns = 280ns
```

NOTE The `end_latency(sync_while)` is accounted for in the `start_delay(mse2).` This imposes a minimum value.

Minimum Start Delay Calculation for Flow-Control and Sync Statement

To calculate the minimum valid start delay for a given flow-control or sync statement, the general rule is to add the End-Latency of the previous statement with the Start-Latency of the current statement:

$$Statement\_MinStartDelay_{EngineX} = PreviousStatement\_EndLatency_{EngineX} + Statement\_StartLatency_{EngineX}$$



From this general rule, we can distinguish 2 subcases:

- First statement of the global HVI sequence: Use the End-Latency of the HVI Start.
- First statement of a sub sequence: Instead of the End-Latency use the Entry-Latency of the parent statement.

The values for each latency can be found in the Timing tables for Local Flow-Control Statements and Sync Statements.

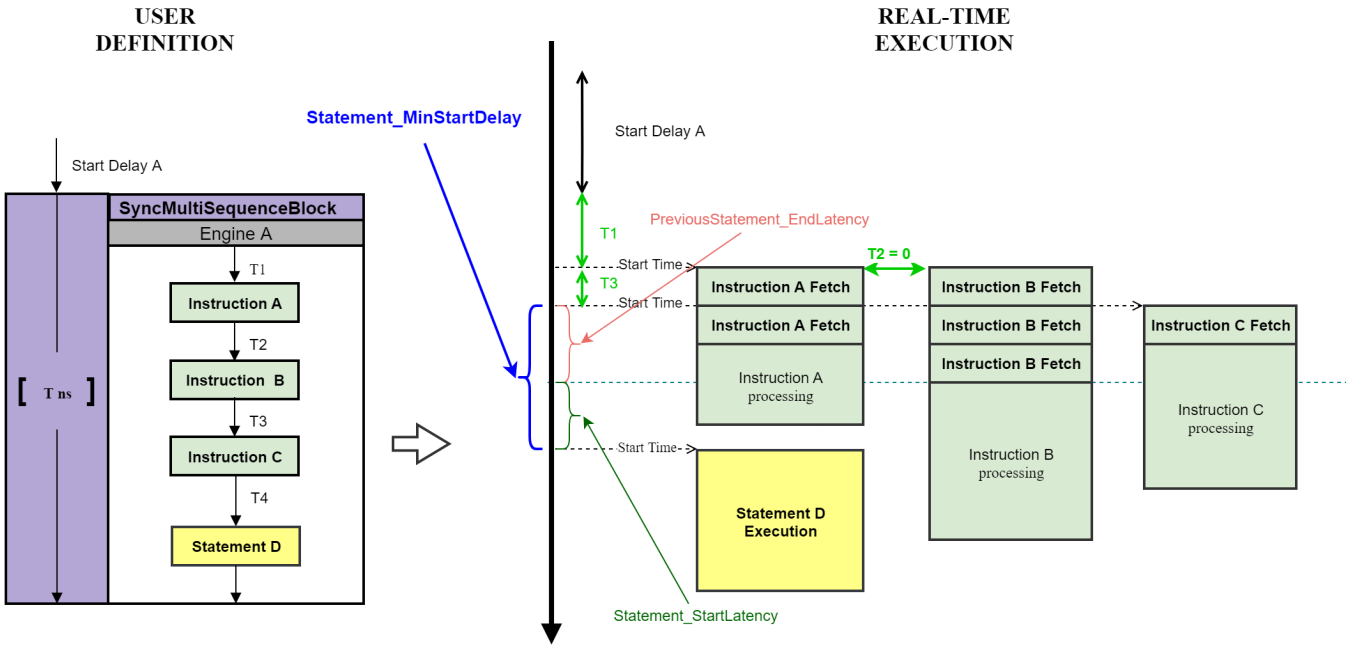
Minimum start delay for Local Flow-Control Statements

The local flow-control statements are following the general rule described above to calculate the minimum start delay.

Minimum start delay after Local instructions

As explained earlier, to find out the minimum Start-Delay of a statement, it is important to know the end-latency of the previous statement. If the previous statement of a flow-control statement is one or more local instructions, the end-latency is calculated as the remaining fetch-cycles of all the local instructions starting from the beginning of the last instruction.

This can be seen in the following figure. Starting from the beginning of Instruction C (last Local instruction), we calculate the remaining fetch cycles of all the instructions executed before Statement D. From the picture we see that there is one fetch cycle where instructions A, B, C are executed together and then, one more fetch cycle for instruction B. So, in total, there are 2 remaining fetch cycles, therefore, the end-latency is 2 cycles.



End-Latency of Local flow-control statements with internal sequence (If and While)

The End-Latency of Local flow-control statements with internal sequence, like If and While statements, depend on the End-Latency of the last statement of their internal sequences. When the last statement of the internal sequence is one or more local instructions, for the calculation of the End-Latency, the same principle applies as described in the previous section, i.e. the end-latency is cal-

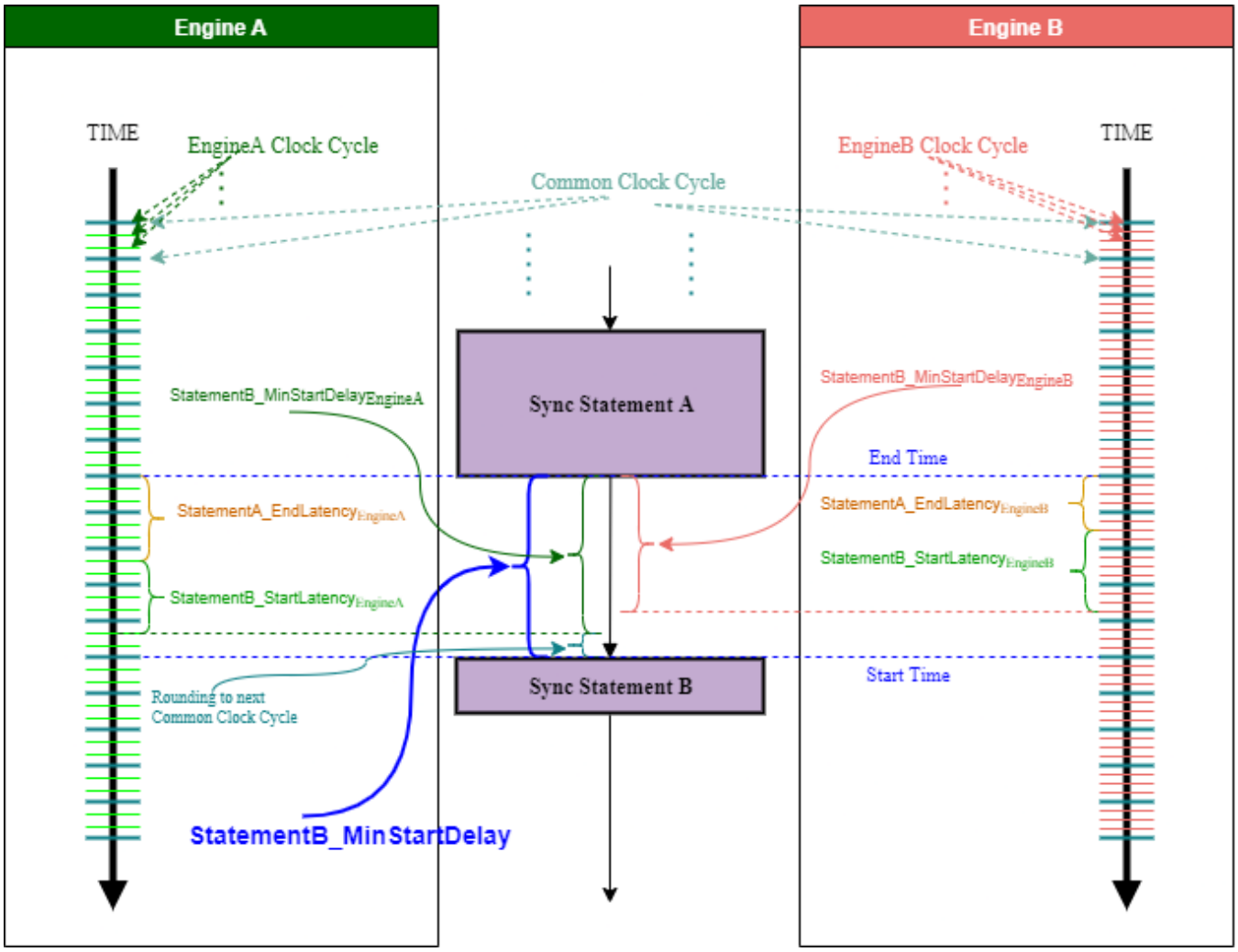
culated as the remaining fetch-cycles of all the local instructions starting from the beginning of the last instruction.

Minimum start delay for Sync Statements

For the Sync Statements, the minimum start delay is the maximum of all the minimum start delays calculated for each HVI engine rounded to the next multiple of the *HVI Common Clock* period:

$$Statement\_MinStartDelay = round_{ncc}(max\{Statement\_MinStartDelay_{Engine1}, Statement\_MinStartDelay_{Engine2}, ..., Statement\_MinStartDelay_{EngineN}\})$$

In the following diagram, we show graphically the minimum start delay calculation process between two sync statements in a system that has two engines with different frequencies, EngineA and EngineB:



Example: Minimum Start delay from HVI Start to Sync While

In this example we show how to calculate the minimum start delay value acceptable for a Sync While statement that is placed as the first statement of the HVI root SyncSequence.


```
# Create system definition object
system_definition = keysight_hvi.SystemDefinition("MySystemDefinition")
system_definition.engines.add(instrument_1.hvi.engines.leader, "HVI_Engine_1")
system_definition.engines.add(instrument_1.hvi.engines.leader, "HVI_Engine_2")
...
# Create sequencer object
sequencer = keysight_hvi.Sequencer("MySequencer", system_definition)
# Iteration counter register for "HVI_Engine_2"iteration_counter = sequencer.sync_
sequence.scopes["HVI_Engine_2"].registers.add("MyRegister", keysight_hvi.RegisterSize.SHORT)
iteration_counter.initial_value = 0
# Define sync while condition
num_loops = 5
sync_while_condition = keysight_hvi.Condition.register_comparison(iteration_counter, keysight_
hvi.ComparisonOperator.LESS_THAN, num_loops)
SyncWhile_MinStartDelay = ... # The calculation for the minimum is explained below
sequencer.sync_sequence.add_sync_while("MySyncWhileStatement", SyncWhile_MinStartDelay,  sync_
while_condition)
```

We assume the following values to be used in the calculations:

Variable	Value	Description
#Register_Conditions	1	*In the example above, we used only one register condition for the Sync While*
$Engine1_{Period}$	5 ns	*We assume HVI Engine 1 to run at frequency of 200 MHz which results in a 5 ns period*
$Engine2_{Period}$	$3.33\overline{3}$ ns	*We assume HVI Engine 2 to run at frequency of 300 MHz which results in a $3.33\overline{3}$ ns period*
$HVI\_Leader\_Engine\_Clock_{Period}$	$3.33\overline{3}$ ns	*The leader engine for the Sync While in the example is HVI Engine 2, since the register used in the condition of the Sync While belongs to that engine. So, the frequency of the leader engine is that of engine 2*
$HVI\_Common\_Clock_{Period}$	10 ns	*This is the result of the GCD of the engines included in HVI: $LCM\{Engine1_{Period}, Engine2_{Period}\}$*

Engine 1:

Using the timing table of HVI Start, we calculate the End Latency for this engine:

HVI Start		
Parameter	Time (cycles)	Result (n s)
$\texttt{HviStart\_EndLatency}_{\texttt{Engine1}}$	2	10

Using the timing table of the Sync While statement, we calculate the Start Latency for this engine:

Sync While		
Parameter	Time (cycles)	Result (n s)
$\texttt{SyncWhile\_StartLatency}_{\texttt{Engine1}}$	Follower Engine: 2	10

Therefore, the minimum for this engine is:

$$\textbf{SyncWhile\_MinStartDelay}_{\textbf{Engine1}} = \texttt{HviStart\_EndLatency}_{\texttt{Engine1}} + \texttt{SyncWhile\_StartLatency}_{\texttt{Engine1}} = \textbf{20 ns}$$

Engine 2:

Using the timing table of HVI Start, we calculate the End Latency for this engine:

HVI Start		
Parameter	Time (cycles)	Result (n s)
$\texttt{HviStart\_EndLatency}_{\texttt{Engine2}}$	2	$6.66\overline{6}$

Using the timing table of the Sync While statement, we calculate the Start Latency for this engine:

Sync While		
Parameter	Time (cycles)	Result (n s)
$\texttt{SyncWhile\_StartLatency}_{\texttt{Engine1}}$	Leader Engine: 6 + #Register_Conditions	$23.33\overline{3}$

Therefore, the minimum start delay for this engine is:

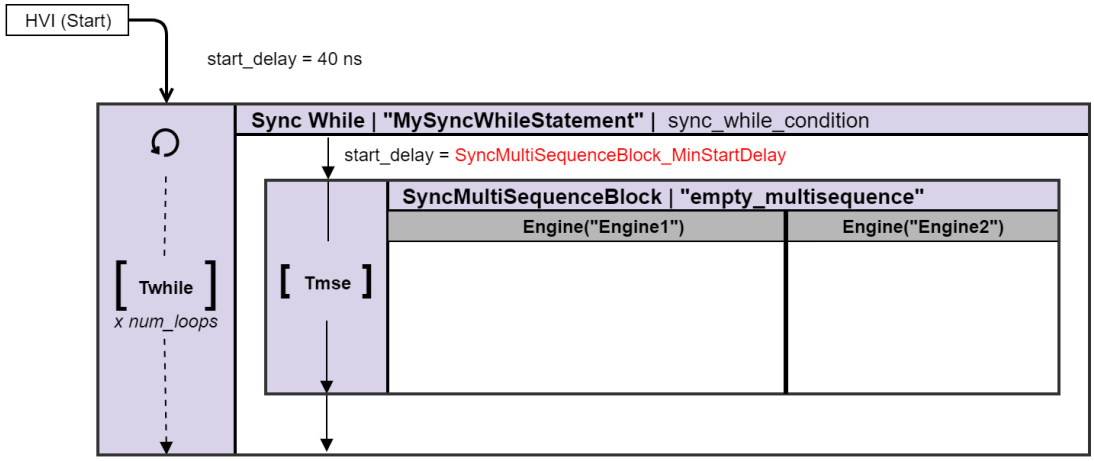
$$\mathbf{SyncWhile\_MinStartDelay_{Engine2}} = \mathrm{HviStart\_EndLatency_{Engine2}} + \mathrm{SyncWhile\_StartLatency_{Engine2}} = \mathbf{30\ ns}$$

Minimum Start Delay:

$$\mathbf{SyncWhile\_MinStartDelay = round_{ncc}(max\{SyncWhile\_MinStartDelay_{Engine1}, SyncWhile\_MinStartDelay_{Engine2}\}) = 30\ ns}$$

Example: Minimum Start delay for the first statement inside a Sync While

Continuing from the previous example, in this example we show how to calculate the minimum start delay value acceptable for a Sync Multi-Sequence statement that is placed as the first statement of a Sync While internal SyncSequence.




```
#...
SyncWhile_MinStartDelay = 30 # As calculated earlier
sync_while_statement = sequencer.sync_sequence.add_sync_while("MySyncWhileStatement",
SyncWhile_MinStartDelay,  sync_while_condition)
SyncMultiSequence_MinStartDelay = ... # The calculation for the minimum is explained below
sync_while_statement.sync_sequence.add_sync_multi_sequence_block("empty_multisequence",
SyncMultiSequence_MinStartDelay)
```

In addition to the Variables provided in the previous example, we assume the following values to be used in the following calculations:

Variable	Value	Description
Instrument_SyncResources_Latency	0 cy	This is an instrument dependent value. We assume it to be 0 for this example.
Propagation_delay$_{Cycles}$	30 cy	Assuming that we use only 1 chassis in this example, the Propagation delay would be 100 ns. Translating it to cycles of the leader engine of the Sync While, that gives us 30 cycles.
End-Latency$_{Last\text{-}statement}$	0 cy	The last statement is an empty Sync Multi-Sequence Block which, according to the timing tables, will have 0 cycles of end-latency.

Engine 1:

Using the timing table of the Sync While statement, we calculate the End Latency for this engine:

HVI Start		
Parameter	Time (cycles)	Result (n s)
$SyncWhile\_EntryLatency_{Engine1}$	$match\{LatencyA_{Engine1}, LatencyA_{Engine2\_inEngine1Cycles}\} + 2 + End\text{-}Latency_{Last\text{-}statement}$	160*

**Calculation:*

$LatencyA_{Engine1} = 2\ cy\ (Engine1\ cycles)$

$LatencyA_{Engine2} = 12 + \#Register\_Conditions + Instrument\_SyncResources\_Latency + Propagation\_delay_{Cycles} = 43\ cy\ (Engine2\ cycles)$

$LatencyA_{Engine2\_inEngine1Cycles} = ceil(43 * Engine2_{Period} / Engine1_{Period}) = 29\ cy\ (Engine1\ cycles)$

$SyncWhile\_EntryLatency_{Engine1} = (match\{2,29\} + 2 + 0) * Engine1_{Period} = (round_{ncc}(29) + 2) * 5 = 160\ ns$

Using the timing table of the Sync Multi-Sequence statement, we calculate the Start Latency for this engine:

Sync While		
Parameter	Time (cycles)	Result (n s)
$SyncMultiSequence\_StartLatency_{Engine1}$	1	5

Therefore, the minimum for this engine is:

$\mathbf{SyncMultiSequence\_MinStartDelay_{Engine1}} = SyncWhile\_EntryLatency_{Engine1} + SyncMultiSequence\_StartLatency_{Engine1} = \mathbf{165\ ns}$

Engine 2:

Using the timing table of the Sync While statement, we calculate the End Latency for this engine:

HVI Start		
Parameter	Time (cycles)	Result (n s)
SyncWhile_ EntryLatency$_{Engine2}$	match{LatencyA$_{Engine1_inEngine2Cycles}$, LatencyA$_{Engine2}$} + 2 + End-Latency$_{Last-statement}$	$156.66\overline{6}$ *

**Calculation:*

*LatencyA$_{Engine1_inEngine2Cycles}$ = ceil(LatencyA$_{Engine1}$ * Engine1$_{Period}$ / Engine2$_{Period}$) = 3 cy (Engine2 cycles)*

LatencyA$_{Engine2}$ = 43 cy (Engine2 cycles)

*SyncWhile_EntryLatency$_{Engine2}$ = (match{3,43} + 2 + 0) * Engine2$_{Period}$ = (round$_{ncc}$(43) + 2) * $3.33\overline{3}$ = 156.666 ns*

Using the timing table of the Sync Multi-Sequence statement, we calculate the Start Latency for this engine:

Sync While		
Parameter	Time (cycles)	Result (n s)
SyncMultiSequence_ StartLatency$_{Engine2}$	1	$3.33\overline{3}$

Therefore, the minimum Start Delay for this engine is:

SyncMultiSequence_MinStartDelay$_{Engine2}$ = SyncWhile_EntryLatency$_{Engine2}$ + SyncMultiSequence_ StartLatency$_{Engine2}$ = **160 ns**

Minimum Start Delay:

SyncMultiSequence_MinStartDelay = round$_{ncc}$(max{SyncMultiSequence_MinStartDelay$_{Engine1}$, SyncMultiSequence_MinStartDelayEngine2}) = 170 ns

Errors when setting start delays and how to deal with them

The previous section explains how to calculate the Start delay for each type of statement. Depending on the type of statements as well as the number of engines with different frequencies, this can be a complex and error-prone procedure.

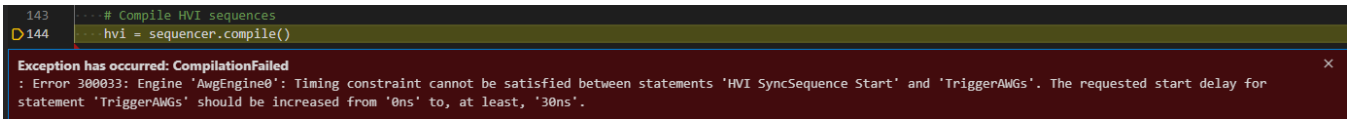
In the case of an error, the HVI compiler will validate the provided values and generate a message with the minimum Start delay applicable.

Example: How to fix your Start Delay by using the compiler message

In the following example code snippet, a Sync Multi-Sequence Block is added as the first statement to an HVI Sync Sequence. To get an error for the minimum Start delay that can be used to add the Sync multi-sequence block, you can set the Start delay to 0 as shown in the following example code snippet.

```
# Create system definition object
my_system = kthvi.SystemDefinition("MySystem")
...
# Create sequencer object
sequencer = kthvi.Sequencer("MySequencer", my_system)
# Add a Sync Multi-Sequence Block (SMSB) with a 0 ns start delay
sync_block = sequencer.sync_sequence.add_sync_multi_sequence_block("TriggerAWGs", 0)
```

When the sequencer object is compiled, the compiler detects that 0 ns does not comply with the minimum latency for the Sync multi-sequence block in the HVI sequence. It returns an error message stating that the specified start delay shall be at least 30 ns. See the following image for an example of the returned error message.



The reason why a minimum latency of 30 ns is required is explained in this chapter in the section Sync Statement Timing Tables. In a similar way, you can set any Start delay to 0 ns and let the compiler error messages provide the minimum latency required for each of those Start delays. The reasons behind the specific minimum values advised by the compiler are explained in the rest of this chapter.

NOTE **Limitations:**

The compiler provides an indication of the minimum start delay for each engine. If you are using different instruments these can be different values. That is because the clock cycle duration is different in each instrument. For example, a minimum latency of 3 cycles lasts 30 ns on M3xxxA instruments and 10 ns on M5xxxA instruments. In case the compiler error messages suggest different values, *pick the highest value of those indicated and round it to the next HVI Common Clock cycle* to set a start delay value that can accommodate the requirements for all the different instruments that are included in the HVI.

Sync Statement Timing Tables

This section provides timing values for Sync statements and Sync flow-control statements, it contains the following sections:

- HVI Start
- Sync While
- Sync Register-Sharing
- Sync FPGA Data-Sharing
- Sync Multi-Sequence Block

How to use the Timing Tables for Sync Statements

All the timings provided in the tables below are expressed in HVI Engine Clock cycles.

Leader Engine

In some of the Sync Statements, one of the engines that leads the statement operation. For example, in a Sync While statement, the engine that leads is the one where the condition is evaluated. For the context of Timing Latency calculation, we are going to call this engine the *Leader Engine*.

Rounding Delays

When a latency value needs to be applied to multiple engines, we must round the Engine cycles to the next HVI Common Clock cycle. We do this using the following formula (below ncc stands for next common clock):

$$\mathbf{round}_{\mathbf{ncc\_cycles}}(\mathbf{TimeValue}_{\mathbf{EngineCycles}}) = \text{ceil}(\text{TimeValue}_{\text{EngineCycles}} * \text{HVI\_Engine\_Clock}_{\text{Period}} / \text{HVI\_Common\_Clock}_{\text{Period}}) * \text{HVI\_Common\_Clock}_{\text{Period}} / \text{HVI\_Engine\_Clock}_{\text{Period}}$$

In the case that all the engines are running at the same frequency, the HVI Engine Clock cycles and the HVI Common Clock cycles will have the same value for all the engines. Therefore, you can skip the rounding calculation because it has no effect:

$$\mathbf{round}_{\mathbf{ncc\_cycles}}(\text{TimeValue}_{\text{EngineCycles}}) == \text{TimeValue}_{\text{EngineCycles}}$$

Matching Delays

Some parts of the latency may need to be aligned between engines. In order to achieve this, we use the following formula:

$$\mathbf{match}\{\mathbf{TimeValue}_{\mathbf{Engine1Cycles}}, \mathbf{TimeValue}_{\mathbf{Engine2Cycles}}, \ldots, \mathbf{TimeValue}_{\mathbf{EngineNCycles}}\} = \mathbf{round}_{\mathbf{ncc\_cycles}}(\ \mathbf{max}\ \{\mathbf{TimeValue}_{\mathbf{Engine1Cycles}}, \mathbf{TimeValue}_{\mathbf{Engine2Cycles}}, \ldots, \mathbf{TimeValue}_{\mathbf{EngineNCycles}}\}\ )$$

In the previous formula, all the `TimeValues` have to first be converted to the `EngineCycles` of the target engine so that the `max` can be applied among similar quantities. This can be done using this formula:

$$TimeValue_{TargetEngineCycles} = ceil(TimeValue_{OtherEngineCycles} * HVI\_OtherEngine\_Clock_{Period} / HVI\_TargetEngine\_Clock_{Period})$$

In the case that all the engines are running in the same frequency, the calculation of the match formula is just the time value of the Leader Engine:

$$match\{TimeValue_{Engine1Cycles}, TimeValue_{Engine2Cycles}, \ldots, TimeValue_{EngineNCycles}\} == TimeValue_{LeaderEngineCycles}$$

Propagation Delay

The Propagation Delay corresponds to the amount of time (expressed in nanoseconds) that a PXIe trigger needs, to cover the path between any given pair of segments in a topology. This value is used when running sync statements because it provides information about how long the execution signaling between modules takes.

The Propagation Delay is expressed in nanoseconds and its value depends on the topology. A table with the values is defined on the page Sync Sequences and Synchronization Points.

In the context of the Timing Tables, the value is expressed in cycles of the HVI Engine, as it is shown in the next tables. To be able to use it, a conversion is required:

$$Propagation\_delay_{EngineCycles} = Round(Propagation\_Delay_{Seconds} / HVI\_Engine\_Clock_{Period})$$

HVI Start

This is the time 0 for the HVI execution. It always matches the rising edge of the Sync signal (in PXIe systems aligned with the PXIe-SYNC100 signal).

HVI start basic timing value:

Parameter	Time (cycles)
End-Latency	2

Sync Multi-Sequence Block

Timing value for Sync multi-sequence blocks:

Execution time (cycles) (1)
$round_{ncc\_cycles}\ (sum_{for\_all\_internal\_statements}(StartDelay_{Cycles}) + sum_{for\_all\_internal\_flow\_control\_statements}(Duration_{Cycles})\ )$ (2)

The following tables shows latency values for Sync Multi-Sequence Blocks:

Parameter	Description		Time(cycles)
Start-Latency	Minimum start-delay for statement		1
Entry-latency	Minimum start-delay for first statement inside any of the contained sequences		1
End-Latency	Minimum start-delay for the next statement	timed-sync (5) Minimum Duration	$round_{ncc\_cycles}(End\text{-}Latency_{Last\text{-}statement\text{-}of\text{-}longest\text{-}branch(3)} - 1)$ * *if the last statement of the longest branch is not starting from a common clock cycle (see section **Sync Multi-Sequence Block Timing and Time Matching** in Sync Statement Timing), the formula is updated to:* $round_{ncc\_cycles}(End\text{-}Latency_{Last\text{-}statement\text{-}of\text{-}longest\text{-}branch(3)} - 1 - DistanceToNextCommonClock)$ *where:* *- $DistanceToNextCommonClock$ is the number of Engine Cycles from the start of the last statement to the following common clock cycle.*
		timed-sync (5) Fixed Duration	0
		triggered-sync (5)	0

Fixed-Duration	Minimum fixed-duration for statement	$round_{ncc\_cycles}([max_{for\_all\_Sequences}[Sequence\text{-}Duration]])$ (4), where Sequence-Duration is calculated as follows: $sum_{for\_all\_internal\_statements}(Start\text{-}Delay) + sum_{for\_all\_internal\_flow\_control\_statements}(Duration) + End\text{-}Latency_{Last\text{-}statement} - 1$

(1) The values provided here apply if the duration property of the statement is set to Minimum (default). If a fixed-duration has been set, then the Execution time is equal to that value.

(2) The values are only calculated for the branch that is being executed, if there are multiple branches available.

(3) If the sequence is empty, the value is 0.

(4) If the sequence is empty, then the duration is 0.

(5) Triggered-sync is required if any of the sequences in a Sync multi-sequence block contains a statement that has unknown execution time at compile time. See section **Synchronization Points and Sync Sequence Start** in Sync Statement Timing.

Sync While

Timing value for Sync while statement:

Execution time (cycles) (1)
$round_{ncc\_cycles}$ (#Iterations * [$sum_{for\_all\_internal\_statements}$(StartDelay$_{Cycles}$) + $sum_{for\_all\_internal\_flow\_control\_statements}$(Duration$_{Cycles}$)])

The following tables shows latency values for the Sync while statement:

<table>
<tr><th>Parameter</th><th colspan="2">Description</th><th>Time (cycles)</th></tr>
<tr><td>Start-Latency</td><td colspan="2">Minimum start-delay for statement</td><td>• Leader Engine: <code>6 + #Register_Conditions</code>
• Follower Engine(s): <code>2</code></td></tr>
<tr><td rowspan="2">Entry/Iteration latency</td><td rowspan="2">Minimum start-delay for first statement inside the while loop</td><td>Minimum Duration</td><td><code>match{LatencyA_{LeaderEngine}, LatencyA_{FollowerEngine1}, ...} + 2 + End-Latency_{Last-statement}(2)</code>
where <code>LatencyA</code> is :
• Leader Engine(3): <code>12 + #Register_Conditions + Instrument_SyncResources_Latency</code> (4) + Propagation_delay_{Cycles}
• Follower Engine(s): <code>2</code></td></tr>
<tr><td>Fixed Duration</td><td><code>match{LatencyB_{LeaderEngine}, LatencyB_{FollowerEngine1}, ...} + 2</code>
where :
■ <code>LatencyB</code> = LatencyA – 1
■ <code>LatencyA</code> as defined above</td></tr>
</table>

Parameter	Description		Time (cycles)
End-Latency	Minimum start-delay for next statement outside the while loop	Minimum Duration	$match\{LatencyA_{LeaderEngine}, LatencyA_{FollowerEngine1}, ...\} + match\{LatencyC_{LeaderEngine}, LatencyC_{FollowerEngine1}, ...\} + End\text{-}Latency_{Last\text{-}statement}(2),$ *where:* ■ *LatencyA as defined above* ■ *LatencyC is 2 for each Engine.*
		Fixed Duration	$match\{LatencyB_{LeaderEngine}, LatencyB_{FollowerEngine1}, ...\} + match\{LatencyC_{LeaderEngine}, LatencyC_{FollowerEngine1}, ...\},$ *where:* ■ *LatencyB as defined above* ■ *LatencyC as defined above*
Fixed-Duration	Minimum fixed-duration for statement	Sync-While Branch with at least one statement inside	$round_{ncc\_cycles}(sum_{for\_all\_internal\_statements}(StartDelay_{Cycles}) + sum_{for\_all\_internal\_flow\_control\_statements}(Duration_{Cycles}) + 1 + End\text{-}Latency_{Last\text{-}statement})$
		Empty Sync-While Branch	$match\{LatencyB_{LeaderEngine}, LatencyB_{FollowerEngine1}, ...\} + match\{LatencyC_{LeaderEngine}, LatencyC_{FollowerEngine1}, ...\} + 1,$ *where:* • *LatencyB as defined above* • *LatencyC as defined above*

Parameter	Description	Time (cycles)
Register Evaluation Latency	Time to evaluate the register condition	`Leader Engine (Only):` • `From start: 2` • `For each iteration: -(3 + #Register_ Conditions)`

(1) This value applies if the duration property of the statement is set to Minimum (default). If a fixed-duration has been set, then the Execution time is equal to that value.

(2) If the sequence is empty, the value of $\texttt{End-Latency}_{\texttt{Last-statement}}$ is `0`.

(3) In the context of this statement, Leader is the engine that contains the register or registers used in the while condition.

(4) `Instrument_SyncResources_Latency` is an instrument specific value. For more information see the instrument documentation.

Sync Register-Sharing

Sync register-sharing latency does not depend on the number of bits shared. For more information on this functionality, see HVI Statements and HVI API Sync Statements.

Timing value for Sync register-sharing statement:

Execution time (cycles) (1)
$round_{ncc\_cycles}(5 + Propagation\_delay_{Cycles})$ (2)

Latency values for Sync register-sharing statement:

Parameter	Description	Time (cycles)
Start-Latency	Minimum start-delay for statement	1
End-Latency	Minimum start-delay for the next statement	0
Fixed-Duration	Minimum fixed-duration for statement	$round_{ncc\_cycles}(5 + Propagation\_delay_{Cycles})$ (2)
Register Evaluation Latency	Time to evaluate the register condition	-1

(1) The value provided here applies if the duration property of the statement is set to Minimum (default). If a fixed-duration has been set, then the Execution Time is equal to that value.

(2) This latency needs to be calculated only on the Leader Engine. In the context of this statement, Leader is the engine that contains the register(s) used as source.

Sync FPGA Data-Sharing

Calculating execution times for Sync FPGA data-sharing can be a complex process. This is because Sync FPGA data-sharing execution time depends on a number of factors:

- Instrument specific delay characteristics.
- The topology of your system.
- The amount of data be transferred.
- If the transfer of data is in a single chassis or if it is between different chassis.
- The scheduling of multiple transactions.

For information about how to calculate timing for a Sync FPGA data sharing statement see Sync Statement Timing.

Latency values for Sync FPGA data-sharing statement:

Parameter	Description	Time (cycles)
Start-Latency	Minimum start-delay for statement	1
End-Latency	Minimum start-delay for the next statement	0

Local Flow-Control Statement Timing Tables

This section provides timing values for Local Flow-control statements, it contains the following sections:

- Local Flow-Control Statement Parameters
- Local Wait-For-Time Statement
- Local Wait-For-Event Statement
- Local Delay Statement
- Local If Statement
- Local While Statement

Local Flow-Control Statement Parameters

Some Local flow-control statements have a parameters and properties you must be aware of for calculating timing:

Branch matching

Branch matching is a concept used in Local If statements. Branches with different instructions can take different times. Match branches enables you to ensure the branches all take the same time irrespective of which one is taken.

NOTE In the following tables, whenever the end-latency of the last-statement contained in a flow-control statement is required and that last statement is a Local instruction, the end-latency is calculated as the fetch-cycles of that instruction.

Local Wait-For-Time Statement

A Wait-for-time statement blocks HVI execution in a Local sequence until a specific amount of time passes. This amount of time is defined in a register that is specified as an argument in the Wait-for-time statement. The value of the register specifies the number of cycles to wait.

Local Wait-for-time statement timing value:

Execution time (cycles)
RegisterValue

Local Wait-for-time statement latency values:

Parameter	Time (cycles)
Start-Latency	1
End-Latency	1
Register Evaluation Latency	1

Local Wait-For-Event Statement

A Local Wait-for-event statement blocks HVI execution in a Local sequence until an event occurs. Events sources can be the Trigger IOs, or internal to the instrument (including FPGA User Sandbox Events).

Local Wait-for-event statement timing values:

Event type	Execution time (cycles)	Fetch time (cycles)
Internal Event	`MAX(Event_Arrival_Time(1) + Instrument_Event_Latency(2) + 1, Fetch_Time) + 1`	3
Trigger IO	`MAX(Event_Arrival_Time(1) + Instrument_Event_Latency(2) + Instrument_Event_Condition_Latency(3), Fetch_Time) + 1`	1 + *Instrument_Event_Condition_Latency*

(1) `Event_arrival_time` *is:*

- *Internal Events*
  - `Event_Arrival_Time = Internal_Event_Generation_Time - WaitForEvent_Start_Time`
- *External Events*
  - `Event_Arrival_Time = Event_At_Module_Connector_Time – WaitForEvent_Start_Time`
  - *The event time can be measured at the front panel or PXIe backplane connector depending on the event.*

(2) `Instrument_Event_Latency` *is the delay from the event source until the event state is available inside the HVI Engine. Events sources can be the Trigger IOs, or internal to the product (including FPGA User Sandbox Events). It is an instrument and event specific value. Refer to the instrument documentation for more information.*

(3) `Instrument_Event_Condition_Latency` *is the time needed for the condition evaluation to be executed once the event has settled inside the HVI Engine. It is an instrument specific value. Refer to the instrument documentation for more information.*

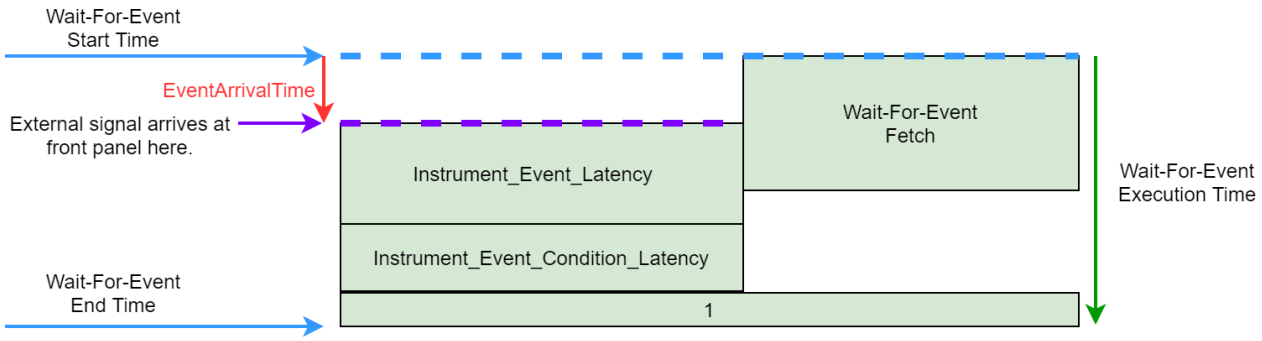
NOTE The *Event_Arrival_Time* can be a negative value if the event enters the module before the Wait-For-Event instruction Start Time. A number of scenarios are shown in the diagrams below.

Local Wait-for-event latency values:

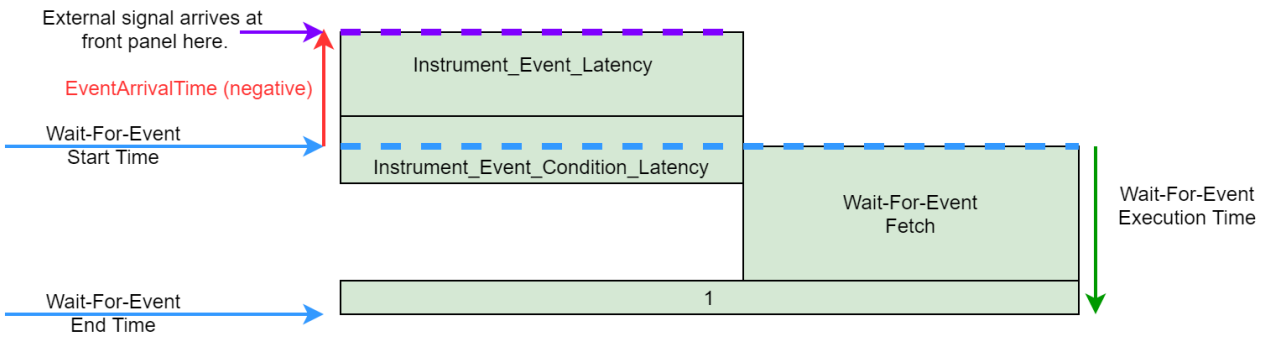
Parameter	Time (cycles)
Start-Latency	0
End-Latency	1

The following diagrams shows scenarios where the execution time of a Wait-For-Event statement can vary:

Case 1: Event arrival + propagation delay after Fetch-Time completion



Case 2: Event arrival + propagation delay completes before Fetch-Time completion



Local Delay Statement

A Delay statement delays HVI execution in a Local sequence until a specific amount of time passes. This amount of time is specified in a parameter in the statement.

Local Delay statement timing value:

Execution time (cycles)
Delay Specified

Local Delay latency values:

Parameter	Time (cycles)
Start-Latency	0
End-Latency	1

Local If Statement

For if statements with multiple If / Else-If / Else branches, the Entry delays are the same for all branches.

If the match-branches attribute is enabled, the HVI ensures that the execution of all of the branches have the same overall delay. If match-branches is not enabled, some branches might take less time than others.

The If statement latency depends on the number or register-conditions used: `#Register_Conditions`.

Local If timing value:

Execution time (cycles) (1) (2)
$sum_{for\_all\_internal\_statements}(Start\text{-}Delay) + sum_{for\_all\_internal\_flow\_control\_statements}(Duration)$ (3)

(1) *The value provided here applies if the duration property of the statement is set to Minimum (default). If a fixed-duration has been set, then the Execution time is equal to that value.*

(2) *This value is only calculated for the branch that is executed, if there are multiple branches available.*

(3) *If the branch is empty, the execution time becomes* $Entry\text{-}Latency_{branch} - 1$.

The following table shows Local If latency values:

Parameter	Description		Minimum time (cycles)
Start-Latency	Contributes to the minimum-possible start-delay for the statement		`5 + #Register_Conditions_IfBranch`
Entry-latency	Contributes to the minimum-possible start-delay for first statement in branch #		• `If-Branch: 3` • `Else-If-Branch:` – *`F` or each else-if branch, we need to add:* ◦ *`6 + #Register_Conditions_Else-If-Branch`* ◦ *For the 1st else-if branch we will have:* ◦ *`2 + #Register_Conditions_IfBranch + 7 + #Register_Conditions_Else-If-Branch`$_1$* ◦ *For the 2nd else-if branch we will have:* ◦ *`2 + #Register_Conditions_IfBranch + 7 + #Register_Conditions_Else-If-Branch`$_1$ `+ 7 + #Register_Conditions_Else-If-Branch`$_2$ ◦ *and so on, so forth...* • *Else-Branch: Equal to last Else-If-Branch value*
End-Latency	Contributes to the minimum-possible start-delay of the next statement outside the if statement	Matching Branches *`disabled`*	`3 + max`$_{\texttt{for\_all\_Branches}}$`[End-Latency`$_{\texttt{Last-statement}}$`]` (1)
		Matching Branches *`enabled`*	`3 + End-Latency`$_{\texttt{Last-statement-of-longest-branch}}$ (2) Where longest branch means the branch with longer execution time.
		Fixed-Duration	`1`

Fixed-Duration	Minimum fixed-duration for statement	2 + $max_{for\_all\_Branches}$[Branch-Duration](3) Where Branch-Duration is calculated as follows: [$sum_{for\_all\_internal\_statements}$(Start-Delay) + $sum_{for\_all\_internal\_flow\_control\_statements}$(Duration) + End-$Latency_{Last\text{-}statement}$] (4)
Register Evaluation Latency	Time to evaluate the register condition	3 *Then, for registers used in the condition of any else-if branch, we need to substract:* – *6 + #Register_Conditions_Else-If-Branch* *Therefore, for the 1st else-if branch we will have:* – *3 - (6 + #Register_Conditions_Else-If-$Branch_1$)* *For the 2nd else-if branch we will have:* – *3 - (6 + #Register_Conditions_Else-If-$Branch_1$) - (6 + #Register_Conditions_Else-If-$Branch_2$)* *and so on, so forth...*

(1) *If the maximum end latency used in this equation corresponds to the if-branch, and the calculated latency is greater than 4, then the End-latency is the calculated value minus 1.*

(2) *If the longest branch is the if-branch, then the End-latency is the calculated value minus 1.*

(3) *If the maximum branch duration used in the equation corresponds to the if-branch, then the duration is the calculated value minus 1.*

(4) *If a branch is empty, then the branch duration is equal to the Entry-latency of the branch.*

Local While Statement

Local While timing value:

Execution time (cycles) (1)
#Iterations * [$sum_{for\_all\_internal\_statements}$(Start-Delay) + $sum_{for\_all\_internal\_flow\_control\_statements}$(Duration)]

(1) *This value applies if duration property of the statement is set to Minimum (default). If a fixed-duration has been set, then this is the Execution time is equal to that value.*

Local While latency values:

Parameter	Description		Time (cycles)
Start-Latency	Minimum start-delay for the statement		`5 + #Register_Conditions`
Entry/Iteration latency	Minimum start-delay for first statement inside the while loop	Minimum Duration	`8 + #Register_Conditions + End-Latency`$_{\text{Last-statement}}$
		Fixed Duration	`8 + #Register_Conditions`
End-Latency	Minimum start-delay for the next instruction outside the while loop	Minimum Duration	`8 + #Register_Conditions + End-Latency`$_{\text{Last-statement}}$
		Fixed Duration	`8 + #Register_Conditions`
Fixed-Duration	Minimum fixed-duration for statement		`[sum`$_{\text{for\_all\_internal\_statements}}$`(Start-Delay) + sum`$_{\text{for\_all\_internal\_flow\_control\_statements}}$`(Duration) + End-Latency`$_{\text{Last-statement}}$`](1)`
Register Evaluation Latency	Time to evaluate the register condition		• `From start: 3` • `For each iteration: -(2 + #Register_Conditions)`

(1) *If the branch is empty, then the duration is equal to the Entry-Latency of the branch.*

Local Instruction Statement Timing Tables

The following sections list the fetch and execution latency for HVI-native Local instruction statements. Unless stated otherwise, all times are in HVI engine clock cycles. The HVI engine clock frequency is instrument specific. For information about the HVI engine clock frequency and instrument-specific instruction latencies, See your instrument documentation.

This section contains the following sections:

- Local Instruction Statement Parameters
- Trigger Write
- Action Execute
- Arithmetic Logic Unit Instructions
- FPGA User Sandbox Instructions
- FPGA-Instruction Statement
- Instrument-Specific Local Instruction Statement Timing Values

Local Instruction Statement Parameters

Local instruction statements have a number of parameters and properties you must be aware of for calculating timing:

TriggerIO groups and Action groups

The following additional parameters are used for calculating timing for some Local instruction statements.

Triggers and actions are organized into groups and the timing can change depending on these:

TriggerIO groups

Trigger Inputs / Outputs are organized together in groups of 16 called TriggerIO groups. Any number of TriggerIO groups can be written at the same time.

Action groups

HVI actions are organized together in groups of up to 16 called Action groups. Any number of Action groups can be executed synchronously.

Trigger Write

Trigger Inputs / Outputs are organized together in groups of 16 called TriggerIOs. Each value can be ON or OFF.

Any number of TriggerIOs can be written at the same time.

- #TriggerIOGroupsON is the number of TriggerIOGroups that contain values set to ON.
- #TriggerIOGroupsOFF is the number of TriggerIOGroups that contain values set to OFF.

The Fetch time of the instruction depends on the number of different TriggerIO groups included in the instruction for the two possible values (#TriggerIOGroupsON or #TriggerIOGroupsOFF).

The following table provides some examples.

Triggers ON	Triggers OFF	#TriggerIOGroupsON	#TriggerIOGroupsOFF	Execution time (cycles)	Fetch time (cycles)
1, 2		1	0	2	1
1, 2, 17, 18		2	0	2	1
1, 2	3, 4	1	1	2	1
1, 2, 17, 18	3, 4	2	1	3	2
1, 2, 17, 18	3, 4, 19, 20	2	2	3	2

See your instrument documentation for information about instrument specific TriggerIO definitions.

NOTE Trigger execution time is instrument specific. For trigger execution timing information, see your instrument documentation.

Example Trigger write basic timing values:

Instruction	Execution time (cycles)	Fetch time (cycles)
TriggerWrite	Instrument_Trigger_Execution + (#TriggerWriteGroups - 1)	#TriggerWriteGroups

#TriggerWriteGroups = ceil[(TriggerIOGroupsON + TriggerIOGroupsOFF)/2], where

- #TriggerIOGroupsON is the number of TriggerIOGroups that contain values set to ON.
- #TriggerIOGroupsOFF is the number of TriggerIOGroups that contain values set to OFF.

Action Execute

The action-execute HVI instruction synchronously executes a list of HVI actions defined by the user. HVI actions are organized in groups called ActionGroups that can contain up to 16 actions. Each instrument defines its own groups of actions. See the instrument documentation for information about instrument action definitions and the way they are grouped. Any number of HVI actions can be executed synchronously, regardless of the group that each action user belongs to.

However, the number of action groups included in the action-execute instruction (#ActionGroups) affects both the Fetch time and the Execution time of the instruction, as shown by the equations in the following table.

NOTE Action execution timing is instrument specific. For action execution timing information, see your instrument documentation.

Example Action execute basic timing values:

Instruction	Execution time (cycles)	Fetch time (cycles)
ActionExecute	Instrument_Action_Execution + INT[(#ActionGroups-1) / 2]	1 + INT[(#ActionGroups -1) / 2]

Where INT is the integer part of a decimal number, for instance INT(1.0)=INT(1.5)=1.

Arithmetic Logic Unit Instructions

Arithmetic Logic Unit (ALU) instructions are the register add, subtract or assign operations that are available in the HVI-native instruction set.

ALU instructions basic timing values:

Instruction	Execution time (cycles)	Fetch time (cycles)
Add	8	1
Subtract	8	1
Assign	5	1

FPGA User Sandbox Instructions

The access latency of the FPGA registers and memory map from HVI depends on the implementation of the specific instrument. The following table summarizes the latency for all FPGA read/write instructions. For the specific value of `Instrument_HVI_FPGA_Latency`, see your instrument documentation.

NOTE FPGA user sandbox timing is instrument specific. For FPGA user sandbox timing information, see your instrument documentation.

Example FPGA user sandbox operations basic timing values:

Instruction	Execution time (cycles)	Fetch time (cycles)
FpgaArrayRead	`2 * Instrument_HVI_FPGA_Latency + 4`	1
FpgaArrayRead (Address from HviRegister)	`2 * Instrument_HVI_FPGA_Latency + 6`	1
FpgaArrayWrite	`Instrument_HVI_FPGA_Latency + 2`	1
FpgaArrayWrite (Address or data from HviRegister)	`Instrument_HVI_FPGA_Latency + 4`	1
FpgaRegisterRead	`2 * Instrument_HVI_FPGA_Latency + 4`	1
FpgaRegisterWrite	`Instrument_HVI_FPGA_Latency + 2`	1
FpgaRegisterWrite (Address or data from HviRegister)	`Instrument_HVI_FPGA_Latency + 4`	1

NOTE
- Consecutive FPGA read instructions must be issued with at least 1 cycle of delay between them.
- If an FPGA instruction that uses an HVI register is issued before an FPGA instruction that does not use an HVI register, the delay between both instructions must be at least 3 cycles.

Local Instruction Position Mapping

The following table show the instruction positions (see Local Instruction Timing) that each HVI-native Local instruction uses during fetch time. For instrument custom instructions, see your instrument documentation:

	Positions					
HVI Native Instruction	1	2	3	4	5	...
ActionExecute	Y				-	-
Add	Y				-	-
Assign	Y				-	-
Fpga Array-Read	Y				-	-
Fpga Array-Write	Y				-	-
Fpga Register-Read	Y				-	-
Fpga Register-Write	Y				-	-
Subtract	Y				-	-
TriggerWrite	Y				-	-

FPGA-Instruction Statement

FPGA-Instruction statement latency depends on a number of factors:

- Instruction fetch time.
- Time to fetch data from any HVI registers it uses.
- Instrument specific delays.

Apart from fetch time and the first two execution cycles spent inside the HVI engine, the rest of the latency is defined by the instrument, this is condensed into the single parameter `Instrument_FpgaInstruction_Latency`. See your instrument documentation for information about the HVI engine clock frequency and FPGA-instruction timing information.

Timing values:

Instruction	Execution Time (cycles)	Fetch Time (cycles)
FPGA-Instruction	`Instrument_FpgaInstruction_Latency + 2`	`1`

Instrument-Specific Local Instruction Statement Timing Values

Instrument-specific local instruction statement latency depends on a number of factors:

- Instruction fetch time.
- Time to fetch data from any HVI registers it uses.
- Instrument specific delays.

Apart from fetch time and the first two execution cycles spent inside the HVI engine, the rest of the latency is defined by the instrument and condensed into the single parameter `Instrument_LocalInstruction_Latency`. See your instrument documentation for information about the HVI engine clock frequency and instrument-specific instruction timing information.

Timing values:

Instruction	Execution Time (cycles)	Fetch Time (cycles)
Instrument-Specific Local Instruction	`Instrument_LocalInstruction_Latency + 2`	`1`

Appendix A: Supported Instruments

PathWave Test Sync Executive supports a number of instruments and PXIe chassis, these require specific minimum software and firmware versions to work with PathWave Test Sync Executive.

The software and firmware version requirements for the supported instruments and chassis are listed on-line here: Instrument and Chassis Software and Firmware Requirements for KS2201A .

Product specific documentation

For product-specific information and documentation please refer to the product pages.

Firmware is available at Keysight PXI Products, on the **Technical Support** page for your specific instruments, see the **Drivers, Firmware & Software** tab.

M3000 Series

The M3000 series (SD1) software provides drivers, programming libraries and software front panels for the M3000 series.

Instruments are shipped with the latest versions of firmware and SD1 software. To use an older instrument with PathWave Test Sync Executive, the firmware and SD1 software must be upgraded to the versions recommended in the product page following the guidelines at the link above. SD1 software is available at Keysight SD1 Software.

Other Instruments

Instruments are provided with their own drivers, programming libraries, and software front panels, and are shipped with the latest versions of firmware and software. To use an older instrument with PathWave Test Sync Executive, the firmware and software must be upgraded to the versions recommended in the product page following the guidelines at the link above.

PXIe Chassis

The Chassis software provides drivers, programming libraries and software front panels for the Keysight chassis.

Chassis are shipped with the latest versions of firmware and software. To use an older chassis with PathWave Test Sync Executive, the firmware and software must be upgraded to the versions recommended in the product page following the guidelines at the link above.

Compatibility with M3601A

M3601A is an older generation of HVI technology that is only programmable by the M3601A Hard Virtual Instrument Design Environment. PathWave Test Sync Executive is a new generation and is not backward compatible with the M3601A generation.

Both PathWave Test Sync Executive and M3601A work with the M3000 series of PXIe products. However, PathWave Test Sync Executive requires newer firmware while M3601A requires older firmware.

Appendix B: Additional Documentation and Examples

This appendix lists the PathWave Test Sync Executive Programming Examples and additional documentation that you can download from the KS2201A Programming Examples page.

NOTE The Programming Examples are often updated so ensure you check for the latest versions.

Programming Example 1: Multi-Channel Sync Playback using M32xxA Arbitrary Waveform Generators

In Programming Example 1, PathWave Test Sync Executive is used to program multiple M3xxxA Arbitrary Waveform Generators (AWG)s. The AWGs synchronously output a front panel trigger pulse followed by a previously queued waveform. All instruments run fully synchronized and actions across the instruments can be controlled at the timing resolution of the M3xxxA AWGs, which is 10ns.

Programming Example 2: Synchronous Signal Generation and Acquisition using M3xxxA PXI Instruments

In Programming Example 2, a M3102A
digitizer performs sequenced acquisition of heterogeneous signals generated by multiple M320xA AWGs. The first AWG generates a train of RF pulses and the other AWGs output a
queued arbitrary waveform. By using PathWave Test Sync Executive, each cycle of the digitizer measurements is precisely synchronized with the AWG output signals.

Programming Example 3: PathWave Test Sync Executive Integration with PathWave FPGA

This Programming Example shows how to use Keysight PathWave Test Sync Executive together with Keysight PathWave FPGA. A custom FPGA block is designed using Keysight PathWave FPGA and loaded into the sandbox of two modular instruments. 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 in any HVI Port Register inserted among the custom FPGA blocks. This example also shows how the 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.

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

In this Programming Example, an M3202A AWG and an M3102A digitizer are used to perform a real-time pulsed characterization experiment on a Device Under Test.

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

Programming Example 5 - Synchronized Multi-Channel Mixed-Signal Generation using M3xxA PXI Instruments

In this Programming Example, KS2201A PathWave Test Sync Executive is used to program multiple M3xxx Arbitrary Waveform Generators to synchronously generate mixed signals. Each instrument can be programmed to output either a front panel marker pulse or a previously queued waveform. All signal channels run fully synchronized and actions across instruments can be controlled with the timing resolution of the M3xxxA AWGs, which is 10ns.

Programming Example 6 - Synchronized MIMO Measurements using M5302A Digital I-O and M3xxxA PXI Instruments

In this programming example, PathWave Test Sync Executive is used to program multiple M5302A Digital I/O (DIO) and M3xxxA PXI instruments. By using HVI (Hard Virtual Instrument) capabilities, DIO instruments can output a pulsed signal from any of their Front Panel (FP) SMB trigger ports and M320xA AWGs can synchronously play a previously queued waveform. Multiple M3102A Digitizers can also be included in the same HVI to synchronously capture all the generated analog and digital signals. This way the example can showcase a Multiple-Input Multiple-Output (MIMO) measurement setup having all his input and output channels fully synchronized.

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). 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.

System Setup Guide

This document describes the different ways you can set up a single or multi-chassis system, with clocking and communications.

Transitioning from M3601A HVI Programming Environment to KS2201A PathWave Test Sync Executive

This Transition Guide is intended for M3601A users and explains how to translate an M3601A project into HVI API Python code programmed using Keysight KS2201A PathWave Test Sync Executive.

Edition 2022_U0_00, June, 2022
Keysight Technologies, USA

KS2201-90000
www.keysight.com