USER MANUAL

PathWave Test Sync Executive

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 21.0, December, 2021
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:

CE

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 주의하시기 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, or for more information see http://about.keysight.com/en/companyinfo/environment/takeback.shtml.

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 2021 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 2021\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 2021\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: Installing 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.

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

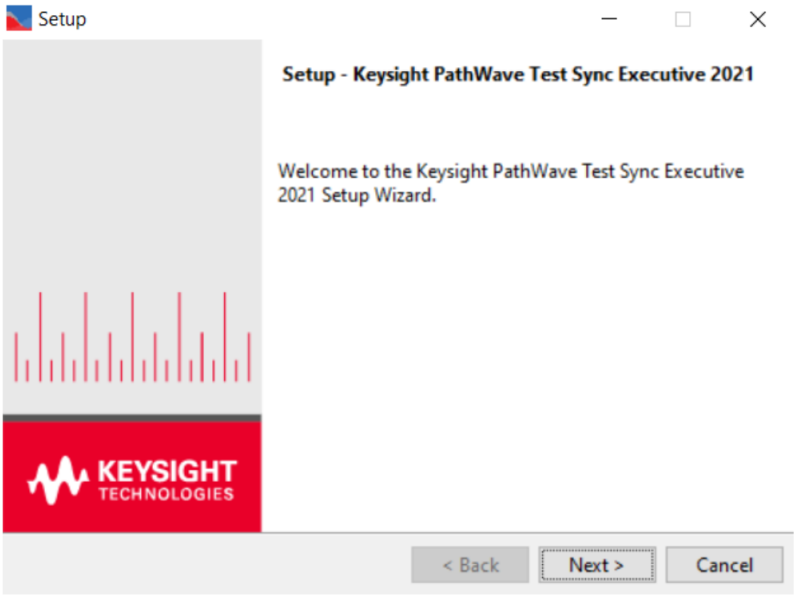
Use the following procedure to install PathWave Test Sync Executive:

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

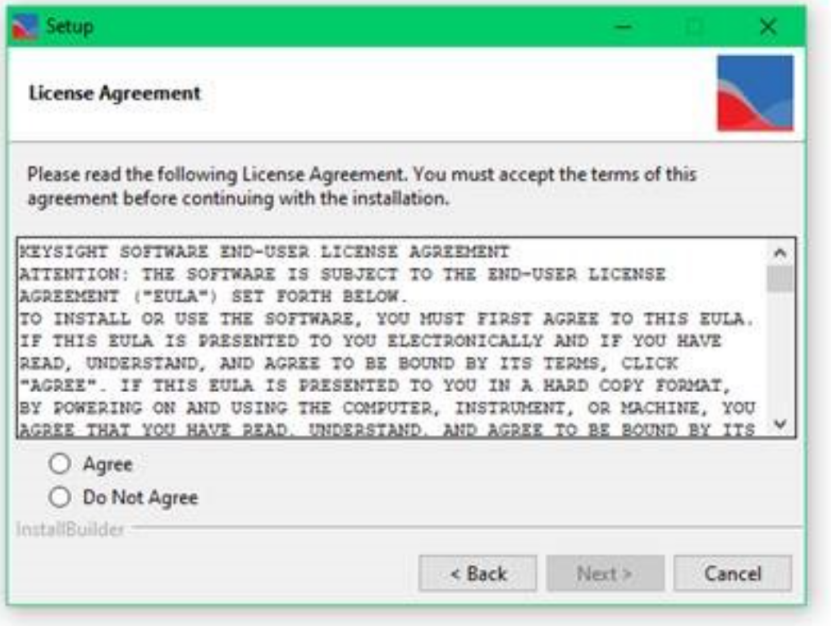
If PathWave License Manager is running, you must close it before installing PathWave Test Sync Executive.

Execute the installer file:

The **Setup** screen is shown:



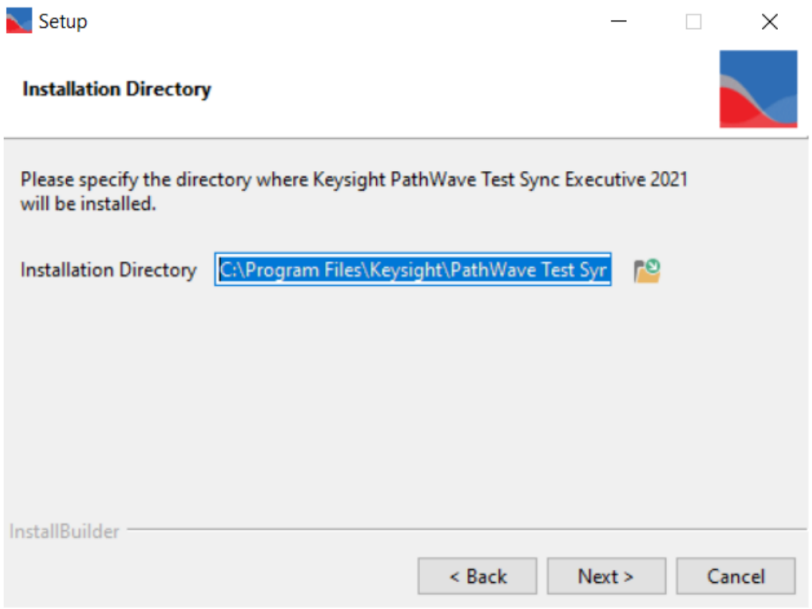
The next screen is the **License Agreement** screen. You must accept the license to continue:



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 2021
```

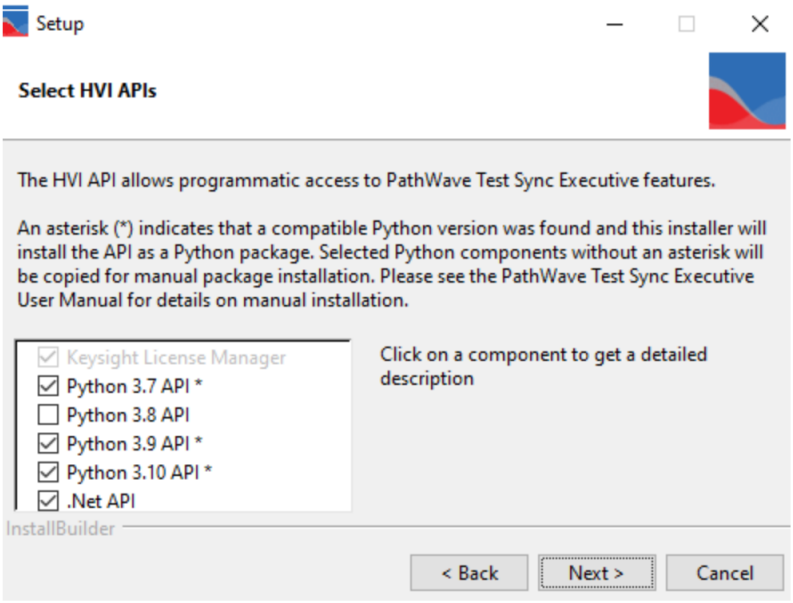


The **Select HVI APIs** screen enables you to select the Python API versions you want to install.

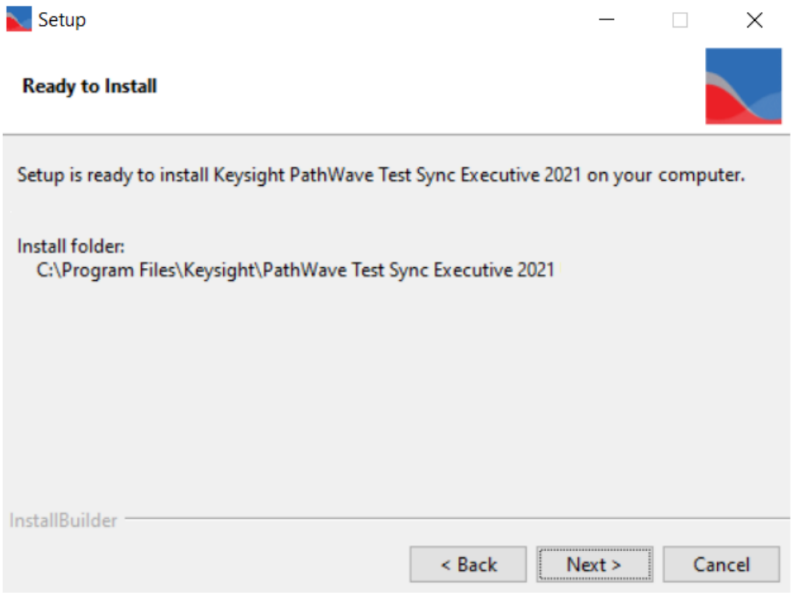
If a Python version component is marked with an asterisk and selected, the installer will install the Python package.

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.

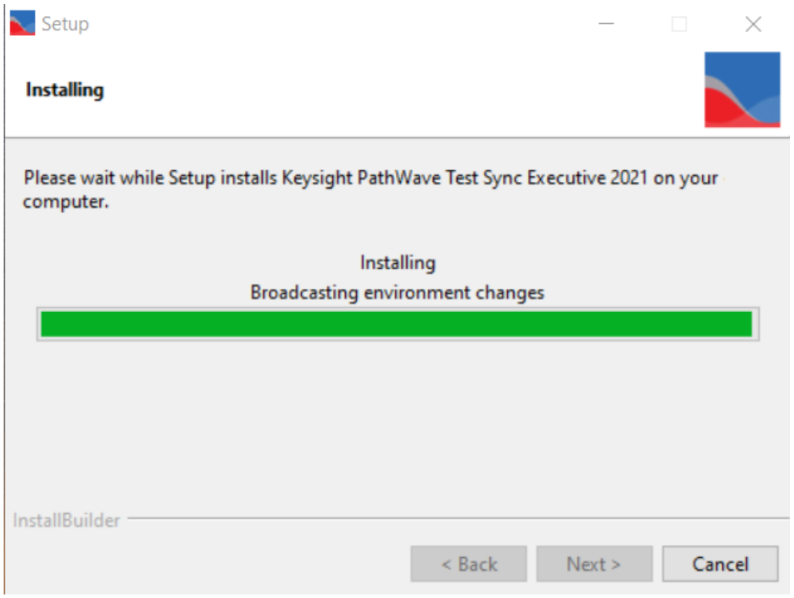
Required components are selected by default and you cannot de-select them.



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 2021\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

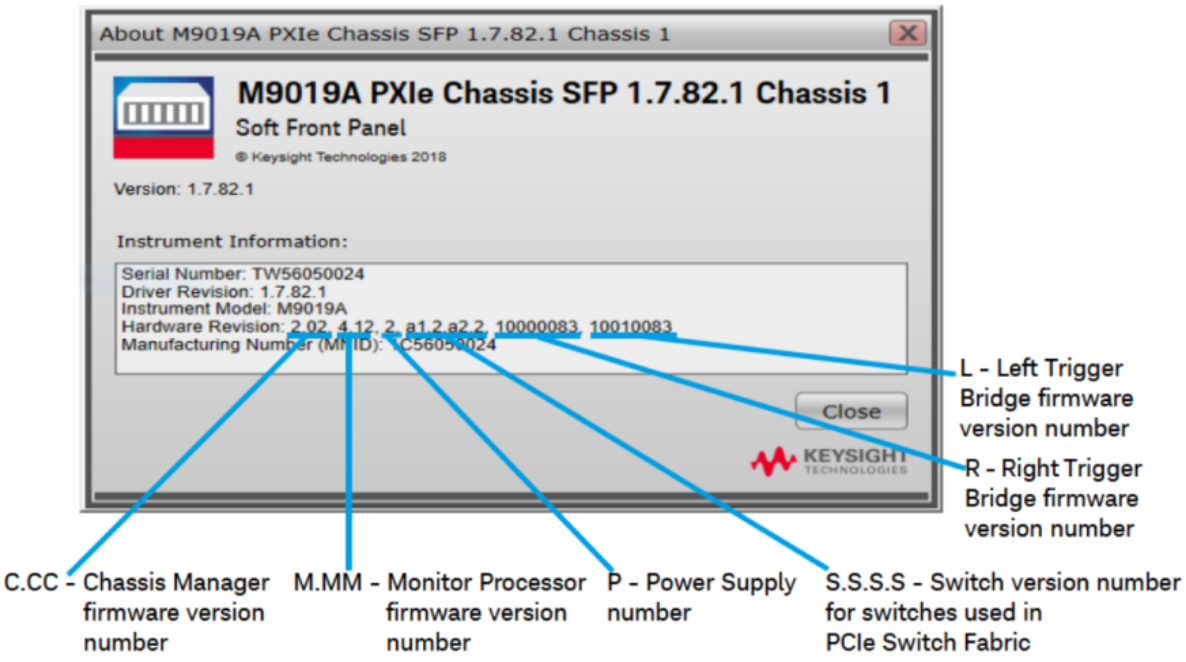
In PXIe Chassis Software Front Panel, you can:

- 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
- Supported Licensing Modes.
- The Licensing Process.
- Installing Licenses with PathWave License Manager.

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 support 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 2021\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 2021\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:

- SystemDefinition.
- Sequencer.
- Hvi.

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.

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.

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 PathWave FPGA.

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, and send 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 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), 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 you can use as resources in your HVI sequences. For more information see Chapter 5: HVI Integration with PathWave FPGA.

NOTE The resources that are 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 user manual.

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.

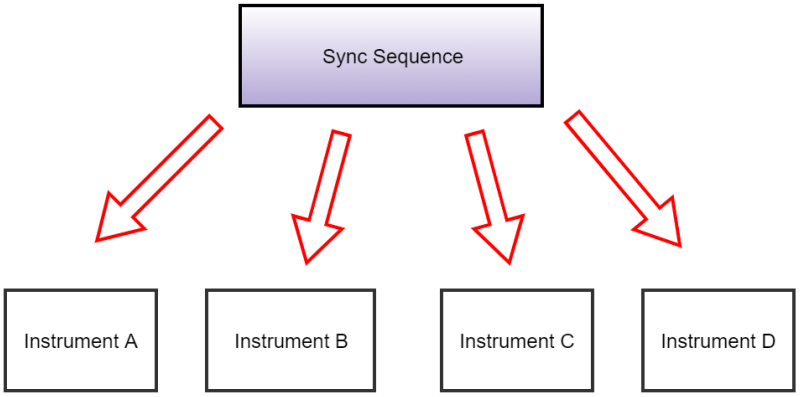
There are two types of sequences:

- 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:

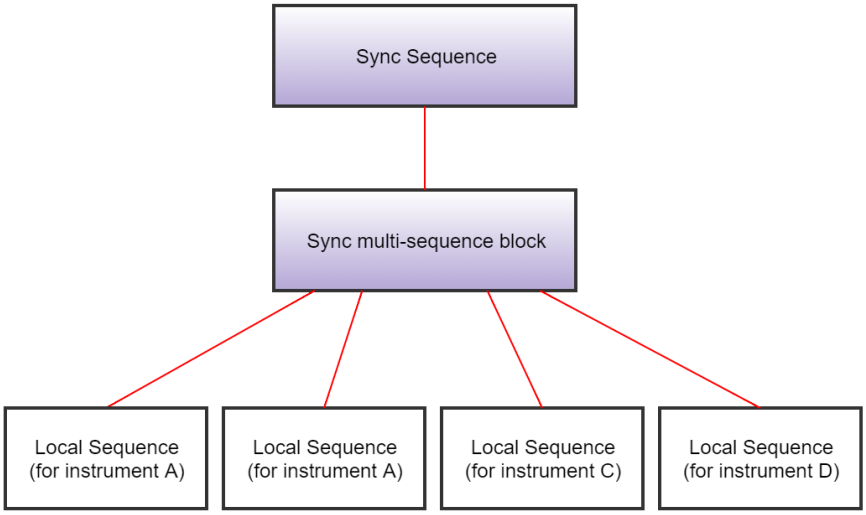


Local sequences

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:

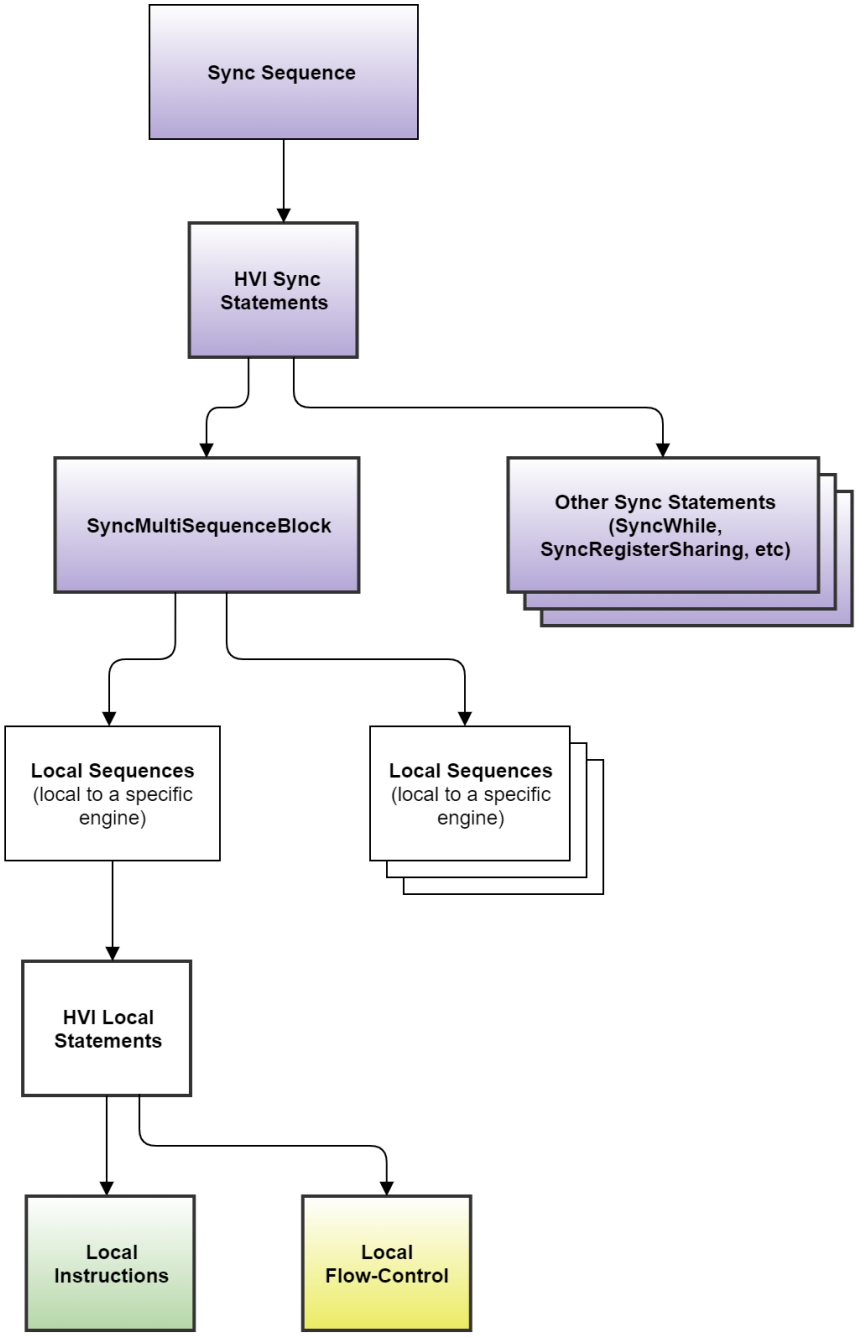
HVI Sync statements

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 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 multi-sequence block

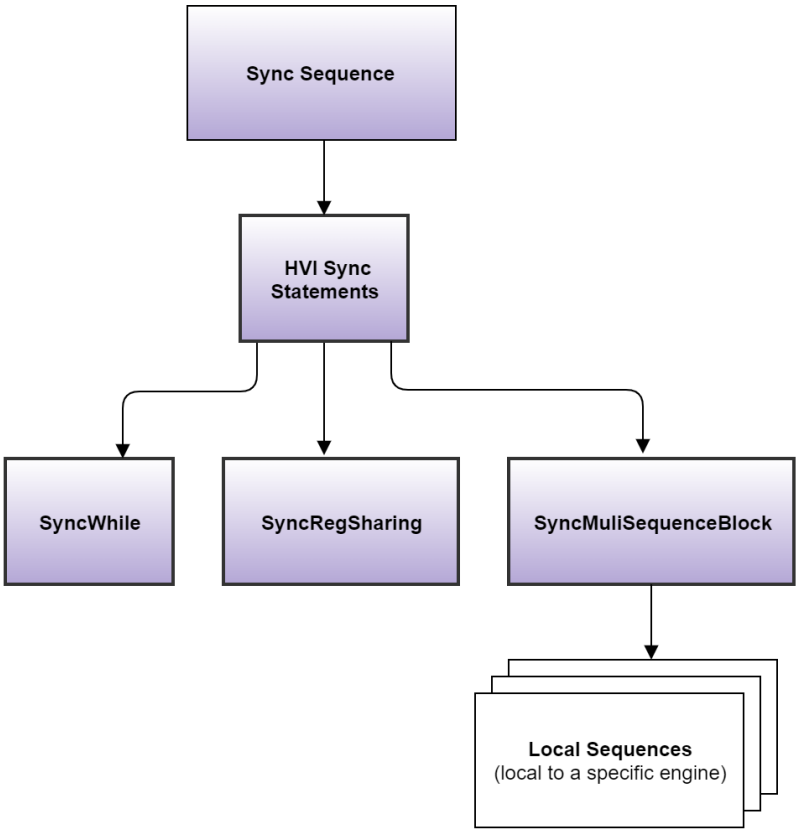
Enables the execution of multiple, simultaneous, engine-specific sequences.

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:

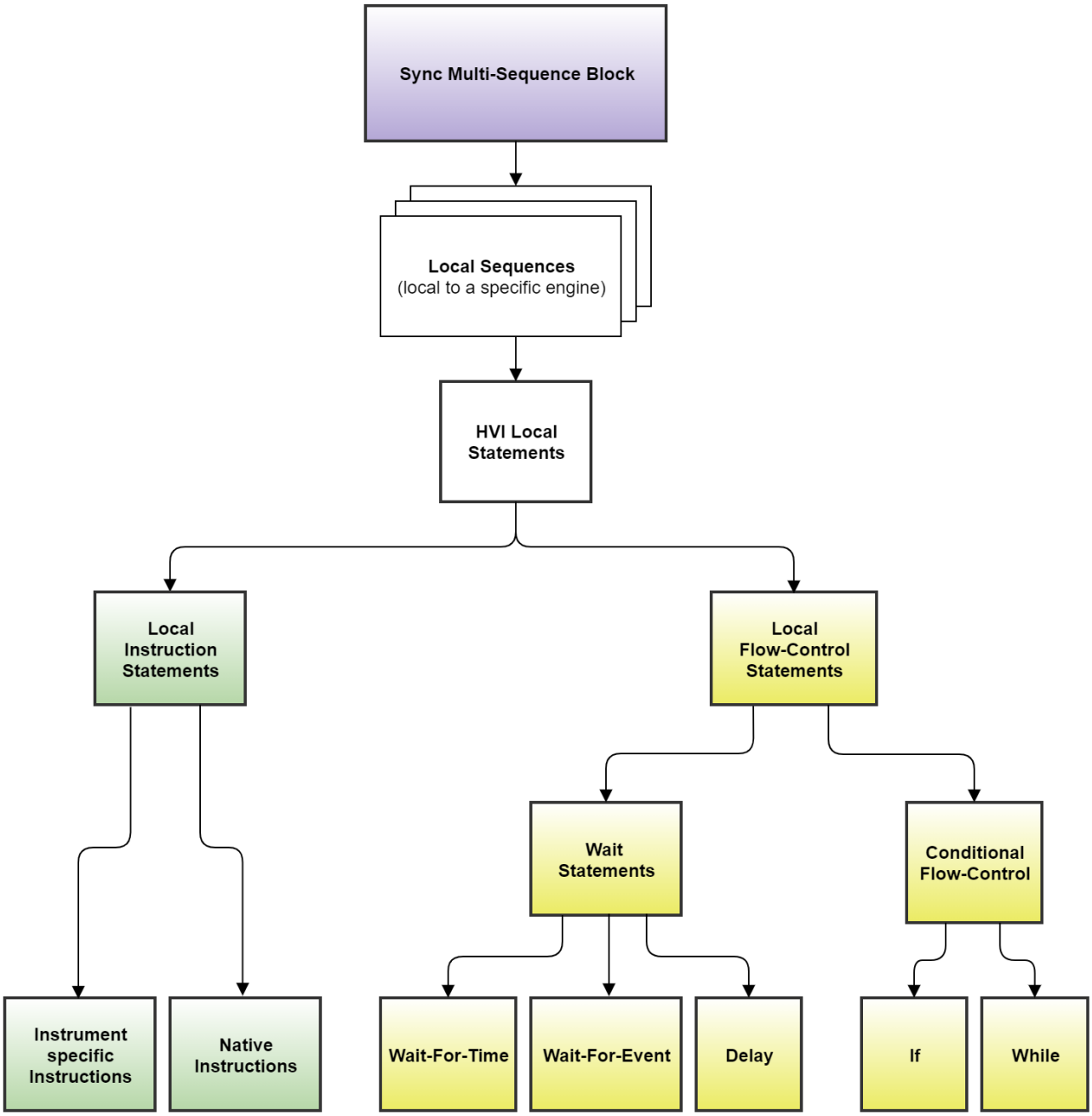
Local If

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.

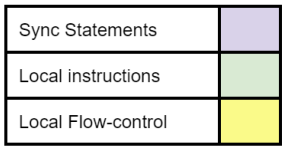
The following diagram shows the different types of Local statements and their relationship to the Local sequences:



HVI Diagrams

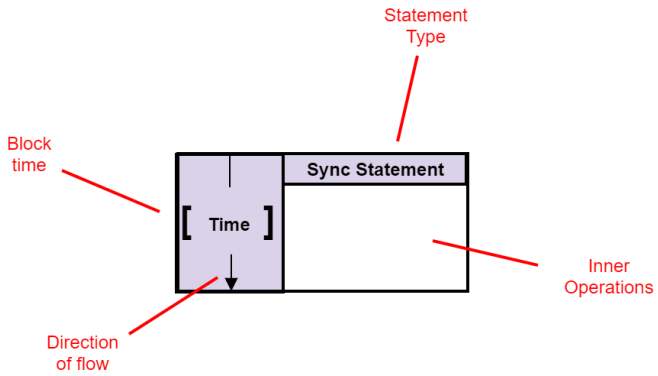
This section shows HVI diagrams. These are used to illustrate HVI sequences.

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

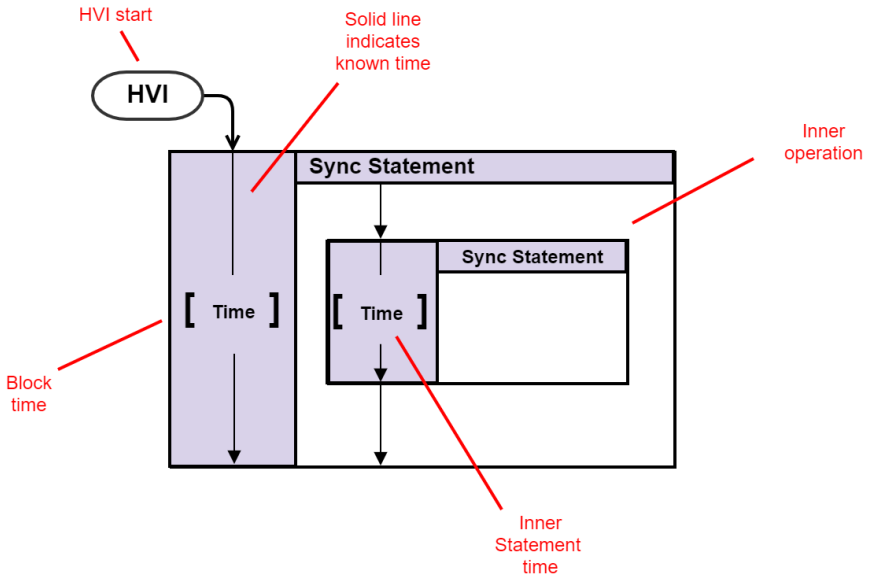


Statement diagram color code

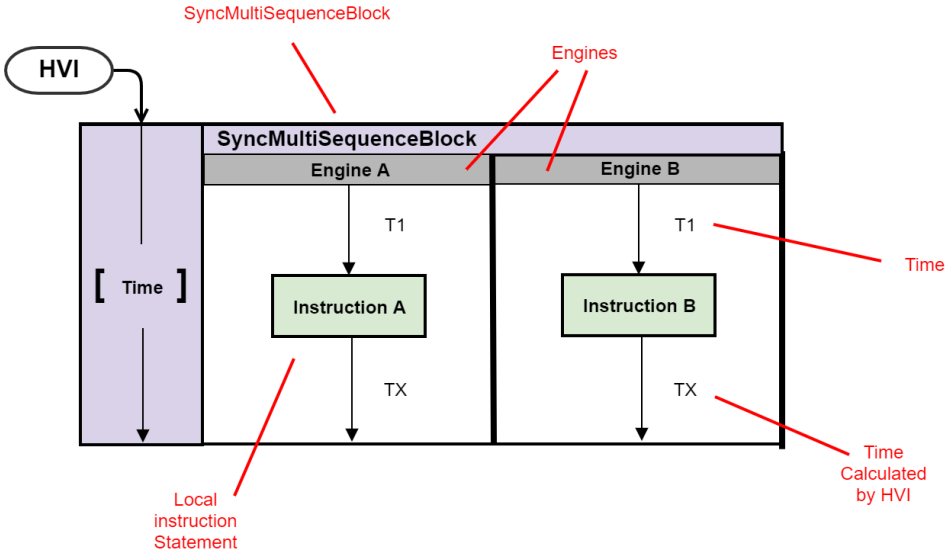
The following diagram shows a single Sync statement with flow and time for the block:



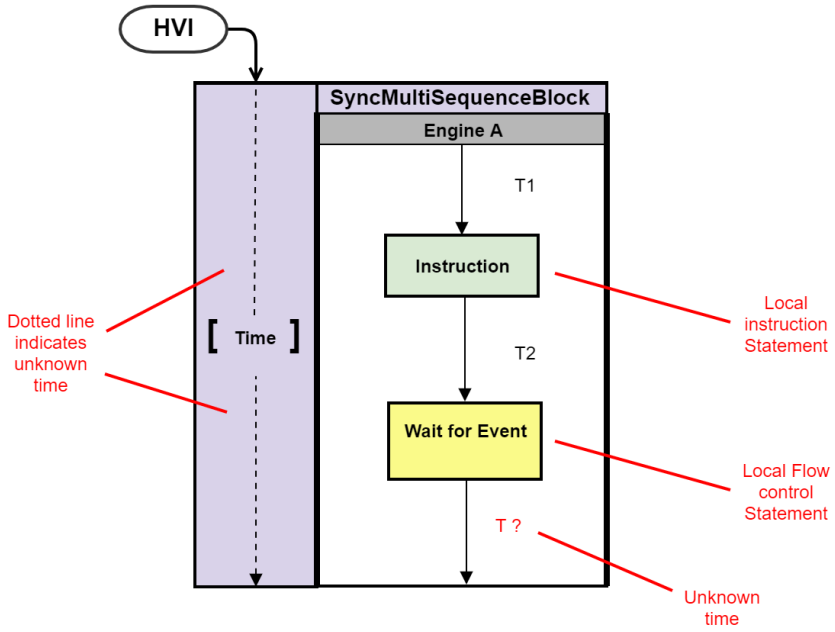
The diagrams can show nesting of statements within statements. For example, the following diagram shows a Sync statement that is within another Sync statement:



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:

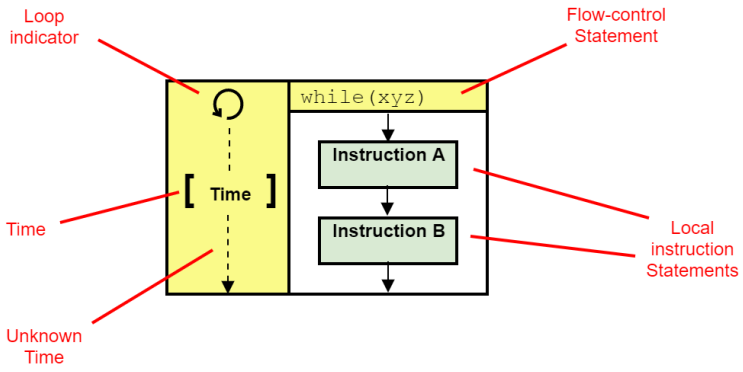


A dotted line indicates that execution time is not known at compile time. This is often the case with flow-control statements. In this case the Wait-for-event statement shall not release until the event occurs. It is not known at compile time when this is, so the time cannot be calculated at compile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

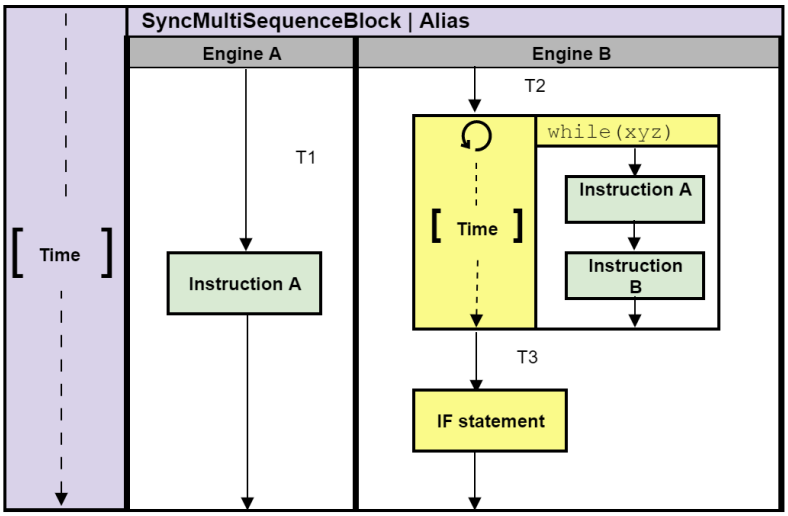


The following diagram shows a Local flow-control statement that encloses a pair of Local instruction statements. The color Yellow indicates a Local flow-control statement.

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 sequences 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

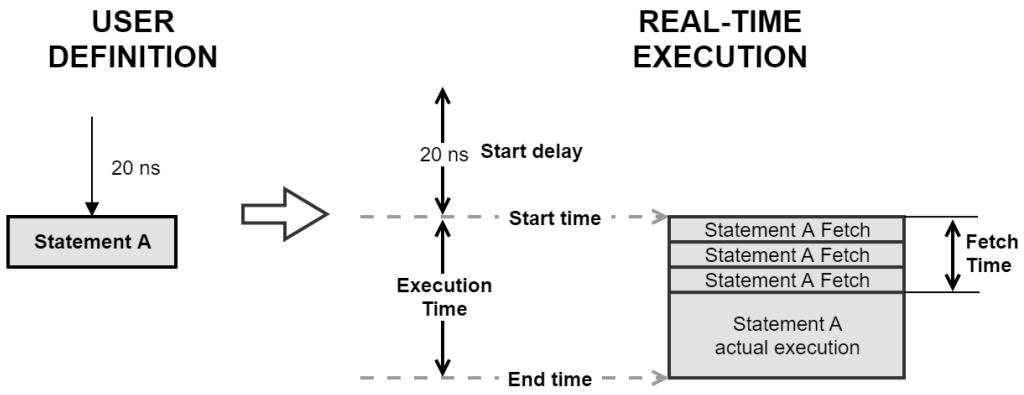
This is the time interval from the Start time to the End time of the statement. This interval is determined by instrument constraints and inherent limits such as propagation delays and resource availability. The Execution time includes the Fetch time.

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:

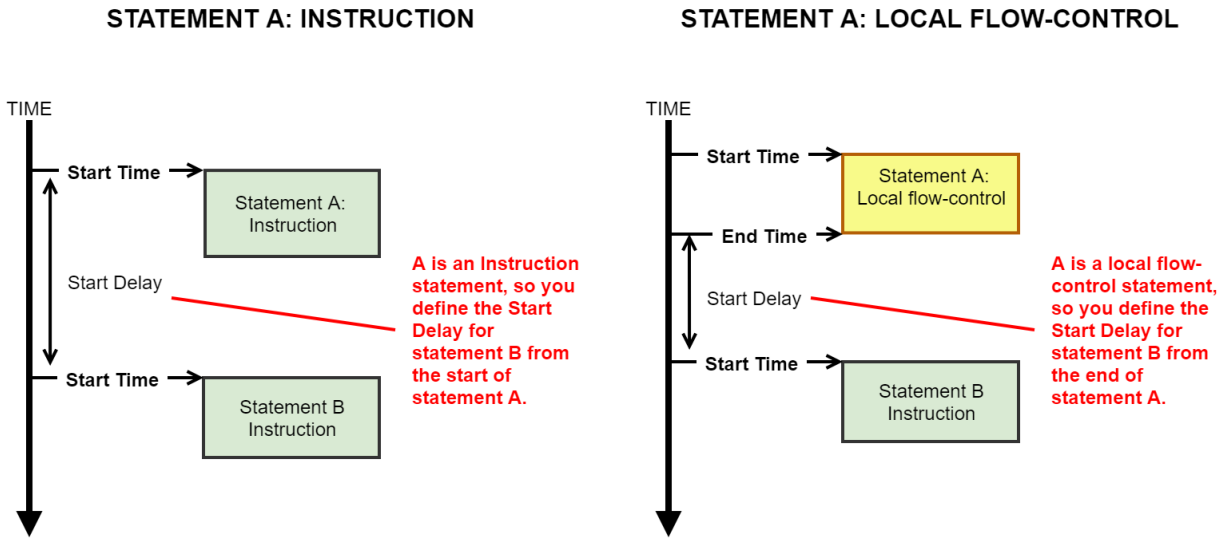
Instruction statements

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:



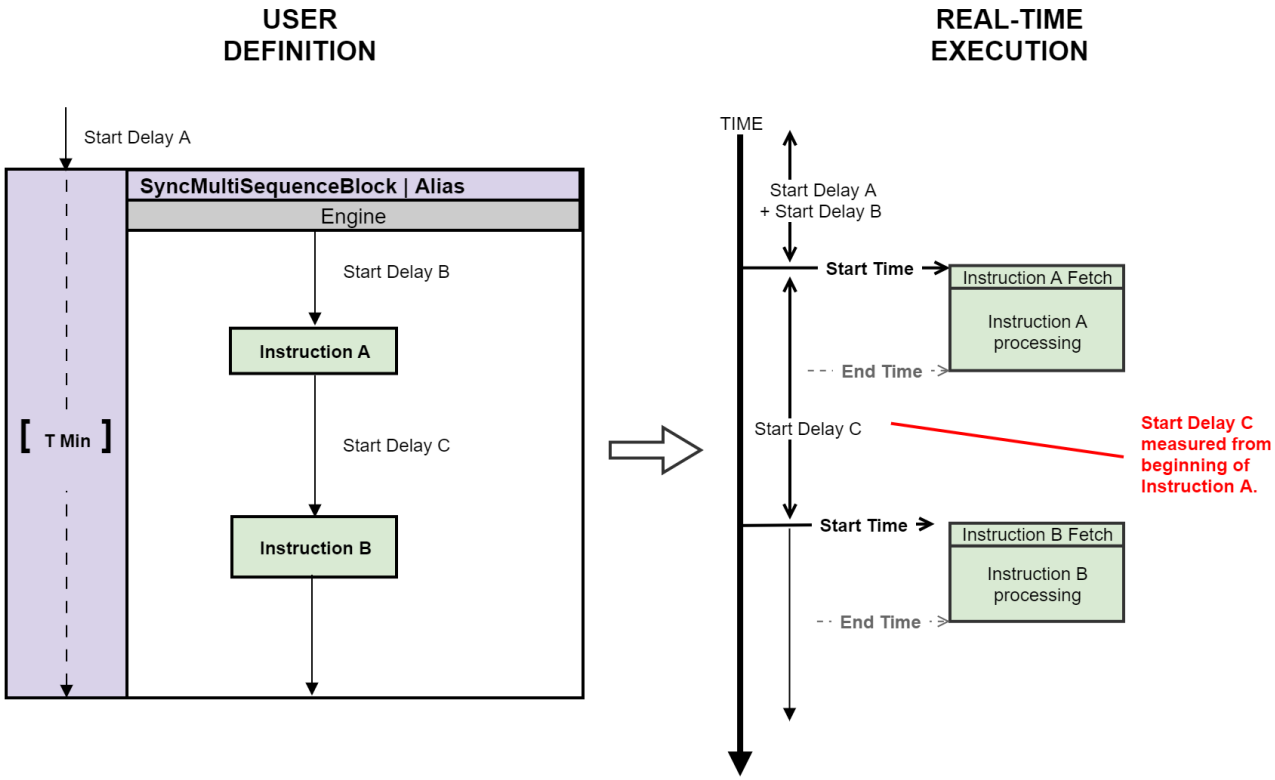
Local instruction timing

The following diagram shows the timing of Local instructions.

For 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

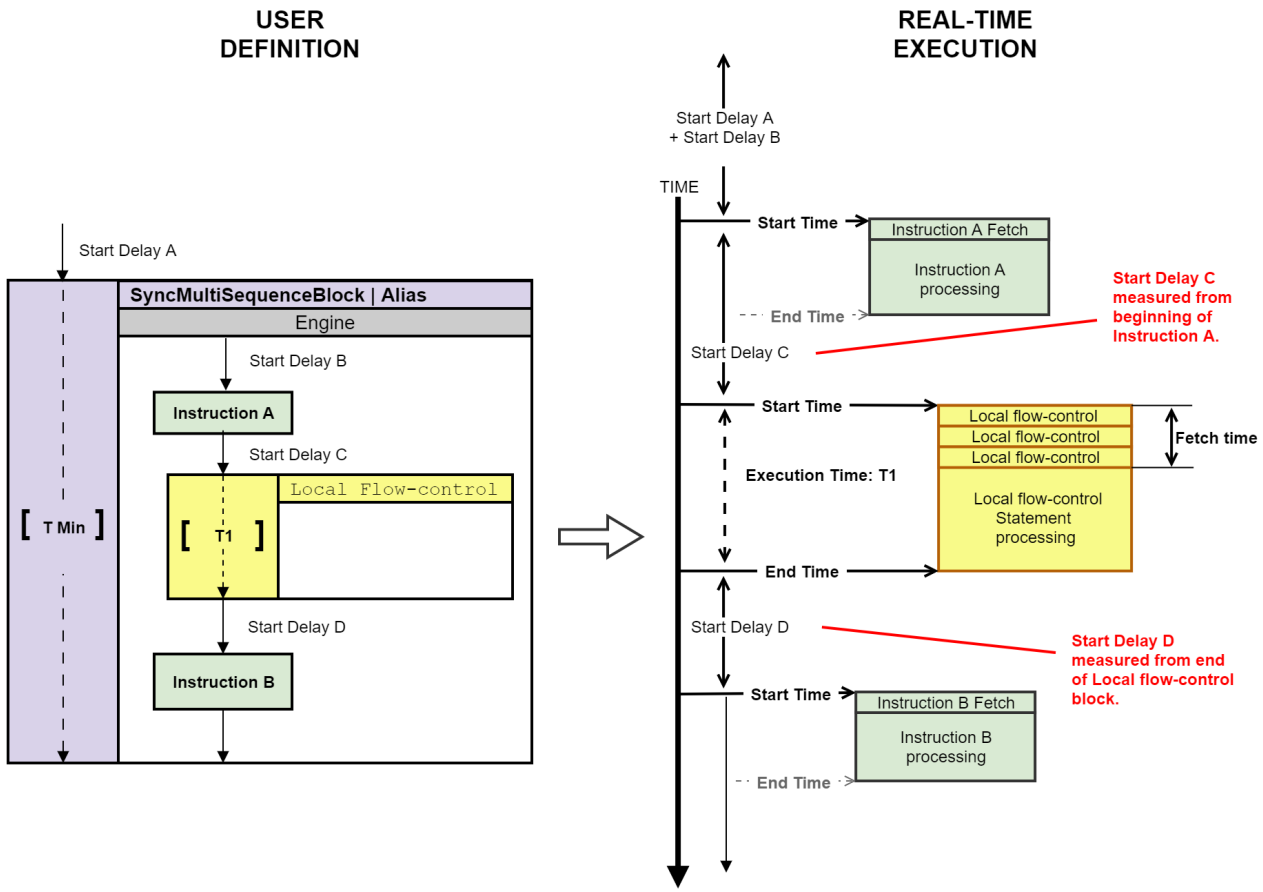
For Local flow-control statements, the Start delay of the next statement is measured from the end of the previous Local flow-control statement. This is because the HVI engine is busy during the execution of the flow-control statement and the execution of a flow-control statements cannot be overlapped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

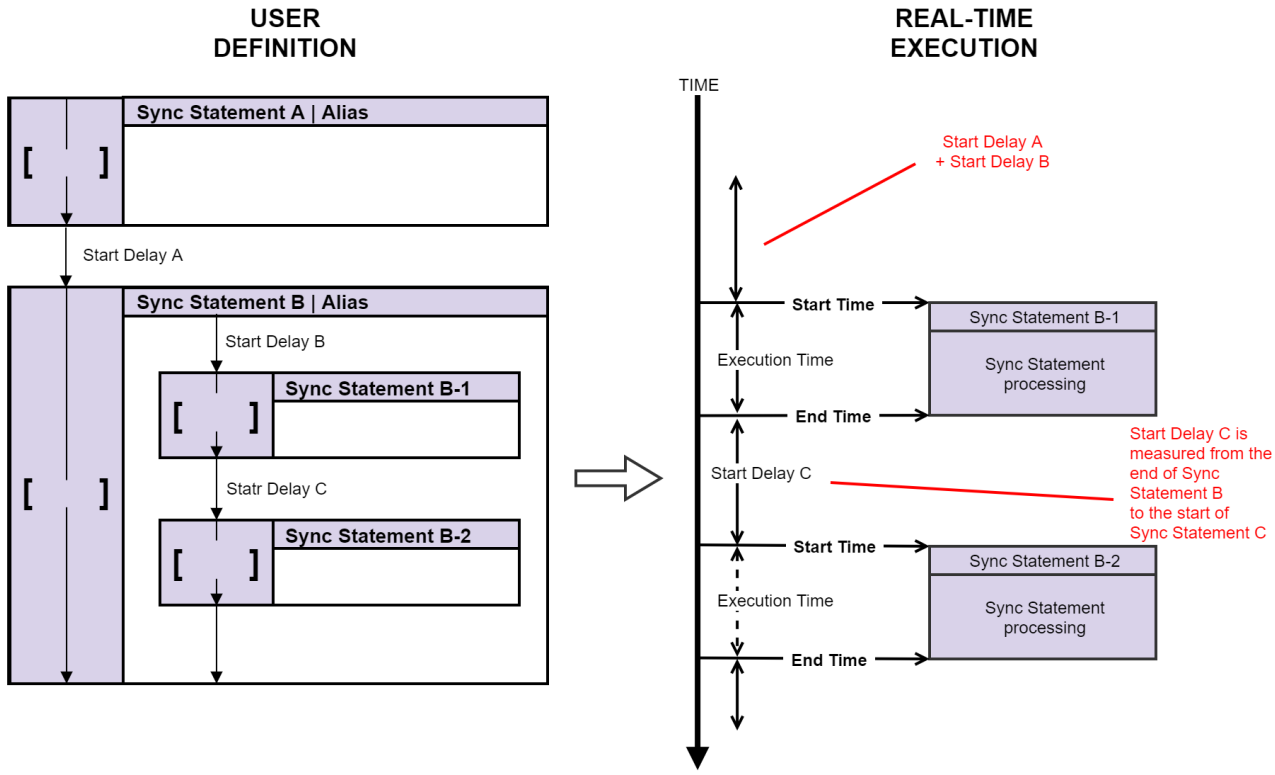
For Sync statements, the Start delayis measured from the end of one Sync statement to the start of the following Sync statement.

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 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.

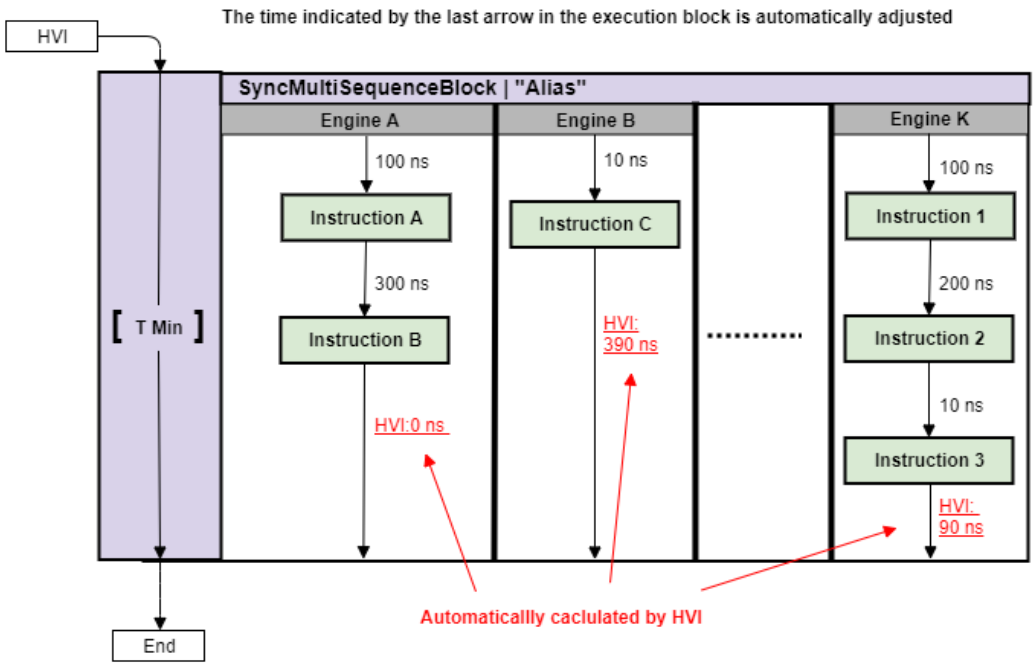
End times of sequences set to match the longest sequence (minimum execution time)

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:

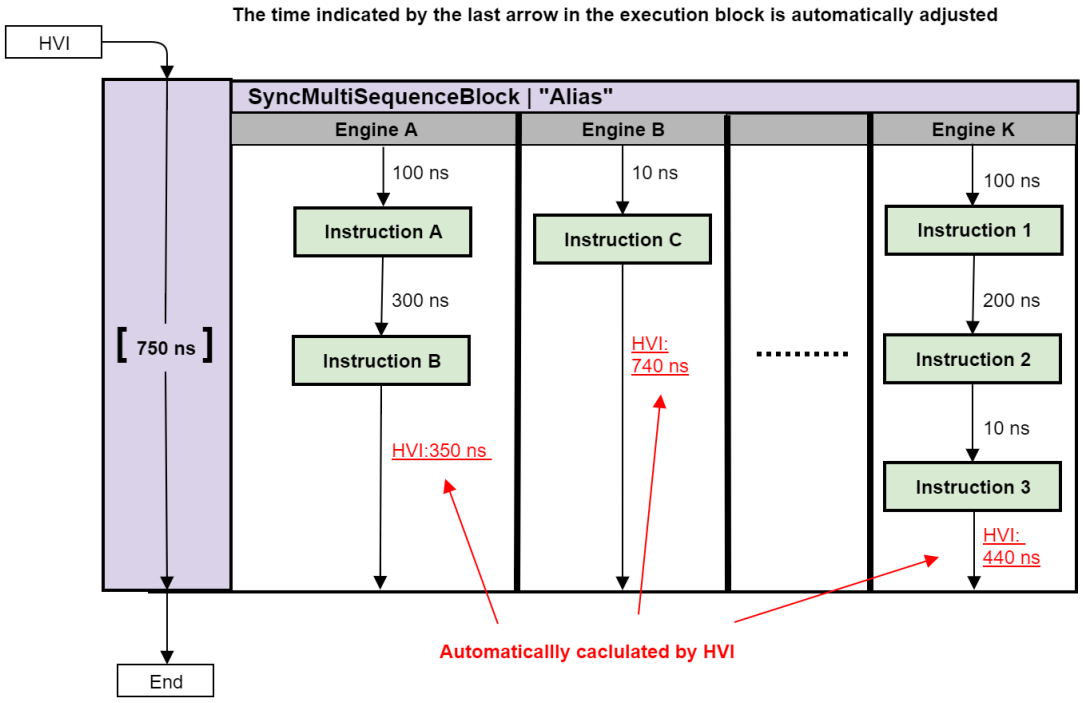


End times of sequences set to match a specific execution time

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 PathWave FPGA

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

FPGAs and HVI Real-Time Control

About FPGAs

PathWave Test Sync Executive enables you to combine your own custom digital signal processing with the precise timing capabilities of HVI by using the capabilities of the *Field Programmable Gate Array* (FPGA) technology.

Many Keysight instruments contain an FPGA. This is an electronic component that can be configured to enable different kinds of functionality. The FPGA in an instrument might include pre-programmed IP for the instrument's functionality and this can include HVI IP components and a region you can configure.

PathWave-FPGA is the software tool than enables you to configure the FPGA.

FPGA Sandbox

In addition to any existing IP and HVI engines, an instrument FPGA can include one (or more) FPGA sandbox. A sandbox is a region in the FPGA that you can configure.

You can configure the FPGA Sandbox to implement your own custom Intellectual Properties (IPs), signal processing and functionalities. This can include custom logic and memory. HVI can interact with this custom logic via an HVI interface.

An FPGA Sandbox contains the following components you can access from your HVI:

- FPGA Sandbox registers.
- FPGA Memory maps.
- FPGA Sandbox HVI interfaces.

FPGA Sandbox registers

FPGA Sandbox registers, are user-defined hardware registers that are similar to Variables in a programming language. Physically, sandbox 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 HVI software calls. Sandbox registers hold values that can be used read and used as values in statements in HVI sequences.

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.

FPGA memory map

An FPGA Memory map is a block of memory in the FPGA sandbox with a size and location that you define.

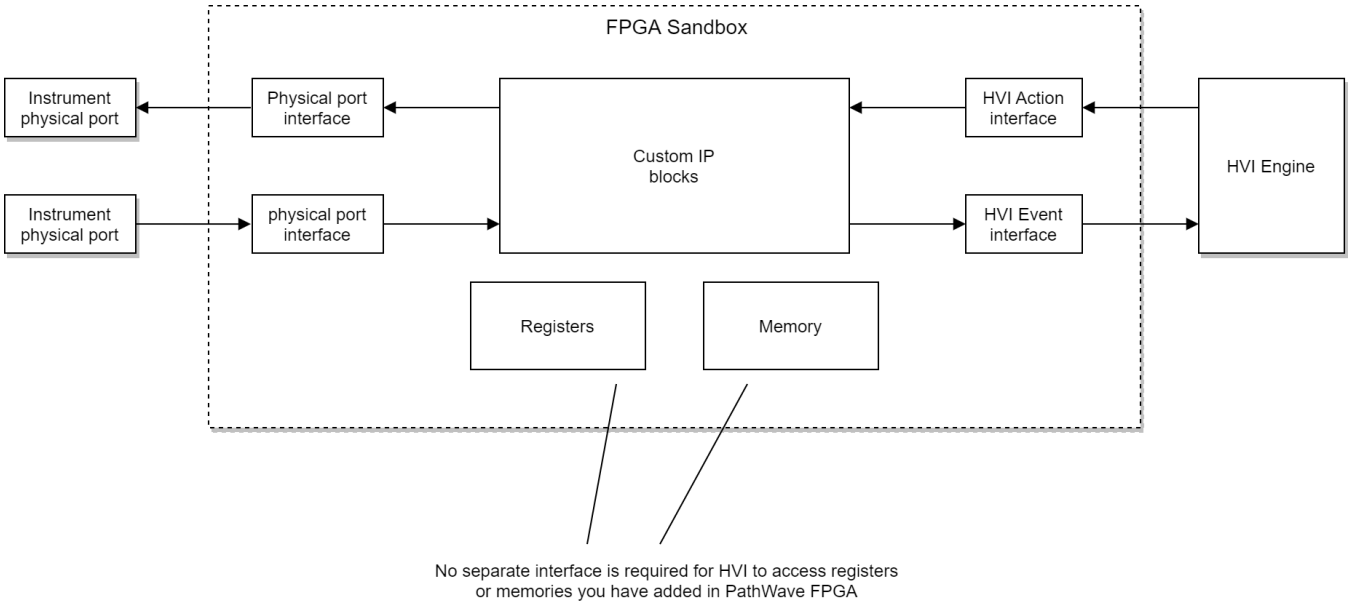
FPGA Sandbox HVI Interfaces

Instruments that support both PathWave FPGA and PathWave Test Sync Executive can benefit of a number of functionalities based on the interaction of HVI with FPGA sandbox IPs. For example, HVI sequences can access data contained in the HVI Registers and Memory Blocks that you have added to the FPGA sandbox using PathWave FPGA. To take advantage of this feature, you must use PathWave-FPGA to create your design in the sandbox. You are not required to add any specific interface components for accessing memory or registers in PathWave FPGA, but you can only access the registers and memory you have added. To do this you must use the same Names you used when you added them in PathWave FPGA.

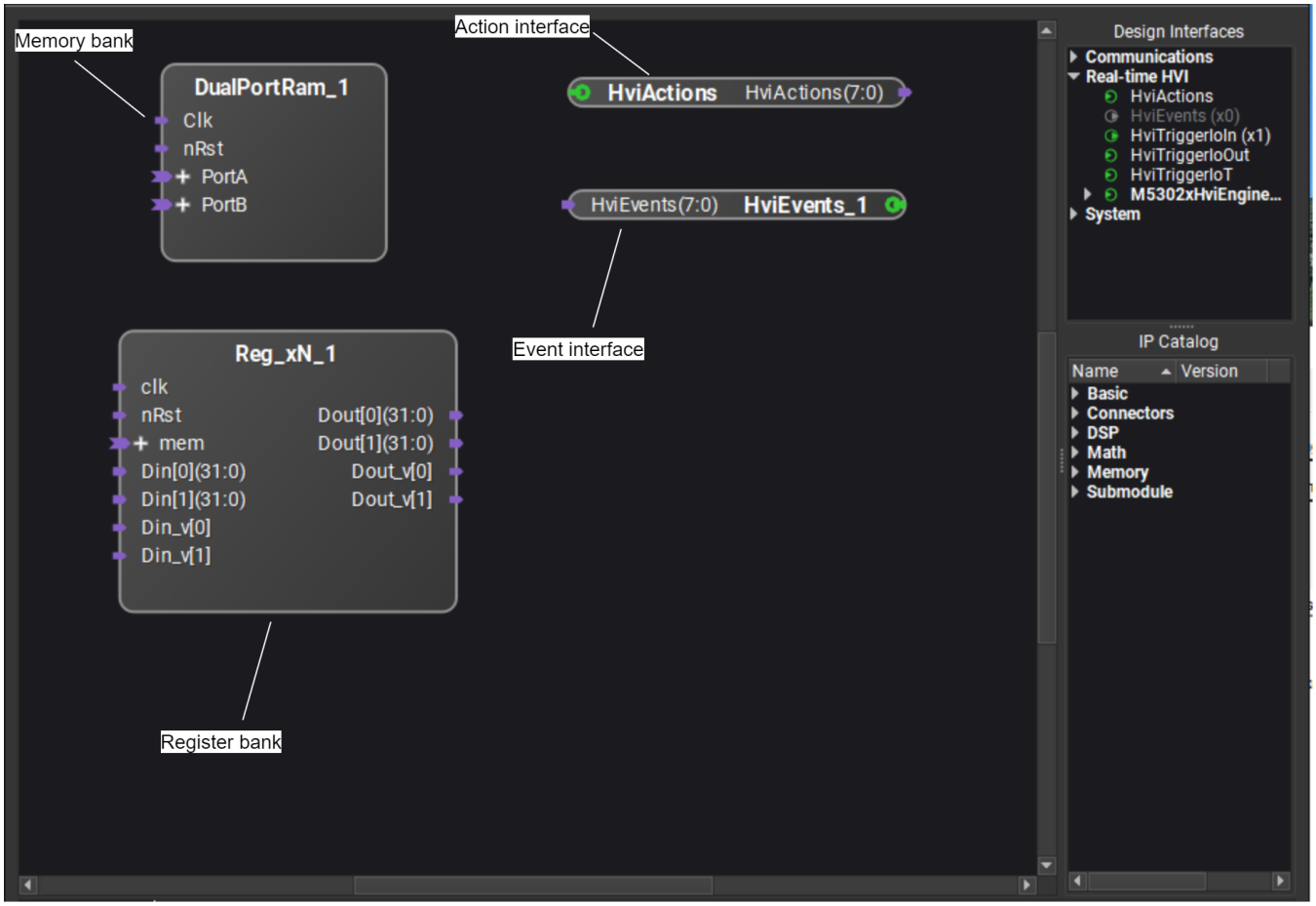
In PathWave FPGA, the **Real-Time HVI** interface catalog contains specific HVI interfaces available for HVI Actions, HVI Events, HVI Triggers and FPGAInstruction statements. PathWave Test Sync Executive includes a number of HVI instructions that enable your HVI sequences to interact with the IP blocks that you can place in the FPGA sandbox. The instructions include access to registers or memory maps, initiate actions, and send commands into the FPGA sandbox.

You add these interfaces for these specific functions. For example, if you want to send an action into the sandbox, you add an HVI Action interface to the sandbox and connect it to the relevant part of your custom logic.

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



The following diagram shows an HVI Action interface, an HVI Event interface, and a memory and register bank in PathWave FPGA:



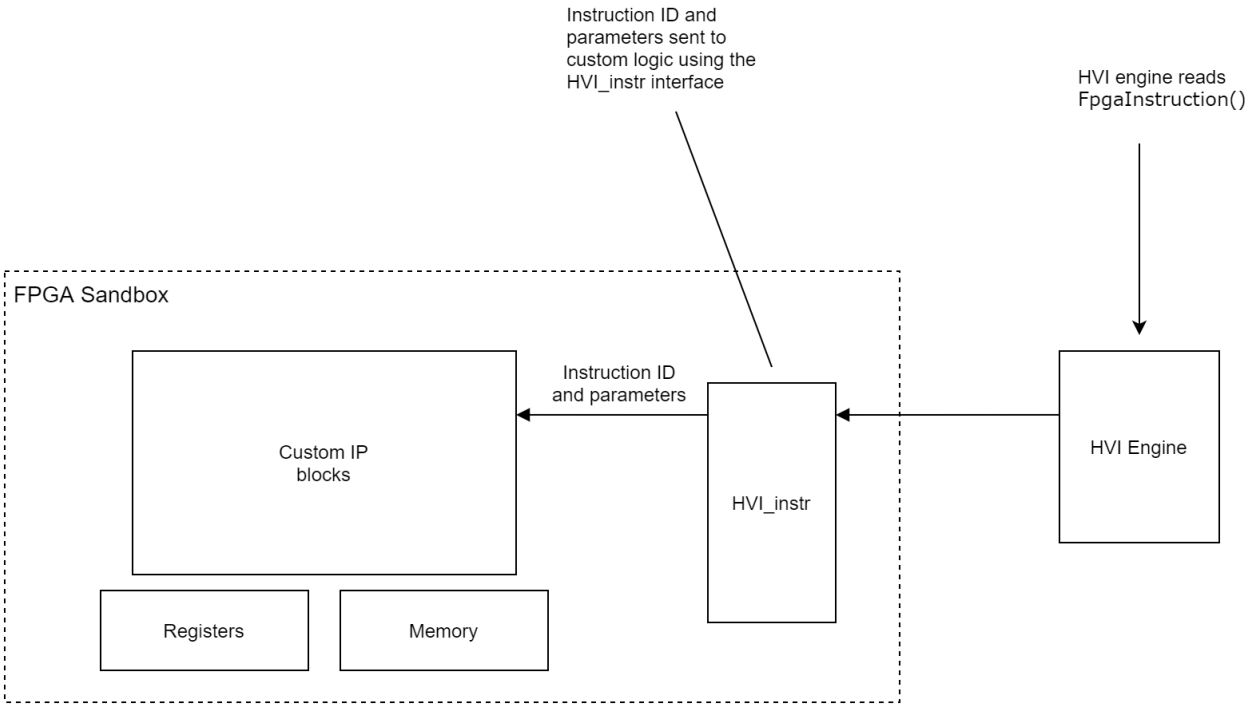
HVI statements and FPGA real-time interaction flow

When you run your HVI, the HVI engine reads and executes your HVI sequences. The HVI engine reads and executes the individual commands.

When the HVI engine executes an HVI statement that involves interaction with IPs placed in the FPGA sandbox using PathWave FPGA, the HVI engine communicates with the FPGA sandbox via an HVI interface or one of the interfaces you have added in PathWave FPGA. This can be to access memory or registers, or initiate an HVI action.

You can also issue commands to your custom logic using the `FpgaInstruction` statement. The `FpgaInstruction` statement sends the commands to your custom logic in the FPGA sandbox using an `HVI_instr` interface. For information on using the `FpgaInstruction` statement, see HVI Statements for using FPGAs.

The flow is shown in the following diagram, the HVI Engine reads the FpgaInstruction and parameters and the instruction ID and the data parameters are then sent into the Sandbox for the logic to process:



Configuring an FPGA for use with PathWave Test Sync Executive

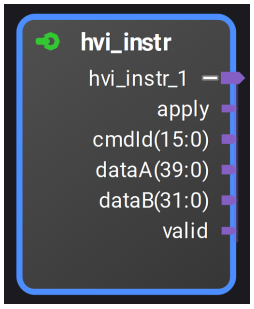
To configure the FPGA with custom logic and use this with PathWave Test Sync Executive, use PathWave FPGA to create your design in the FPGA Sandbox, PathWave FPGA is available from PathWave-FPGA.

You add your logic in PathWave FPGA and add the relevant HVI interfaces as required.

Integrating the FpgaInstruction Statement with FPGA Custom 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.

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



To use the FpgaInstruction, you must include the following in your custom logic:

HVI instruction interface

An HVI Instruction interface is required to receive the commands from the HVI engine.

Instructions defined for `instructionID`

You must define and implement execution logic for any custom commands you intend to include in your custom logic. You also require a means to decode the commands.

Registers for `DataA`

There must be logic to access the register that holds data that is provided by the `DataA` parameter.

Logic to read the Apply bit and act accordingly

There must be logic to handle if the the apply signal is set to `0` or `1` .

Using your PathWave FPGA Design with PathWave Test Sync Executive

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

Load the k7z into your HVI. This file is used by the HVI to get all the definitions required so you can utilize your customizations.

The .k7z file contains information about the Names, addresses, ranges of the registers, and memory-mapped locations that are connected to the HVI interface.

The access latency of the FPGA registers and memory maps from HVI depends on the implementation of the specific instrument. For more information, see Chapter 9: HVI Time Management and Latency and your Instrument User Manual.

HVI Statements for using FPGAs

PathWave Test Sync Executive includes a number FPGA specific HVI statements you can use to interact with the FPGA. This section describes the statements:

- FPGA register read
- FPGA register write
- FPGA memory map write
- FPGA memory map read
- FpgaInstruction statement

FPGA register read

The instruction `fpga_register_read` is an HVI-native instruction that enables you read from an HVI register. The value read from the HVI 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 register placed in an FPGA sandbox. The value to be written to the HVI 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 memory map. The value read from the HVI 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 memory map that is located in an FPGA sandbox. The value to be written to the HVI 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 your 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 reads the `FpgaInstruction` statement and sends the command and data parameters to your logic in the FPGA sandbox. Your logic reads the parameter data and executes the command as indicated.

FpgaInstruction 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 bits are set to 0. • If the source register is a `long` (48 bits), the top 8 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	Default is 1 • 0 = Set up now, apply the instruction later • 1 = Apply instruction immediately

[1] To apply the instruction after setup, you initiate it with the HVI Action, Apply action. For more information about the HVI Actions available in your instrument, see your instrument User Manual.

The set up now, and apply later option enables you to set up the command and then delay the execution so it happens at a specifically timed interval. This can enable you to set up a number of commands and then have them execute simultaneously.

The following code shows an example of an FpgaInstruction:

```
# 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)
```

Using FPGA Resources in an HVI

You use FPGA resources in your HVI in a similar way to other resources. The following section explains what to do in the different HVI programming stages:

SystemDefinition

When you are setting up your HVI, you define FPGA sandbox resources in `SystemDefinition`.

When you must define and configure the HVI resources, you must get the `SandboxCollection` class for the FPGA sandbox.

The object for each instrument is already populated with the relevant sandbox or sandboxes if there are more than one in that instrument. Sandbox objects do not need to be added to the collection, you only need to access them.

```
# NOTE: The M5xxxA_sandbox_Name is not arbitrary and cannot be changed

 M5xxxA_sandbox_Name = "Application"

#

# Get engine sandbox

sandbox = sys_def.engines["M5302A_Engine"].fpga_sandboxes[M5xxxA_sandbox_Name]
```

Programming sequences

When you are programming sequences, you must first ensure the sandbox is loaded. This involves loading the **`.k7z`** file that was produced by PathWave FPGA.

```
# Populate the HVI sandbox with a list of IP blocks contained in the bitfile, so that
HVI can use them
sandbox.load_from_k7z(bitfile)
#
# Partially reprogram the instrument FPGA using the bitfile compiled by PathWave FPGA
module.fpga_sandboxes[M5xxxA_sandbox_Name].configure_from_k7z(bitfile)
```

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 your HVI sequences. You must use the same Names used in the PathWave FPGA project to access the FPGA resources.

In the following example, the register Name `Register_Bank_MyCounter` is not arbitrary but the Name of a register bank in the PathWave FPGA project that generated the file **`MySandboxProject.k7z`** :

For example:

```
sandbox = engine.fpga_sandboxes["sandbox0"]
sandbox.load_from_k7z("MySandboxProject.k7z")
counter_register = sandbox.fpga_registers["Register_Bank_MyCounter"]
```

The following code shows how to write to an FPGA register:

```
# 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)
```

The following example shows how to read from an FPGA array:

```
# 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))
```

FPGA Actions, Events and Triggers

You can set Actions, events and triggers for FPGAs.

Actions

You set Actions to send signals into the FPGA.

For example, you can use an action to tell an FPGA to send a signal.

Events

You set Events to inform your HVI of events in the FPGA.

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

You use the Wait for event statement to command your HVI to wait until an event occurs. The signal going into the FPGA can initiate the event.

Triggers

You can set Triggers to go into or out of the FPGA.

For example, a trigger can initiate an event in the FPGA. You use the Wait for event statement to command your HVI to wait until an event occurs. A trigger going into the FPGA can initiate the event.

You can also use an action to initiate the FPGA to send a trigger out.

Load to Hardware

Once the instrument has been loaded to hardware, you can write to the FPGA memory map.

For example, in Python:

```
memory_map.write(0, 1)
```

A full example is listed in Chapter 8: Building an Application with the HVI API.

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:

System Synchronization Modules

This section describes System Synchronization Modules.

KS2201A 2021 release introduces new multi-chassis topologies that use the Keysight M9032A/M9033A System Synchronization Modules (SSM). The previous means of interconnecting multiple PXI chassis using M9031A modules is discontinued starting from the KS2201A 2021 release. The SSM has a much wider range of functions including compared to the discontinued M9031A module:

- Distribution of a precise clock reference.
- Chassis interconnectivity.
- Synchronization of all PXI instruments in the multi-chassis.

Keysight M9032A/M9033A PXIe System Synchronization Modules

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

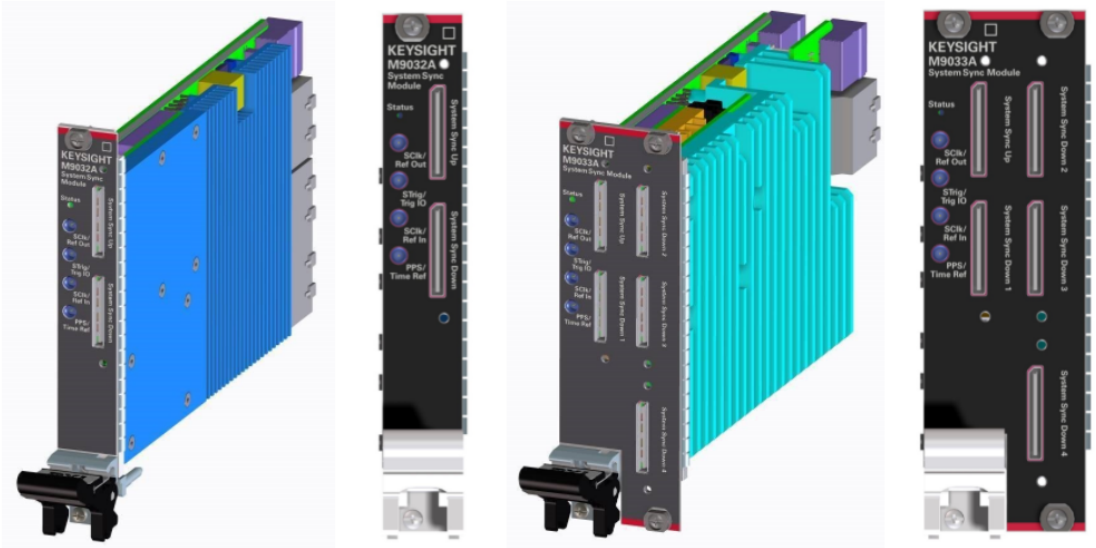
For further information about the System Synchronization Module (SSM) including detailed performance specifications please consult the **M9032A/M9033A User's Guide,** available at Keysight PXI Products.

M9032A and M9033A

Keysight PXIe System Synchronization Module is available in two form factors: M9032A and M9033A.

- M9032A is a one-slot PXIe System Synchronization Module that goes into the system timing slot of a PXIe chassis.
- M9033A is a two-slot version of PXIe System Synchronization Module that occupies two slots including the system timing slot of a PXIe chassis.

The following image shows the physical M9032A and M9033A SSMs:



System Sync Module Connectivity

The M9032A and M9033A System Sync Module front panel contains various connectors that can be used for both multi-chassis interconnection and configuration of the reference signal source.

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

SClk/Ref Out:

Outputs a copy of the system clock or a timebase reference clock signal with a frequency of 10 or 100 MHz.

STrig/Trig IO:

Receive an arbitrary trigger signal with a frequency between DC and 10MHz.

SClk/Ref In:

Receives the system clock or a timebase reference clock signal with a frequency of 10 or 100 MHz.

PPS/Time Ref:

Receives a Pulse Per Second (PPS) signal.

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

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.

All System Sync ports use PCIe OCuLink (Optical Copper (Cu) Link) connectors. System Sync ports can be used for chassis interconnection and synchronization in the multi-chassis system. The signals shared over System Sync connections can be shared for this. Each System Sync Downstream port can connect to the System Sync Upstream port of another System Sync Module (placed in a different chassis) or to up to two System Link ports placed on PXIe modules. Keysight System Sync and/or System Link cables can be used for such connections.

System Sync Module Reference Clock

One important function of a System Sync Module is the distribution of a high-fedility system reference clock to achieve precise synchronization among various measurement instruments distributed across different chassis. in some configurations, this system clock also acts as a timebase reference clock.

Each SSM is equipped with an onboard high-quality 10MHz Oven Controlled Crystal (Xtal) Oscillator (OCXO) that you can use as a timebase reference clock source.

The reference clocking can be set in two different modes:

Internal:

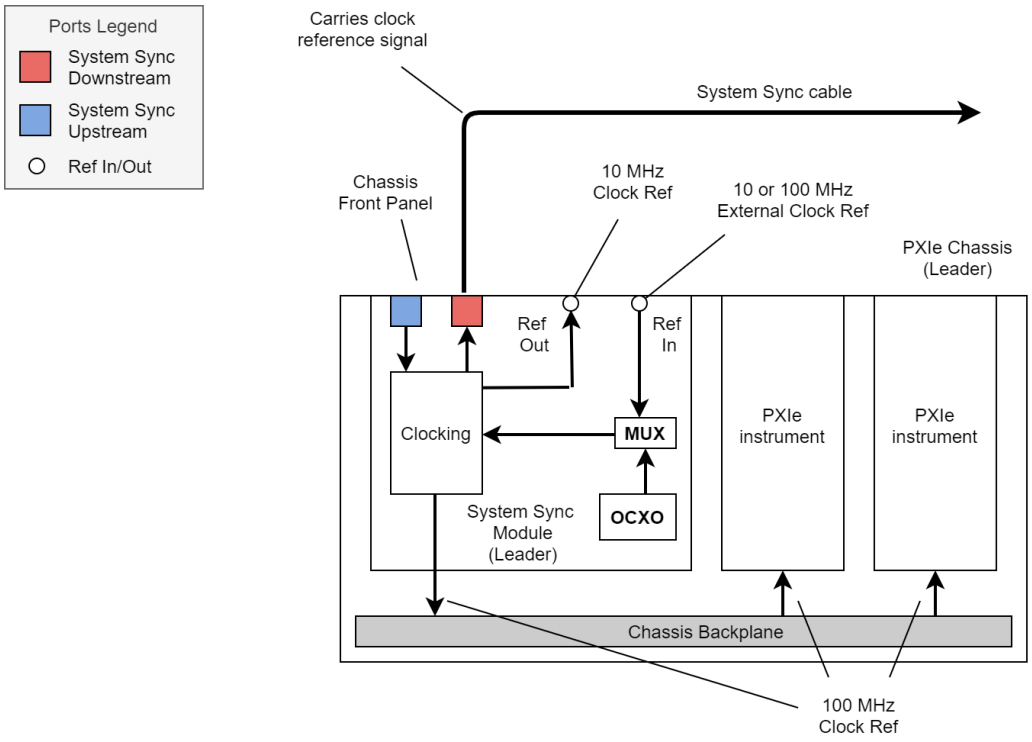
This is the default mode. The reference clock generated by the SSM or chassis get propagated across the mutli-chassis system.

External:

The multi-chassis reference clock signals synchronize with a clock signal coming from an external source, e.g. a DUT (Device Under Test) or another component of the measurement system.

In all possible configurations, the reference clock signal source always drives an internal PLL (Phase Locked Loop) inside the System Sync Module. The PLL generates a 100 MHz clock signal whose phase noise and signal characteristic depend on the purity of the source driving the PLL. The 100 MHz reference clock gets propagated to all the PXIe instruments within the same chassis through the DSTARA signal path and to the next SSM through a connection by means of System Sync cable.

The following diagram illustrates the SSM functionality as a reference clock and the ports and connections used to connect or propagate the clocking signals.



Multi-Chassis Interconnection, Synchronization and Data-Sharing Functionalities

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

- Multi-chassis interconnections are made with System Sync connections via their capability to interconnect two SSMs together through their System Sync Downstream/Upstream ports.
- Multi-instrument interconnections inside chassis are made with point-to-point DSTARA/B/C connectors. The SSM can share the precise reference clock over the DSTARA bus, trigger/data over the DSTARB/C buses, and PXI sync signals over the PXI_TRIG[0:7] bus.

The high-precision reference clock can also be shared between two interconnected SSMs using the System Sync connection between System Sync Downstream/Upstream ports. The same System Sync connection can share from one SSM to the next one, the signals that are sent over the PXI_TRIG[0:7] trigger buses. This way the SSMs can share PXI sync resources used by PathWave Test Sync Executive for the HVI (Hard Virtual Instrument) to execute across the multiple chassis. The SSM can route the data interchanged by two modules located in the same chassis by using the DSTARB/C buses. Data can be passed through the System Sync connection to route it to instruments located in a different chassis. Finally, data sharing can be routed also by connecting any of the SSM System Sync Downstream ports to up to two different instrument System Link ports.

Setting up a Multi-Chassis System with System Synchronization Modules and PXI Chassis

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

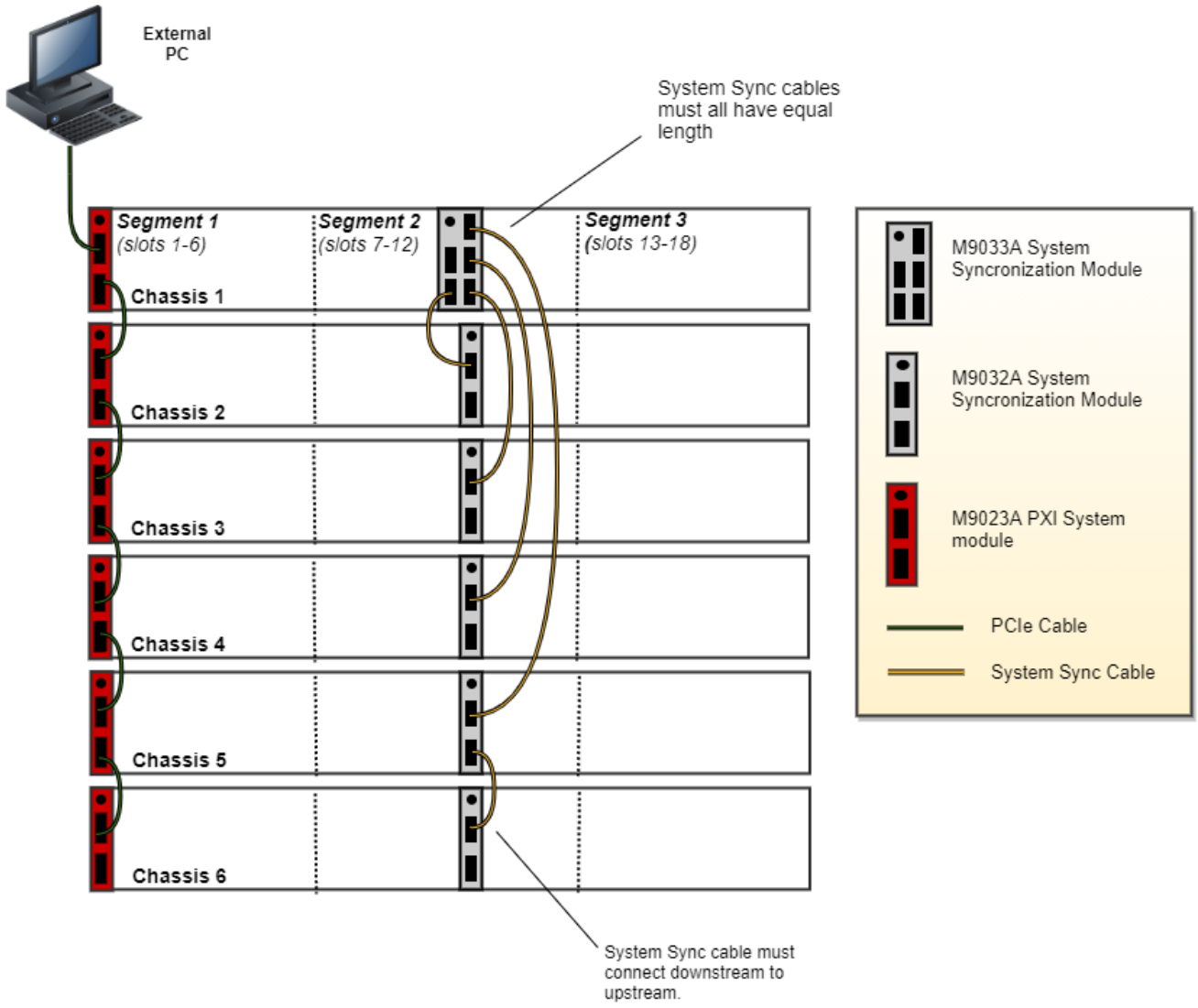
In a multi-chassis system connected with Keysight PXIe System Synchronization Modules (SSM), you must plug one SSM in each chassis that is part of the system. Each SSM must be inserted in the **timing slot** of your chassis. This is typically slot 10 in Keysight 18-slot chassis, but it can be a different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

One SSM is chosen as a leader and it is used to synchronize all the instruments included in the multi-chassis system. The SSM acting as a leader is passing the reference clock signal to the other SSMs located in the other chassis through point-to-point connections between System Sync Downstream/Upstream ports. The leader SSM is the one that has no incoming connection to his System Sync Upstream port and it only distributes on the reference clock (and other signals) from its System Sync Downstream port(s). In the example multi-chassis system depicted in the following figure, the leader SSM would be the one placed in Chassis 1.

A multi-chassis PXIe system may be configured to use many different measurement timebase reference options. For a list of those options and descriptions of how to configure them, see the section *Reference Clock Configuration* in this document. For one of those timebase reference options, one SSM is chosen as a leader and uses its internal Oven Controlled Crystal (Xtal) 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 2021, the SSM assumes the function of the **reference clock signal generator** , by sharing the a 100 MHz 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 REF Input port, to sync it together with other clusters.

The following diagram shows an example of a 6 chassis system connected with SSMs:

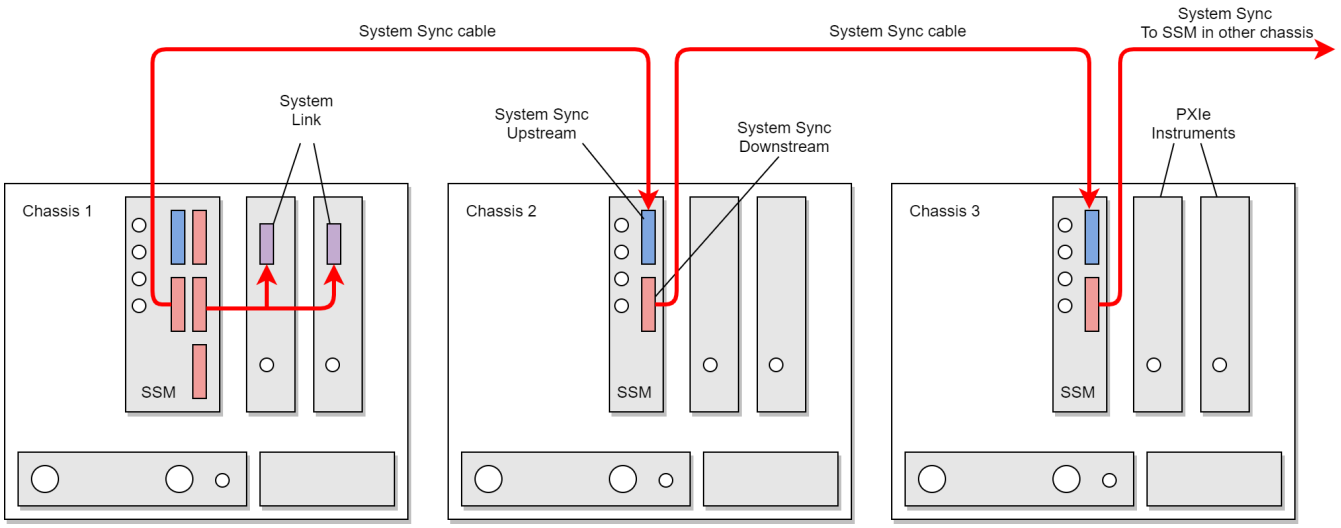


Multi-Chassis Configuration using the HVI Python API

This section shows how to use the HVI Python API to define and use SSMs in a multi-chassis system. The code snippets refer to a two-chassis example topology, but they can be leveraged to implement a multi-chassis system containing an arbitrary number of chassis. Each System Sync Downstream port connects to the System Sync Upstream port of another System Sync Module (placed in a different chassis).

The following block diagram shows a system with 3 chassis. The code examples below show how to implement this topology:

3 Chassis system with System Sync and System Link Connections



The first step is to define the SSMs placed in each of the two chassis during the system definition phase. In the following code in the options, the SSMs are simulated with `Simulate=true` and there are a number of parameters. For the hardware SSM instruments, set options to an empty string.

```
# 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'
#
options1 = "Simulate=true,DriverSetup=Model=M9033A, HviEngineIpVersion=1.1.0,
HviGatewareFeatureVersion=2,model=M9033"options2 =
"Simulate=true,DriverSetup=Model=M9032A, HviEngineIpVersion=1.1.0,
HviGatewareFeatureVersion=2,model=M9032"#
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, options2)
```

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 be connected only Downstream to Upstream.

```
# Define System Sync connectors
ssm1_downstream_sync = sync_module_1.connectivity.system_sync_down[0]
ssm2_upstream_sync   = sync_module_2.connectivity.system_sync_up[0]
ssm2_downstream_sync = sync_module_2.connectivity.system_sync_down[0]
ssm3_upstream_sync   = sync_module_3.connectivity.system_sync_up[0]
#
# Define connections among System Sync connectors of the SSMs
ssm1_downstream_sync.setConnection(ssm2_upstream_sync)
ssm2_downstream_sync.setConnection(ssm3_upstream_sync)
```

The SSM can also be used to manage data transfer across different PXI instruments. To do this, the System Link port of the instrument that is to transmit or receive data, must be connected to one of the System Sync ports of the SSM. Each System Sync port can be connected to two System Link ports using a dedicated cable that separates the top and bottom of the System Sync port connection.

```
# Define instrument engines
inst_engine_1 = my_system.engines.add(inst_1_engine_id, "HviEngine1")
inst_engine_2 = my_system.engines.add(inst_2_engine_id, "HviEngine2")
#
# Define System Link and System Sync ports to be connected
instrument_1_link = inst_engine_1.connectivity.system_link[0]
instrument_2_link = inst_engine_2.connectivity.system_link[0]
ssm1_downstream_sync1 = sync_module_1.connectivity.system_sync_down[1]
#
ssm1_downstream_sync1_top = ssm1_downstream1_sync.top
ssm1_downstream_sync1_bottom = ssm1_downstream1_sync.bottom
#
# SystemSync - SystemLink connection
ssm1_downstream_sync1_top.setConnection(instrument_1_link)
ssm1_downstream_sync1_bottom.setConnection(instrument_2_link)
```

Setting up System Synchronization Modules and Links

For instructions on how to set up your physical chassis and modules, see the *Keysight PXIe Chassis Family Startup Guide.* When you set up your system ensure you keep a record of which chassis you have, and which chassis contains which modules and in what slots they are. If you do not have this information, you can use the chassis and instrument Soft Front Panels (SFPs) to find it out.

System Synchronization Modules, have SystemSync connectors of two types: downstream and upstream. Each SSM can share signals from its SystemSync downstream connector to the SystemSync upstream connector of the next SSM. Connections upstream-to-upstream and downstream-to-downstream are not permitted. The SystemSync connectors (upstream or downstream) can also be connected to one or more instrument SystemLink connectors for data-sharing.

There are two types of SystemSync connectors: 4x and 8x.

An 8x SystemSync connector can connect to two 4x connectors with a special cable. The 4x connections can be SystemSync and a SystemLink, or a pair of SystemLink connections. On the 8x connector these are labelled top and bottom.

To set up connections in the HVI API do the following:

- Define the SystemSync modules.
- Add the SSMs to the interconnects collection.
- Specify the connectors and their type.
- Set the connection.

Define the SystemSync modules

You must first define the SystemSync modules. The options specify a number of parameters about each module.

```
# 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"
```

Add the SSMs to the interconnects collection

You must add the modules to the interconnects collection within the system definition. The options specify a number of parameters about each module.

```
# 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)
```

Specify the connections and their type

NOTE The items in the collection `systemsync_downstream` are indexed from 0.

```
# Get the 8x SystemSync downstream connector on first SSM
ssm_m9032_down = ssm_m9032.connectivity.systemsync_downstream[0]
# Optional: check Name and type
assert ssm_m9032_down.Name == "System Sync Down"assert ssm_m9032_down.type == keysight_
hvi.oculink_type.SYSTEM_SYNC_DOWNSTREAM_X8
# Get the 8x SystemSync upstream connector on second SSM
ssm_m9033_up = ssm_m9033.connectivity.systemsync_upstream[0]
# Optional: check Name and type
assert ssm_m9033_up.Name == "System Sync Up"assert ssm_m9033_up.type == keysight_
hvi.oculink_type.SYSTEM_SYNC_UPSTREAM_X8
```

Set the connection

You must set the connection between the connectors. This tells the HVI that these connections are connected together.

```
# Set the connection
ssm_m9032_down.set_connection(ssm_m9033_up)
```

Connecting 8x SystemSync to 4x SystemLink connectors

You cannot directly connect a 8x connection to a 4x connection. You must split the 8x into top and bottom 4x connections and connect one of these to the 4x connection.

An 8x connector can connect to a pair of two 4x (SystemSync or SystemLink) connectors with a special cable. In the API, the 8x connector is divided into a pair of 4x connections labelled top and bottom.

The upstream must still connect to downstream.

In the following code snippets the top 4x connection from an 8x connector is connected to a 4x SystemLink connector.

```
# Add SystemSync Modules to chassis
sync_module_1 = my_system.interconnects.add_sync_module(resource_id, options1) # ssm
module with SystemSync connectors of 8x links
sync_module_2 = my_system.interconnects.add_sync_module(resource_id2, options2) # ssm
module with SystemSync and SystemLink connectors of 4x links each
```

Get the 8x downstream SystemSync connection on module 1 and get the top 4x connection from it.

```
# Get a 8x SystemSync type connector on sync_module_1
system_sync_downstream1 = sync_module_1.connectivity.systemsync_downstream[0] # get a
SystemSyncDownstream connector of 8x links
# Get the top 4x connection
system_sync_1_top = system_sync_downstream1.top
# Check the type of connection
assert system_sync_1_top.type == keysight_hvi.oculink_type.SYSTEM_SYNC_DOWNSTREAM_X4
```

Get the SystemLink connector.

```
# Get a 4x SystemLink type connector on sync_module_2
system_link_2 = sync_module_2.connectivity.system_link[1]; # get a SystemLink connector
of 4x links
# Check the type of connection
assert system_link_2.type == keysight_hvi.oculink_type.SYSTEM_LINK_X4
```

Set the connection.

```
# set connection between module1 and module2
system_sync_1_top.setConnection(system_link_2)
```

Chassis Supported for Multi-Chassis Systems

The following Keysight chassis models are supported:

- M9010A
- M9018B
- M9019A

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.

Troubleshooting Multi-Chassis Systems

When you are using more than 1 chassis, you must:

- Use the latest chassis driver and firmware.
- Specify the connections between the chassis in the HVI API.

Chassis numbering

- Ensure your chassis are numbered from 1 upwards.

  The PXI standard does not permit chassis to be numbered as 0. If this happens, it indicates there has been an incorrect installation of the firmware, PXI chassis driver, software, or PXI resource manager.

Ensure you are using correct firmware and software components

- For PathWave Test Sync Executive to work correctly, the PXI chassis, firmware, driver, software, and PXI Resource Manager must be all be installed correctly, regardless of the chassis vendor.

  Compatibility requirements for PathWave test Sync Executive are listed at Instrument Software and Firmware Requirements for KS2201A.

Using non-Keysight chassis with PathWave Test Sync Executive

- Keysight recommends you use PathWave Test Sync Executive with Keysight chassis. It is It is possible to use non-Keysight chassis with the following limitations:

  Only a single PXIe chassis is supported if you are using a non-Keysight chassis. Multi-chassis operation requires the recommended Keysight PXIe Chassis.

  The proper PXIe resource manager and chassis VISA driver installation is required.

  PathWave Test Sync Executive has not been validated with non-Keysight PXIe chassis, if you encounter any issues, contact support.

Using non-Keysight chassis with Keysight Instruments and PathWave Test Sync Executive

- Check the documentation of each PXI instrument that you are using with PathWave Test Sync Executive, to ensure they comply with the instrument limitations on compatibility with non-Keysight chassis or controllers.

Reference Clock Configuration

This section describes how to configure the Reference Clock:

The Reference Clock is here intended as the main system clock that all the instruments in the multi-chassis system lock to. In a single or multi-chassis system, you require a source for a reference clock and a means to distribute it. You can set the source can to internal mode (the default) or external mode. In external mode, an external clock source drives the Keysight instrument chosen as a reference clock source.

Reference Clock Distribution using System Sync Modules

In a multi-chassis system based on the Keysight PXIe SSM and Keysight PXIe chassis, the SSM is always in charge of the reference clock distribution. There are different options for the source driving the SSM clocking, you can select this by configuring one of the options outlined later in this section.

To distribute a reference clock, you must choose one SSM to act as the leader. The leader SSM outputs the reference signal and shares it with the other SSMs using System Sync cables. Each SSM shares the reference clock across all the instruments in its chassis using the DSTARA signal path located in the backplane of each Keysight PXIe chassis.

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.

The SSM also has a Ref Out port. This enables the SSM to provide its own reference clock to the Device Under Test (DUT). If you want to use the SSM clock output from the Ref Out port, use your DUT or external system API to set the SSM Ref Out as the clock source for the external system.

Reference Clock Source and Modes

If you are using a multi-chassis with System Synchronization Modules to synchronize the system, you have multiple system components that can be selected to act as a primary source for the system reference clock. Once the primary source is defined, all the other instruments in the system are secondary and these are synchronized to the primary.

You must set the reference clock source you want to use and program it into the HVI API to reflect the HW connections that you made.

In PathWave Test Sync Executive release 2021, the source of the reference clock signal can be the internal OCXO clock of the SSM chosen as leader.

In addition to the reference clock source, you must define the reference clock mode of operation. The possible modes are:

- Internal: this is the default mode of operation. It uses the defined reference clock source and it propagates across the system.
- External: in this mode the defined reference clock source is synchronized to an external reference clock that must be connected to the Ref In port of the clock source. Later sections provide more details on the hardware connections for each case.

The possible sources and modes for the system reference clock in a multi-chassis system are summarized in the following table:

Clock Source	Signal Path	Description	Clock Reference Mode
SSM	OCXO	The OCXO generates a 10 MHz sine wave that gets transformed into a 100 MHz reference clock by the clocking circuitry inside the SSM	Internal
SSM	SSM Reference input	The external reference clock can have a 10 or 100 MHz frequency. As an example, it can come from a DUT, an atomic clock reference, or another instrument that is part of the setup, etc.	External

According to the configuration chosen for the reference clock source and mode, additional connections to the SSM may be required:

- If you want to use the SSM internal OXCO clock you are not required to attach any other clock sources.
- If you want to use the internal OXCO clock synced to an external source, you must attach the external clock source to the SSM External Ref In.

Setting the Reference Clock for a Multi-Chassis System

All of the options for the clock primary source also have the ability to synchronize the primary to an external reference clock. This is useful if you want to for example, synchronize with a clock coming from a DUT (Device Under Test) that you are measuring.

To use an external clock, you must connect it to the relevant **External Reference In** and add the relevant definition in the API.

NOTE Some instruments might have additional clocking requirements. For more information, see your instrument manuals.

Configuring clocks with the HVI API

You define the system wide clocking with the `clocking` interface in the `SystemDefinition` class.
See Clocking API.

Reference Clock Configurations for Multi-Chassis Systems

The following section describes the configurations available for multi-chassis systems:

Multi-Chassis SSM System with SSM OCXO Clock as Reference Clock

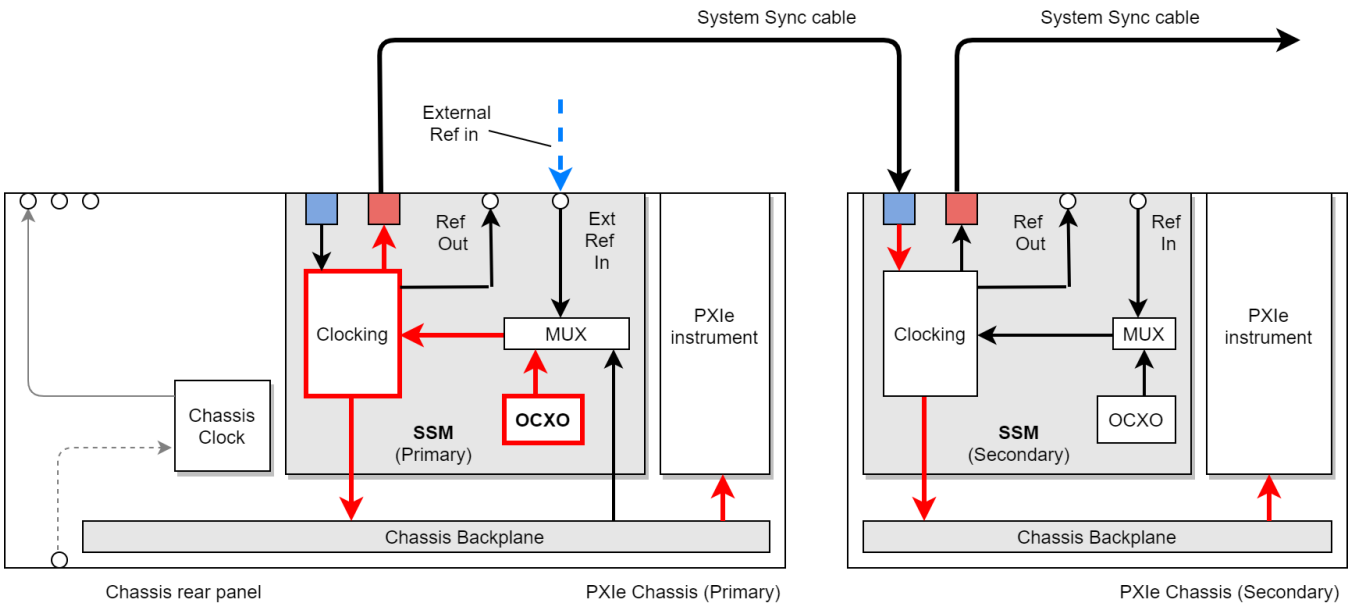
This configuration is for single or multiple chassis with SSMs. This configuration is compatible with Keysight PXIe chassis models M9010A and M9019A. The SSM internal OCXO is used as the reference clock source. In the HVI API you must configure the corresponding clock source and mode options.

You can use the SSM as a clock source in the following ways:

- Use the internal OCXO clock as the reference clock source.
- Use the internal OCXO clock as the reference clock source synchronized to an external reference clock.

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

Multiple chassis with SSMs, OCXO as clock source



Using the internal OCXO clock

By default, 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. Make sure the connections defined in SW match the physical hardware connections between SSMs. In the HVI API, no additional definition other than the connections between SSM is required to identify the leader SSM.

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 secondary SSMs depending the number of chassis
leader_ssm = my_system.interconnects.add_sync_module(resource_id_1, options)
my_system.interconnects.add_sync_module(resource_id_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 clock mode to INTERNAL
clockSource.set_mode(keysight_hvi.ClockingReferenceMode.INTERNAL)
#
# Set the SSM clock source
systemDefinition.Clocking.reference_source = clockSource
```

Using the internal OCXO clock with an external reference clock

The SSM leading the synchronization with its reference clock can optionally be connected to an external reference. the external reference can come from, for example a DUT or another source such as an atomic clock. This way, the time bases of the measurement instruments and the DUT are in sync. The external reference can be connected to the SSM using the Ref In port.

To use an external reference clock, you must:

- Connect the external reference source to the SSM **External Reference In** .
- In the HVI API you must set the SSM to synchronize to an external reference clock. To do this, set the mode to `External` .

The HVI API delivered by PathWave Test Sync Executive enables you to set the system reference clock to be in External mode and sync with an upcoming clock output from a DUT or another system. To use the external reference, modify the previous code snippet to use the external mode:

```
# Set clock mode to EXTERNAL and set frequency to 10MHz
clockSource.set_mode(keysight_hvi.ClockingReferenceMode.EXTERNAL, 10.0)
```

Single chassis as a reference clock with no SSM

This is the default configuration for a single chassis with no SSM. You are not required to configure this.

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.

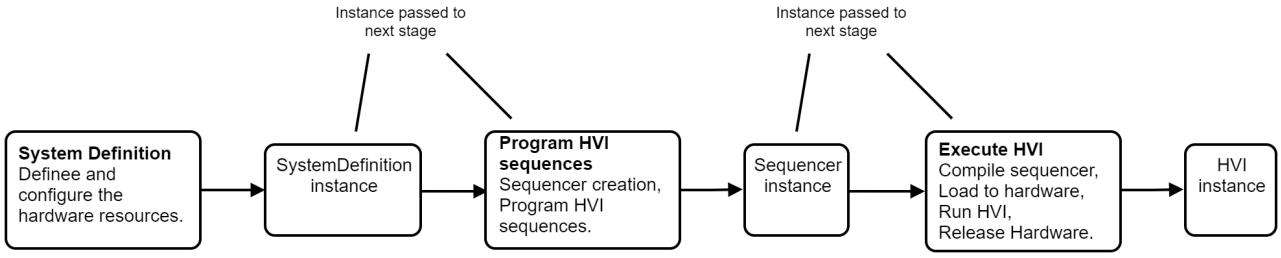
This chapter contains the following sections:

HVI API Main Classes and Use Model

PathWave Test Sync Executive has three primary 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 2021\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.

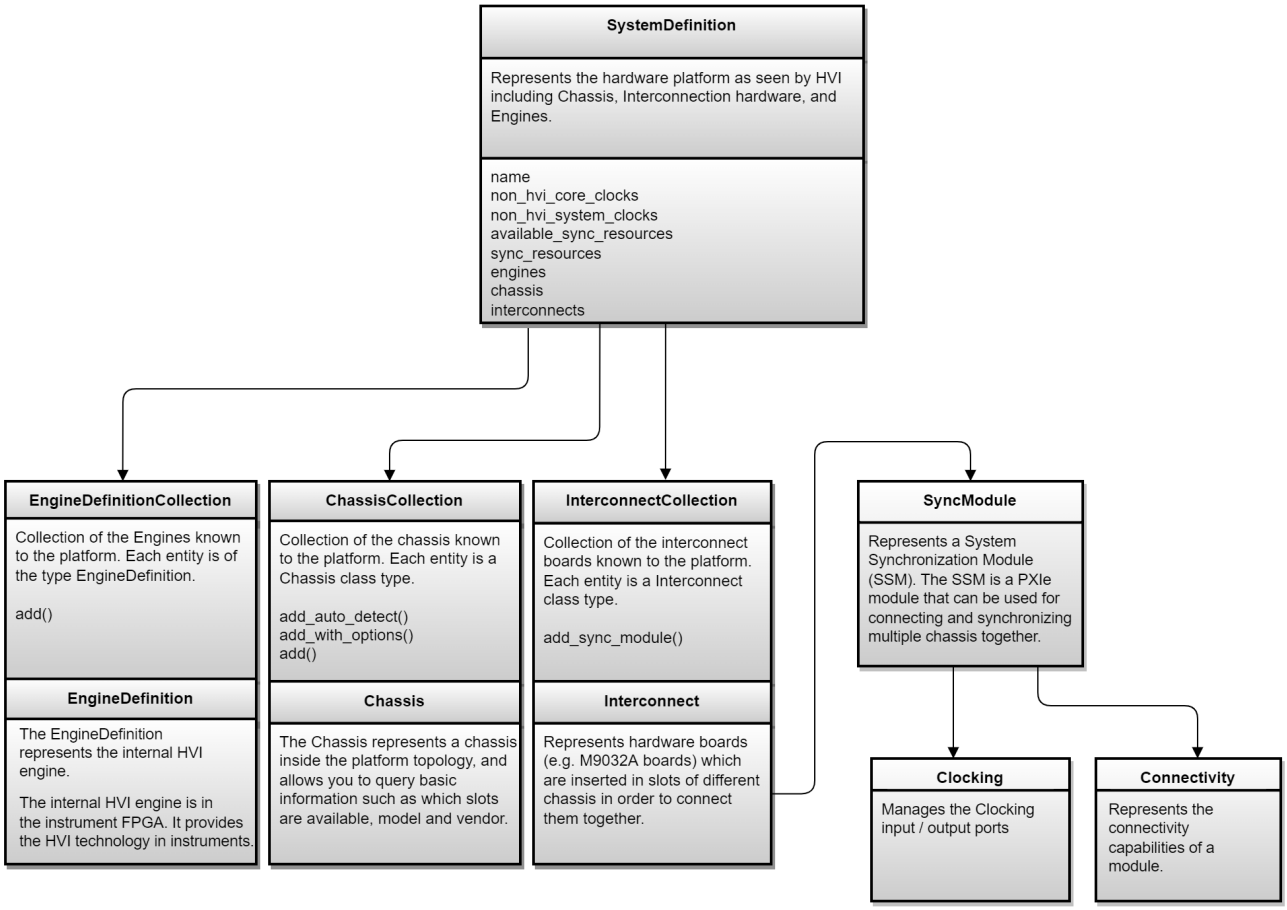
SystemDefinition

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

- Engines
- Chassis, Interconnects and SyncModules Classes
- Synchronization Resources and Clocks
- System Initialization
- Clocking API

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:



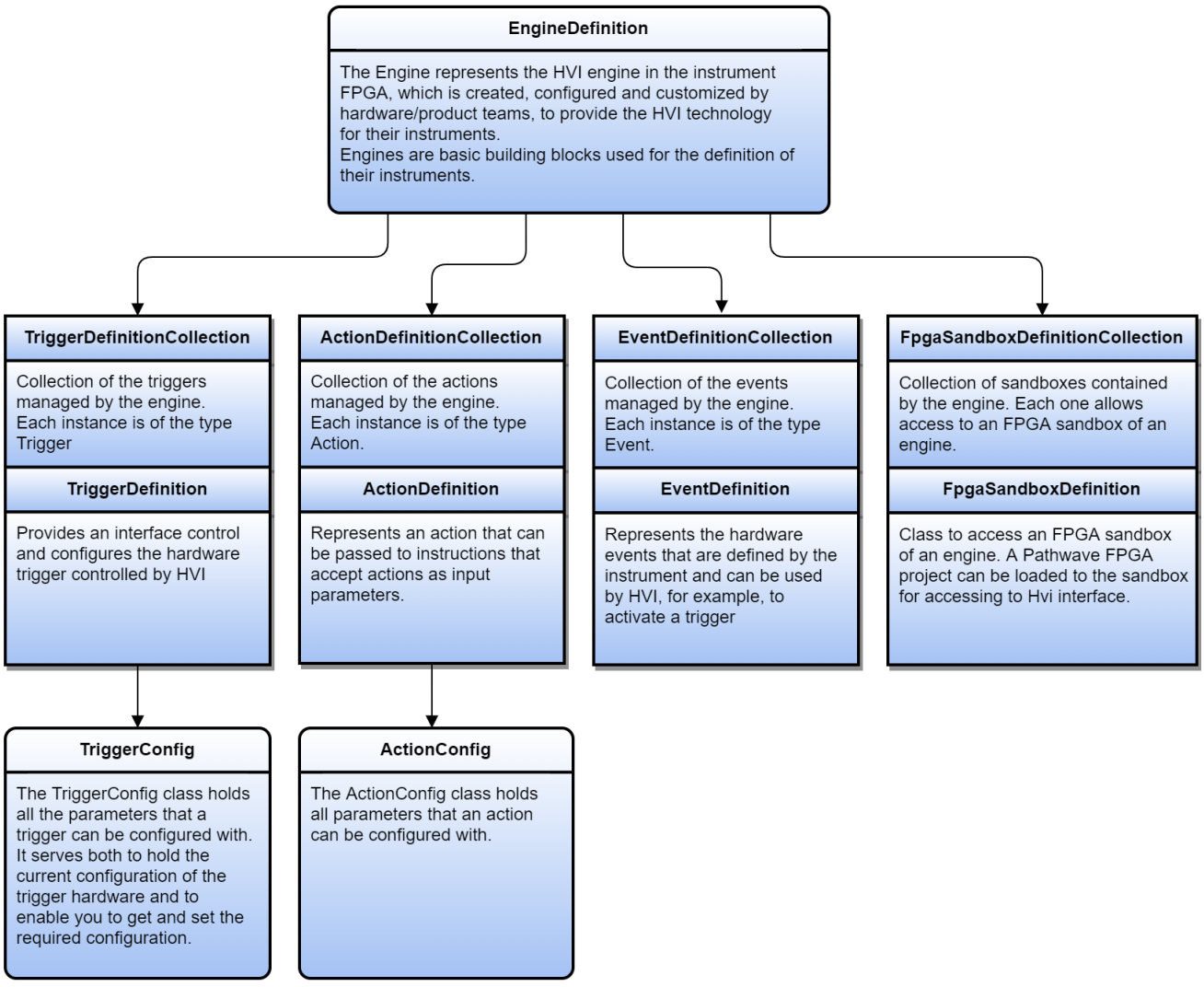
Engines

The Engine class provides access to the HVI Engine in the instrument.

You create instrument objects, where each object represents a physical PXI 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 is a class used to list and manage all the trigger signal lines to be used by each HVI engine for triggering purposes.

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.

`TriggerDefinition` provides an interface to control and configure the hardware trigger controlled by HVI. The `TriggerConfig` class holds all the parameters that a trigger can be configured with. It serves both to hold the current configuration of the trigger hardware and for you to get and set the desired configuration.

You use the `TriggerConfig` class to configure the trigger to:

- Turn a trigger ON or OFF.
- Write to a trigger line.
- Get the hardware Name or ID of a trigger resource.
- Configure settings for a given trigger.

To configure the trigger settings, you must set up the following parameters:

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

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 initiated 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]
```

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

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

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 precision clock reference across the multi-chassis system.

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')
```

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 resourcesyou 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.

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 Setting up a Multi-Chassis System with System Synchronization Modules and PXI Chassis.

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	
Only 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

Synchronizing HVI to non-HVI instruments

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 supports the definition of the following types of clocks:

Non-HVI system clocks

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 clocks that directly impact the operation of the specific feature that the HVI must trigger. Core clocks are used by the HVI to determine everything except for the Sync-Base period.

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 HVI Sync signals according to the following definition:

- `Sync_Base= functionOf(CLK100, CLK10, all core clocks, all system clocks )`

System Initialization

The SystemDefintion class includes an `initialize()` method that performs system initialization and clock alignment.

There are 3 cases where system initialization and clock alignment can occur:

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

The initialize() method

There are two ways of calling the `initialize()` method

With no parameters

```
sys_def.initialize()
```

This performs minimal initialization, which is the default initialization performed when no alignment mode is specified. The default initialization tries to minimize the necessary operations to obtain the fastest initialization / synchronization time.

With a parameter

```
sys_def.initialize(keysight_hvi.AlignmentModes.Full)
```

The parameter specifies an alignment mode or modes for initializing the system.

The parameter value is:

Mode	Description
Full	Force full system complete clock alignment and initialization.

Use cases for Full Mode

If you have a system that has been power-cycled and is therefore is not aligned. If you start a test and call `sys_def.initialize()` (no arguments). This initiates a full clock alignment since the system has not been aligned yet. The time the initialization procedure can take several minutes depending on on the size and structure of your system, including the number of chassis and instruments.

If you make a second call to `sys_def.initialize()` after the first call, it will be very fast. For instance, if you run another test with the very same setup it will take just around 100ms.

Forcing a full clock alignment

You can force a full clock alignment by calling:

```
sys_def.initialize(keysight_hvi.AlignmentModes.Full)
```

Calling Initialize() in Sequencer Creation

If you call `inialize()` in Sequencer Creation, `sys_def.initialize()` is executed without parameters, this only performs a minimal update to the initialization and clock alignment.

Calling Initialize() in Load To Hardware

If you call `inialize()` in Load to Hardware, `sys_def.initialize()` is executed without parameters, this only performs a minimal update to the initialization and clock alignment..

Clocking API

In a 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 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
```

Alternatively, 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
```

Set 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 of the external sources.

The following code show these options for a chassis:

```
clockSource = chassis.clock_source
#
# Set clock mode to INTERNAL
clockSource.set_mode(keysight_hvi.ClockingReferenceMode.INTERNAL)
#
# Set clock mode to EXTERNAL and set frequency to 10MHz
clockSource.set_mode(keysight_hvi.ClockingReferenceMode.EXTERNAL, 10.0)
#
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
#
mode = clockSource.mode
frequency = clockSource.frequency
```

In a 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.

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.

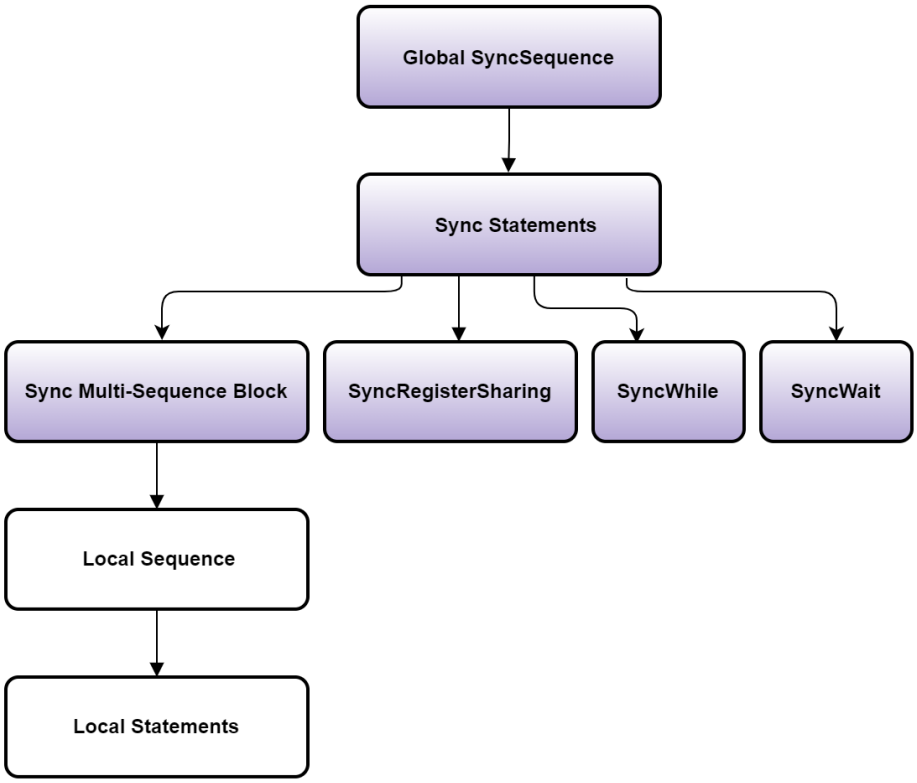
The Sequencer object defines a top level Global Sync Sequence. Within this:

- 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.

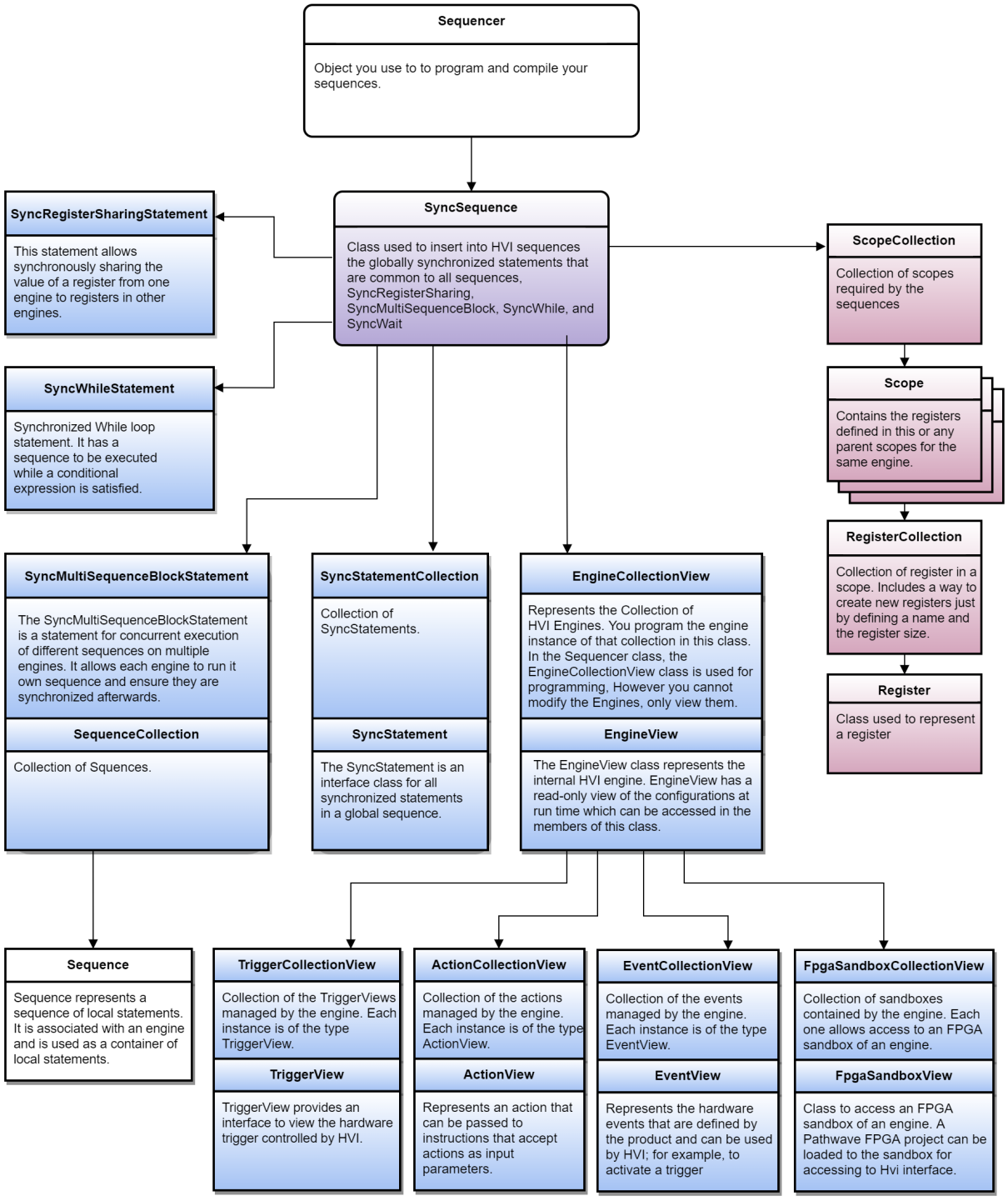
The sequences and statements you add can access the resources you previously added via the `EngineCollectionView` class. The view classes enable you to see the definitions you have set up, but you cannot modify them.

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.

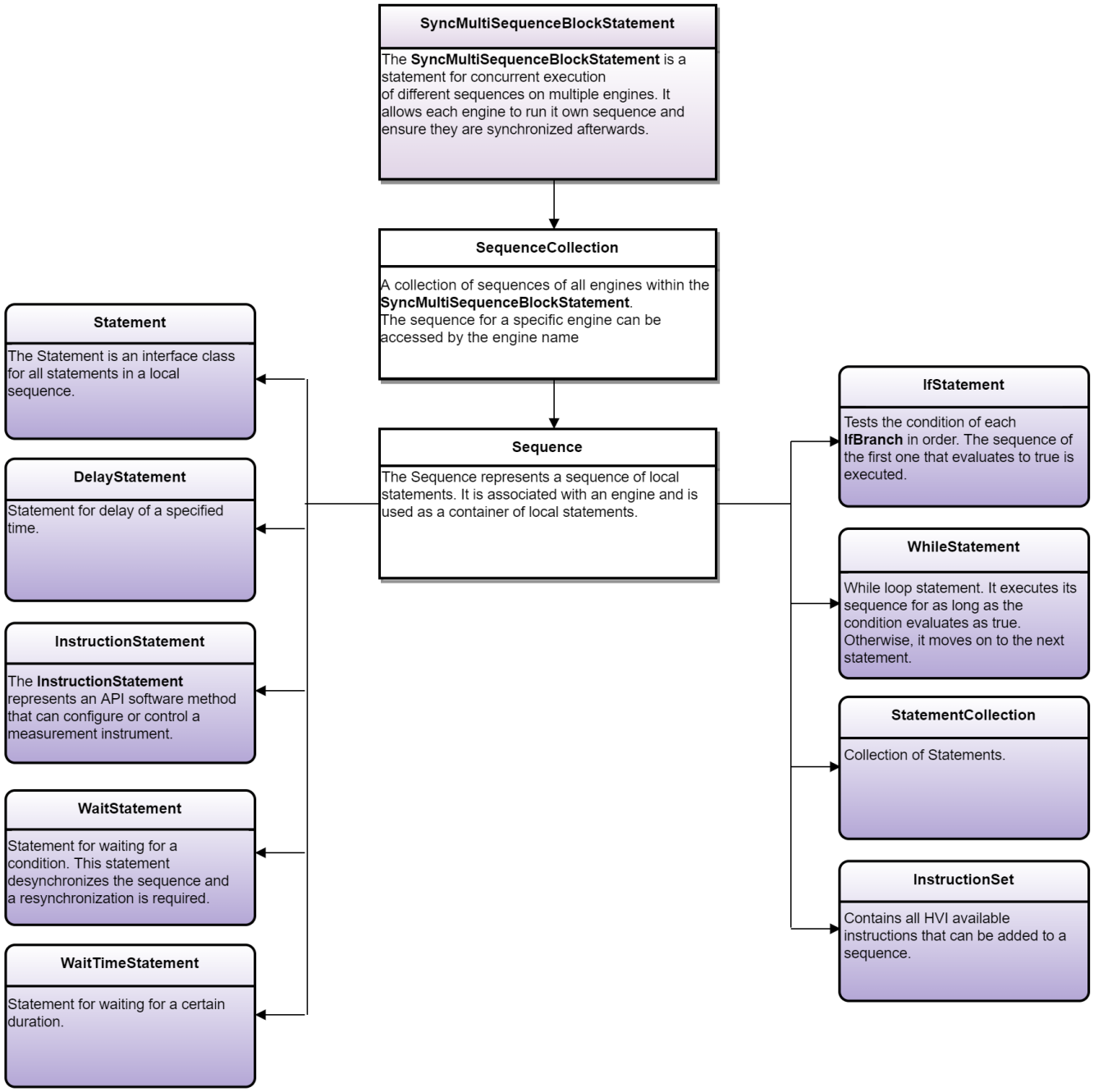
Synchronization and timing information are added within each Sync statement so that all sequences across the HVI are coordinated precisely. The `SyncMultiSequenceBlockStatement` exposes local flow control and instruction statements that are sent by the `Sequence` object. The other Sync statements are 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.

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:

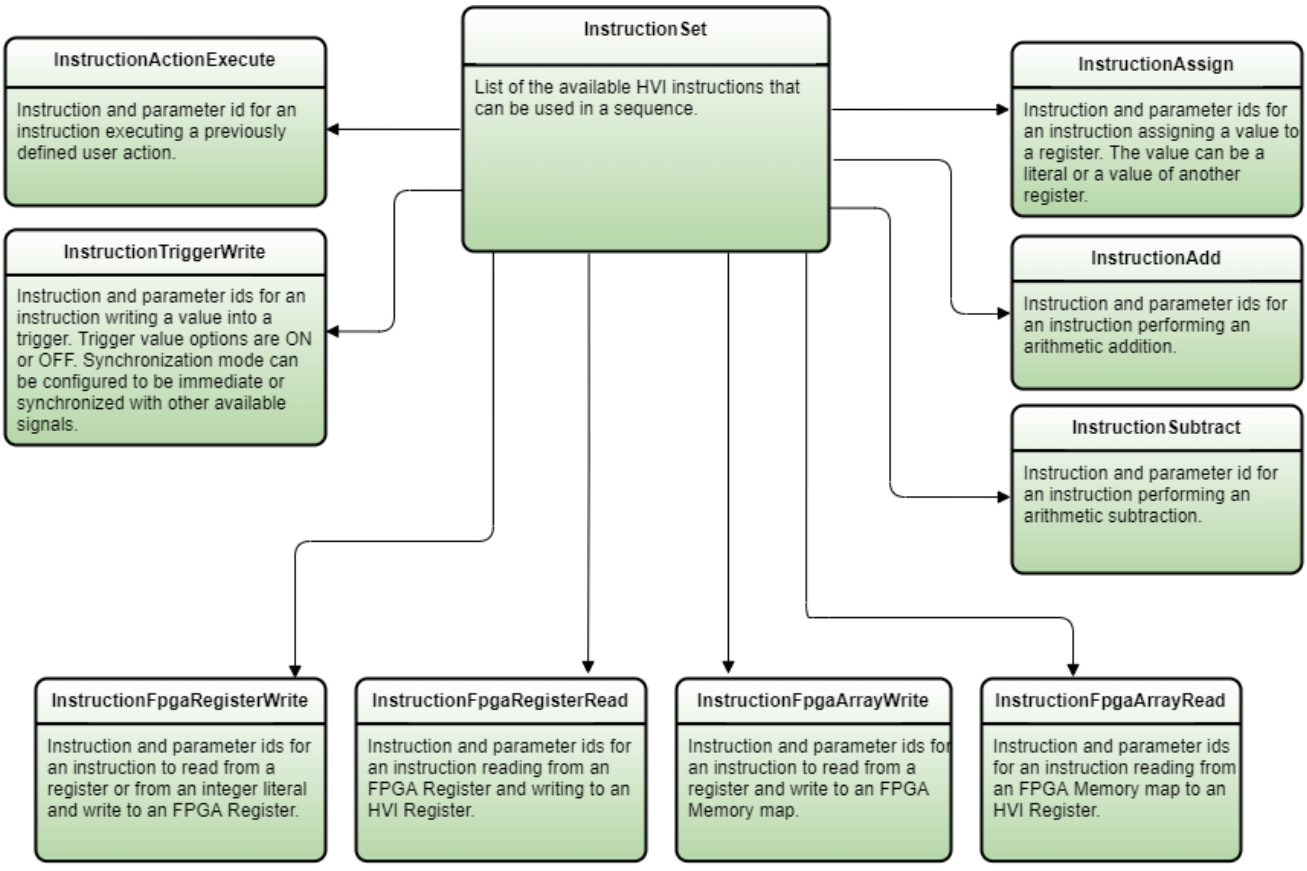
- HVI-native instructions are part of the `InstructionSet` class.
- Instrument-specific instructions are documented in instrument user guides.

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 2021\api\python\Help`

`C:\Program Files\Keysight\PathWave Test Sync Executive 2021\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
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 = 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.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)
```

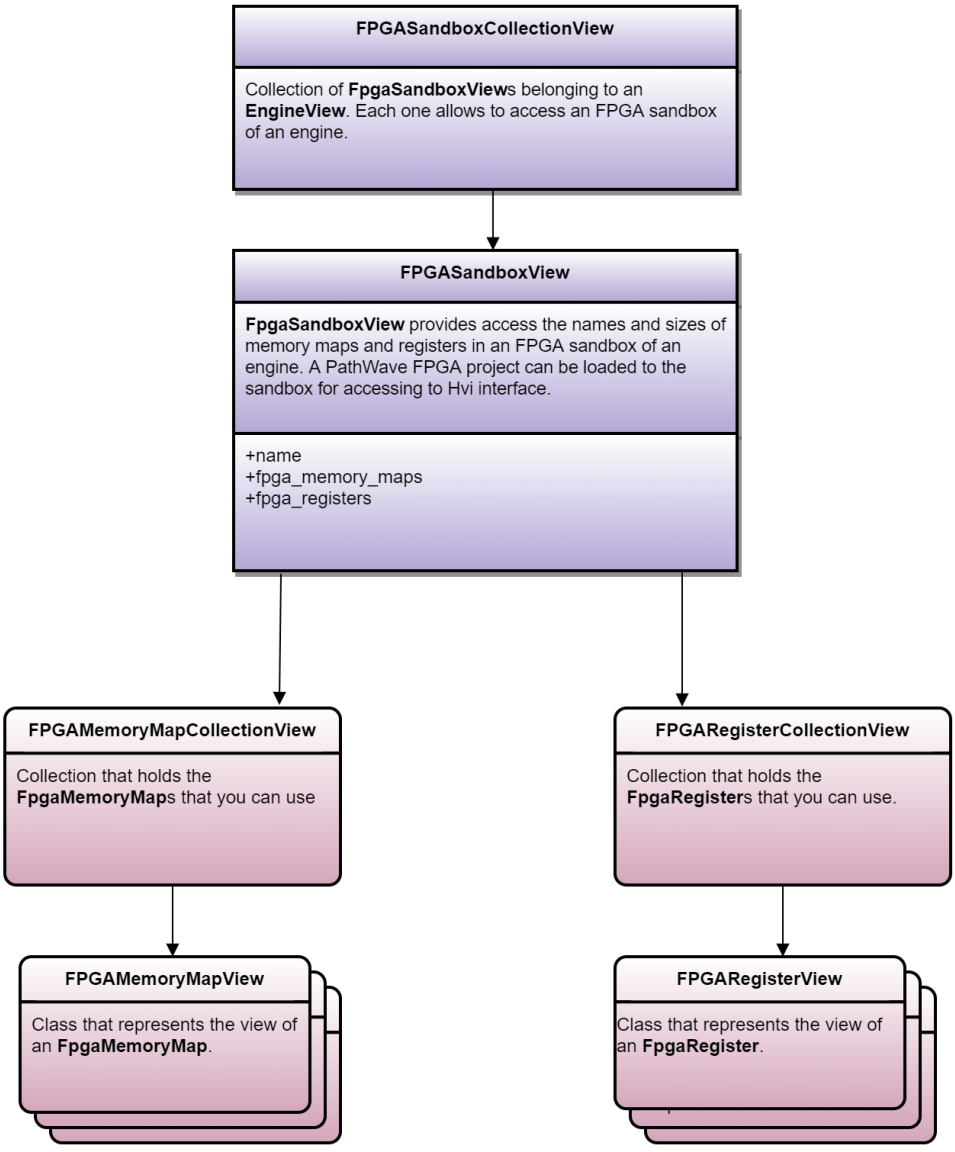
Instrument-specific HVI instructions

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))
```

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.

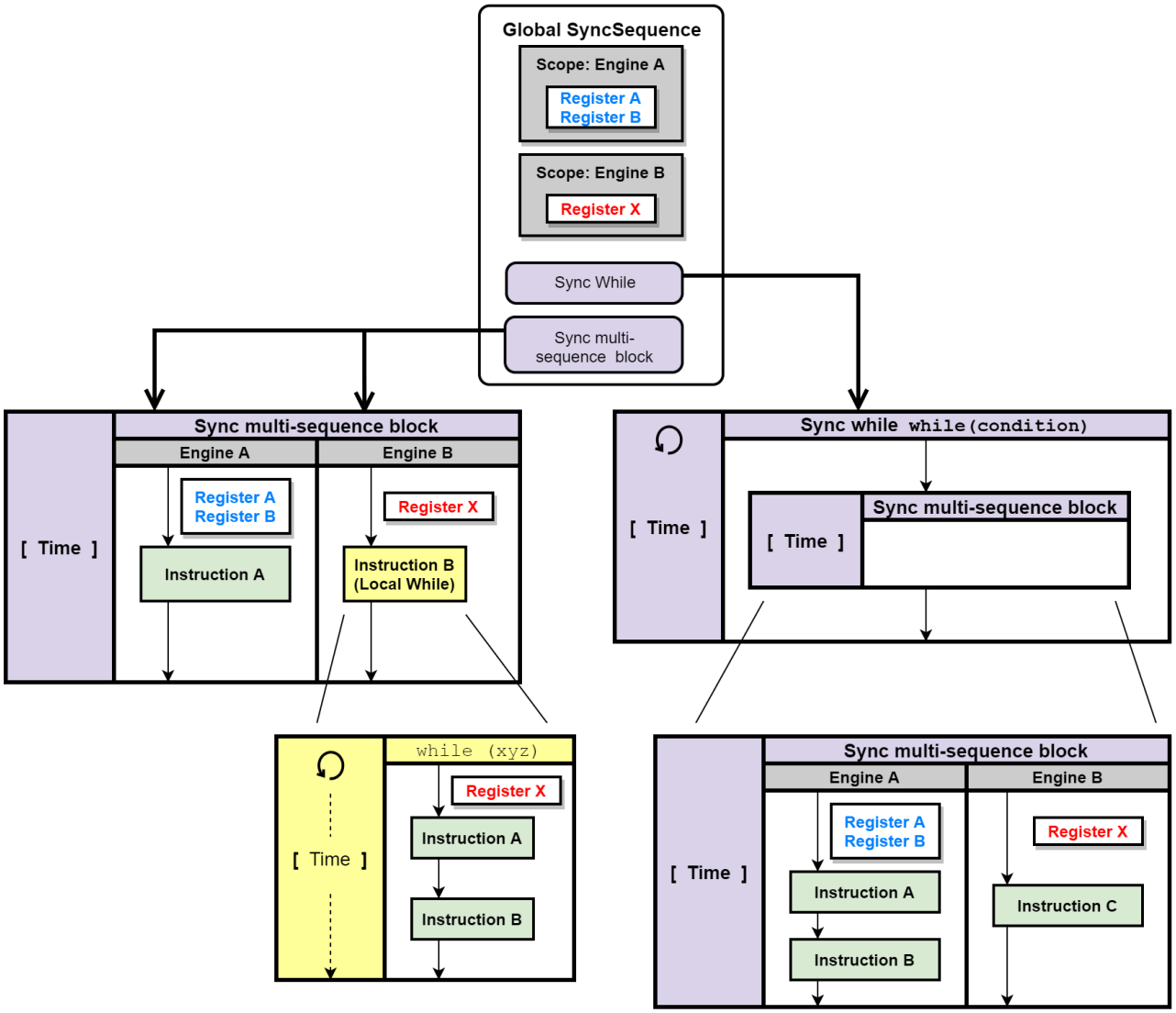
The following diagram shows the scope concepts. The registers available are shown in the sequences and child sequences.

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

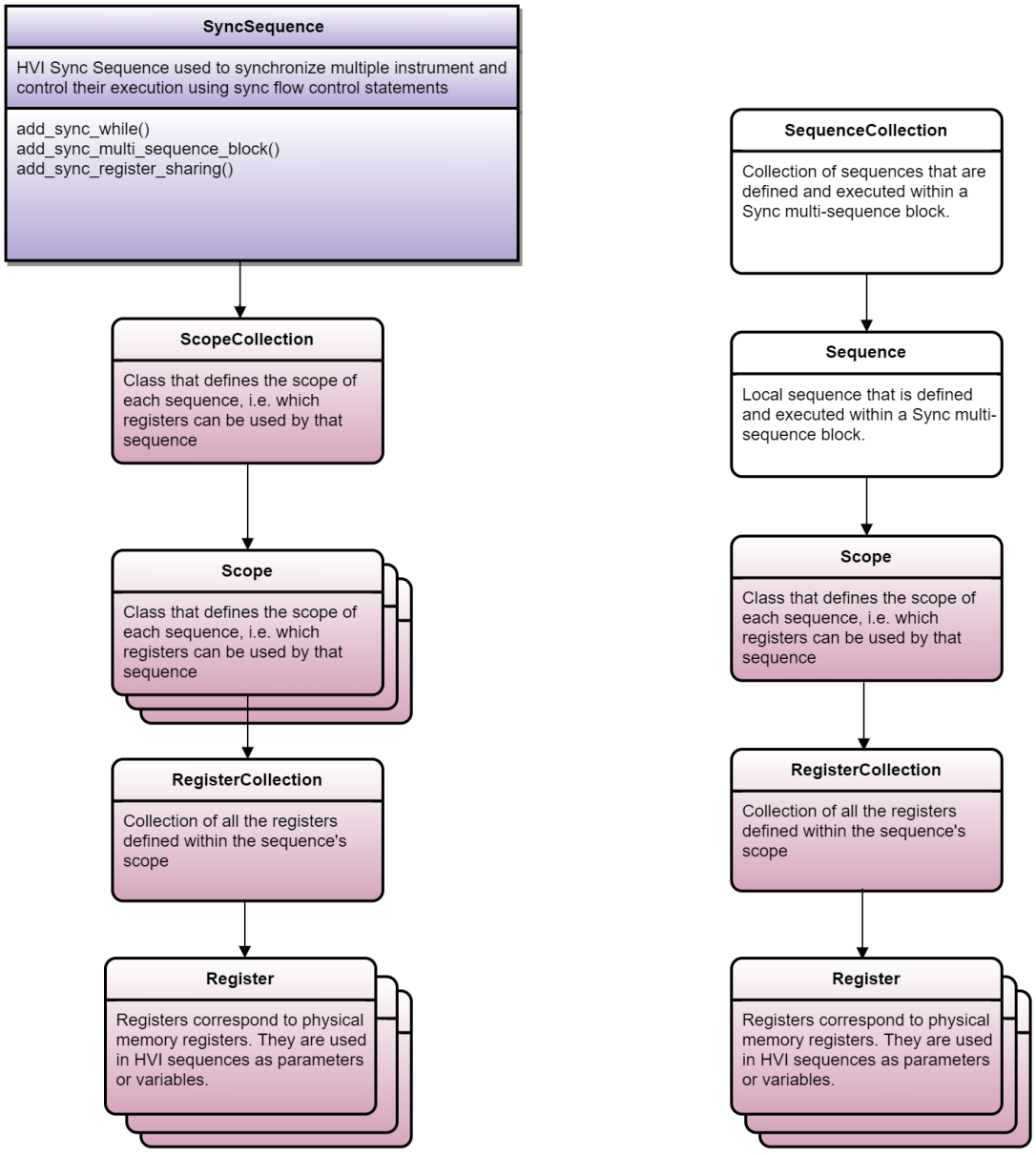
The scopes of HVI sequences are managed through the Scope class. Each Local sequence is an instance of the Sequence class, it is associated to a specific HVI engine and has its own Scope object. SyncSequences are associated to multiple HVI engines and consequently have an HVI Scope collection that contains 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:

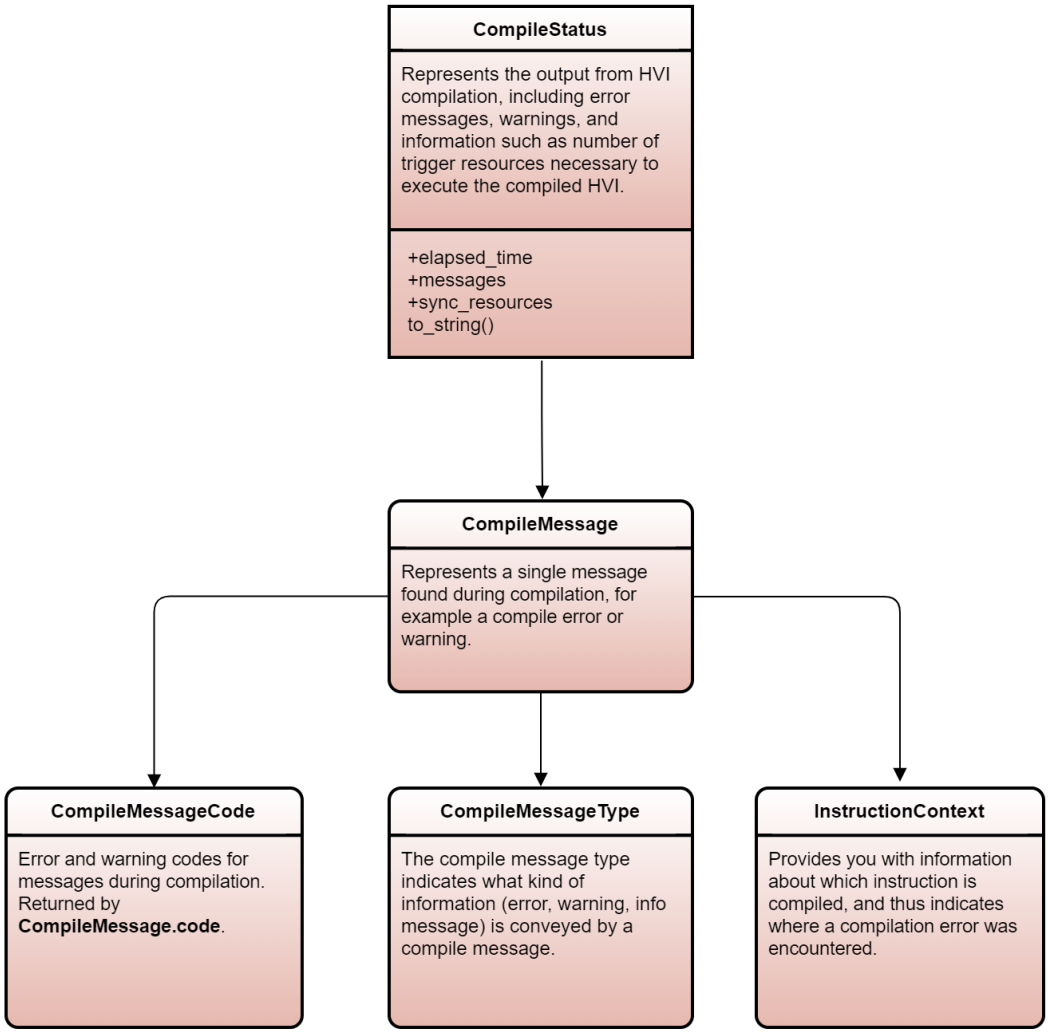
```
try:
   hvi = sequencer.compile()
   print('Compilation completed successfully!')
except kthvi.CompilationFailed as err:
   print('Compilation failed!')
   compile_status = err.compile_status
   print(compile_status.to_string()) # This line will print all the errors and warnings
collected during 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"PrimaryEngine.Loop Counter" =
reg"PrimaryEngine.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"PrimaryEngine.Loop Counter" =
reg"PrimaryEngine.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
`static 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

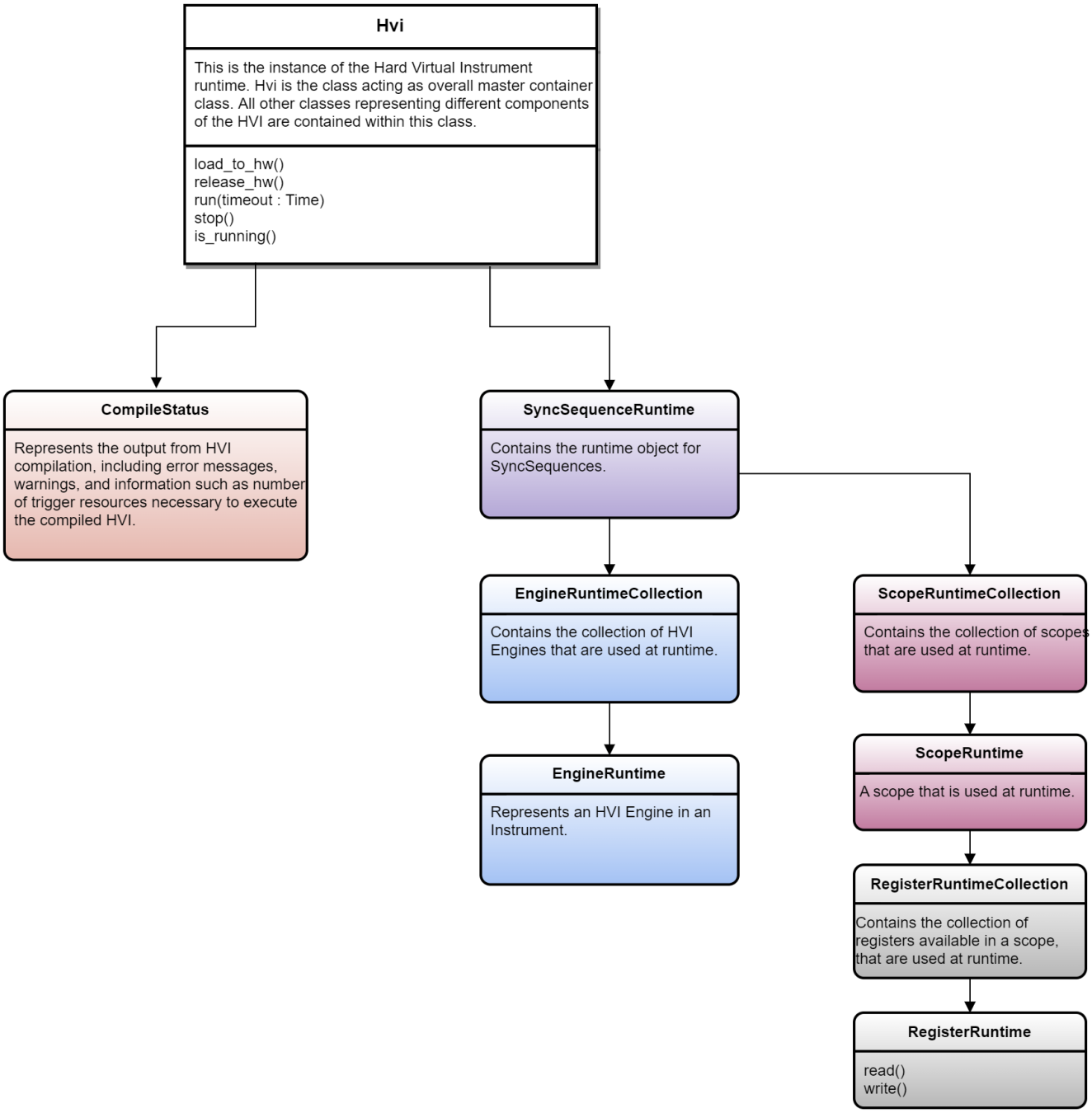
This section describes the Hvi object, it contains the following sections:

- EngineRuntime Components
- Load to Hardware and Run
- HVI Real-time Hardware Execution Error Handling

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.

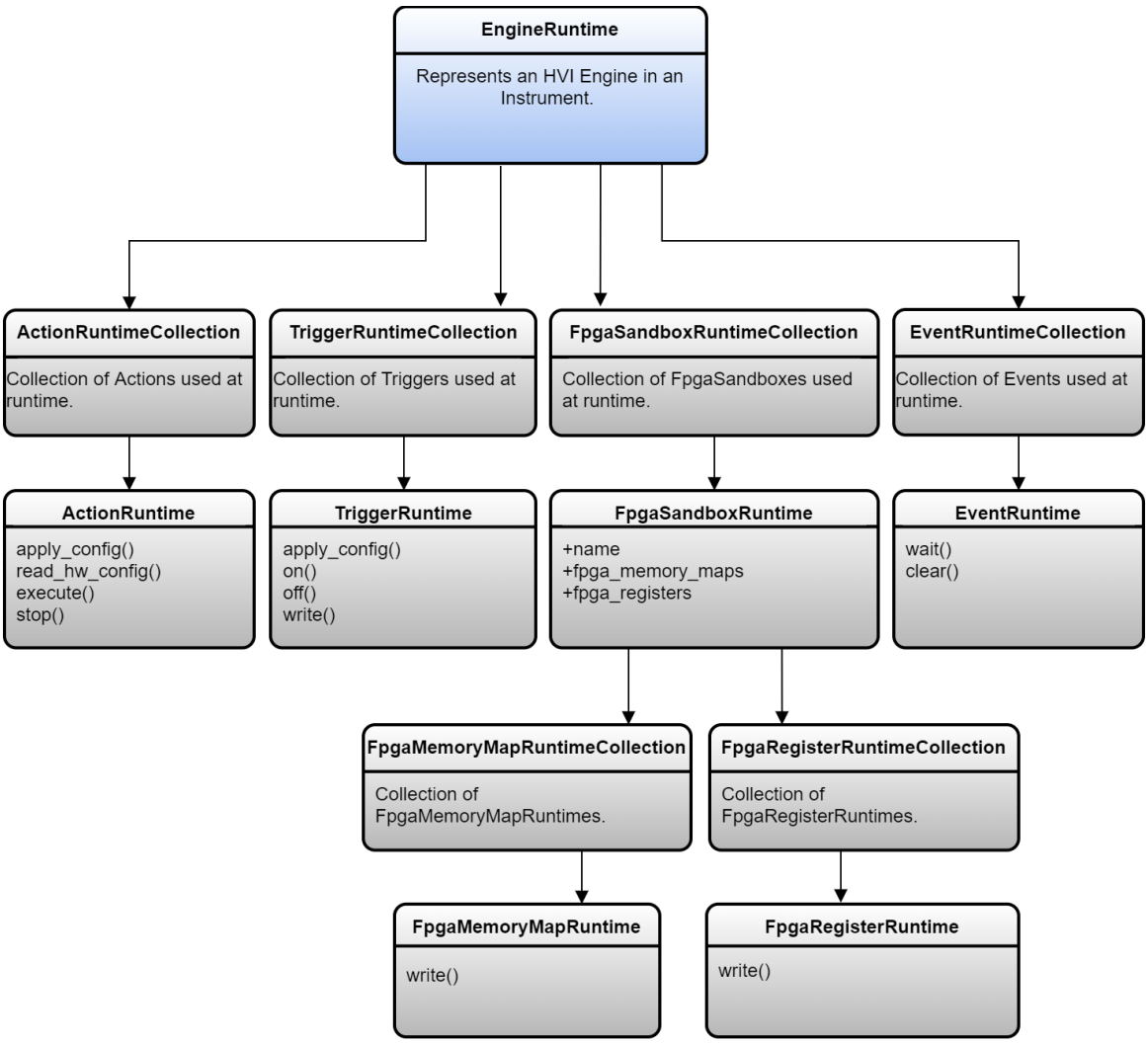
The following diagram shows the classes:



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` .

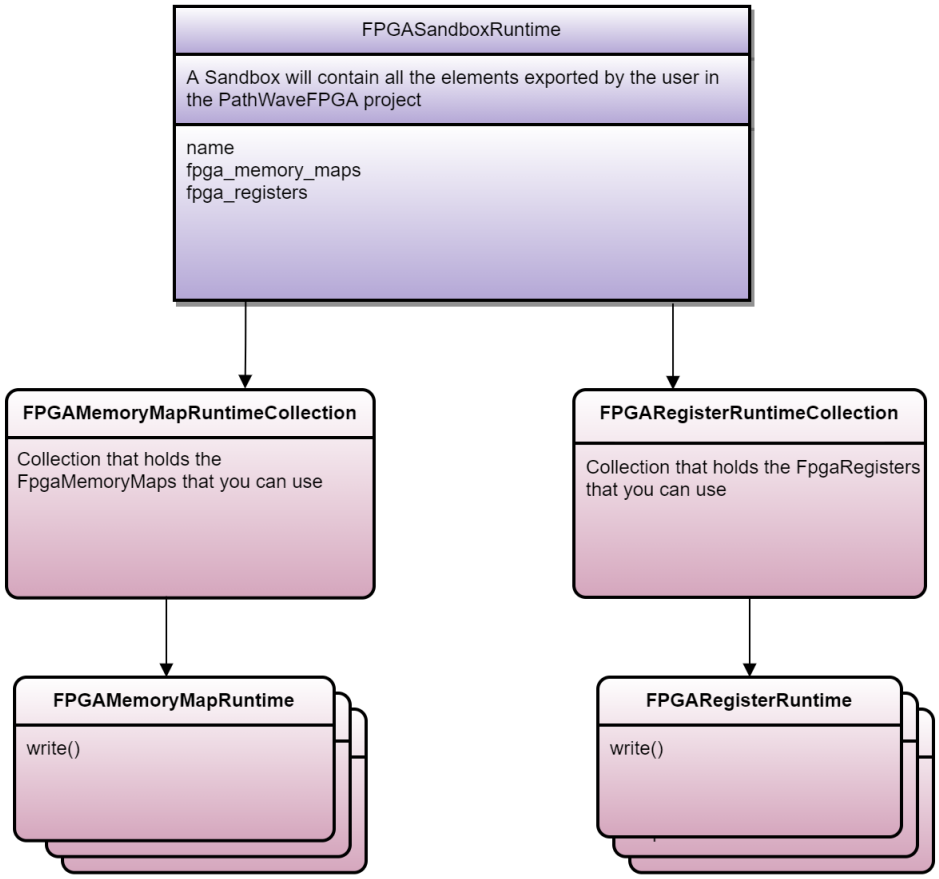
RegisterRuntime

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
```

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.

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.

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:

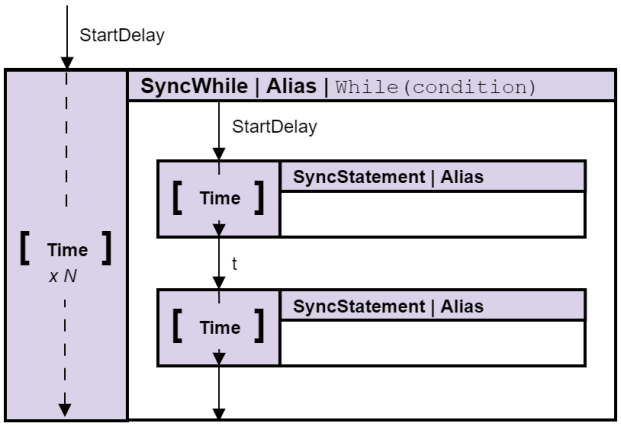
- Sync while.
- Sync multi-sequence block.
- Sync register-sharing.

Sync 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 *Primary* and the remaining engines have the role of *Secondary* :

Primary

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.

Secondary

A Secondary engine monitors the result of the condition and acts on it, following the Primary.

The condition expression assigned to the Sync while must use resources that belong to the same HVI engine. The Primary 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.

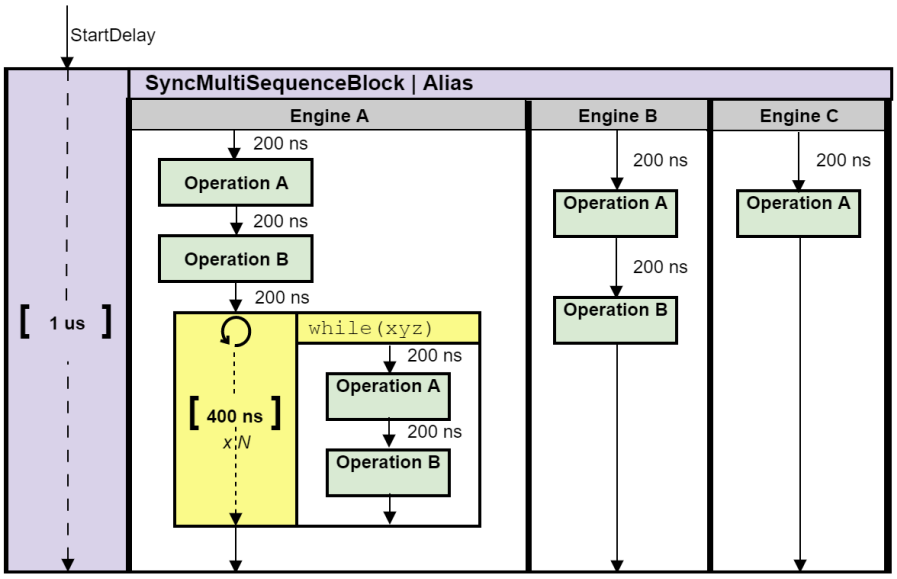
```
# Configure Sync While Condition
sync_while_condition = keysight_hvi.Condition.register_comparison(reg, keysight_
hvi.ComparisonOperator.GREATER_THAN, 10)
#
# Add Sync While to a sync-sequence
sync_while = my_sync_seq.add_sync_while("sync_while", 10, sync_while_condition)
#
# Access the sync sequence 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)
```

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 = module.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)
```

FPGA register read

The instruction `fpga_register_read` is an HVI-native instruction that enables you read from an HVI port 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 port register placed in an FPGA sandbox. The value to be written to the HVI port 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 port memory map that is located in an FPGA sandbox. The value to be written to the HVI port 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 port memory map. The value read from the HVI port 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.

For full details of the FPGA instructions see Chapter 5: HVI Integration with PathWave FPGA.

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.

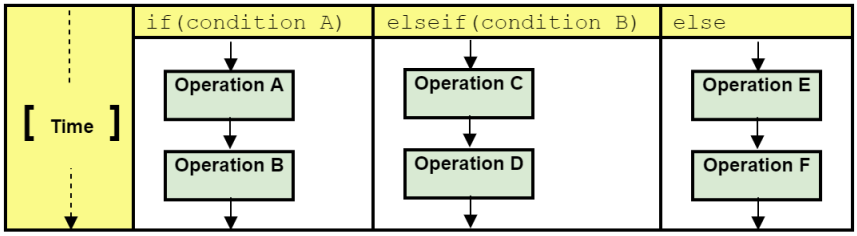
Local if statement

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.

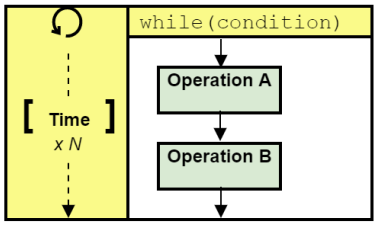
The following code is an example of a Local If statement:

```
# 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)
```

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

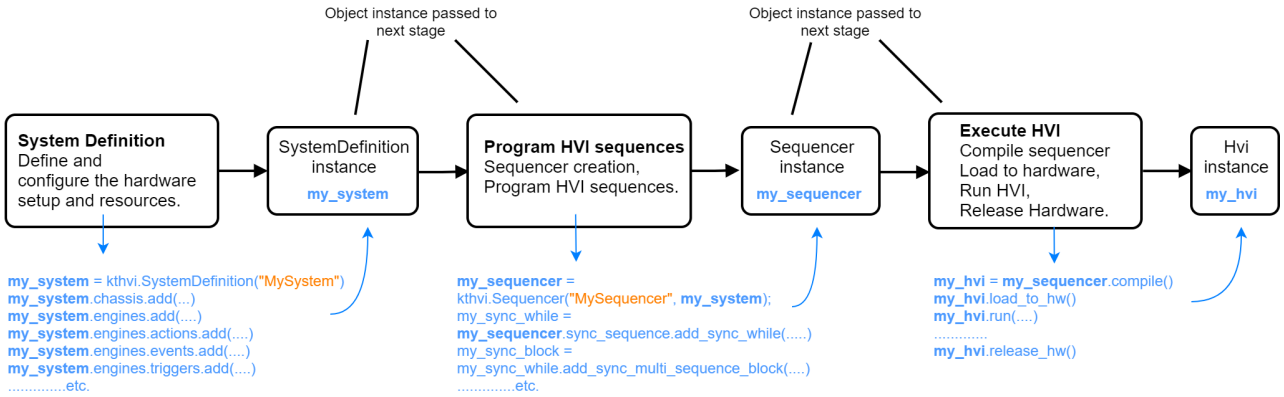
When you have programmed your sequences, you call the compile software method to create an instance of the `Hvi` class from the `Sequencer` object instance containing the information about the programmed sequences.

After a successful sequencer compilation, the 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
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
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=M9019A")
```

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]
# Specify a connection between a M9032 and a M9033 SSMs
ssm_m9033.connectivity.systemsync_downstream[0].set_connection(ssm_
m9032.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

This is only required when dealing with instruments that do not support HVI technology, or *Devices Under Test* that have specific clocking requirements. This is an advanced feature that most users do not require. If you think you require it, please contact your application or support engineers.

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**

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);
}
```

Adding 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 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 HVI Engine clock. For example, for an engine that runs at 100 MHz, an engine cycle will be equal to 10 ns.

Start Time

This can be distinguished in the following definitions:

HVI Execution Start Time :

This is the time 0 for the HVI execution. It always matches the rising edge of the Sync Pulse.

Statement Execution Start Time :

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 of the Start Delay are multiples of the *HVI engine clock period* . For a clock rate of 100MHz the *clock period* is 10 ns, so the acceptable values are the multiples: 0 ns, 10 ns, 20 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.
- 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 of the Least Common Multiple of the clock frequencies of all engines included in the HVI that can be represented in a signed 64-bit integer 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.

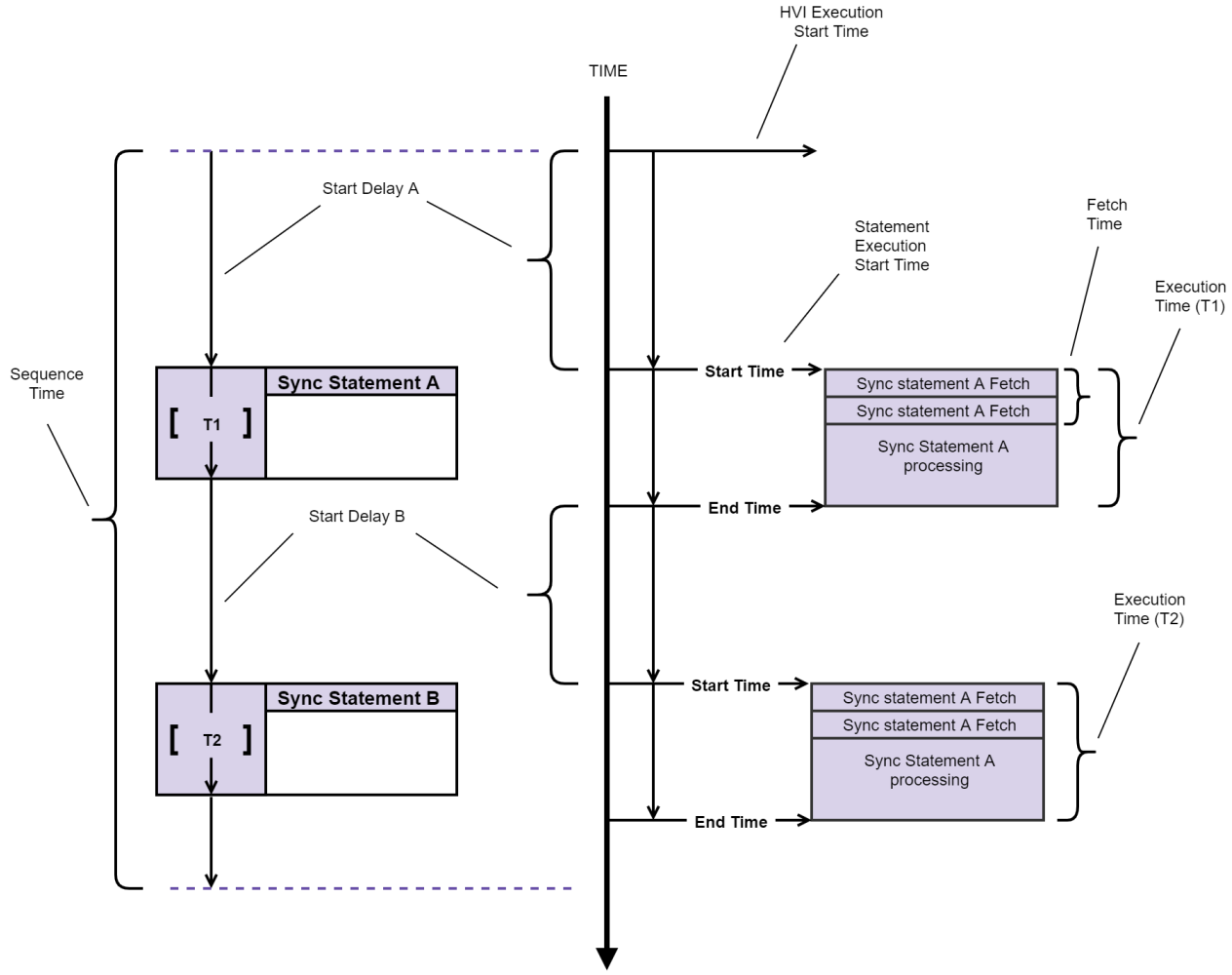
Internal Sequences

Sync statements and some Local 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

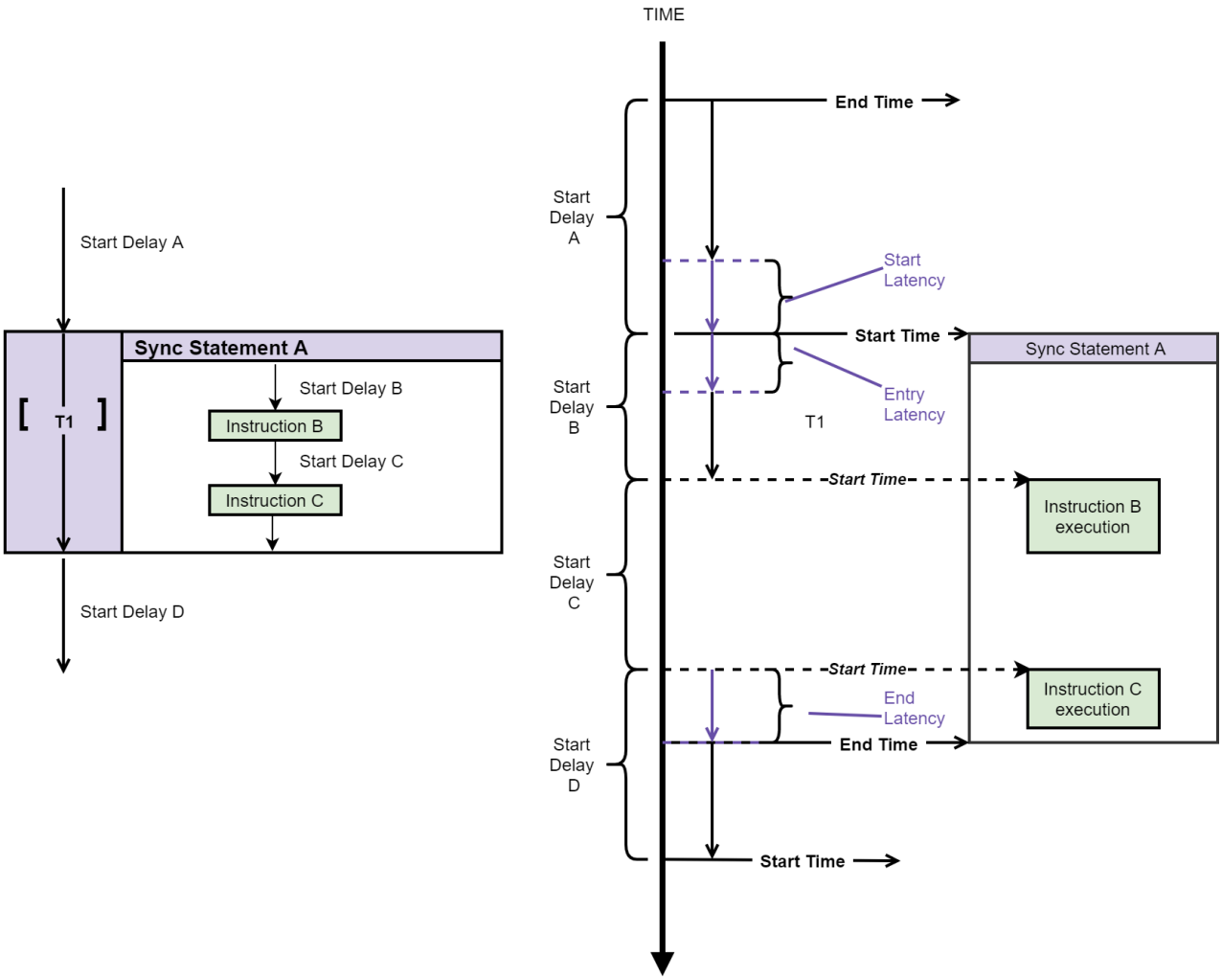
This is the minimum number of clock cycles a statement requires to exit its execution, before another statement can be executed.

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 Timingand Local Statement Timing. The Latency values are listed in Sync Statement Timing Tables, Local Flow-Control Statement Timing Tablesand Local Instruction Statement Timing Tables.

The following diagram shows the Start, Entry and End Latencies and how they relate to Start delays:



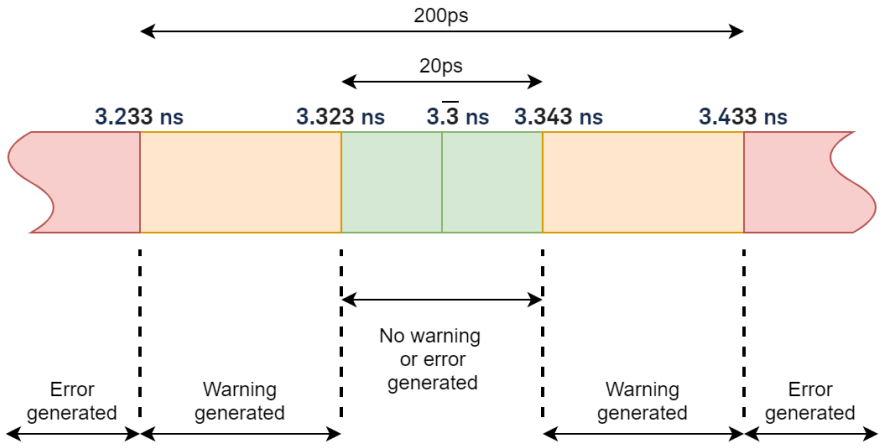
Error and Warning Margins

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.

Setting Start Delays

This chapter explains the formulae to calculate the Start delay for each type of statement, however, the compiler can calculate this value for you. If you set a Start delay too low, the compiler provides the minimum Start delay possible.

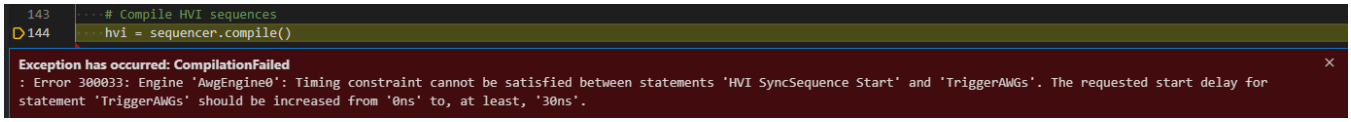
If you don't know what to set and you don't need to calculate the minimum Start delay using the minimum latency and explanations provided in this chapter, try setting the start delay to 0. When you compile, the compiler will calculate the minimum and indicate it.

Example: How to Use Compiler Messages to Set a Start Delay

In the following example code snippet, a Sync Multi-Sequence Block is added as the first statement to an HVI Sync Sequence. To find out 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 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 to set a start delay value that can accommodate the requirements for all the different instruments that are included in the HVI.

Duration Property of Statements

The Sync statements and Local flow-control statements If and While include a `duration` property you can set. The `duration` property enables you to specify the time interval a statement takes to execute. Its 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 of the duration are multiples of the engine clock period. For a clock rate of 100MHz, the clock period is 10 ns, so the acceptable values are the multiples: 0 ns, 10 ns, 20 ns, etc. The acceptable margin of the value is defined in the Error and Warning Margins section in About Time Management and Latency Concepts.
- The minimum value depends on the internal processes of the statement, the start delays, and executions times of the included statements.
- The maximum boundary is only limited by the actual representation of the value in hardware and software. While this limit in hardware can vary, in software it is defined as: The maximum value of the *Least Common Multiple* of the clock frequencies of all engines included in the HVI 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.

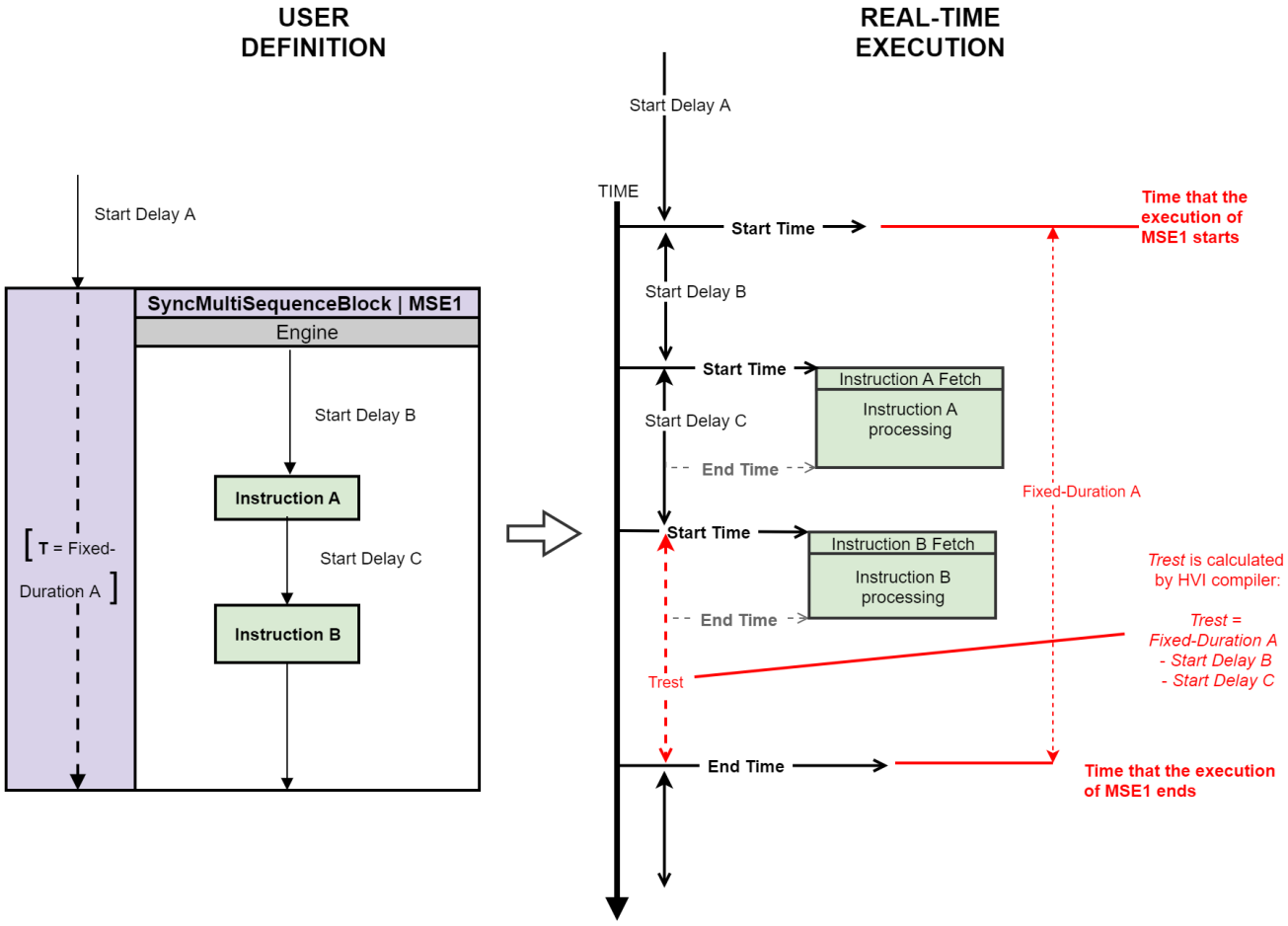
If the `duration` is set to a **fixed-time** interval, then the execution time of the statement shall match the value specified in the `duration` property. If this time cannot be matched an error is generated. For example, this can happen with an if-statement when more time is required to complete the 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:

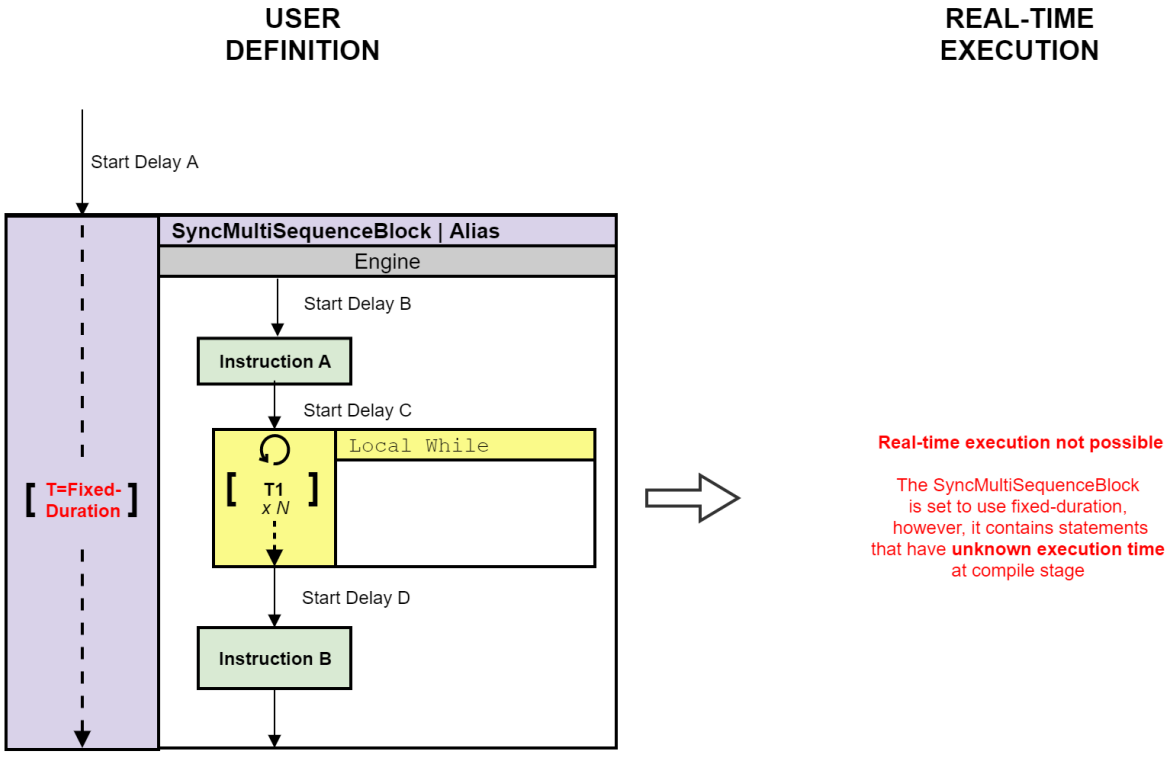


Python code for the preceding diagram:

```
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:



Local Statement Timing

This section describes Local statement timing. It contains the following sections:

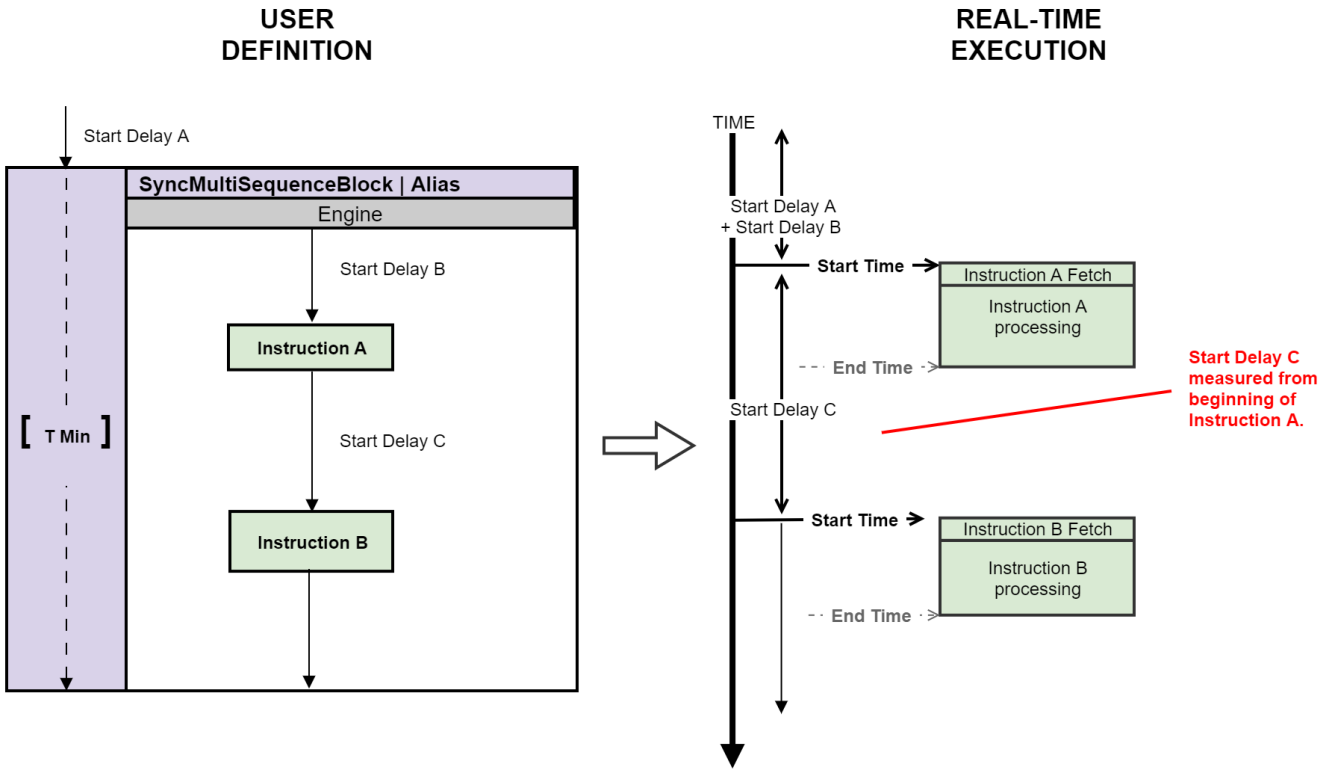
- Local Instruction Timing.
- Local Flow-Control Statement Timing.
- Calculating Local Instruction Start Delay Requirements.

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:

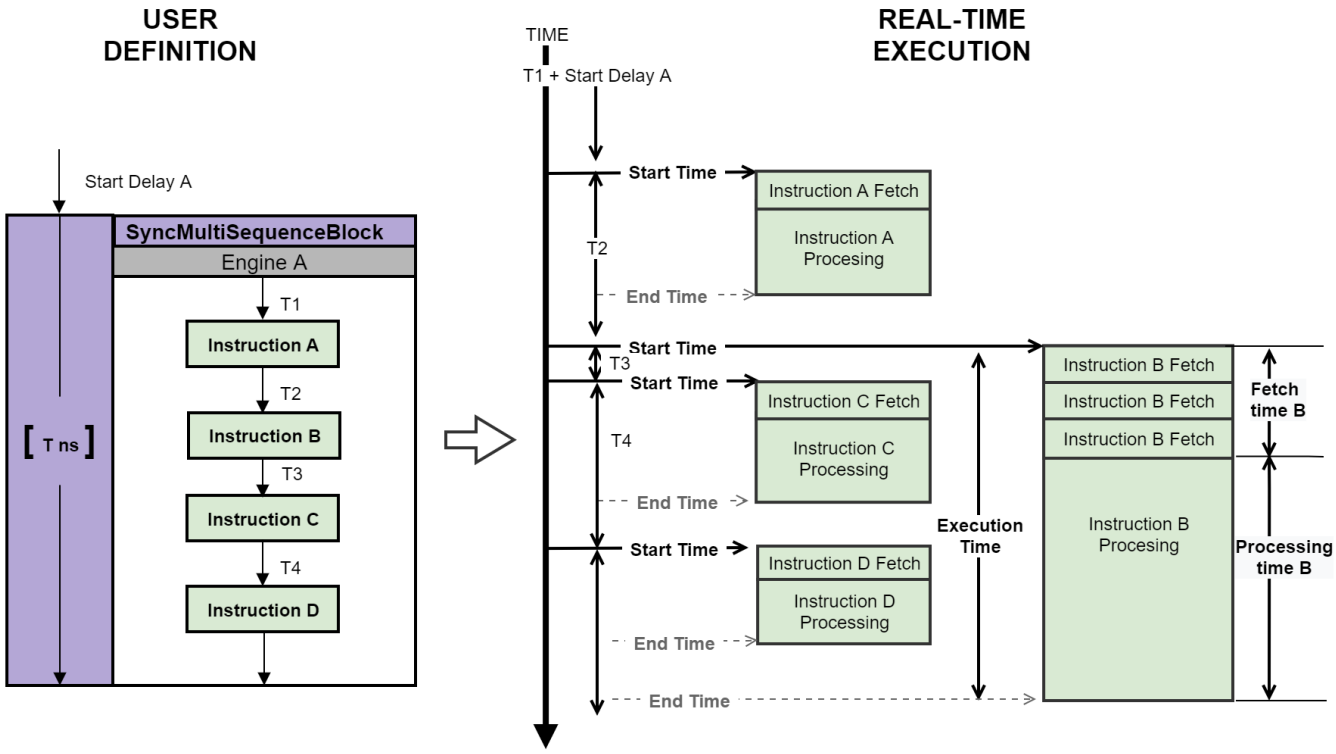


Overlapping Local instructions

In some cases, 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.



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.

Statements 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.

Statements can be scheduled for execution together, however, depending on the statements 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.

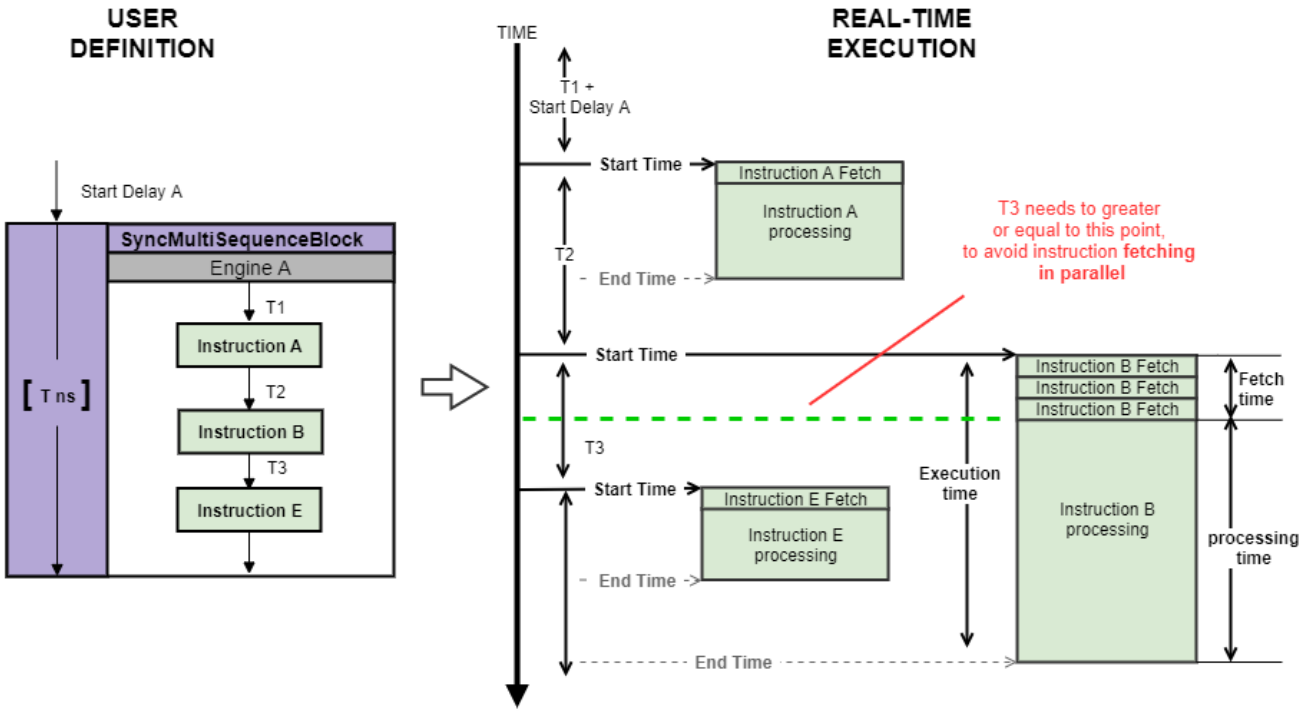
At compile time, HVI maps the instructions to be executed to their respective supported positions. If an instruction cannot be mapped to its supported position because another instruction is already mapped there, HVI generates an error and informs the user. For example:

- 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.

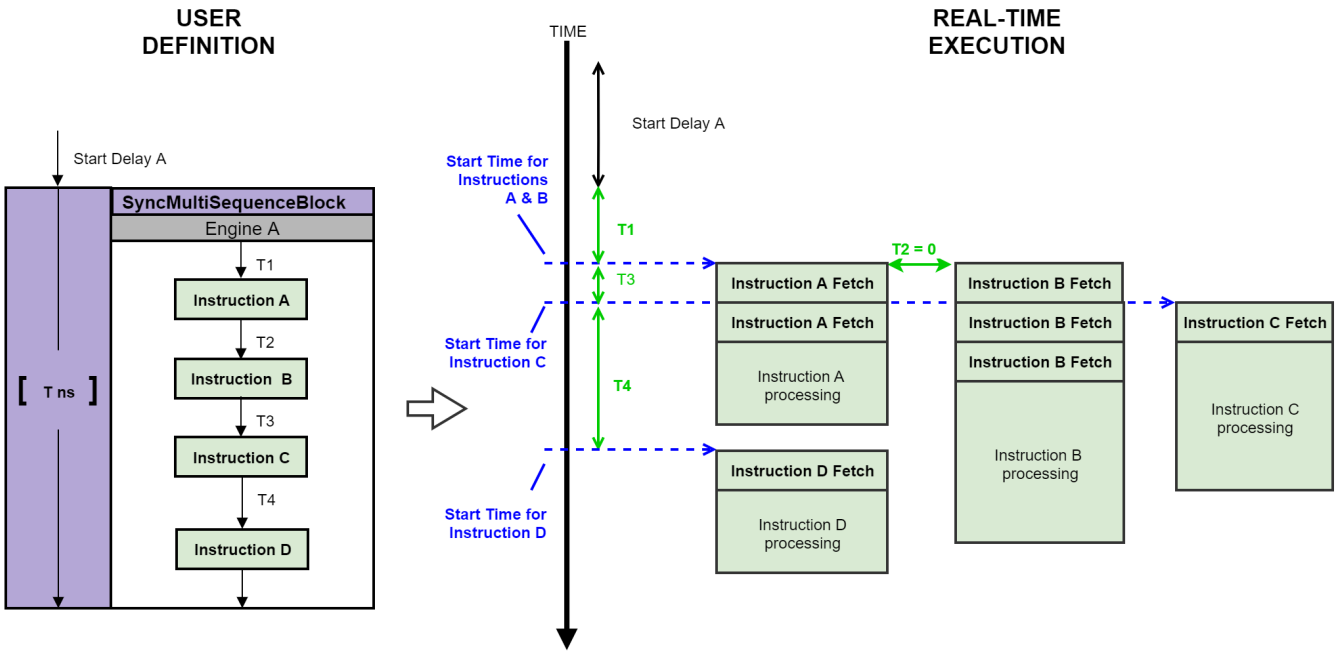
The following figures show examples of instruction fetching in parallel. For the instruction positions that are being used by each instruction, the values are from the previously defined example table.

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.

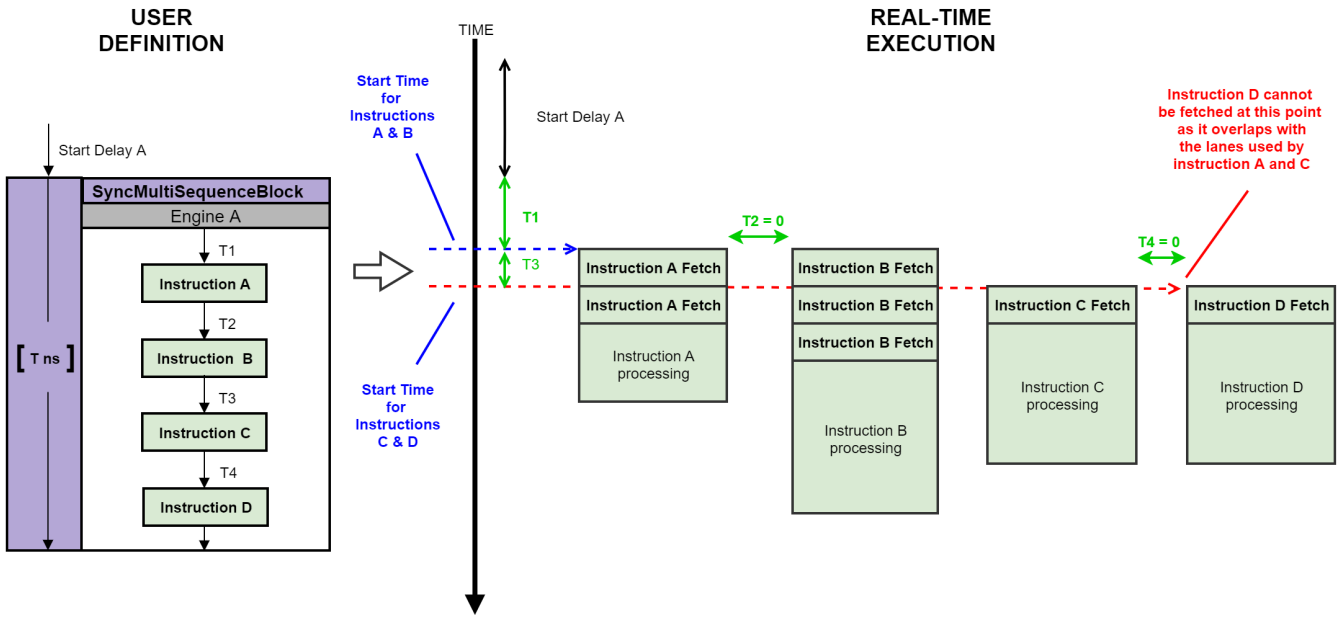


Example B. In this example start delay T1 > 0 cycles, T2 = 0 cycles, T3 = 1 cycle, and T4 = 0 cycles.

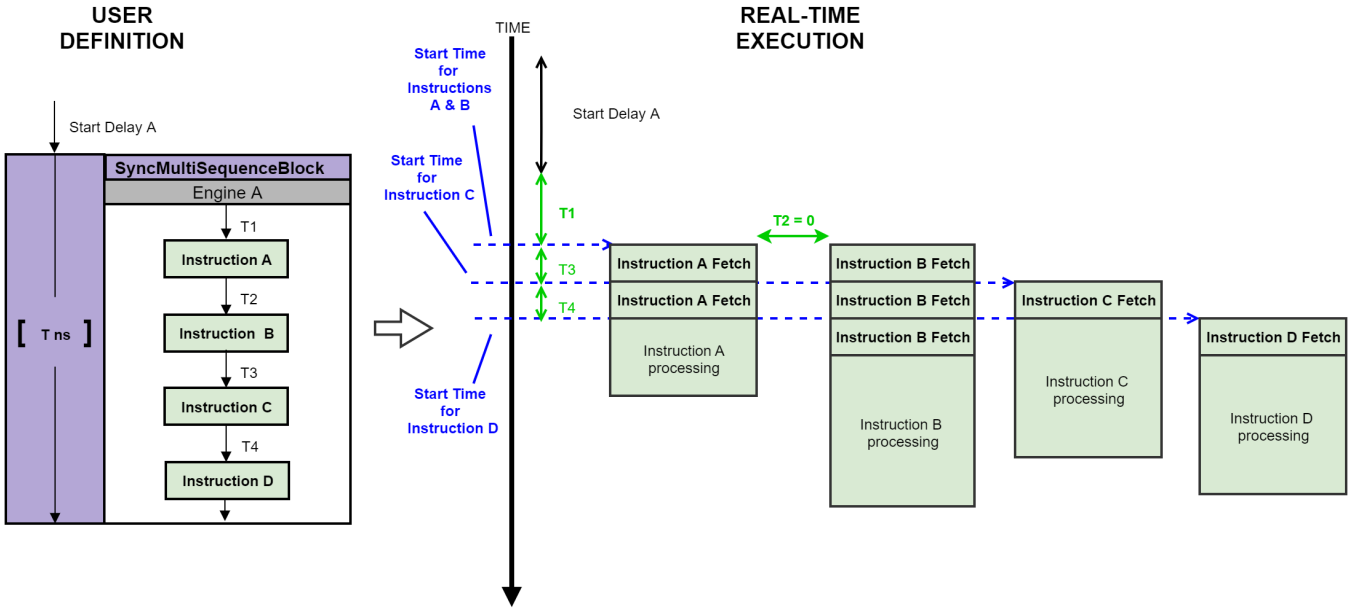
At real-time execution, after the T1 delay has passed, instructions A and Instruction B are fetched at the same time because the Start delay T2 for instruction B is equal to 0.

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.**



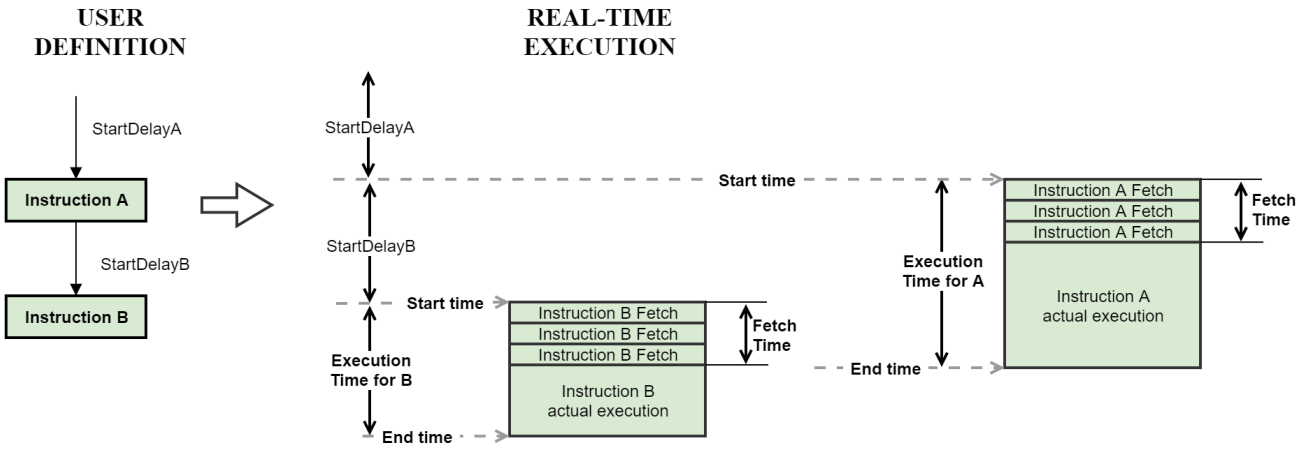
One way to fix the issue is to increase the Start delay T4 of instruction D so that it is not fetched at the same time as instruction A and C. This can be done by increasing T4 by at least 1 cycle. This is shown in the following figure:



Overlapping instruction execution with 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.

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 Start DelayB is greater or equal to the Processing Time of instruction A, minus the Fetch Time of instruction B. This way, the execution of instruction B will start after the end of execution of instruction A.

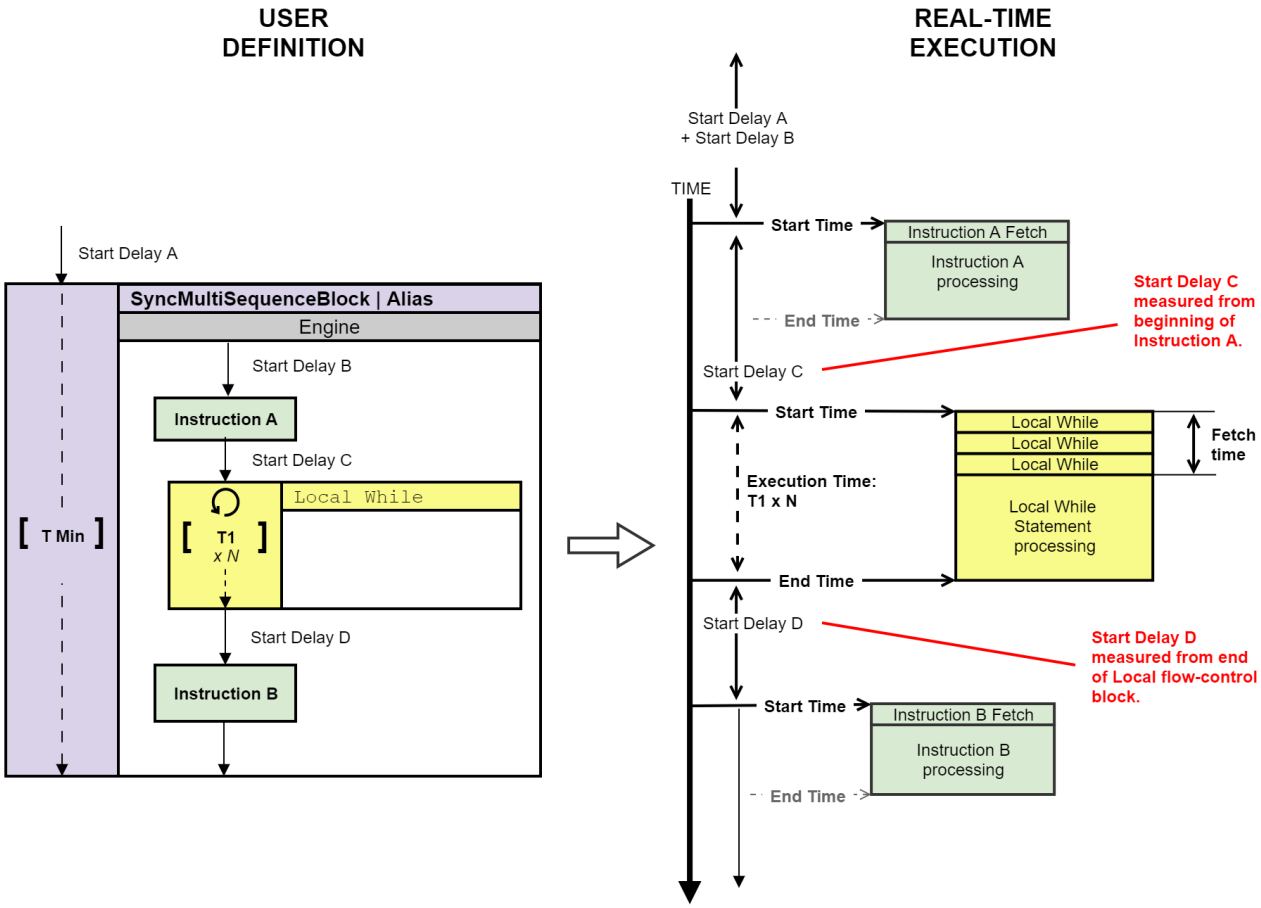


NOTE It is important to consider the effects of overlapping instructions with dependencies, because HVI does not track dependencies. It is your responsibility to ensure you have specified sufficient Start delay between instructions with dependencies.

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

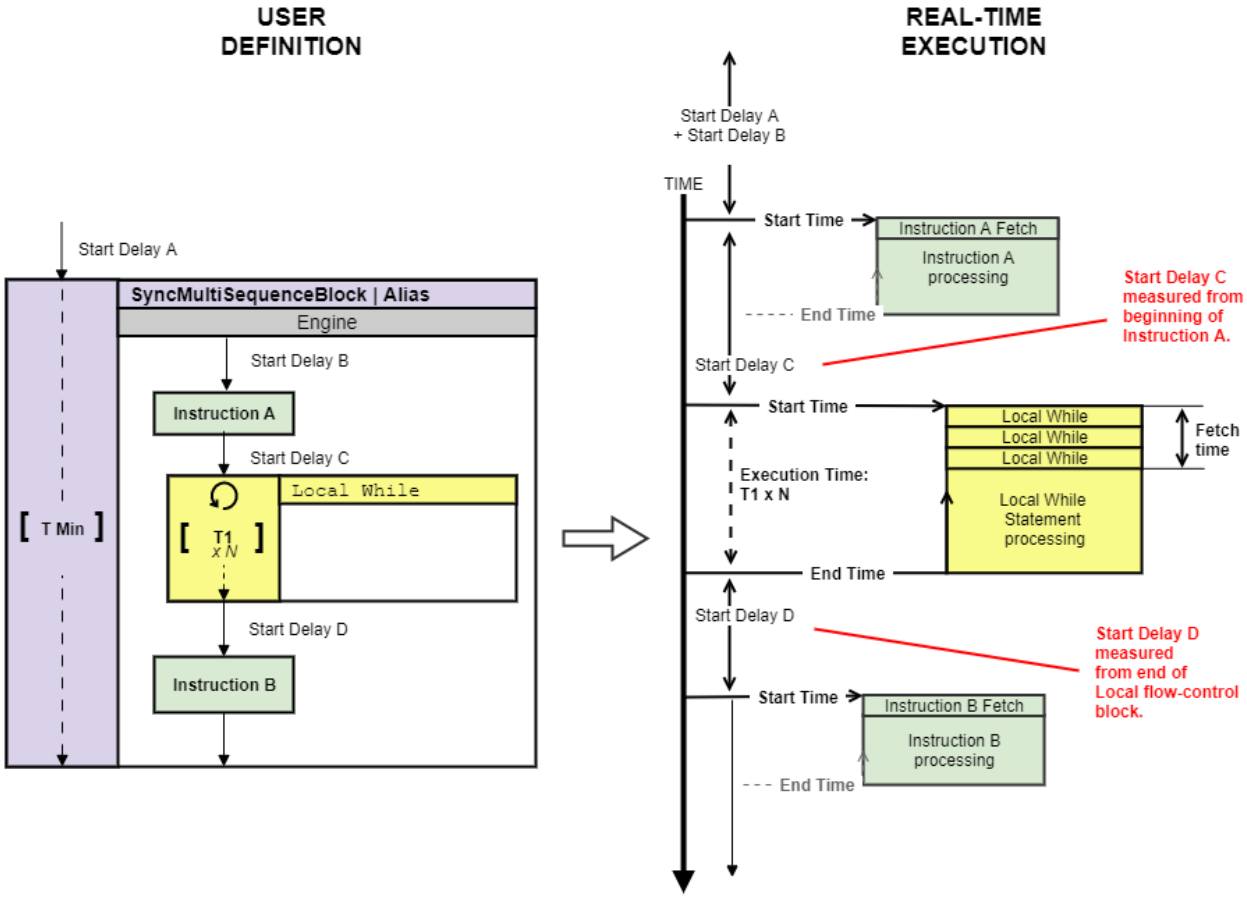
The Local while statement continues execution while a condition is met and finishes the execution when the condition is no longer met. This has the same timing as Sync while statements.

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.



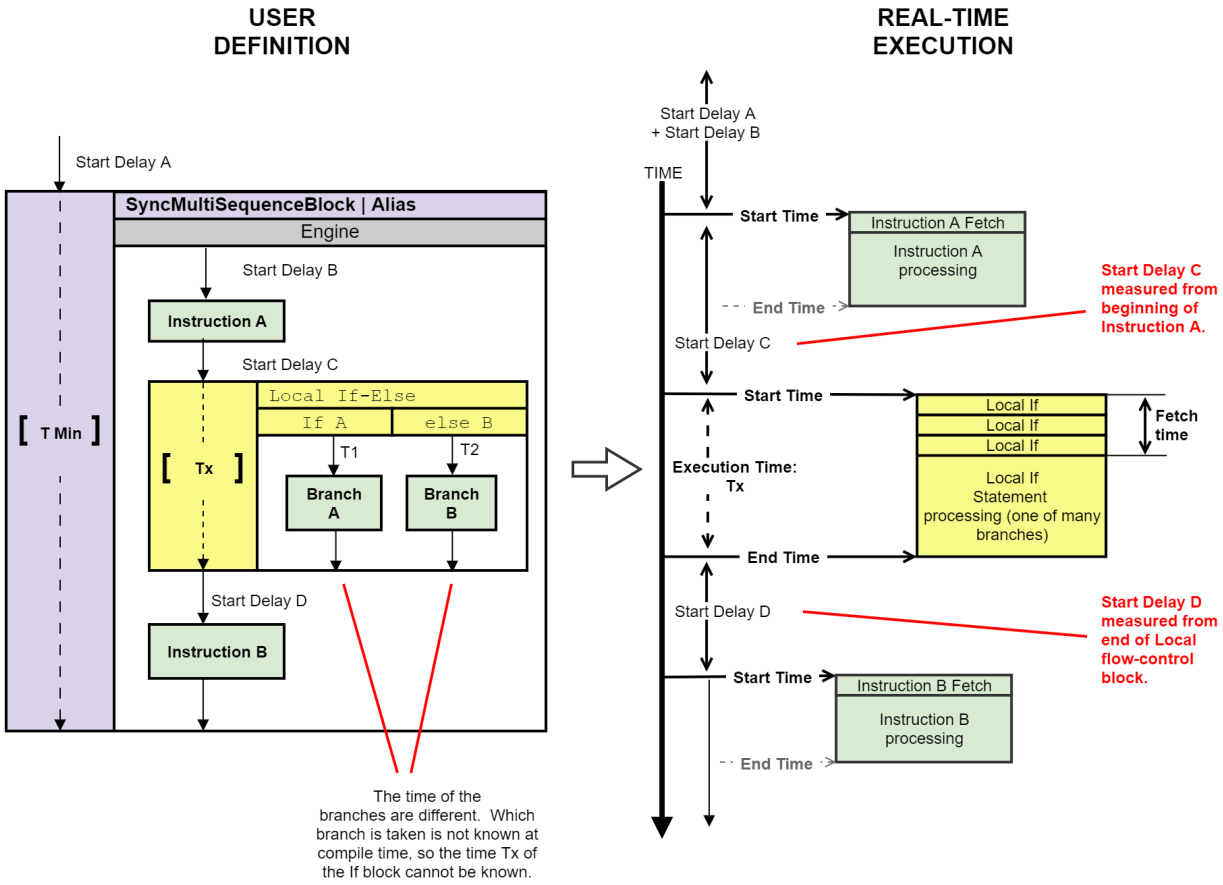
Local if

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

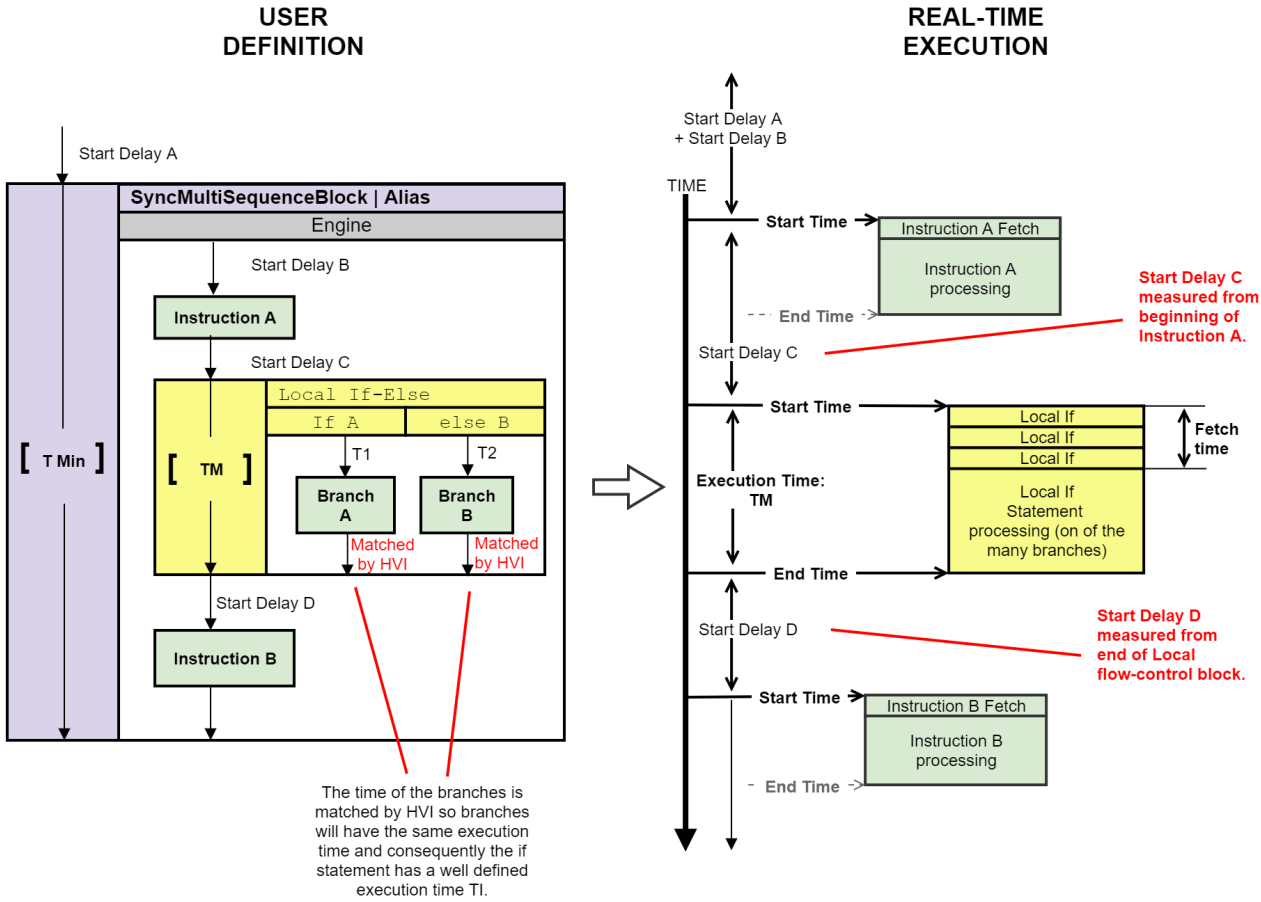
Unlike other flow-control options, the Local if statements can have different execution paths, each with different times. The matched branches option enables you to control how the HVI deals with them.

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.



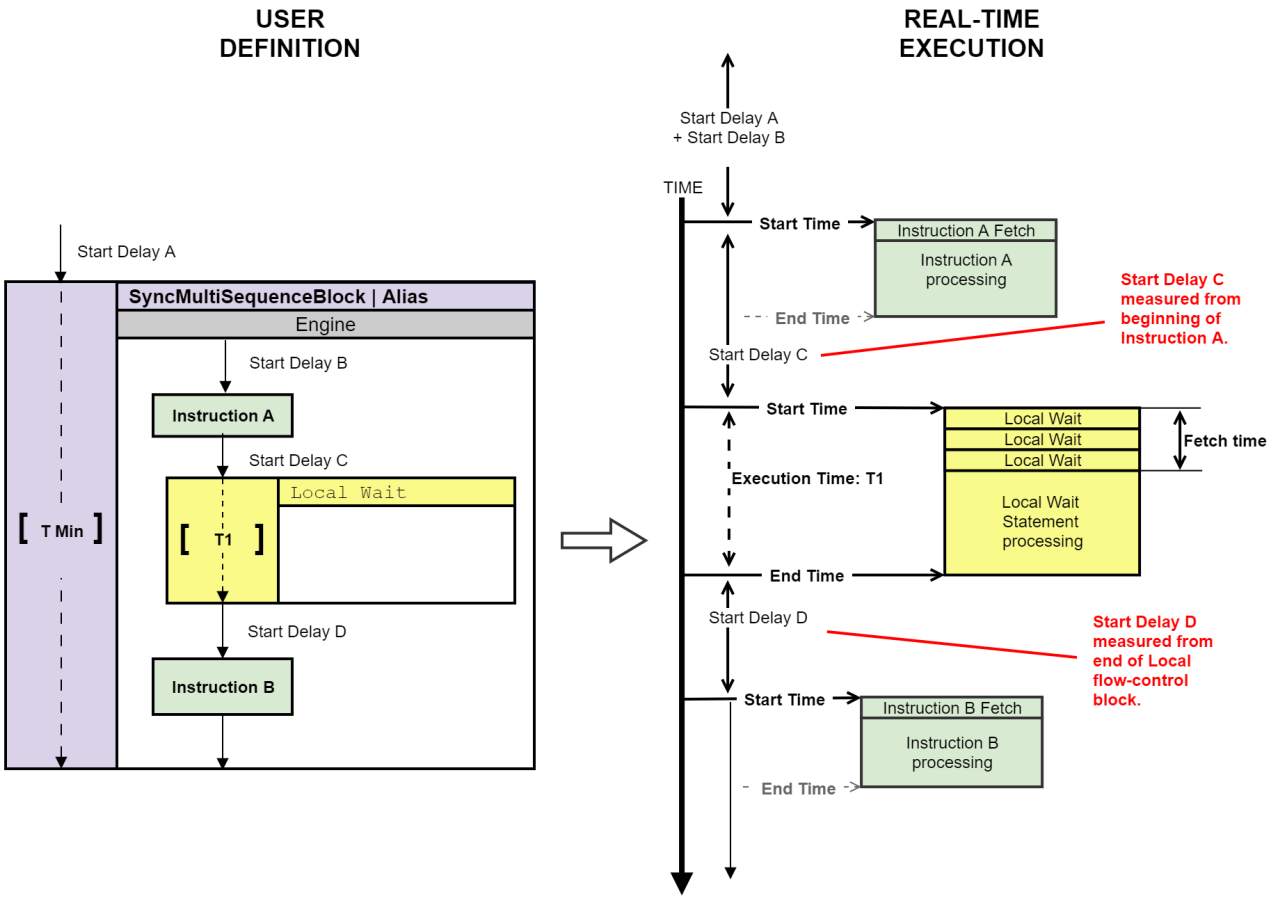
Local wait (event or time in register)

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.



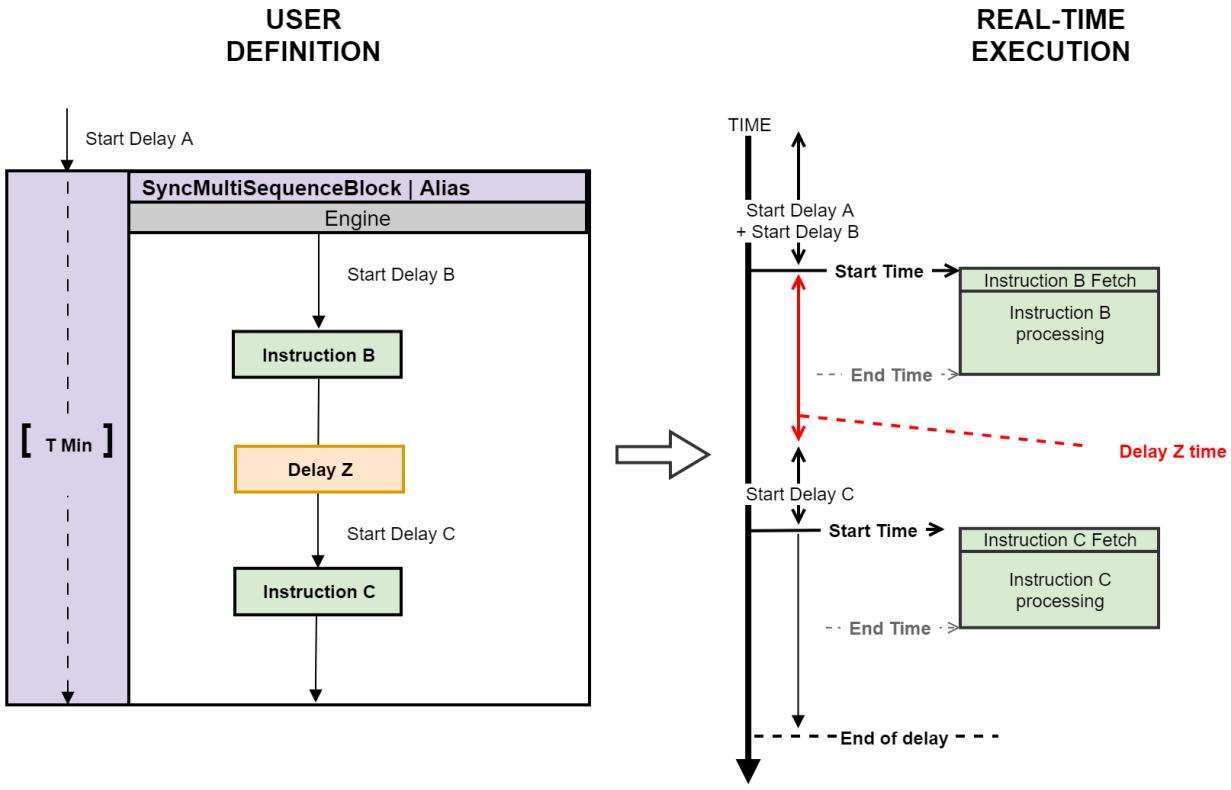
Local delay statement

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** :

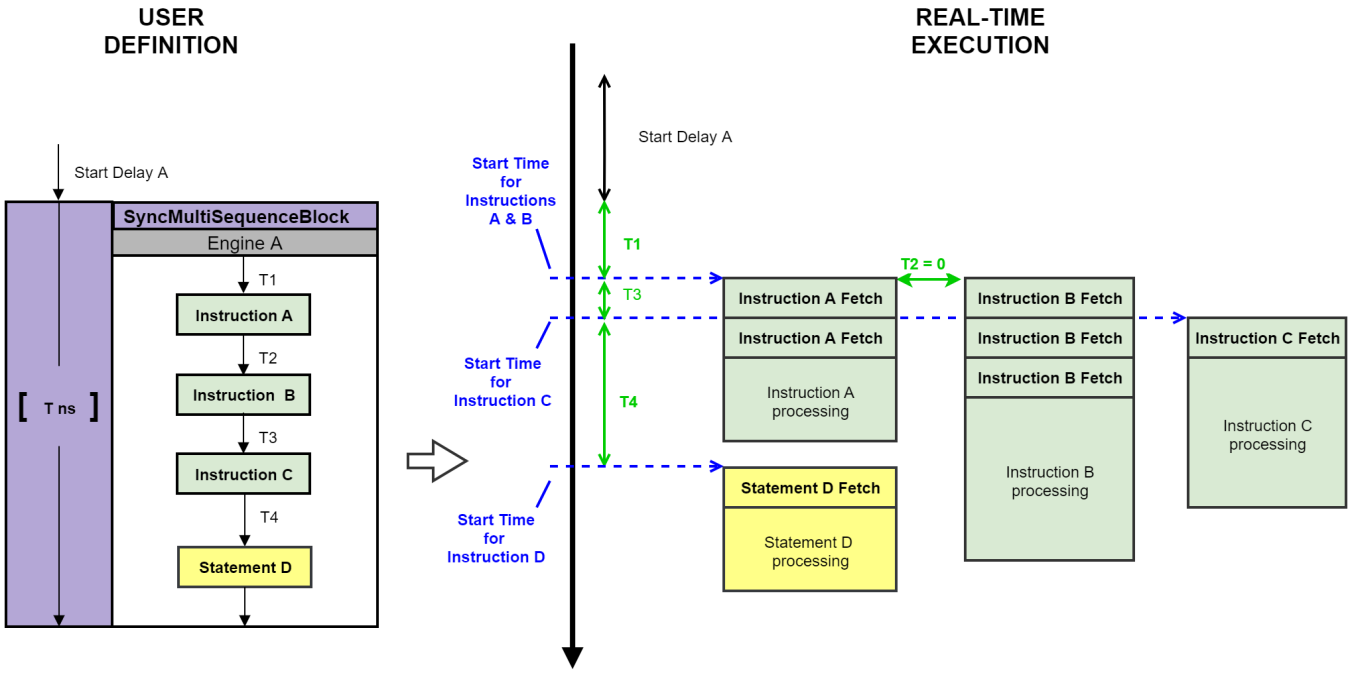


Calculating Start Delay Requirements for flow-control statements after Local instructions

To find out the minimum valid value for the Start delay of a flow-control statement, it is important to know the end-latency of the last statement. If there are local instructions before a flow-control statement, 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.

The same principle applies when calculating the end-latency of a local sequence that has local instructions as its last statements.



Calculating Local Instruction Start Delay Requirements

The following examples show how to calculate the latency for Local instruction statements using the latency information provided in Local Instruction Statement Timing Tables and in the instrument documentation.

When an instruction requires the result from a previous operation, the *minimum delay* (MinDelay) from the previous (1st instruction) to the current (2nd instruction) is given by this equation:

```
MinDelay_Instr1_to_Instr2 = Instr1_ExecutionTime - Instr2_FetchTime
```

For those statements that include a minimum Start delay, the minimum time is the highest value of the minimum Start delay or the value calculated in the equation above.

NOTE The MinDelay resulting from the previous calculation using the Fetch time is not enforced by the compiler. 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.

Example 1: Add instruction 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_FetchTime = 8 - 4 = 4 cycles
```

To use this value, you must add the start-latency of the `If` , which, for this specific case, is 6 cycles.

The minimum start-delay that you must use to make sure that the result of the `Add` operation is used by the `If` statement is:

```
MinDelay_If = 4 + 6 = 10 cycles
```

Example 2: Add instruction 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 this equation:

```
MinDelay_If = Add_ExecutionTime - While_FetchTime = 8 - 4 = 4 cycles
```

However, since you cannot specify iteration_delay, you must make sure that this extra time is consumed before reaching the end of the internal while sequence. This can be done by adding a delay statement with at least 3 cycles of delay. This way the final delay will become 4 cycles (including the fetch time of the Delay statement).

Example 3: Add instruction followed by a Local Sync register-sharing statement

In this example an `Add` instruction writes its result to a register and then then new value is shared to other modules. The Sync register-sharing statement is not a Local instruction statement, but the timing calculation and fetch time applies in the same way.

1. `Reg1 = RegN + 10` (Add).
2. `SyncRegisterShare(Reg1)` (sharing the result of the previous `Add` instruction).

In this case, the minimal delay between the Sync register-sharing and the previous `Add` is calculated with this equation:

```
MinDelay_SyncRegShare = Add_ExecutionTime - SyncRegShare_FetchTime = 8 - 1 = 7
cycles
```

The Sync register-sharing has no minimum Start delay, so the result of the equation can be used as:

```
MinDelay_SyncRegShare
```

For a description of Sync register-sharing, see HVI API Sync Statements.

Sync Statement Timing

This section describes Sync statement timing. It contains the following sections:

- About Sync Statement Timing.
- Sync While Timing.
- Sync Register-Sharing
- Basic Local Statement Timing Across Sync Multi-Sequence Blocks
- Sync Multi-Sequence Block Timing and Time Matching.
- Synchronization Points and Sync Sequence Start.

About Sync Statement Timing

Sync statements consume HVI engine execution time and cannot overlap their execution with other statements. The Start delay of a Sync statement is measured from the end of previous Sync statement to the start of the current one.

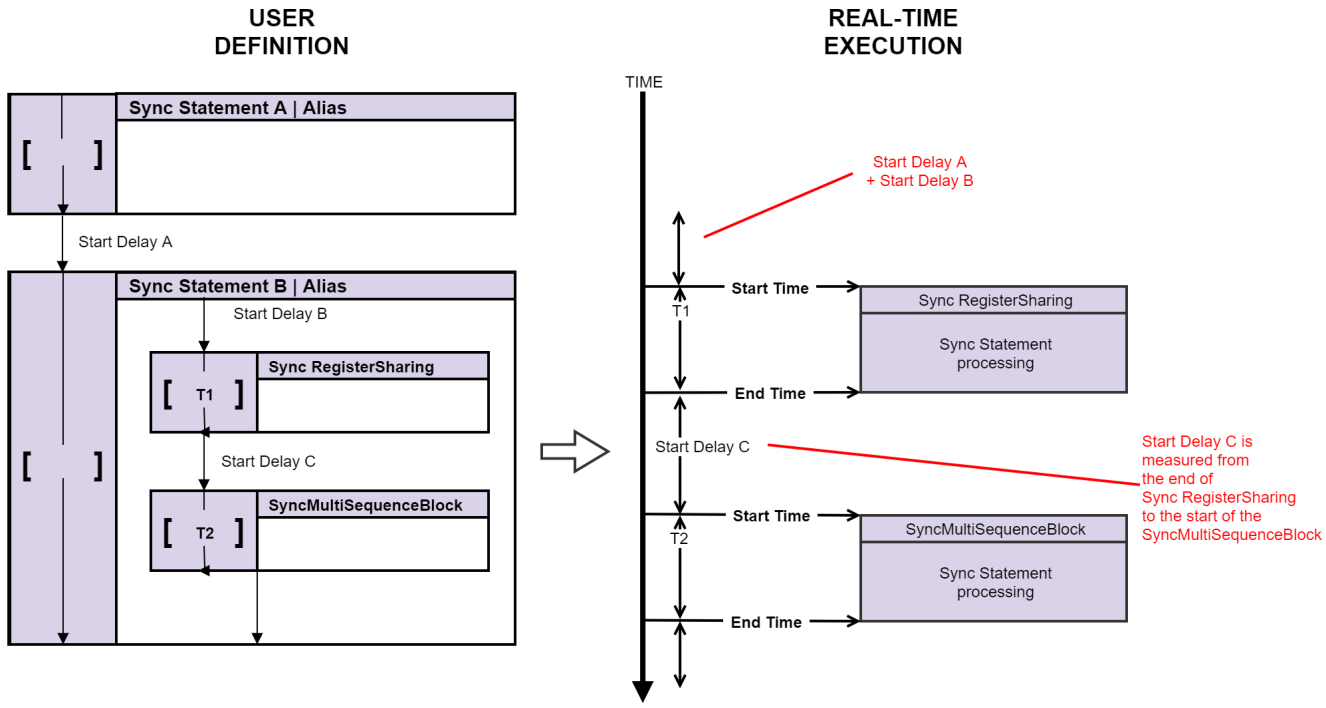
The following diagram shows the timing between a number of Sync Statements including a Sync register-sharing statement and Sync multi-sequence block statement.

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:

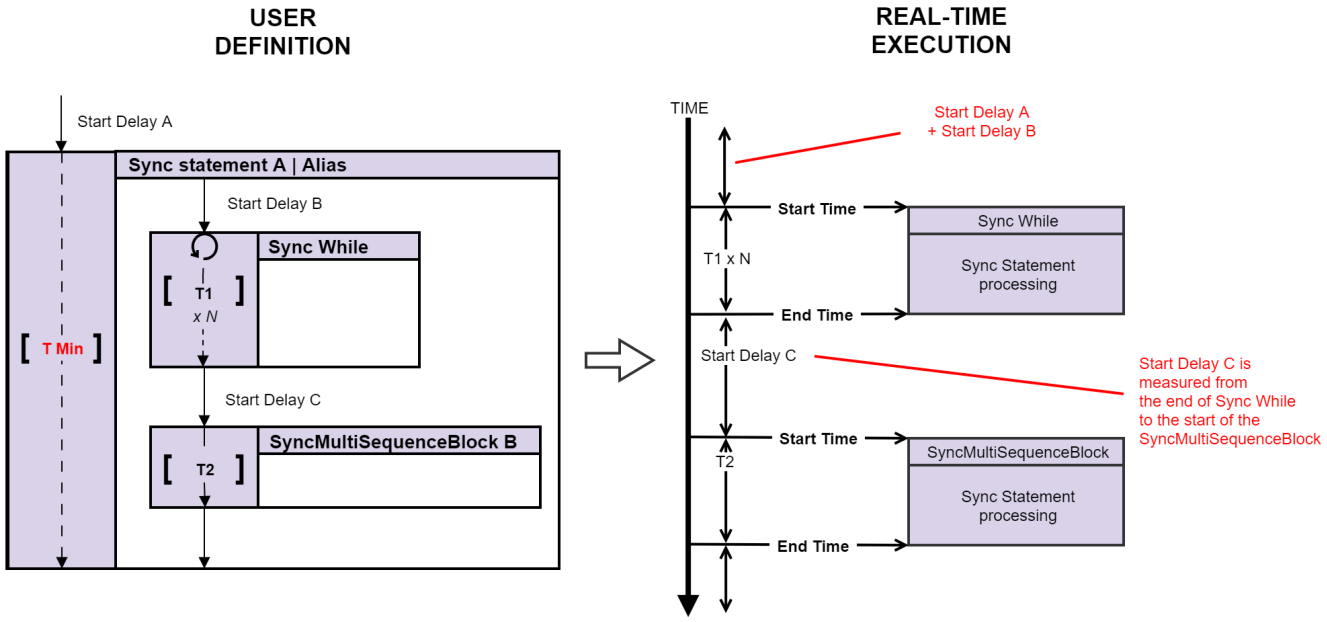


Sync While Timing

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.

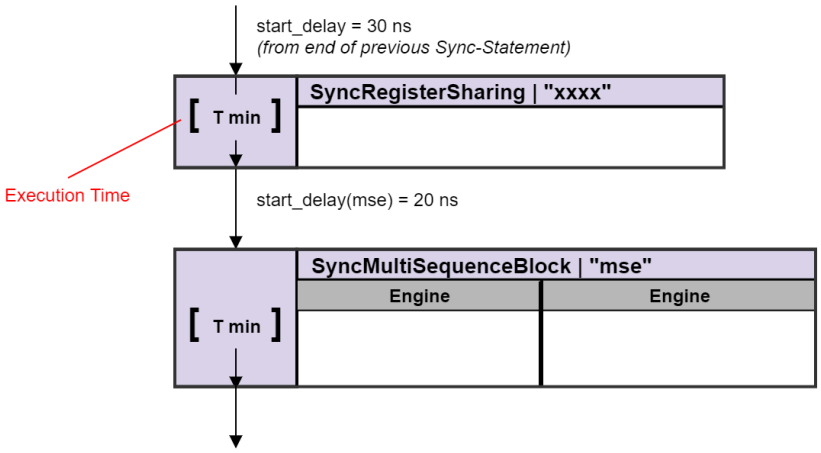


Sync Register-Sharing

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.



Basic Local Statement Timing Across Sync Multi-Sequence Blocks

This section shows basic examples of Local statements within Sync Multi-sequence Blocks and how the timing is calculated.

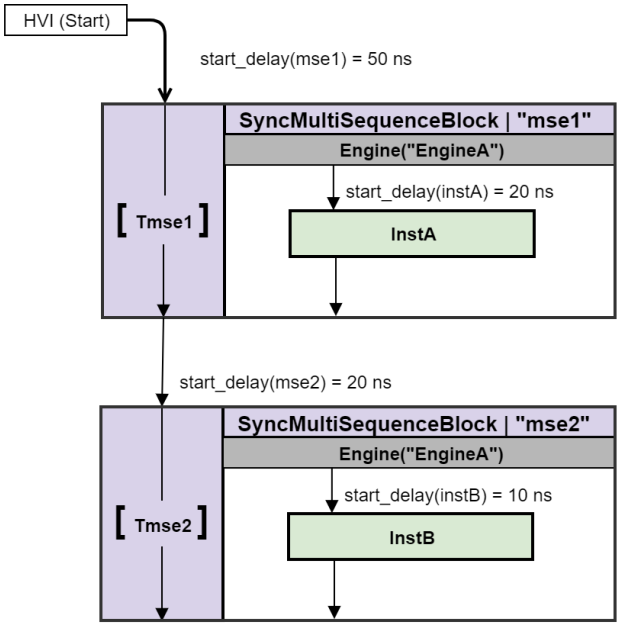
It contains the following sections:

- Simple Local instruction timing calculation example.
- Simple Local instruction with Local if timing calculation example.

Simple Local instruction timing calculation 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
```

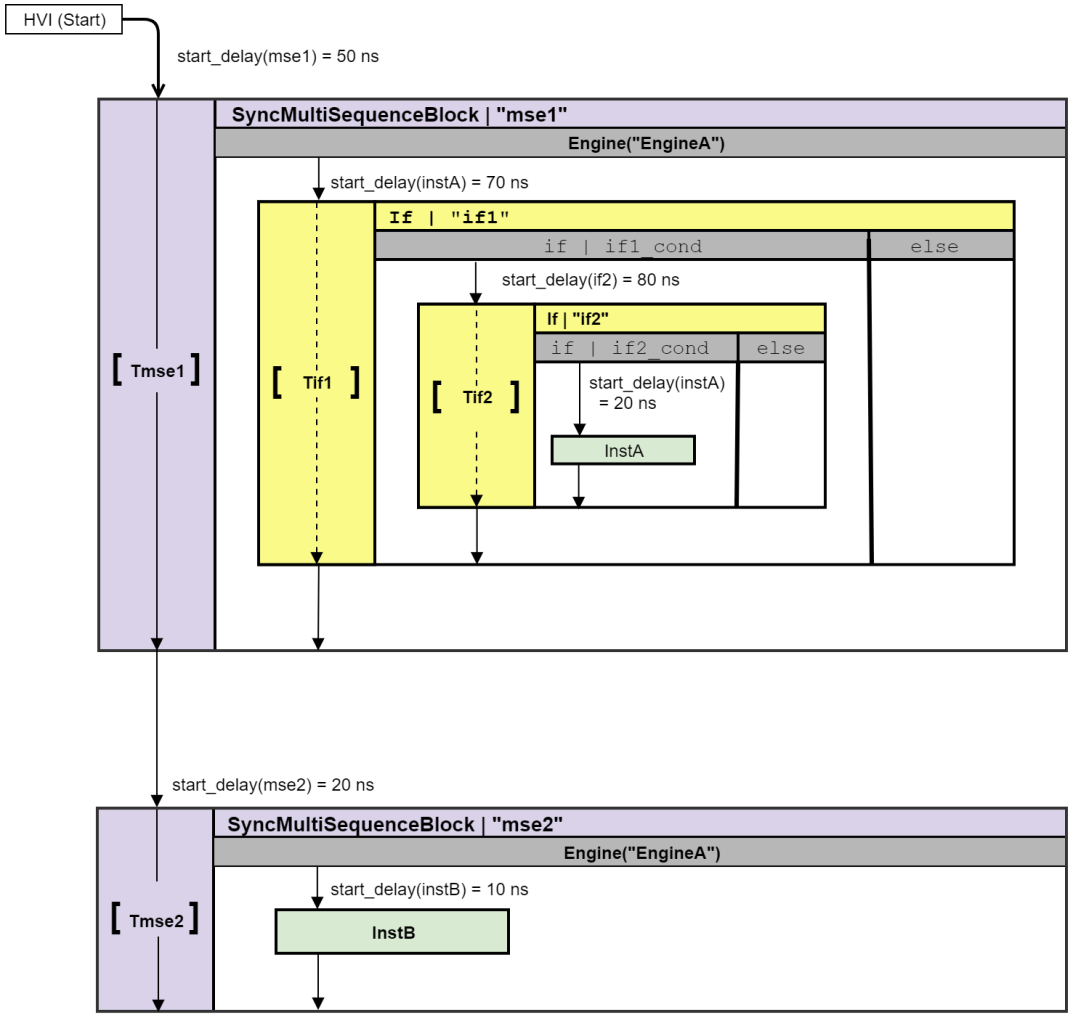
Time from InstA to InstB (`T_InstA_InstB`):

```
T_InstA_InstB = start_delay(mse2) + start_delay(instB) = 20ns + 10ns = 30ns
```

Simple Local instruction with Local if timing calculation 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:

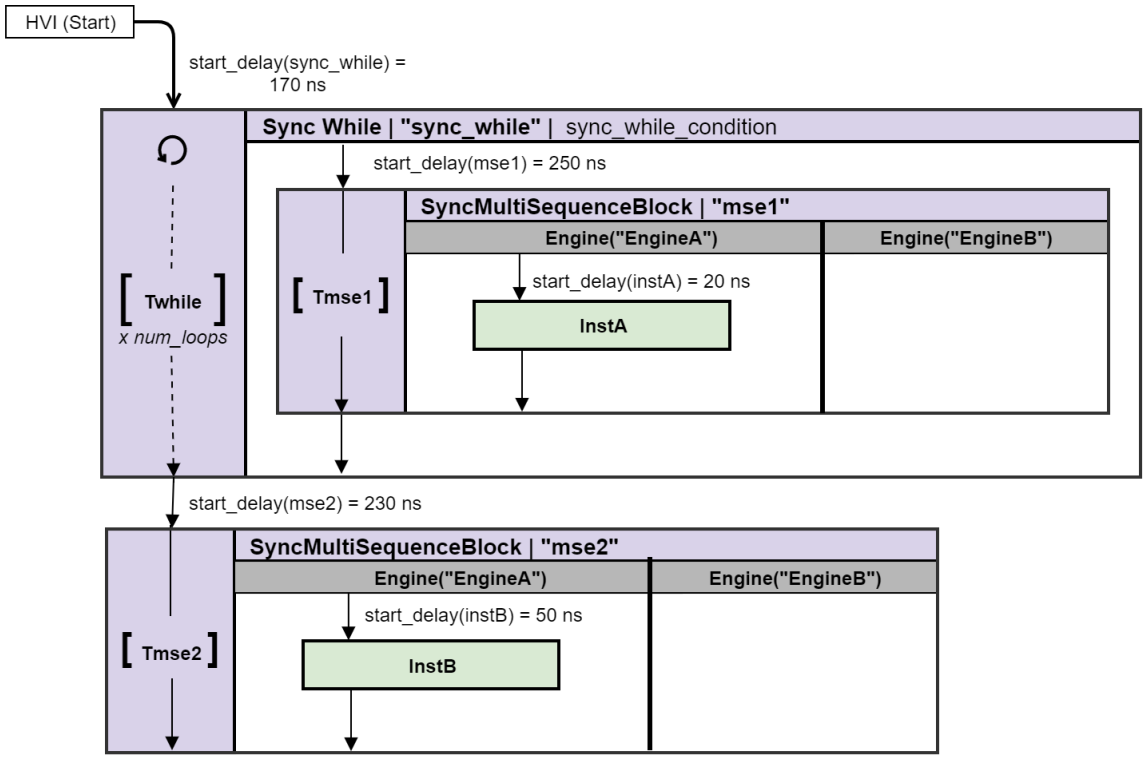
```
T_InstA_InstB = start_delay(mse2) + start_delay(instB) = 20ns + 10ns = 30ns
```

NOTE The `end_latency(mse1)` is accounted for in the `start_delay(mse2)` , this imposes a minimum value.

Calculating Sync Flow-Control Statement Latency

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.

Sync Multi-Sequence Block Timing and Time Matching

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 that are treated in a different way by HVI:

- Execution time is known at HVI compilation time for all Local sequences within the Sync multi-sequence block.
- Execution time is unknown at HVI compilation time for one or more Local sequences within the Sync multi-sequence block.

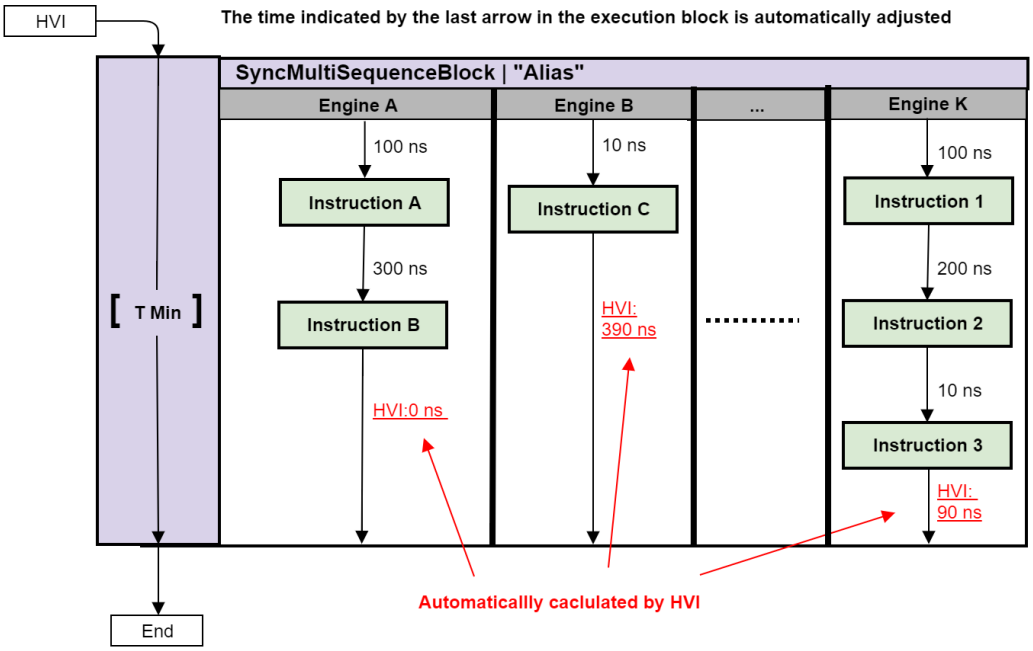
Sync multi-sequence block containing Local sequences with known total execution time

A Sync multi-sequence block can contain instructions or flow-control statements with execution times that are known at HVI compilation time. In this case the HVI accounts for the different sequence execution times during compilation and then adjusts the final times. This ensures all of the Local sequence reach the end of the Sync multi-sequence block at the same time.

Sync multi-sequence block with minimum execution time

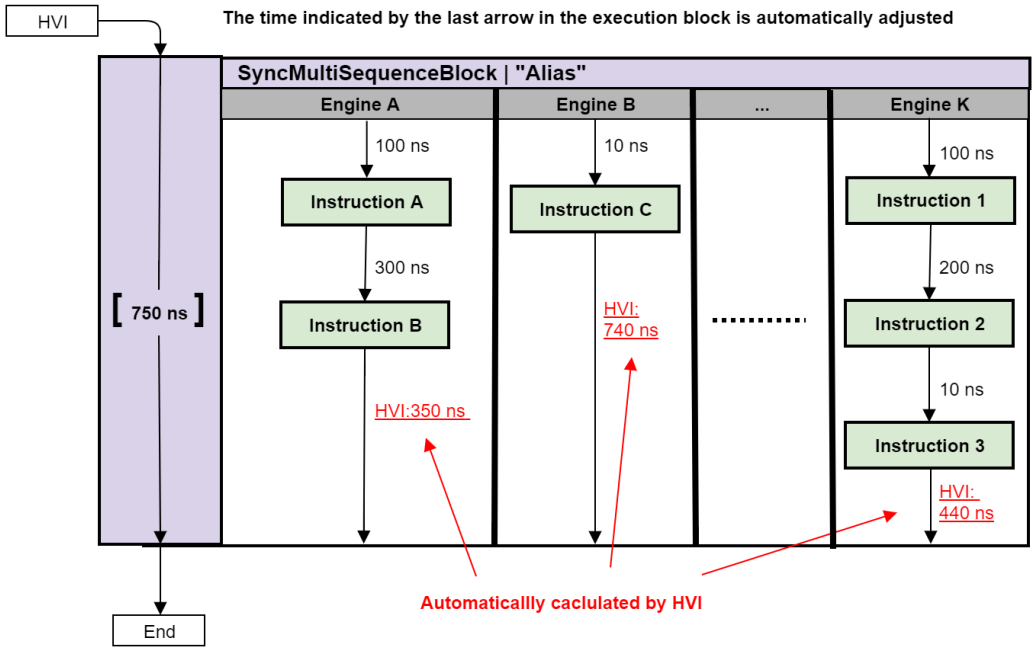
In the following diagram, the time of the Sync multi-sequence block is not specified, so the compiler adjusts the total execution time of all Local sequences to the longest one. The times of the instructions and the delays between them are known, so the timing between them and the timing of the entire block can be calculated. The Sync multi-sequence block execution time is set to the minimum possible time given by the longest Local sequence.

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.



Sync multi-sequence block with a specific execution time (duration property)

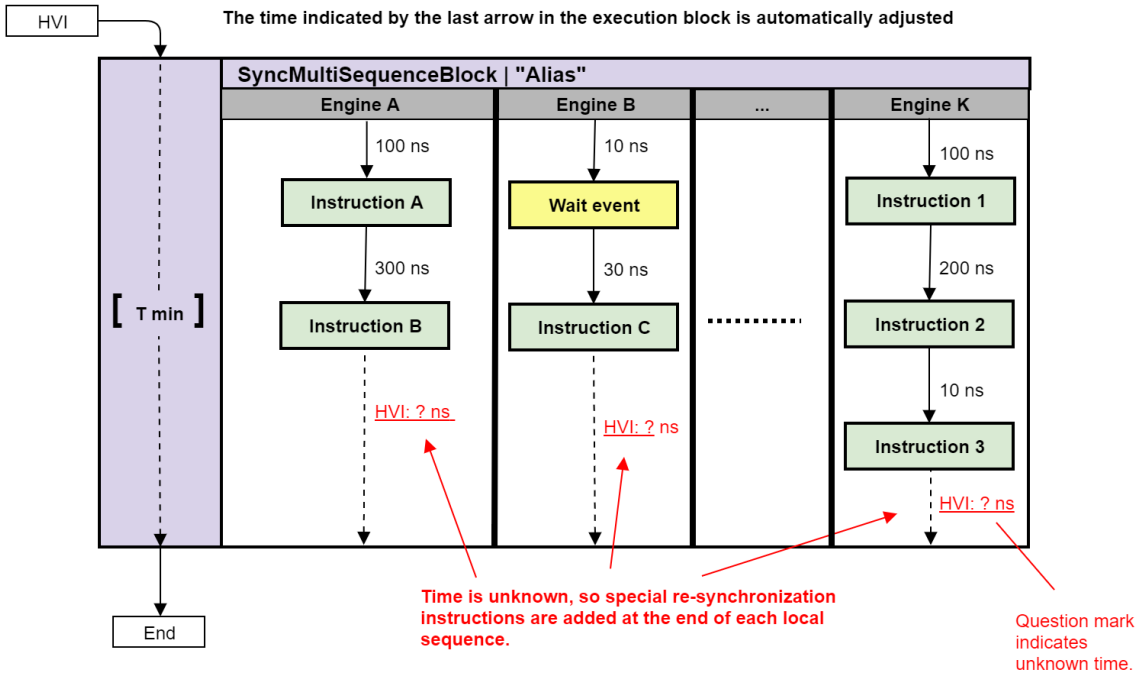
In the following diagram, the times of the instructions and the delays between them are known, so the timing between them and for the entire block can be calculated. In this case the total time is specified at 750 ns. The HVI calculates the times required for all the other engines to finish at the same time. For Engine A this is 350 ns, for Engine B this is 740 ns, for Engine K this is 440 ns.



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. Since HVI ensures that all the Local sequences in a Sync multi-sequence block end at the same time when there is such a Local flow-control statement, HVI implements a special re-synchronization procedure at the end of the Sync multi-sequence block.

In the following diagram, the time of the instructions and the delays between them are known, except for the execution time of the Wait event. This means the execution time of the complete Sync multi-sequence block cannot be specified. HVI still enforces all Local sequences to end at the same time, but in this case the time required at the end of each sequence is not known since it cannot be calculated during the HVI compilation, this is indicated by the dotted lines. The time of the full Sync multi-sequence block is also unknown, so this is indicated as T min with a dotted line. To enforce that all Local sequences synchronize again at the end of the Sync multi-sequence block, special re-synchronization instructions are added at the end of each local sequence in the Sync multi-sequence block. This re-synchronization procedure relies on triggering resources to re-synchronize the execution of the Local sequences on all the HVI engines. This procedure is explained in detail in the following section.



Synchronization Points and Sync Sequence Start

All Sync statements enforce synchronization points across instruments and HVI Engines. The start and the end of a Sync multi-sequence block or Sync while statement are examples of Synchronization points. In addition to Sync statements, the start of the sequence is also a critical synchronization point, it ensures that all HVI engines start execution at the same time.

There are two types of synchronization points:

Timed sync points

These points correspond to Sync statements where the timing of execution of all HVI engines in the HVI can be determined without ambiguity at compilation time. In this case, the HVI compiler adjusts the timing before the Sync point in each HVI engine to ensure all engines leave the Sync point at exactly the same time.

Triggered sync points

The triggered sync points are the points where an active triggering process is required to re-synchronize the execution of all HVI engines. They are necessary in those cases when the execution time of one or more HVI engines cannot be determined at compile time. In the HVI diagrams in this User Manual, a dotted arrow is used to indicate this. Triggered sync points occur in the following cases:

- At the start of the HVI Sequence, that is, the Global Sync sequence.
- At the end of a Sync multi-sequence block statement, where one of the local sequences contains one or more statements with an execution time that is unknown at compile time. Possible cases of the unknown execution time are:
  - 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.

Timing with triggered-sync points

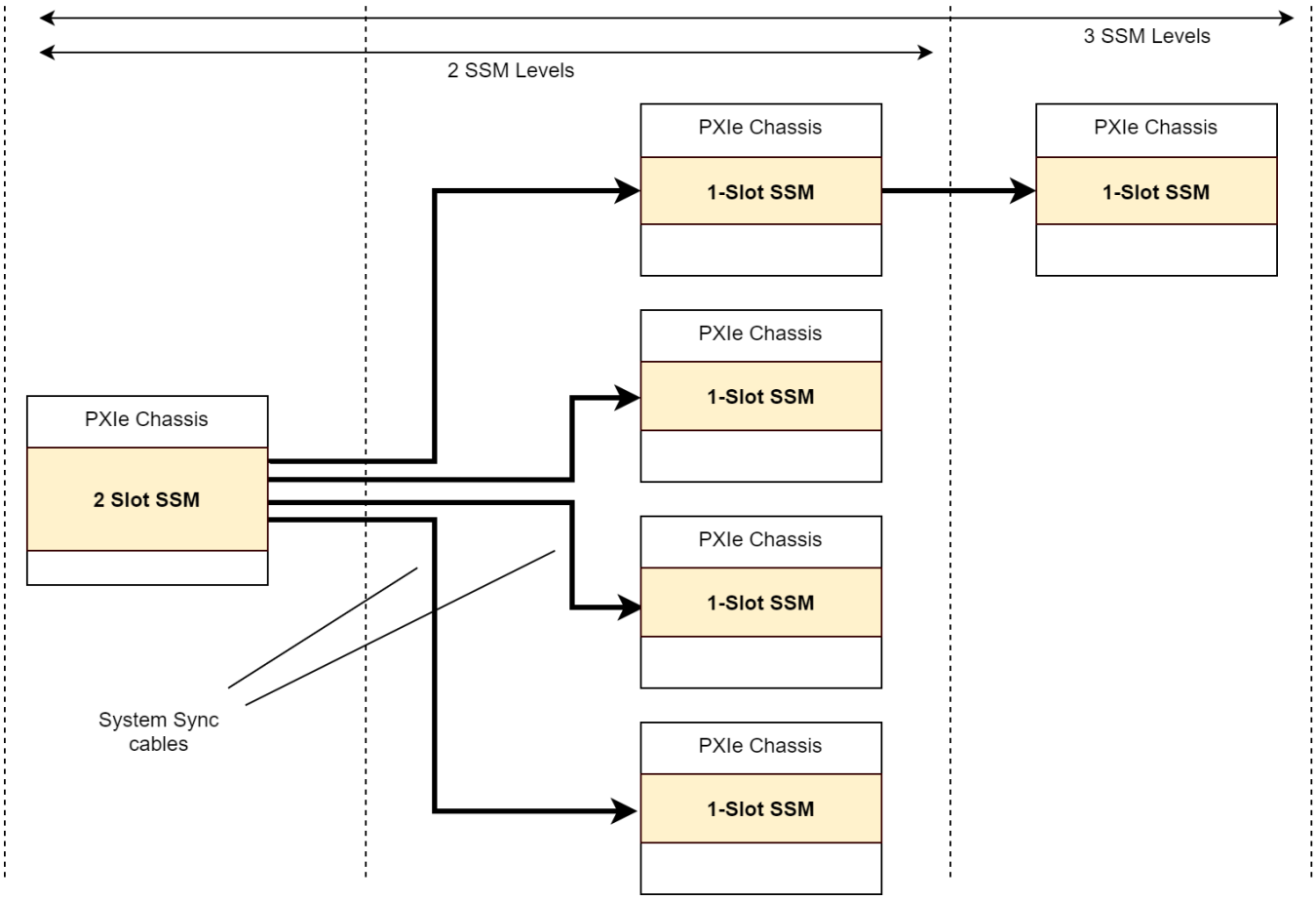
Triggered-sync points re-synchronize all of the HVI engines, this guarantees that all the HVI engines then continue execution at exactly the same point after the Sync point. The execution resumes in all HVI engines at the same time, aligned with a sub-sequence Sync pulse, this forces the execution to 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.

The time of the Sync-period a system synchronizes to depends on 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 2021 release supports up to 6 chassis.

The following diagram shows a 6 chassis system with 3 SSM levels:



The following tables shows the Sync period for different numbers of chassis and SSM levels:

Number of Chassis	Number of SSM levels	Sync-period (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 2021

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:

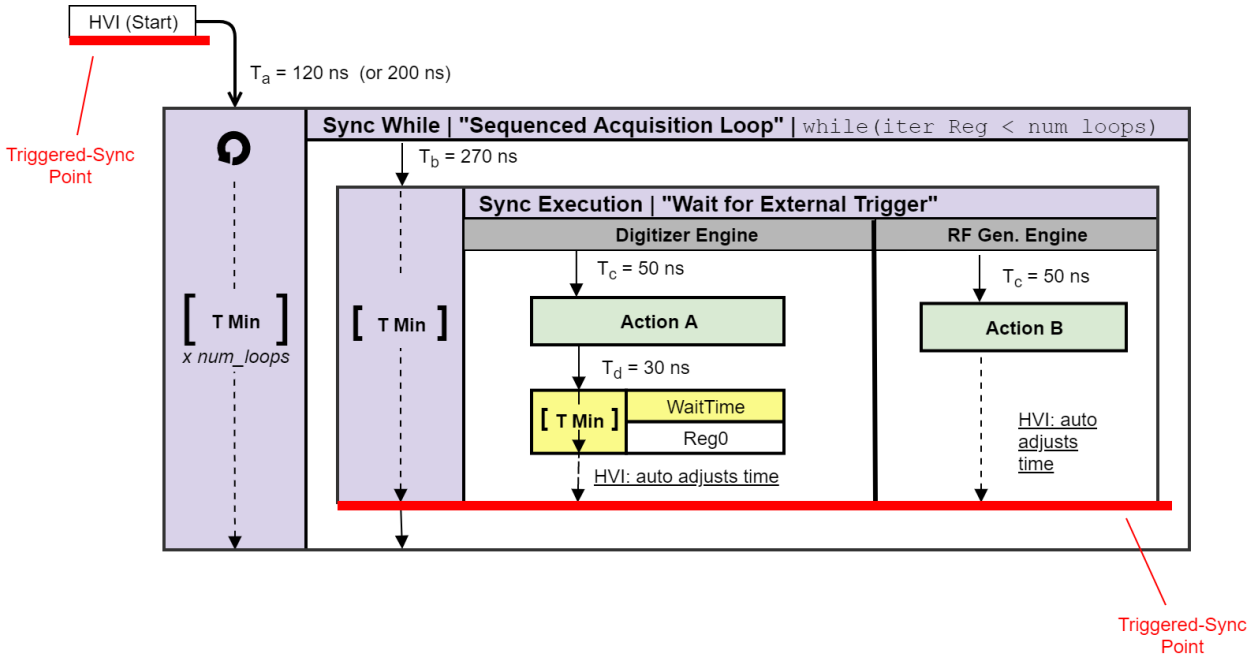
```
triggered_sync_delay = end_latency + sync_overhead + edge_offset + sync_period
```

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

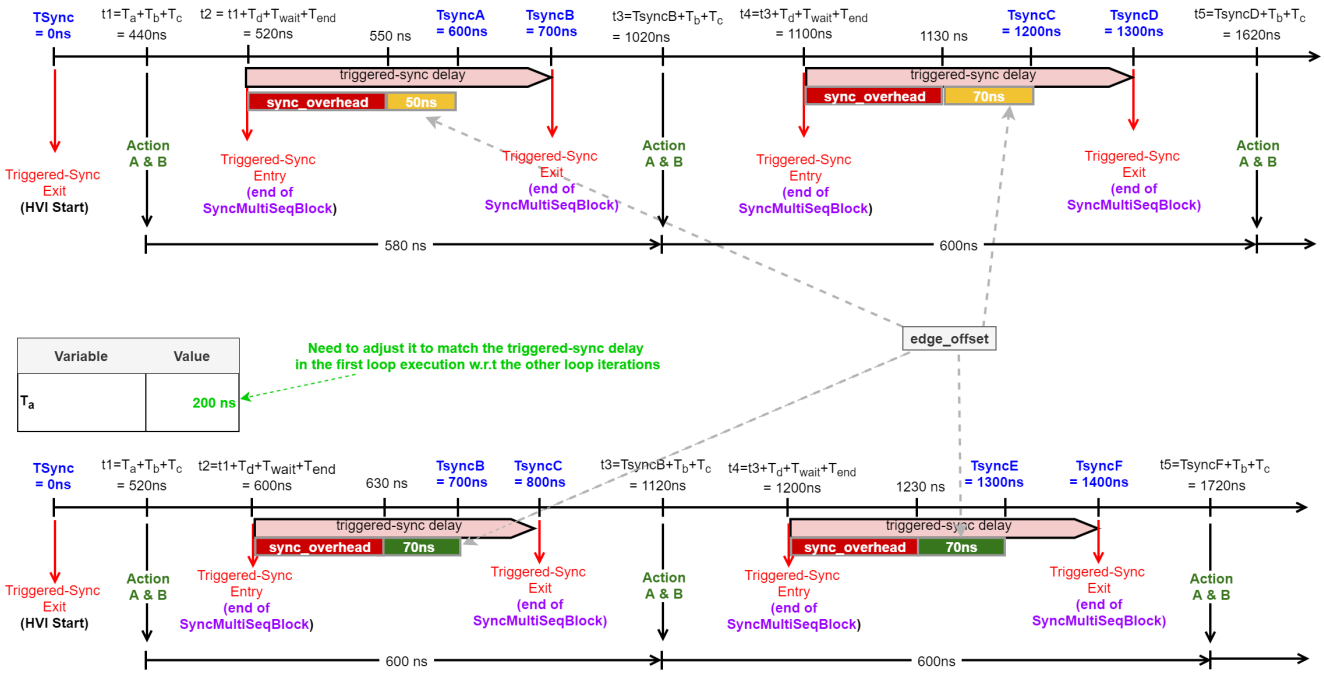
The following diagram shows an example with a simple sequence where the triggered-sync points have been marked in red. There are two triggered-sync points, the HVI start which is always present, and a second one at the end of the Sync multi-sequence block. The second sync point is required because there 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 statement
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 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 constant

The following diagrams shows the execution timeline for the first 3 iterations of the sequence shown in the previous diagram:



Timing management with triggered-sync as a result of a Wait-for-event statement

In the case that the re-synchronization process is taking place because of a Wait-for-event statement inside a Sync multi-sequence block statement, there are two possible scenarios:

- **The event is in-sync with the Sync pulse, that is, it is happening at a constant offset compared to the Sync pulse.**

  In this scenario, the previous example applies to this case. You just need to adjust the `Triggered-Sync Entry` to the event arrival time and the result will be similar.

- **The event is out-of-sync with the Sync pulse.**

  In this scenario, the same time from the execution of the actions from one iteration to the other cannot be guaranteed. Depending on the time of the event arrival, the triggered-sync latency might change in number of cycles from iteration to iteration. In this case, all of the HVI sequence statements following the Wait statement will execute with a jitter equal to one Sync period.

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 Register-Sharing Statement
- Sync Multi-Sequence Block Statement
- Sync While Statement

HVI Start

This is the time 0 for the HVI execution. It always matches the rising edge of the Sync Pulse.

HVI start basic timing value:

Parameter	Time (cycles)
End-Latency	3

Sync Register-Sharing Statement

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)	Fetch time (Primary Module, cycles)
`5 + Propagation_delay_cycles` (2)	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) *The Propagation delay depends on the structure of your system and the Sync period. See the section* ***Timing with triggered-sync points*** *in Sync Statement Timing .*

Latency values for Sync register-sharing statement:

Parameter	Description	Time (cycles)
Start-Latency	Minimum start-delay for statement	0
End-Latency	Minimum start-delay for the next statement	1
Fixed-Duration	-	`5 + Propagation_delay_cycles` (1)

(1) *The Propagation delay depends on the structure of your system and the Sync period. See the section* ***Timing with triggered-sync points*** *in Sync Statement Timing .*

Sync Multi-Sequence Block Statement

Timing value for Sync multi-sequence blocks:

Execution time (cycles) (1)	Fetch time (Primary Module, cycles)
$sum_{for\_all\_internal\_statements}$ (Start-Delay) + $sum_{for\_all\_internal\_flow\_control\_statements}$ (Duration) (2)	N/A

(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.*

Latency values for Sync Multi-Sequence Blocks:

Parameter	Description		Time(cycles)
Start-Latency	Minimum start-delay for statement		0
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 (3) Minimum Duration	$End\text{-}Latency_{Last\text{-}statement\text{-}of\text{-}longest\text{-}branch}$ (1)
		timed-sync (3) Fixed Duration	1
		triggered-sync (3)	1
Fixed-Duration			$[max_{for\_all\_Branches}$ [Branch-Duration] - 1] (2) , where Branch-duration is calculated as follows: $sum_{for\_all\_internal\_statements}$ (Start-Delay) + $sum_{for\_all\_internal\_flow\_control\_statements}$ (Duration) + $End\text{-}Latency_{Last\text{-}statement}$

(1) *If the sequence is empty, the value is constant 1.*

(2) *If all branches are empty, then the duration is 0.*

(3) *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 Statement

Timing value for Sync while statement:

Execution time (cycles) (1)	Fetch time (Primary Module, cycles)
#Iterations * [$sum_{for\_all\_internal\_statements}$ (Start-Delay) + $sum_{for\_all\_internal\_flow\_control\_statements}$ (Duration)]	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.*

Latency values for Sync while statement

Parameter	Description		Time (cycles)
Start-Latency	Minimum start-delay for statement		5 + #Register_Conditions
Entry/Iteration latency	Minimum start-delay for first statement inside the while loop	Minimum Duration	14 + #Register_Conditions + Propagation_delay_cycles (2) + Instrument_SyncResources_Latency(1) + $End\text{-}Latency_{Last\text{-}statement}$
		Fixed Duration	14 + #Register_Conditions + Propagation_delay_cycles(2) + Instrument_SyncResources_Latency (1)
End-Latency	Minimum start-delay for next statement outside the while loop	Minimum Duration	14 + #Register_Conditions + Propagation_delay_cycles(2) + Instrument_SyncResources_Latency(1) + $End\text{-}Latency_{Last\text{-}statement}$
		Fixed Duration	14 + #Register_Conditions + Propagation_delay_cycles (2) + Instrument_SyncResources_Latency(1)
Fixed-Duration	-		[$sum_{for\_all\_internal\_statements}$ (Start-Delay) + $sum_{for\_all\_internal\_flow\_control\_statements}$ (Duration) + $End\text{-}Latency_{Last\text{-}statement}$](3)

(1) `Instrument_SyncResources_Latency` *is an instrument specific value. For more information see the instrument documentation.*

(2) *The Propagation delay depends on the structure of your system and the Sync period. See the section* ***Timing with triggered-sync points*** *in Sync Statement Timing .*

(3) *If the branch is empty, then the duration is equal to:* `8 + #Register_Conditions`

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)	Fetch time (Primary Module, cycles)
RegisterValue	1

Local Wait-for-time statement latency values:

Parameter	Time (cycles)
Start-Latency	1
End-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.*

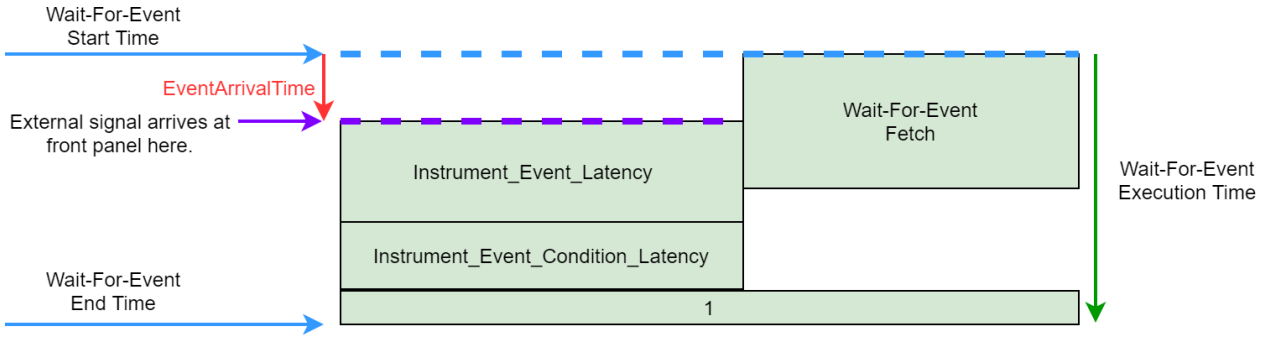
NOTE The *Event_Arrival_Time* can be a negative value if the event enters the module before the Wait-For-Event instruction Start Time. A number of scenarios are shown in the diagrams below.

Local Wait-for-event latency values:

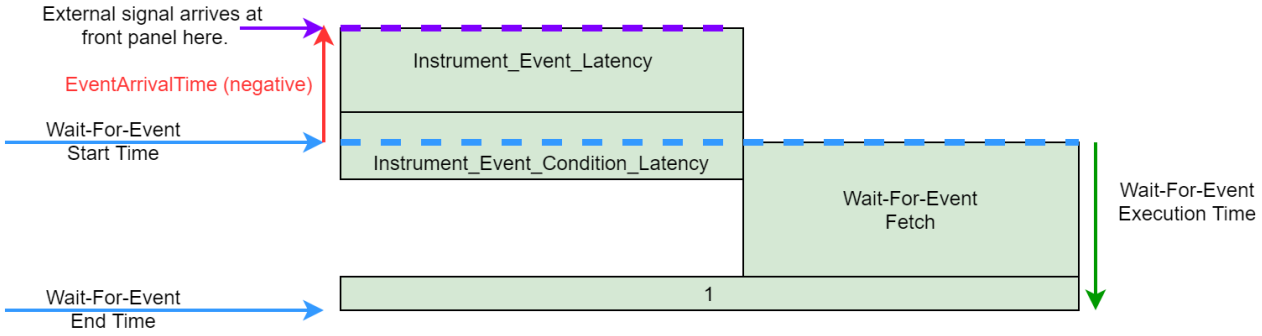
Parameter	Time (cycles)
Start-Latency	0
End-Latency	1

The following diagrams shows a number of 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:

Parameter	Execution time (cycles)	Fetch time (Primary Module, cycles)
Delay	Delay Specified	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)	Fetch time (Primary Module, cycles)
$sum_{for\_all\_internal\_statements}$ (Start-Delay) + $sum_{for\_all\_internal\_flow\_control\_statements}$ (Duration) (3)	3 + #Register_Conditions

(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* $Entry\text{-}Latency_{branch} - 1$.

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 #		`3 + (6 + #Register_Conditions) * Index`$_{branch}$ where Index$_{branch}$ is the branch index starting the count from 0: • Index$_{IF\text{-}branch}$ = 0 • Index$_{ith_ELSEIF\text{-}branch}$ = i • Index$_{ELSE\text{-}branch}$ = #if&elseif_Branches
End-Latency	Contributes to the minimum-possible start-delay of the next statement outside the if statement	Matching Branches *disabled*	`3 + max`$_{for\_all\_Branches}$ `[End-Latency`$_{Last\text{-}statement}$ `]` **(1)**
		Matching Branches *enabled*	`3 + End-Latency`$_{Last\text{-}statement\text{-}of\text{-}longest\text{-}branch}$ **(2)** Where longest branch means the branch with longer execution time.
		Fixed-Duration	`1`
Fixed-Duration			`2 + m ax`$_{for\_all\_Branches}$ `[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)**

(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

Execution time (cycles) (1)	Fetch time (Primary Module, cycles)
#Iterations * [$sum_{for\_all\_internal\_statements}$ (Start-Delay) + $sum_{for\_all\_internal\_flow\_control\_statements}$ (Duration)]	2 + #Register_Conditions

(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 + $End\text{-}Latency_{Last\text{-}statement}$
		Fixed Duration	8 + #Register_Conditions
End-Latency	Minimum start-delay for the next instruction outside the while loop	Minimum Duration	8 + #Register_Conditions + $End\text{-}Latency_{Last\text{-}statement}$
		Fixed Duration	8 + #Register_Conditions
Fixed-Duration			[$sum_{for\_all\_internal\_statements}$ (Start-Delay) + $sum_{for\_all\_internal\_flow\_control\_statements}$ (Duration) + $End\text{-}Latency_{Last\text{-}statement}$] (1)

(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
- 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 Statement 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	✓					
Add	✓					
Assign	✓					
Fpga Array-Read	✓					
Fpga Array-Write	✓					
Fpga Register-Read	✓					
Fpga Register-Write	✓					
Subtract	✓					
TriggerWrite	✓					

Instrument-Specific Local Instruction Statement Timing Values

See the instrument-specific documentation for information on the HVI engine clock frequency and instrument-specific instruction timing information.

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.

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.