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
PrintedInAddress_Line1
PrintedInAddress_Line2
PrintedInAddress_Line3

Edition

Edition 1.14, February, 2021
PrintedInCountry

Regulatory Compliance

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

Warranty

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

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

Technology Licenses

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

U.S. Government Rights

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

or DFAR 227.7103-5 (c), as applicable in any technical data.

Safety Notices

CAUTION

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

WARNING

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

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

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

WARNING

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

The types of product users are:

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

WARNING

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

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

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

Operators of this product must be protected from electric shock at all times. The responsible body must ensure that operators are prevented access and/or insulated from every connection point. In some cases, connections must be exposed to potential human contact. Product

operators in these circumstances must be trained to protect themselves from the risk of electric shock. If the circuit is capable of operating at or above 1000V, no conductive part of the circuit may be exposed.

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

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

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

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

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

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

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

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

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

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

Instrumentation and accessories shall not be connected to humans.

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

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

WARNING

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

PRODUCT MARKINGS:

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

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

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

South Korean Class A EMC Declaration. This equipment is Class A suitable for professional use and is for use in electromagnetic environments outside of the home. A 급 기기 (업무용 방송통신 기자재) 이 기기는 업무용 (A 급) 전자파적 합기기로서 판 매자 또는 사용자는 이 점을 주 의하시기 바라 며 , 가정외의 지역에서 사 용하는 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 previous version, 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.

About Keysight PathWave Test Sync Executive

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, an IP block. or processor, integrated into the instrument. The code executes on these Engines in parallel, across multiple instruments. The new user-defined hardware operation of the 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 with HVI support. The Keysight M3xxxA family of PXI instruments is one product family with HVI support. This user manual includes code examples from the SD1/M3xxxA API. These snippets complement the code examples that explain functionality of the HVI 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. The HVI API 2020 update 1 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 2020 Update 1.0\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 2020 Update 1.0\api\dotNet\Help
```

API Use Model: HVI API versus 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, which enable an 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:

For HVI instrument-specific definitions of the Keysight M3xxxA PXI product family, see:

SD1 3.x Software for M320xA / M330xA Arbitrary Waveform Generators User's Guide.

and

SD1 3.x Software for M310xA / M330xA Digitizers User's Guide.

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:

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

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

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, 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 you use 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 there are limitations in terms of 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 and located online at the KS2201A Programming Examples page.

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

Instruments

For information about compatibility, software and firmware versions requirements for specific PathWave Test Sync Executive releases, and firmware update procedures, see your instrument documentation.

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

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.

1: Install Python 3.7.x, 64-bit

PathWave Test Sync Executive requires Python 64-bit version 3.7.x

1. Download the Python installer from the Python web site: python.org .
2. Run the installer.

    a. Add Python 3.7.x to the PATH system Variable. To do this, ensure the check-box **Add python 3.7 to PATH** is checked. This is shown in the following screenshot:

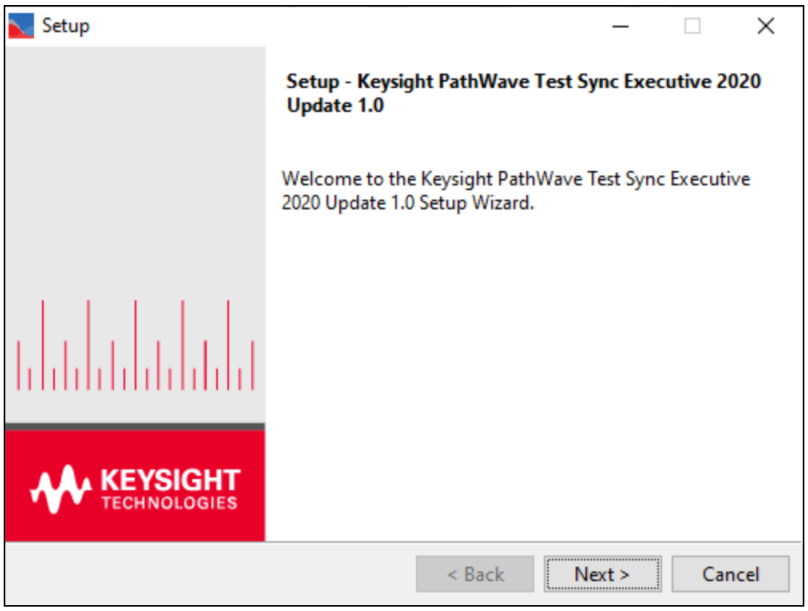
2: Install PathWave Test Sync Executive

Use the following procedure to install PathWave Test Sync Executive:

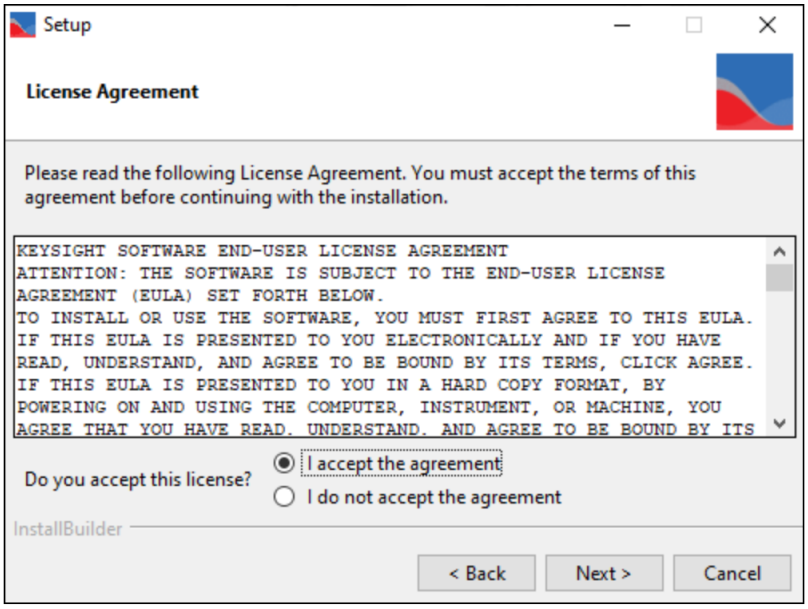
NOTE **You must install Python 3.7.x 64-bit before installing PathWave Test Sync Executive.**

Execute the installer file:

The **Setup** screen is shown:



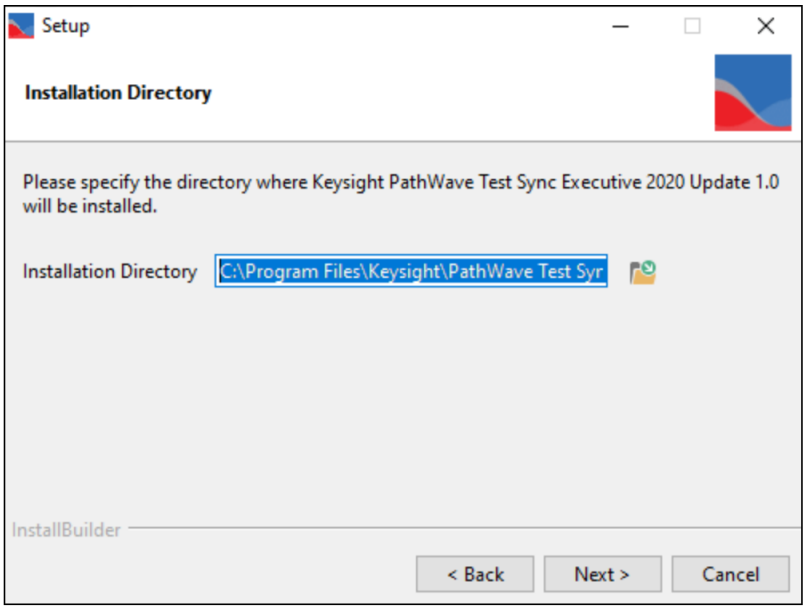
The next screen is the **License Agreement** screen. You must accept the license to continue:



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

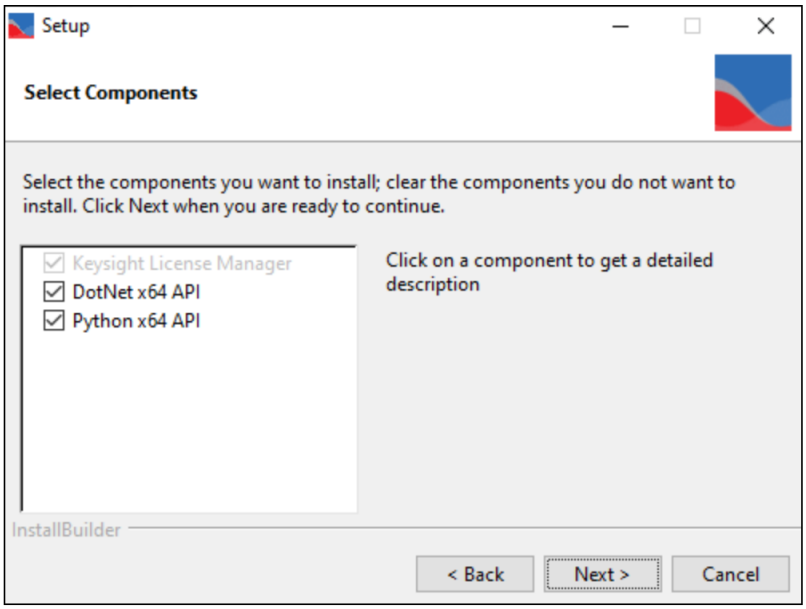
By default, PathWave Test Sync Executive is installed to:

```
C:\Program Files\Keysight\PathWave Test Sync Executive 2020
```

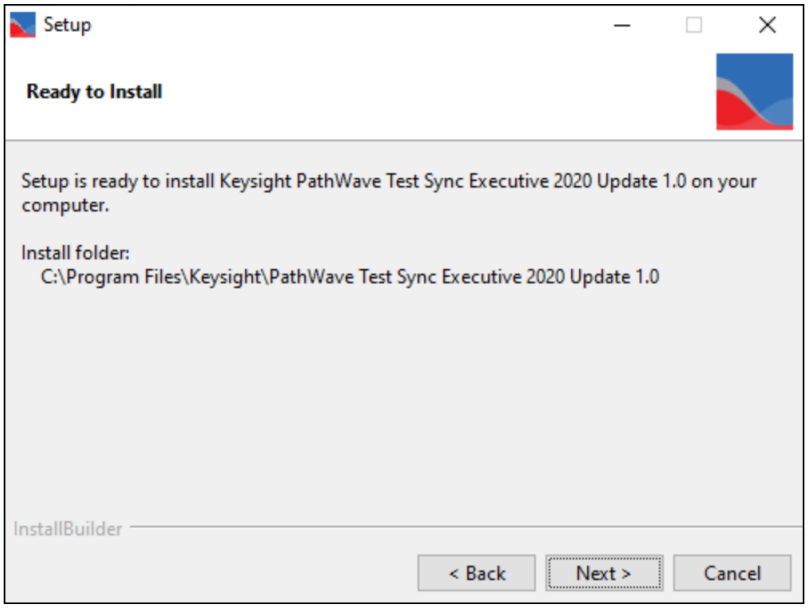


The **Select Components** screen enables you to select the components you want to install.

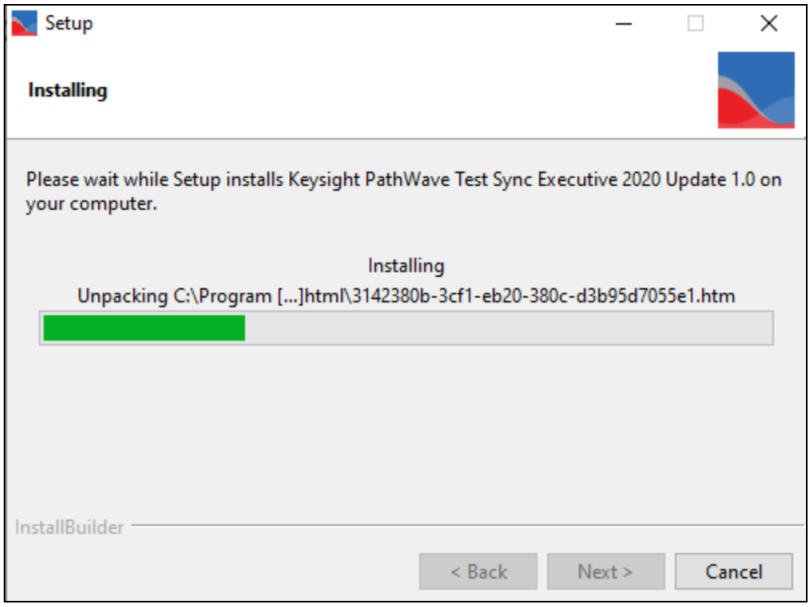
Required components are selected by default and you cannot de-select them.



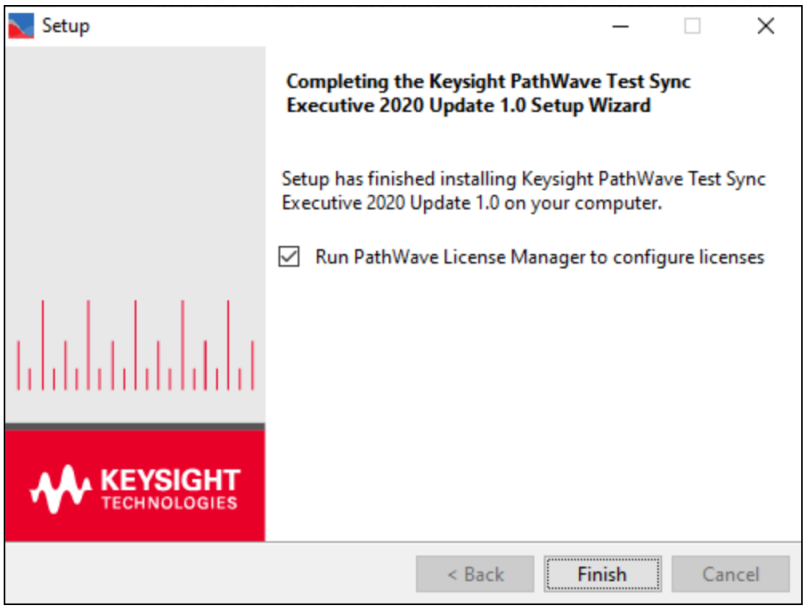
When you have selected the components, the next screen is **Ready to Install**. Select **Next** to install PathWave Test Sync Executive.



The Installer first installs the Keysight 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.



Install Additional Components

To use PathWave Test Sync Executive, you require both hardware and software. Ensure you have all of the following components and they are up to date:

1. Keysight IO Libraries.
2. Keysight SD1 Drivers, Libraries, and Software Front Panel.
3. Keysight Instrument FPGA Firmware.
4. Keysight Chassis Family Driver.
5. Keysight Chassis Driver and Firmware.

Install Keysight IO Libraries

Install the IO Libraries, these are available at Keysight IO Libraries Suite.

Install Keysight SD1 Drivers, Libraries, and SD1 Software Front Panel

Install the SD1 Libraries and driver, both of these are available at Keysight SD1 Software. When you install the Keysight SD1 Driver, SD1 Software Front Panel (SFP) software is automatically installed.

NOTE Ensure you check the SD1 driver release notes, so that you install a SD1 driver that is compatible with the version of PathWave Test Sync Executive you have installed.

For detail about the specific versions you require for PathWave Test Sync Executive, see: Appendix A: Supported Instruments.

Update Keysight Instrument FPGA Firmware

You can update the FPGA firmware of your PXI instruments from the Hardware Manager window of the SD1 SFP. For information about how to install SW and FPGA firmware for SD1 and M3xxxA Keysight instruments, see the documents:

- *Keysight M3xxxA Product Family Firmware Update Instructions*.
- *M3xxxA User Guide*.

These are available at Keysight PXI Products.

NOTE Ensure you check the M3xxxA firmware release notes, so that you install a driver that is compatible with the version of PathWave Test Sync Executive you have installed.

For detail about the specific versions you require for PathWave Test Sync Executive, see: Appendix A: Supported Instruments.

Install Keysight Chassis Family Driver

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

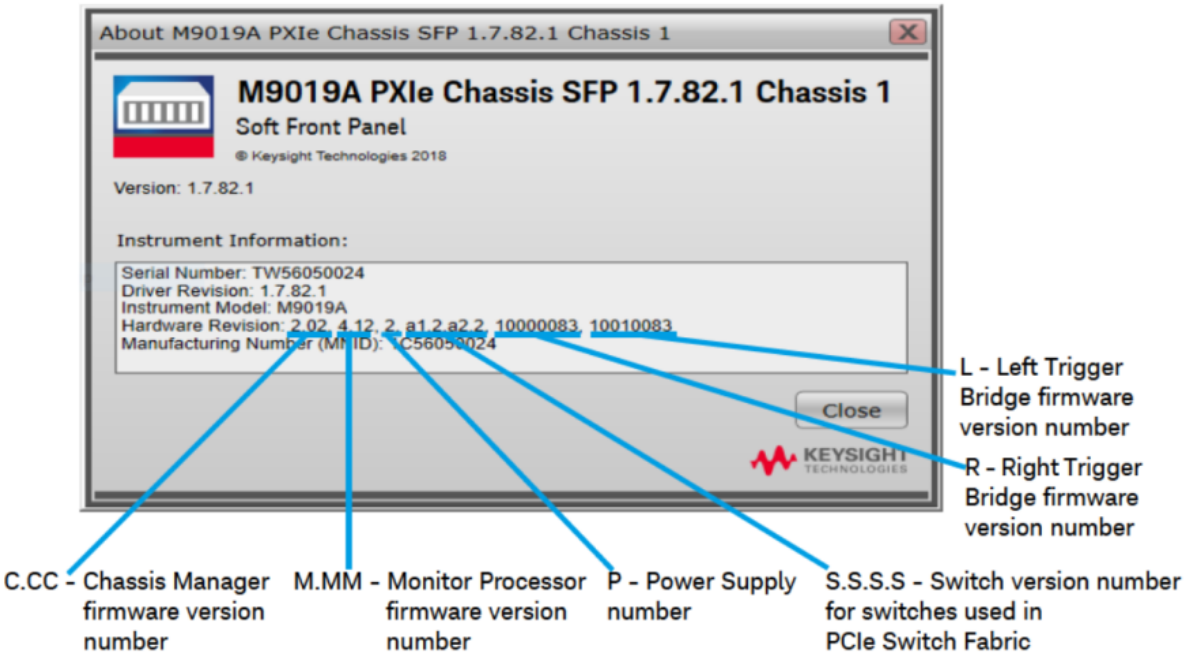
Update Keysight Chassis Firmware

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.

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 explains how to install PathWave Test Sync Executive licenses. It contains the following sections:

- About PathWave Test Sync Executive Licenses.
- Installing Licenses with PathWave License Manager.

About PathWave Test Sync Executive Licenses

PathWave Test Sync Executive requires one license per chassis that is used in your HVI implementation. If you are using one chassis and want to use a second chassis, you are required to have purchased and installed at least two licenses to use the second chassis. All HVI instances running in the same process share the same licenses, but HVI instances running on different processes require different licenses.

For example:

Description	Licenses required
1 HVI instance using 1 chassis	1
2 HVI instances in the SAME process using 1 chassis each	1
3 HVI instances in the SAME process using 2 chassis each	2
2 HVI instances in 2 DIFFERENT processes using 1 chassis each	2
3 HVI instances in 3 DIFFERENT processes using 2 chassis each	6

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/Networking, perpetual and 6, 12, 24, and 36 months, subscription.
- Transportable, perpetual and 6, 12, 24, and 36 months, subscriptionn.

Trial licenses:

- 90 days Node-locked.

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, so you don't have to keep buying new licenses.

For example, if 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.

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.

More documentation is available at http://www.keysight.com/find/licensingdoc.

Installing Licenses with PathWave License Manager

You can install licenses from the PathWave License Manager 2.3. 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.

Troubleshooting

By default, PLM 2.3 saves its log files 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 of 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 so they can operate tightly orchestrated together with other instruments, like one instrument.

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

- Execute real-time sequences of operations with full time determinism.
- Precisely synchronize instruments operations.
- Fast exchange information and decisions among instruments in real-time by hardware.

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 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. For example, 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 *2020 Update 1* supports the Python and C# languages.

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

- Class Names are in camel case, 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 underscores between words in class Names.
- The first letter of methods or functions is capitalized.

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

For example, the following code is 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 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 2020 Update 1.0\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 2020 Update 1.0\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.

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
- HVI 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
- Triggers.
- Actions.
- Events.

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.

HVI sequencer

You use this 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 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 the HVI instance Hvi.

Hvi

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

Execution Flow

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 a *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 controls the instrument. The HVI Sequences are compiled into the binary executables that the HVI Engine executes.

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 following resources:

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

Actions

HVI actions are digital electronic pulsed or level signals that are sent from the HVI engine to control instrument operations. Typically, actions are associated to instrument-specific operations.

For example, in a digitizer module, a `StartAcquisition` action sends a digital pulse to start an acquisition operation.

Events

HVI events are digital electronic pulsed or level signals input to the HVI engine that represent state, or an event in the instrument. HVI events are associated with instrument-specific functions. For example, they can be used to trigger instrument operations or, control the execution in an HVI application.

For example, a `WaitForEvent` statement pauses execution until the Event occurs.

Triggers

Triggers are electronic logic signals shared between instruments, you can use these to initiate operations, communicate states, or communicate information among instruments.

Clock signals

You can use clock signals to synchronize instruments.

HVI Registers

HVI Registers are very fast access physical memories that are located in the HVI Engines in instruments. HVI Registers can be used for storing parameters for operations and modified during Sequence execution. You use them in the same way as Variables in a programming language. The number and size of Registers is defined by each instrument.

FPGA Sandbox Registers and Memory-Maps

For the modules that contain an FPGA with a user-configurable sandbox, HVI can, if the configuration of the module allows it, access (read/write) the Registers and memory-mapped locations that you define in that sandbox.

To accomplish that, you must obtain the .k7z file for the FPGA sandbox that was generated by the PathWave FPGA application. This file contains all the necessary information to enable you to access the Registers and the memory-mapped locations by Name.

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

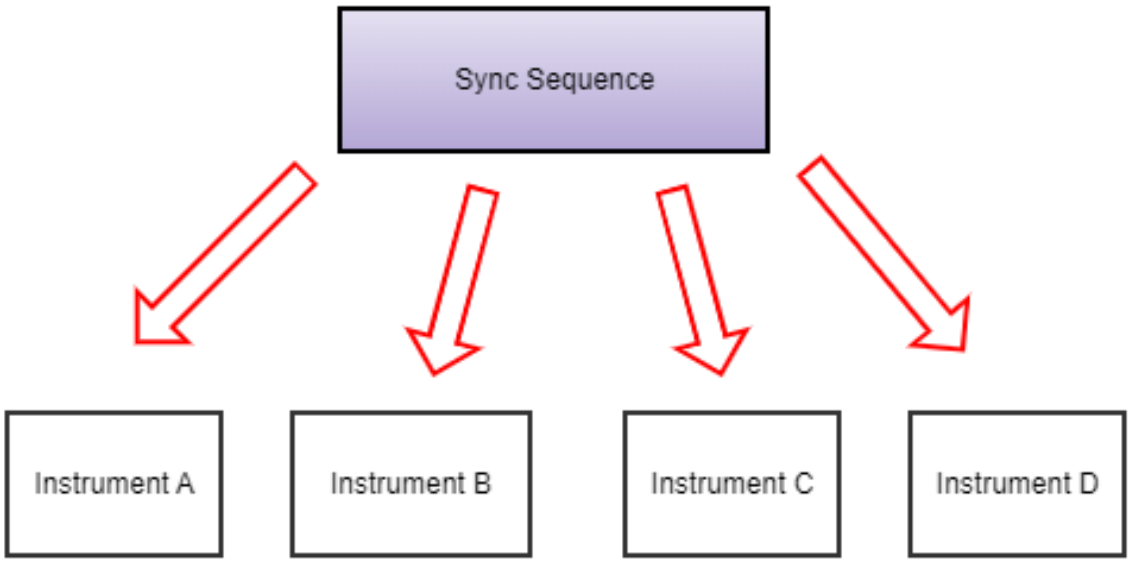
An HVI instance consists of HVI sequences, these are the foundations of HVI technology. An HVI sequence is an ordered list of HVI statements with associated timing information. A sequence is executed in a time-deterministic manner by the HVI hardware engine located within an instrument. An HVI instance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

There are two types of sequences:

- Sync sequence
- Local sequences

Sync Sequences

HVI sequences are organized in a hierarchy. A Sync sequence contains commands known as Sync statements that execute across multiple instruments:

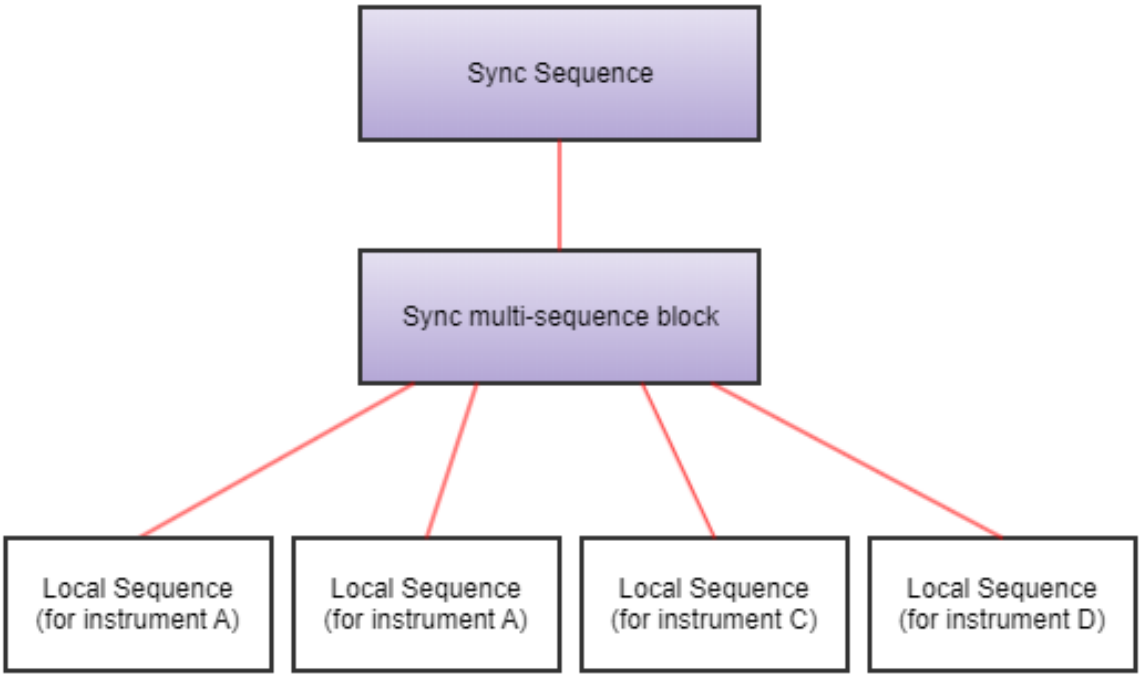


Local Sequences

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 general-purpose FPGA-level operations that can be executed by the HVI engines.

HVI statements are broadly divided into two groups:

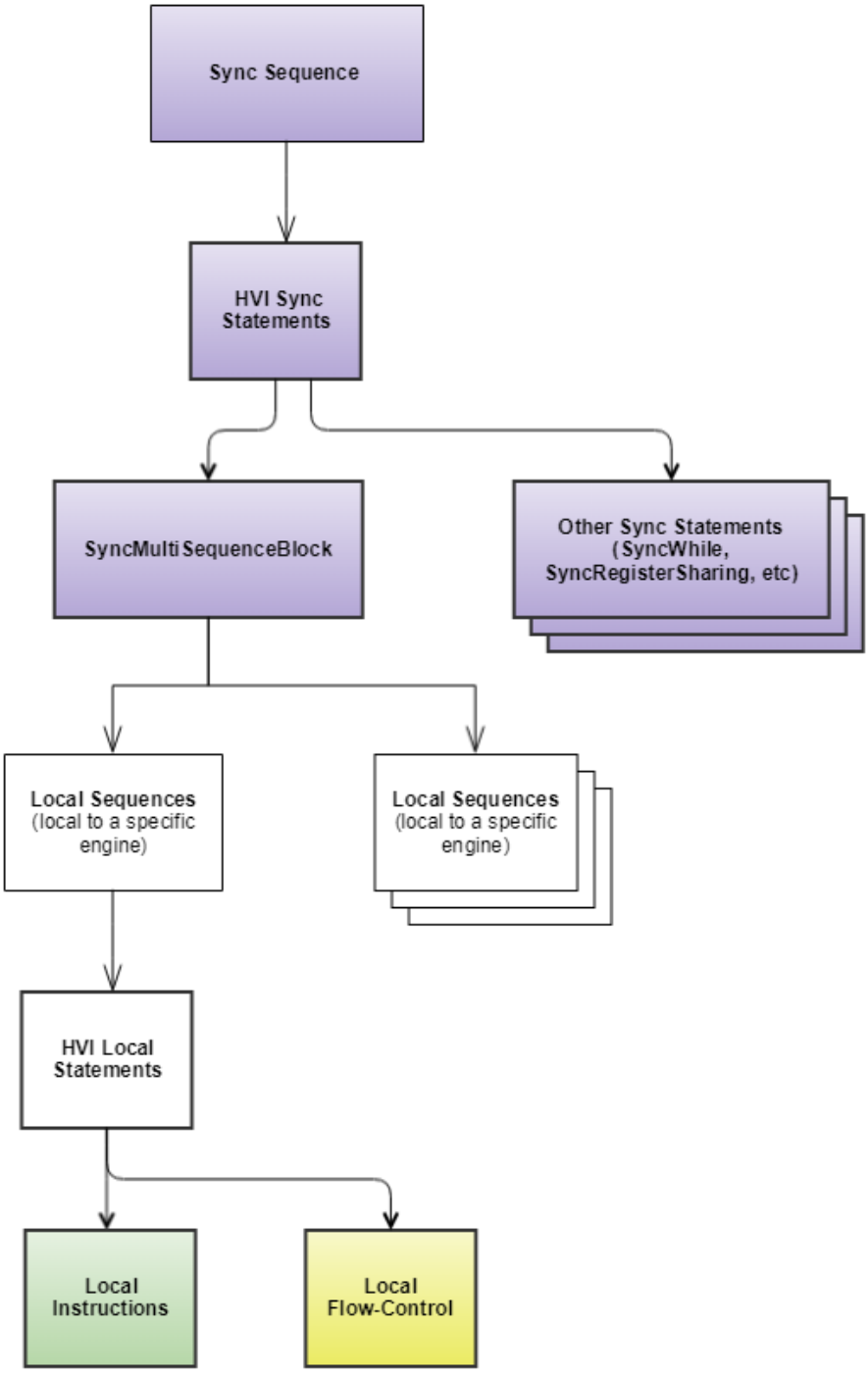
HVI Sync statements

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 Local statements

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.

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. For example: Sync While, Sync Register-sharing and Sync Multi-Sequence Block.

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

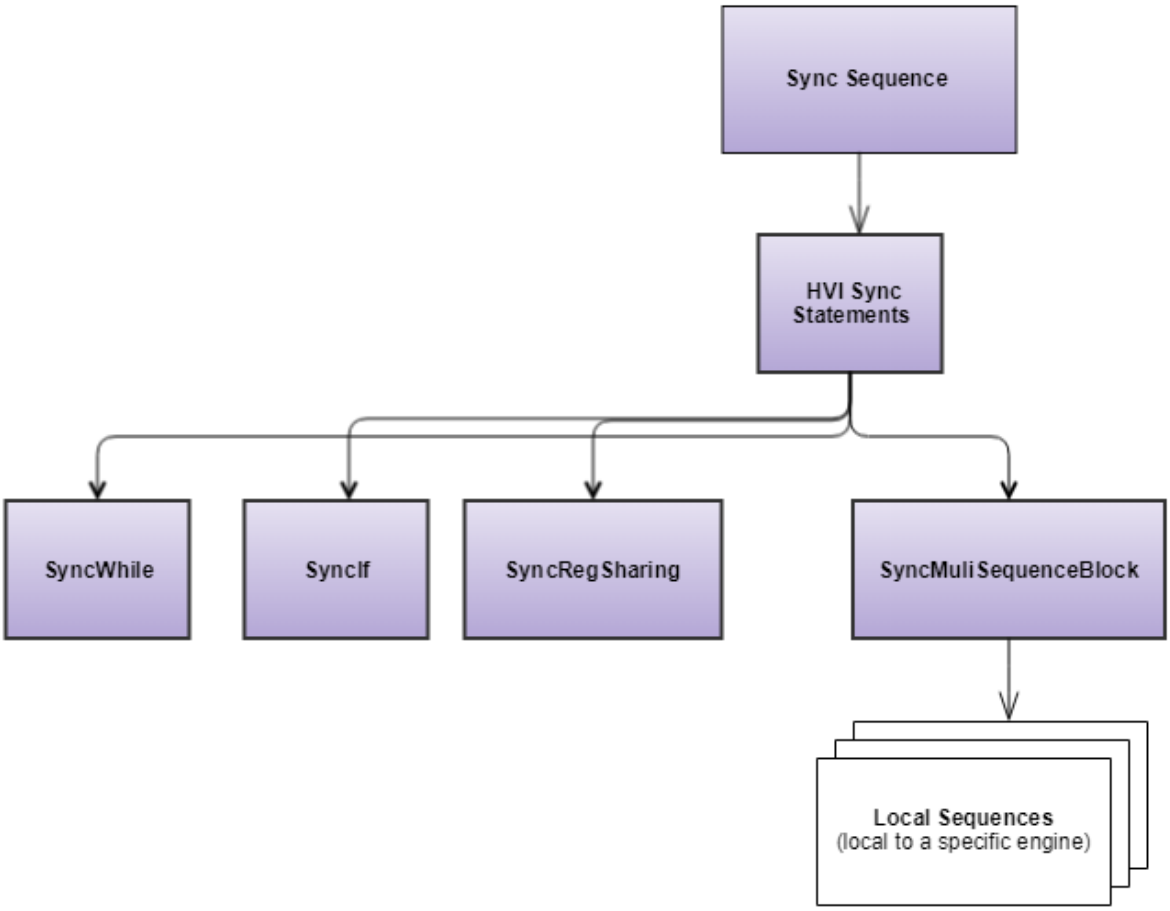
HVI Sync Statements are contained in a Sync sequence. HVI Sync statements execute across all instruments.

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

The following HVI Sync statements are available:

Type	Description
Sync While	Enables a While loop to execute synchronously on all engines.
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.
Sync Register-Sharing	Enables you to share data from a source Register to a destination Register in any other HVI Engine.

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



Sync While

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 continue the execution with the next Sync statement.

Sync Multi-Sequence Block

Sync multi-sequence blocks are a type of Sync statement that contain a set of Local sequences. Each Local sequence executes on an individual HVI engine within a specific instrument.

The Sync multi-sequence block enables you to run different sequences on each engine concurrently. It ensures that the execution of all 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.

Sync Register-Sharing

The Sync Register-sharing statement 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.

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

Instrument-Specific Instructions

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

These instructions can change instrument settings such as amplitude, 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 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 locally 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, such as loops and waits. 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.

Conditional flow-control statements:

Local If-Elseif-Else

Executes one of a set of possible Local sequences depending on the value of a defined condition.

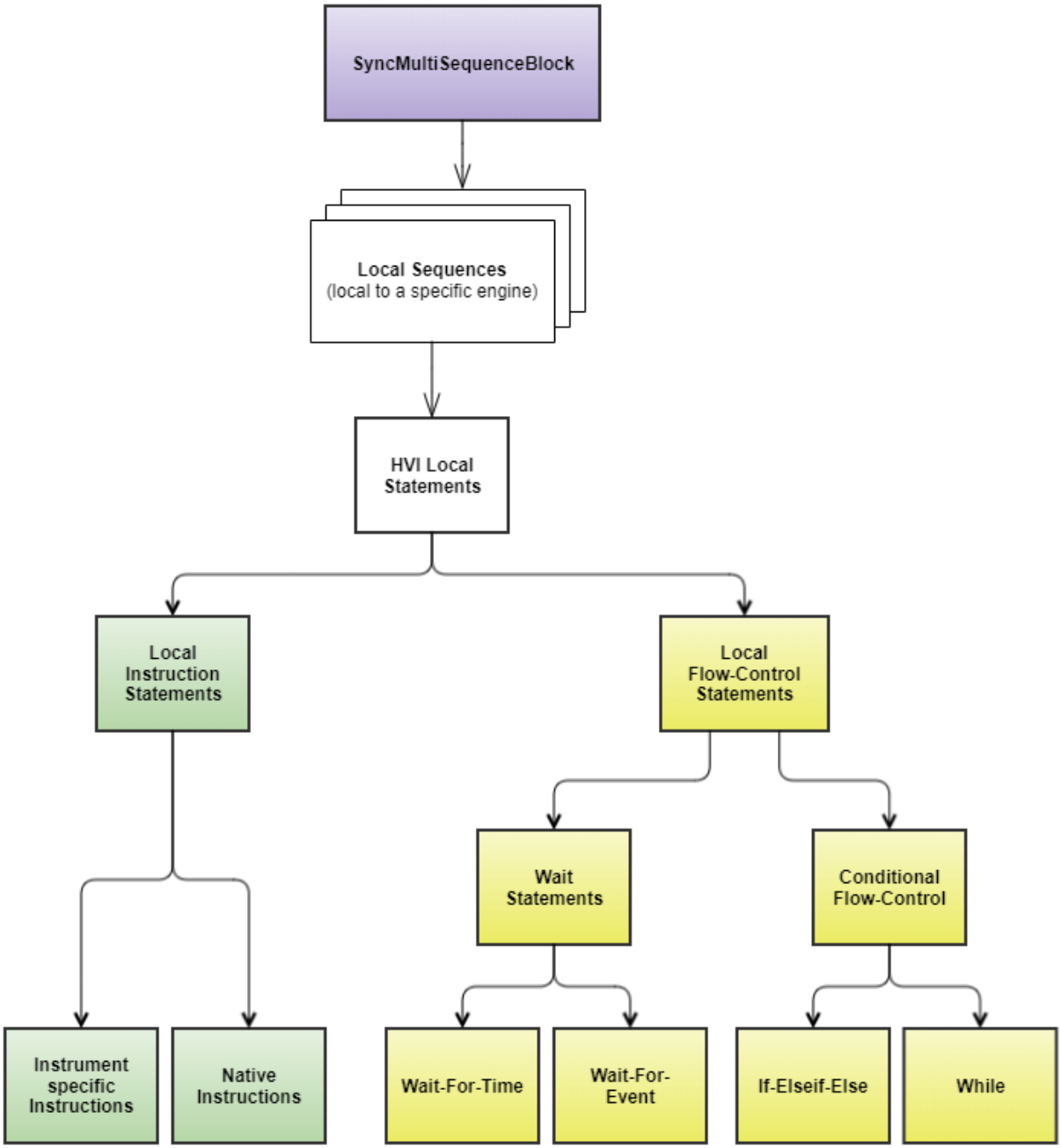
Local While

Executes while a condition is true.

Local Delay

Delays a sequence for a time you specify.

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



HVI Diagrams

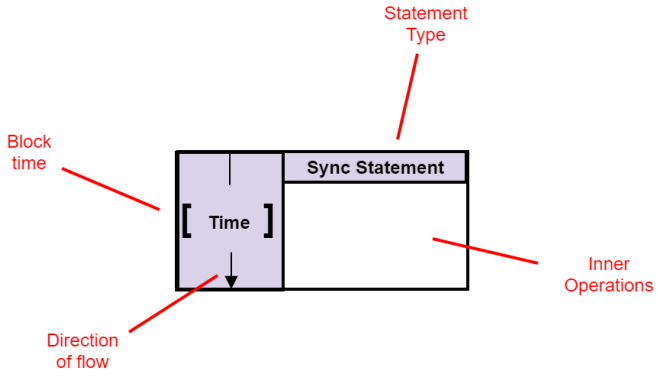
This section shows HVI diagrams. These are used to illustrate HVI sequences.

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

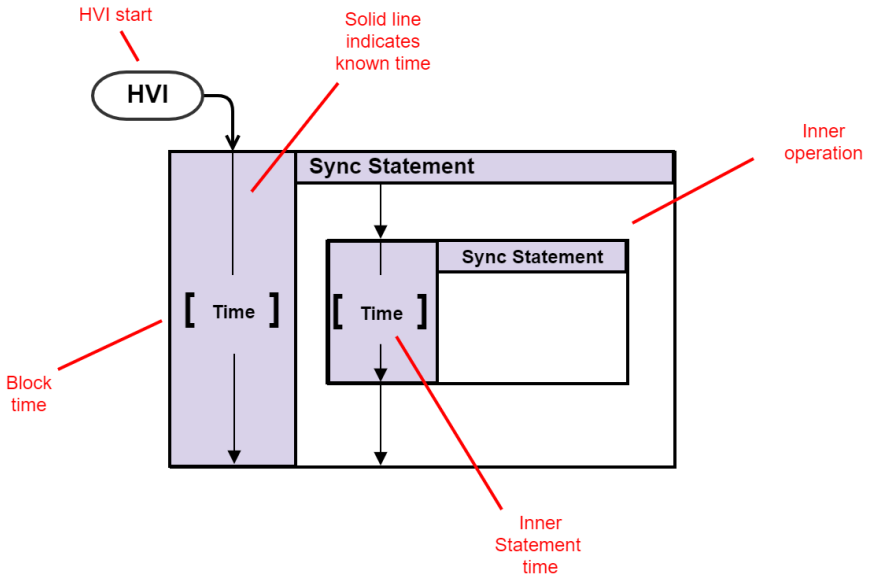
Sync Statements	
Local instructions	
Local Flow-control	

Statement diagram color code

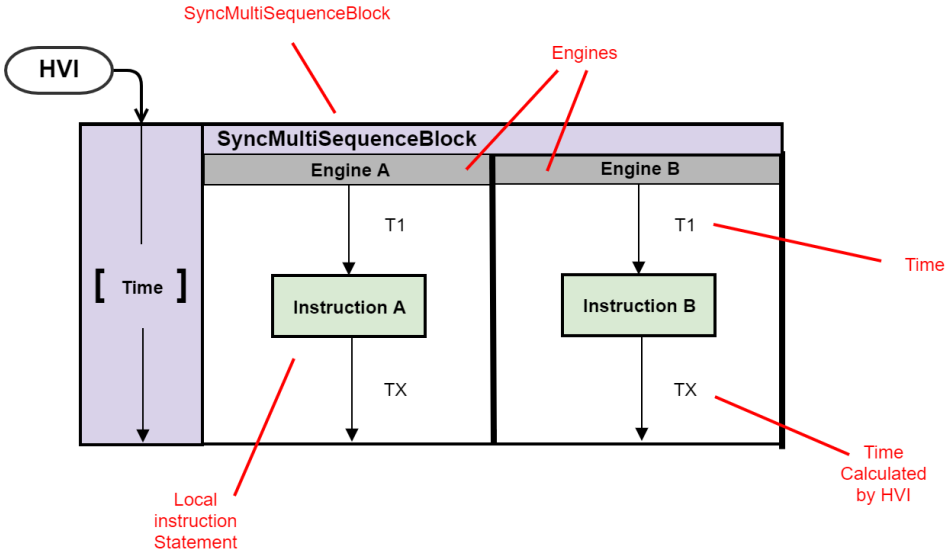
The following diagram shows a single Sync Statement with flow and time for the block:



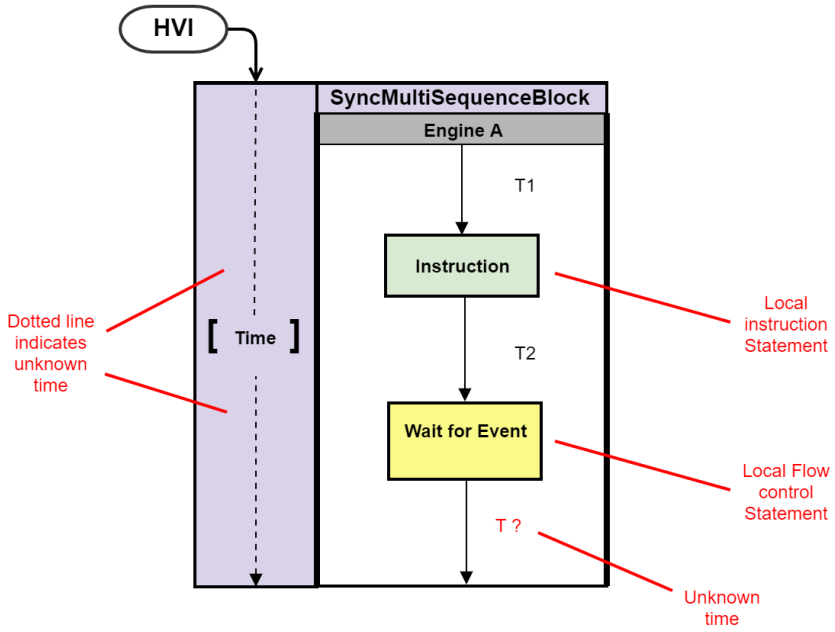
The diagrams can show nesting of Statements within Statements. For example, the following diagram shows a Sync Statement that is within another Sync Statement:



Local Sequences are placed within their 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:

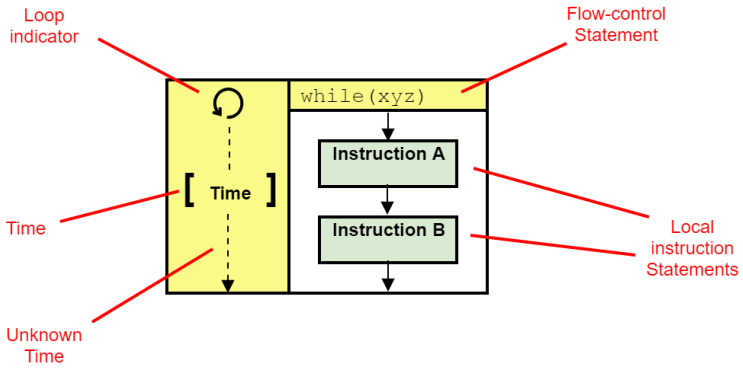


Dotted lines indicate that time is not known at compile time. This is often the case with flow-control statements. In this case the Wait-for-event statement shall not release until the event occurs. It is not known at compile time when this is, so the time cannot be calculated at compile time.

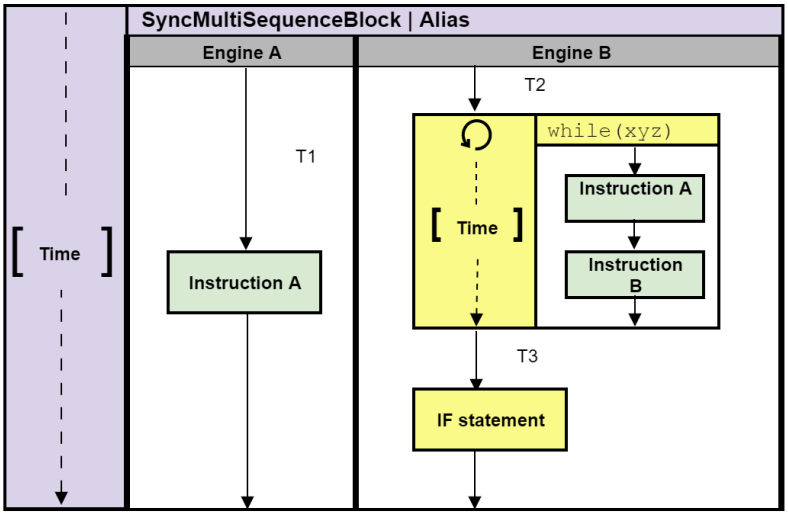


The following diagram shows a Local flow-control statement that encloses a pair of Local instruction statements. The color yellow indicates this is a Local flow-control statement.

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



T he 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 and you must calculate the timing between statements. The calculations required and parameters involved are described in detail in Chapter 7: 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 the actual execution. Depending on their characteristics, some statements can take several HVI engine cycles to complete the fetch before the actual execution can start.

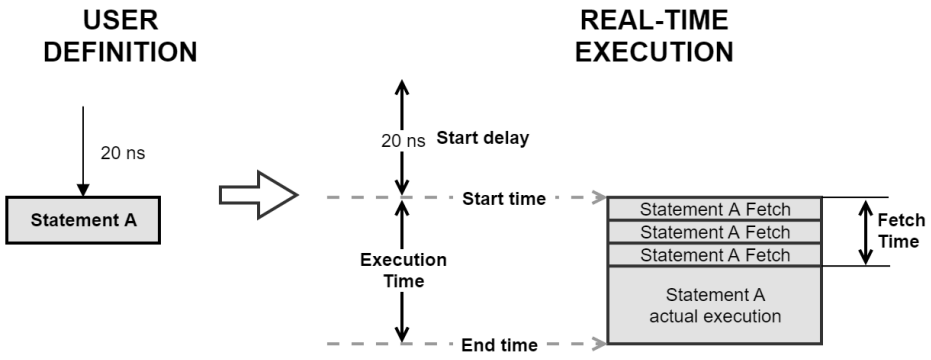
Execution time

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.

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 the start delay of statement B depends on the type of the previous statement A:

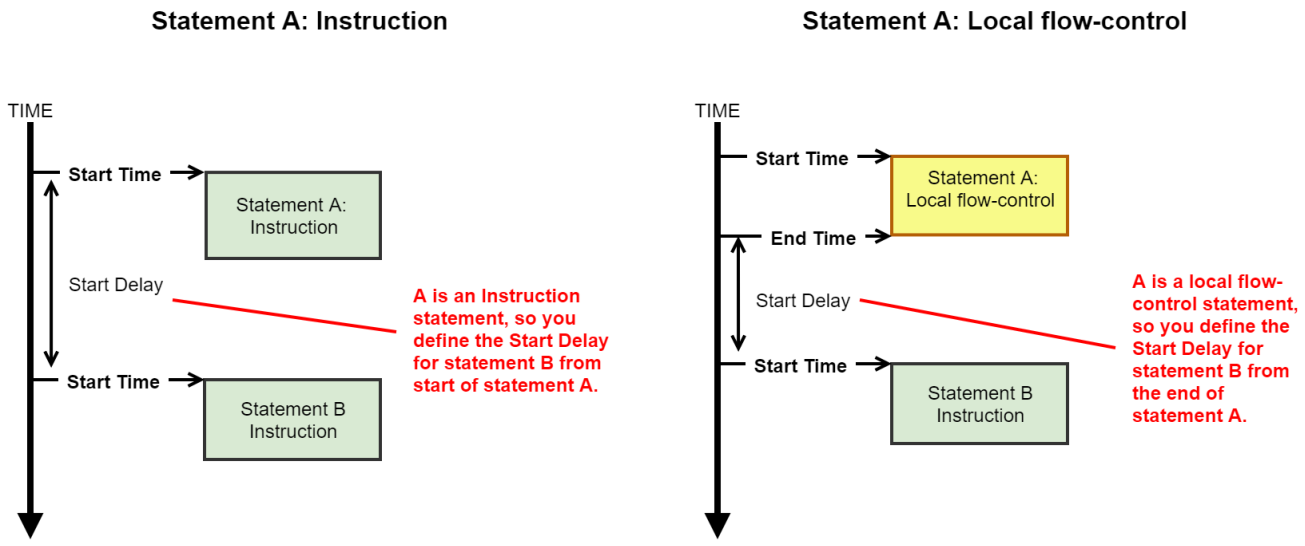
Instruction statements

If statement A is an **instruction** statement, the start delay of statement B is measured from, or starts at, 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 is measured from, or starts at, the **End time** of statement A.

The following diagram shows the different start delay definitions:



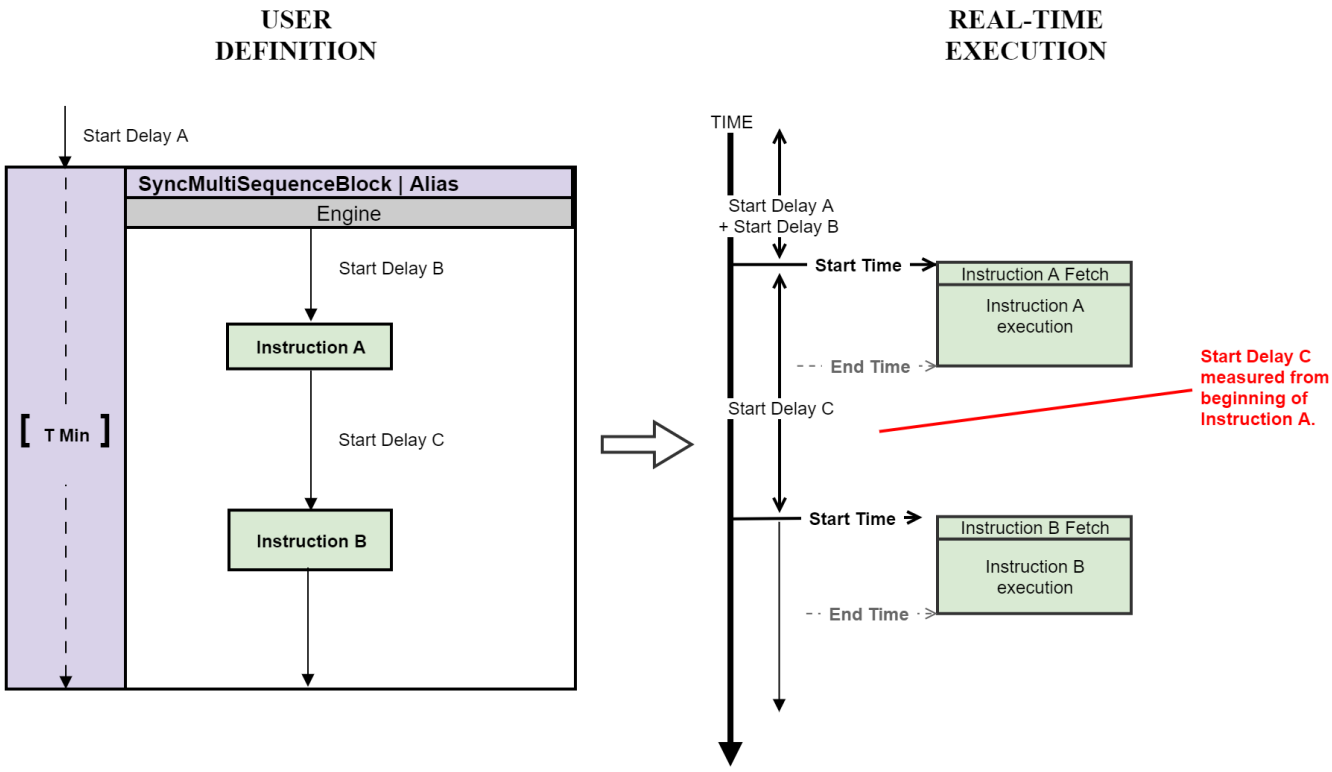
Local instruction timing

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 at least greater or equal to the fetch time of the previous instruction.

The following diagram shows two Local instructions and their timing:

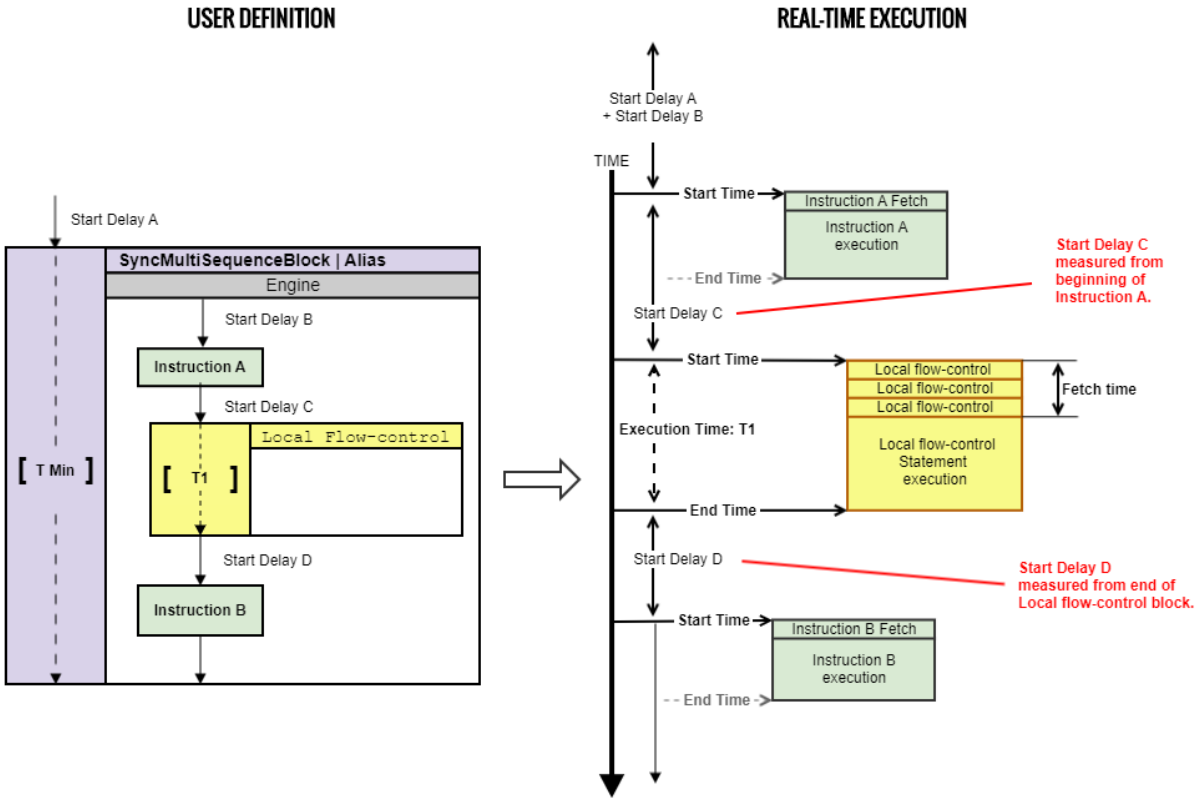


Local flow-control timing

For Local flow-control statements, the Start delay of the next statement is measured from the end of the previous Local flow-control statement. That's because the HVI engine is busy during the execution of the flow-control statement and no overlapping execution is possible between a flow-control statements and any other following statement. The following diagram shows the difference between measuring timing of instructions and Local flow-control statements.

For instructions, the Start delay of the next statement is measured from the start of the previous instruction, while for the Local flow-control statements, Start delay D is measured from the end of the previous flow-control block.

The execution time of local flow-control statements can be known at compile time, or 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:



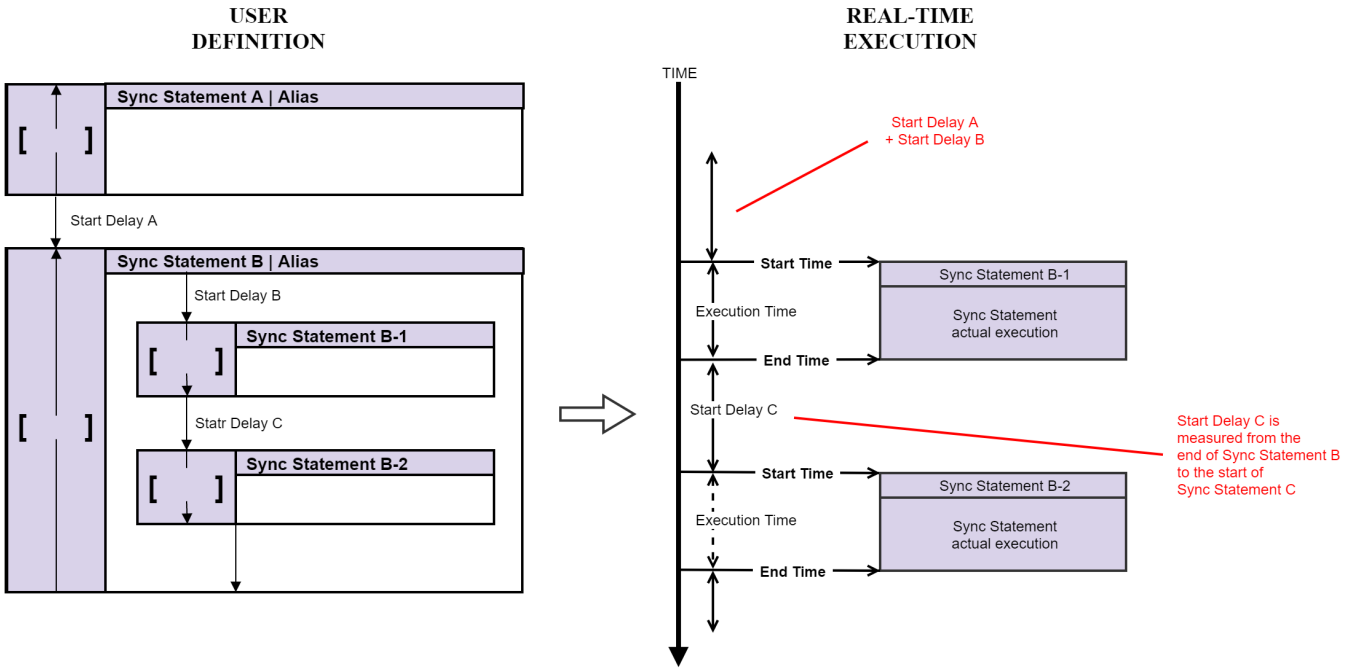
Sync statement timing

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

The following diagram shows the timing between Sync statements. The 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. If the execution time is unknown, it is represented by a dotted line.



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 also synchronized to end simultaneously. The individual sequences can take different execution time, 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.
- The end times of the sequences are set to match a 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.

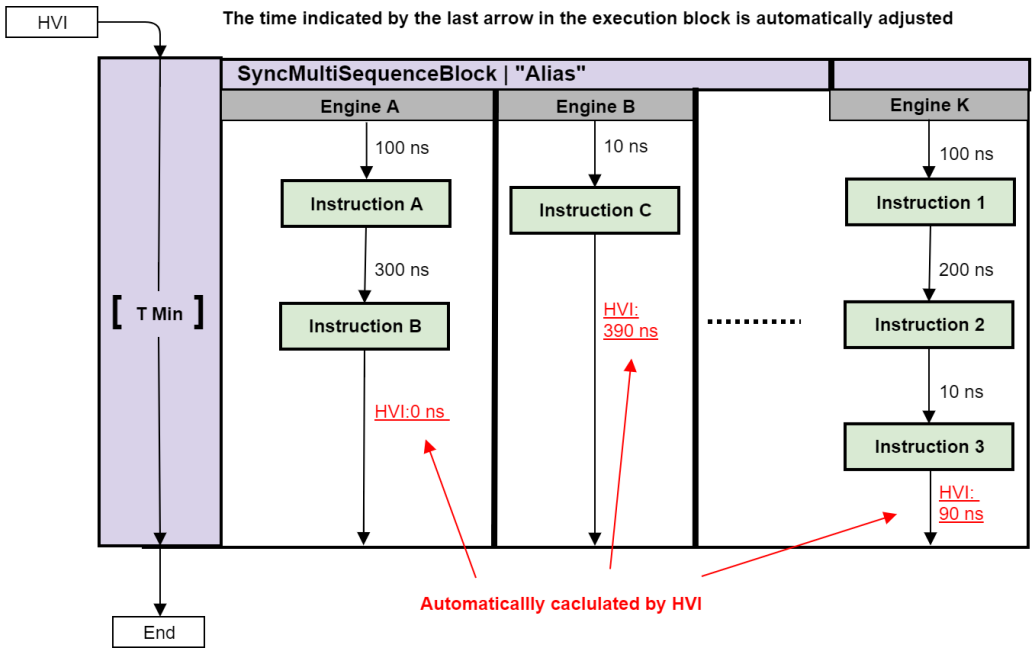
The time of the sequences set to match the longest sequence

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 of 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 and the different HVI Engines clocking constraints is 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:

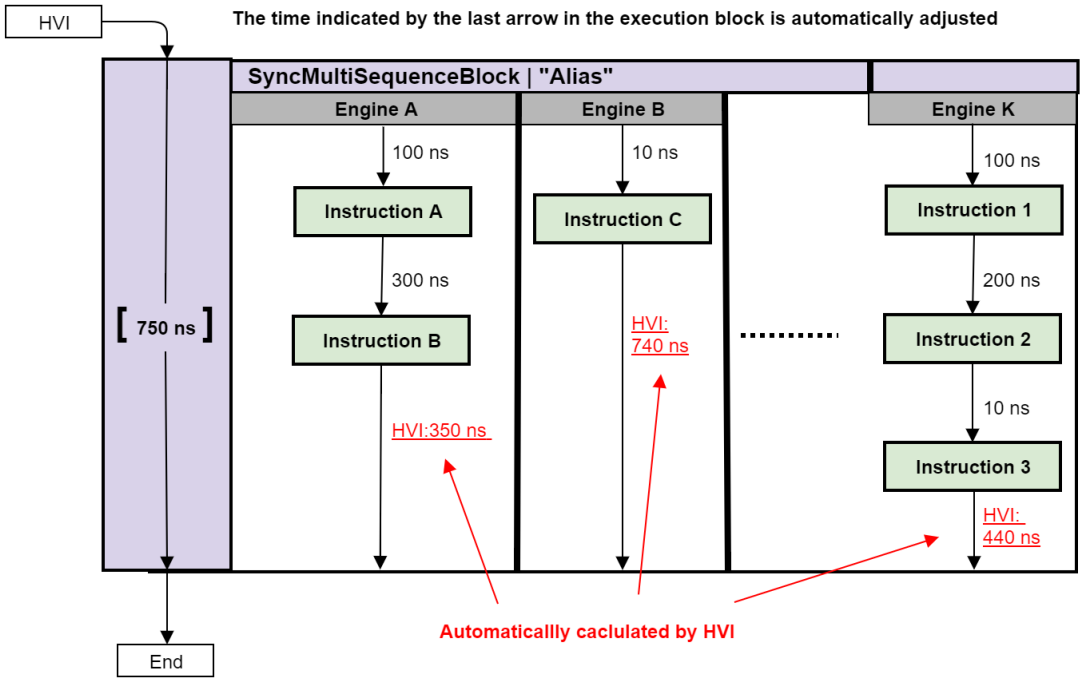


The times of the sequences set to match a specific execution time

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: 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 any instrument that supports 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 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 M9031A modules.
- Access to HVI memory resources in the FPGA user Sandbox.
- Real-time sequencing:
  - Synchronized flow-control, for example, While loops.
  - Synchronized multi-sequence block statements that provide access to local instructions and flow control.
  - Local flow-control, such as While loops and If statements.
  - Local Instructions and operations. These include HVI-native and instrument-specific instructions.

HVI Collections

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

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

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.

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. If a component is not registered with a collection, it cannot be used. To register components, add them to the collection of items of that function, for example, you must add triggers to a trigger collection.

When you are defining an HVI instance, you must define several resources and add them to the corresponding collections. You can then use them inside an HVI sequence. You cannot use 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:

You add new collection items by calling the `add()` method. This takes a Name as its first parameter and returns the new item. For example, the following code declares and returns a 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 refers to their order 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, but it has no insight of the parameters or attributes of its members. Definition and management of the instances within a collection are not managed in their own class, not the collection class. For instance, you manage an Engine with the `Engine` class, not the `EngineCollection` class. Once an instance is defined, it is then added to the collection using the methods above.

HVI API Error Management

Error handling in 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.

HVI API Organization

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

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

SystemDefinition

You first define the hardware and resources you have in the SystemDefinition class. You do this by adding each of the resources to the relevant collection. SystemDefinition contains classes for:

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

Sequencer

After defining the SystemDefinition, you define and program HVI Squences with the 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 in turn contains Instruction set, Register Scope, and Registers.

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 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 classes for:

- SyncSequenceRuntime.
- EngineRuntimeCollection.
- ScopesRuntimeCollection.

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:\ProgramFiles\Keysight\HVI\api\python\doc\keysight_pathwave_hvi.htm`

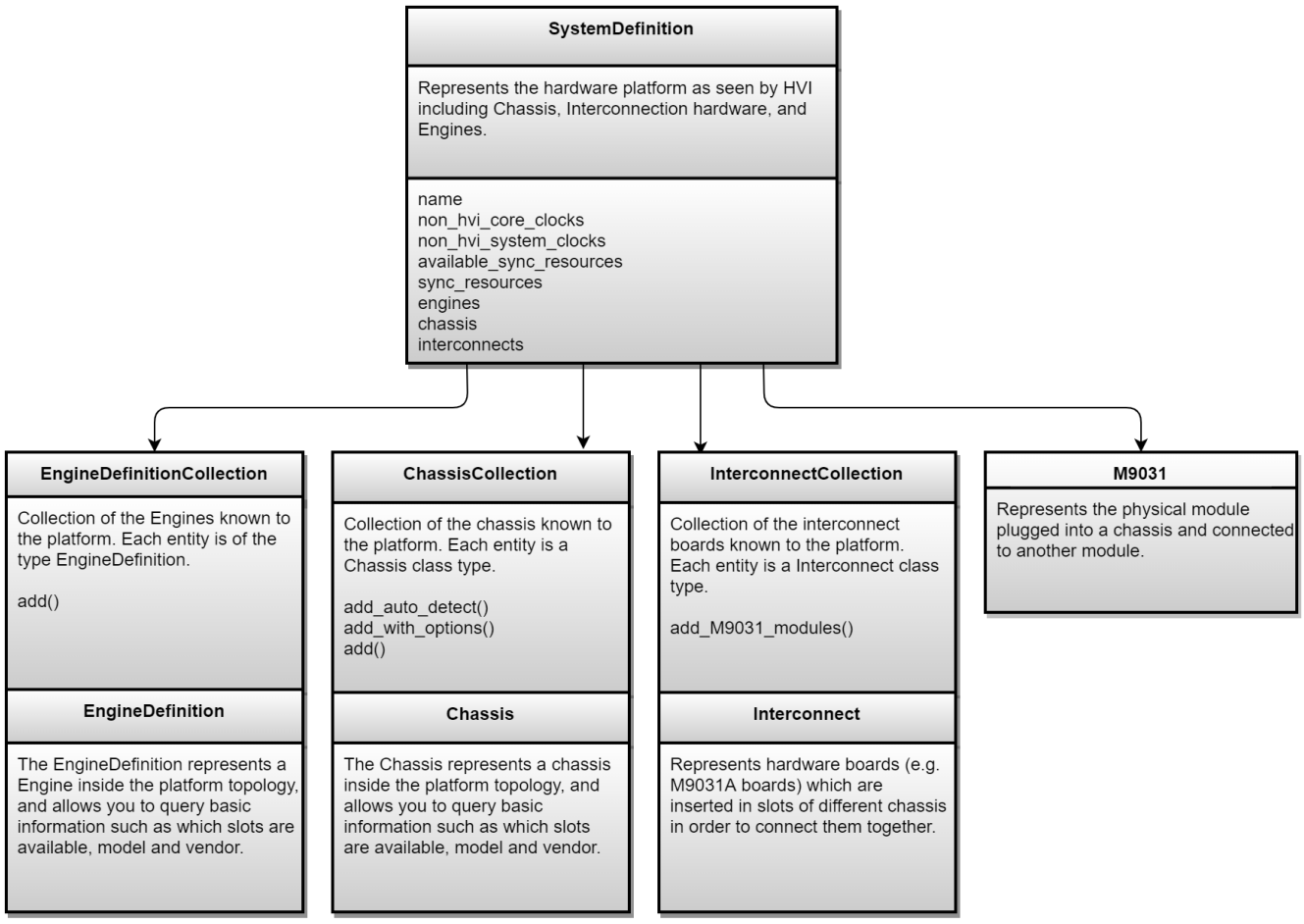
SystemDefinition

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

- Engines
- Chassis and Interconnects
- Synchronization Resources and Clocks

You use `SystemDefinition` to configure the physical hardware resources available to the HVI. This class has interfaces to the Chassis, Engines, and interconnected M9031A board pairs.

The following diagram shows the classes:



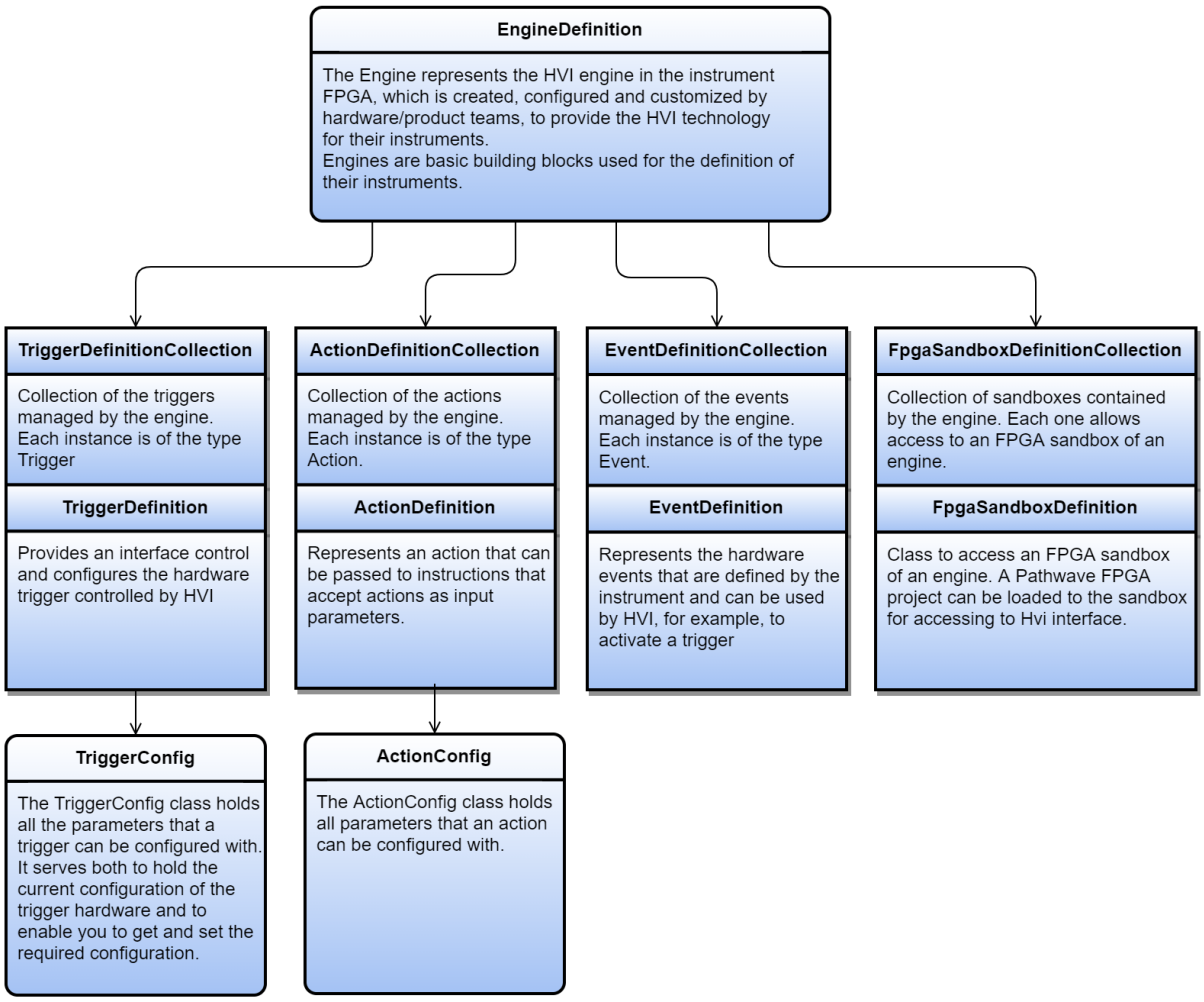
Engines

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

You create instrument objects where each object represents a physical PXI instruments placed into a specific chassis and slot. You can the obtain its Engine 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. Trigger signal lines include PXI triggers, Front Panel (FP) triggers, and any other trigger lines enabled within the instrument.

`TriggerDefinition` provides an interface control and configure the hardware trigger controlled by HVI. The `TriggerConfig` class holds all 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.

The class can be used 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`
`drive_mode`	Get or set the drive mode	`DriveMode enum: OPEN_DRAIN, PUSH_PULL`
`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, SYNC_CDC, SYNC_FAST, SYNC_USERx`
`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 for use by the HVI instance when it is loaded to hardware.

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 will use this event. Registration locks the resource for the HVI instance when it is loaded to hardware.

FPGA Sandbox Definition

An FPGA Sandbox is a user-configurable region in the FPGA. For the modules that support it, an HVI interface is provided into the sandbox. Through this interface, HVI can access read/write Registers and memory inside the sandbox.

To take advantage of this feature, 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 modules 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 will return an FPGA Sandbox Definition object with which the user can load the k7z file, exported from PathWave FPGA, to load the information related to this sandbox.

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

Once the sandbox project is loaded, you can access the contents of the FPGA sandbox, that is the Register and Memory-Map definitions.

FPGA Register Definition Class

Using an FPGA Sandbox Definition object that has already loaded a k7z file, the user can access the list of 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, the user 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 and Interconnects

This section describes the Chassis and Interconnect classes.

Chassis

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

It 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

This class represents hardware M9031A boards that are inserted in slots of different chassis to connect them together.

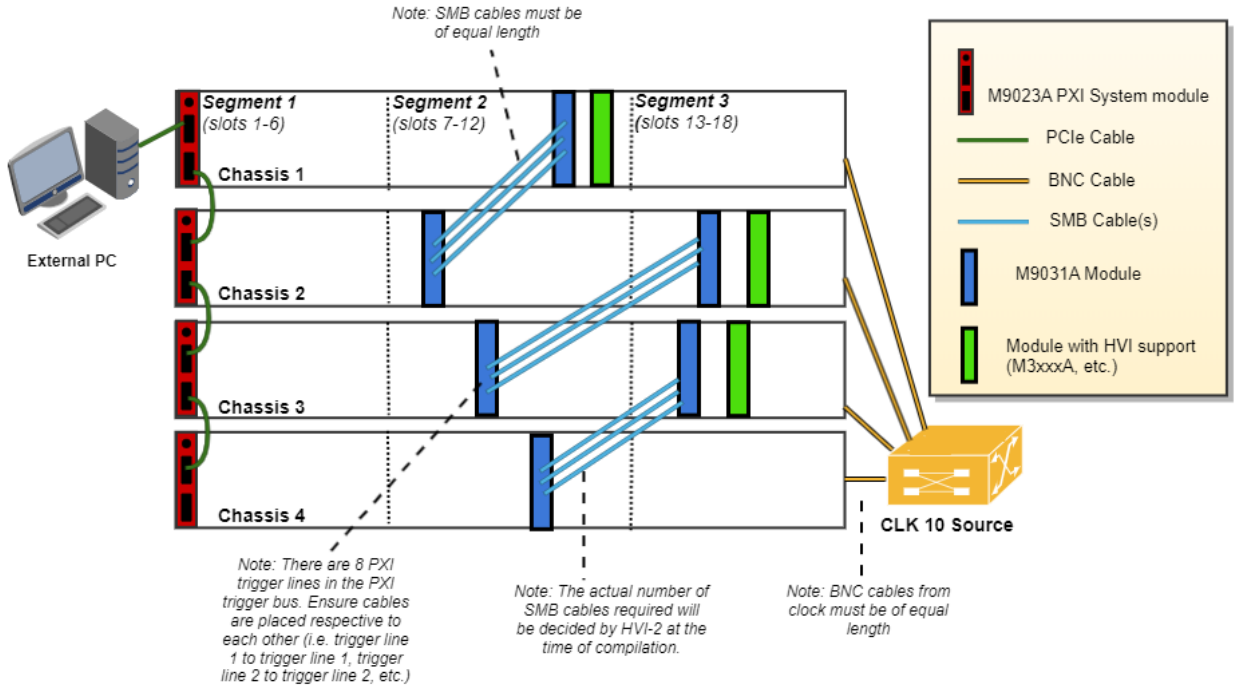
It has the following properties:

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

Multi-Chassis Setup

The reference examples provided with this document can be executed on a multiple-chassis setup with only the few modifications explained below. In a multi-chassis setup, it is necessary to interconnect the PXI triggers and clocking of the multiple chassis.

With the currently available infrastructure to interconnect PXIe backplane triggers a pair of M9031A boards must be placed in a specific segment in each chassis to be interconnected.



NOTE Ensure the SMB cables used to connect the M9031A modules are as short as possible. The chassis should be stacked in the same rack, on top of each other, and as close as possible to each other to enable the SMB cables that connect them to be as short as possible

On the two M9031A boards, the connectors corresponding to the same PXI line(s) are connected between each other. There are mainly three rules to consider when choosing the chassis slot where to place a M9031A board:

1. Only one M9031A board can be placed in a chassis segment. M9031A boards are connected in pairs. Each pair of M9031A connects two chassis together and shares info through their PXI lines
2. If no other M9031A board is already placed in the central segment, then the M9031A board should be placed there as a preferred choice, to minimize the signal path length

3. A PXI module included in the HVI application needs to be placed in the same chassis segment where the first M9031A board of each pair is placed, in order to control the exchange of PXI line values through the pair of boards.

The previous image illustrates the PXI modules in green. These must be placed in the same segment as the blue M9031A modules:

- The 1st chassis must include a M9031A together with a PXI module with HVI in segment 2
- All Middle chassis must have a M9031A in the segment 2, and a M9031A together with a PXI Module with HVI support in Segment 3
- The last chassis must include a M9031A in segment 2.

All the chassis that are part of the multi-chassis setup should be connected in a daisy chain. Chassis connections with M9031A are made to share the PXI lines that are used as sync resources. PXI trigger lines are shared using M9031A boards, connecting the ports corresponding to the same PXI line on both M9031A boards. The first and last chassis of the daisy chain each require one M9031A board; all the middle chassis in the daisy chain require two M9031A boards. A multi-chassis including N chassis requires a number of M9031A boards equal to 2*(N-1).

Additionally, a very clean 10 MHz source should be used to provide the same reference signal to all chassis. One option is to use a multi-output 10 MHz source, for best performance probably driven by an atomic clock, connecting each output to the 10 MHz reference input of each chassis using cables that have the same length. It is extremely important for the correct operation of HVI and in particular for synchronization that all chassis are running with their CLK10 and CLK100 fully locked and aligned, the skew between these clocks in the different chassis will result in skew in the instrument operation.

Adding Chassis

Each chassis included in the multi-chassis setup can be added using any of the HVI API methods below. You can either add them manually, or call the `add_auto_detect()` method once to automatically detect and add all the chassis connected to the system.

For example:

Python code:

```
# To add chassis resources use:
sys_def.chassis.add(chassis_number)
#
# Add a chassis and set options
sys_def.chassis.add_with_options(chassis_number, options).
#
# Automatically detect and add chassis:
sys_def.chassis.add_auto_detect()
```

Opening a chassis in simulation mode

You can use PathWave Test Sync Executive in simulation mode. This enables you to test your sequences without using real hardware.

To enable simulation mode, 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 specify the model of chassis and other characteristics of the simulated chassis.

To enable simulation mode, set `Simulate=True` and set the options you require after `DriverSetup`.

`Simulate=True` starts the chassis in simulation mode. If simulation is enabled, the chassis does not perform instrument I/O. For output parameters that represent instrument data, the chassis driver functions return simulated values.

`DriverSetup` specifies custom chassis options that are not standard for all instruments. HVI specifies the following options:

Option	values	Comment	Default	Notes
`model`	-	Specifies the model of the chassis you want to simulate.	-	
`NoDriver`	True/False	Specifies if a hardware driver is used or not. If set to True, HVI uses a simulated driver.	False	-
`EnhancedPxiTrigger`	True/False	Enables more than one writer module in a chassis. If set to True, the triggers in the chassis have active and idle values.	• True for chassis model M9018B, M9019A or M9010A. • False for others.	This option is ignored if NoDriver is set as False.

For example, the following code is an example of a command to start chassis number 2 in Simulation mode:

```
sys_def.chassis.add_with_options(2, "Simulate=True,DriverSetup=model=M9018B,NoDriver=True")
```

Adding M9031A Boards

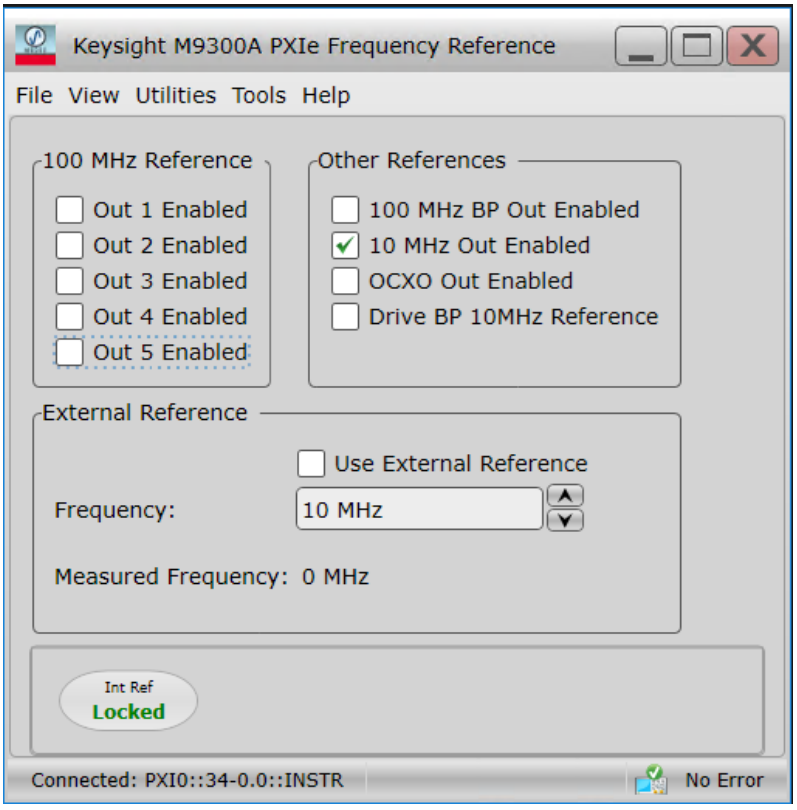
In the HVI API each M9031A board pair needs to be declared using the following software method:

Python code:

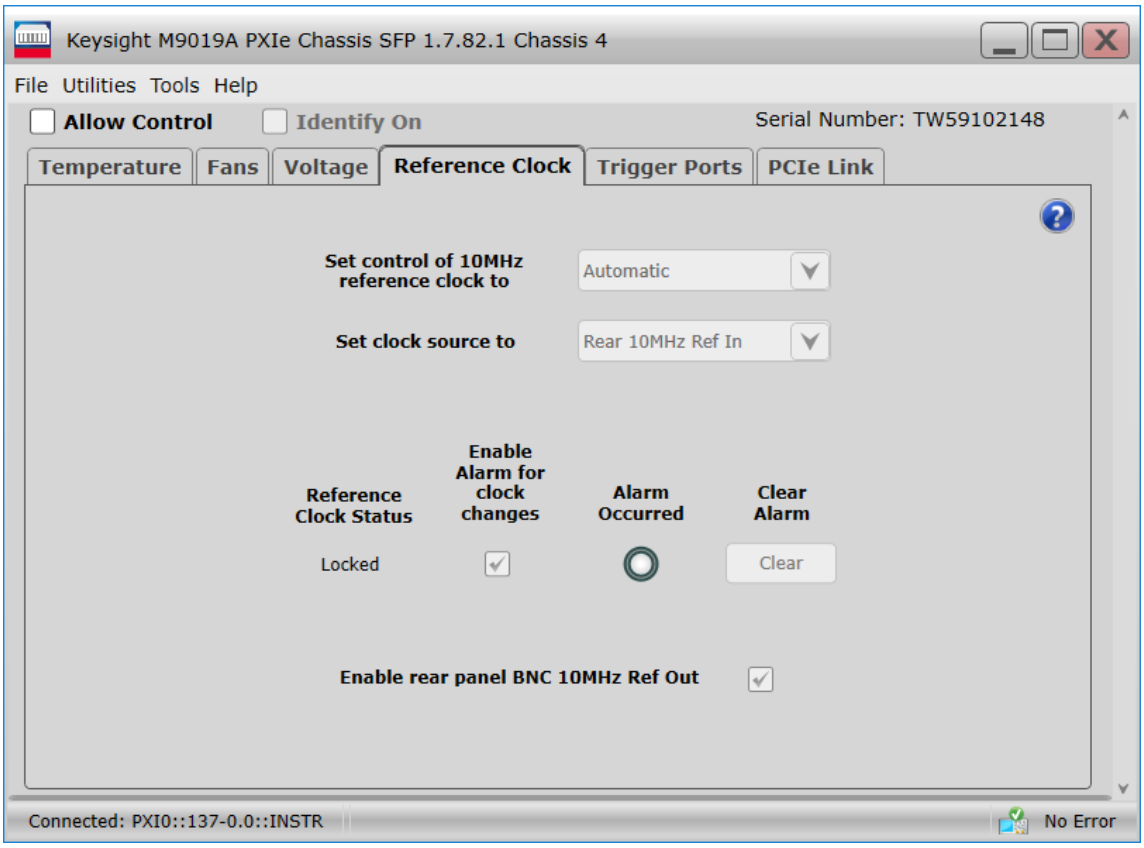
```
# To add each interconnected pair of M9031 modules use:
interconnects.add_M9031_modules(1st_M9031_chassis_number, 1st_M9031_chassis_slot,
    2nd_M9031_chassis_number, 2nd_M9031_chassis_slot)
```

10 MHz Clock Reference Source

One option is to use as a 10 MHz Reference source the PXI module Keysight M9300A PXIe Frequency Reference. Place this module in one of the chassis and use splitters to divide the 10 MHz clock output into N cables to be connected to the 10 MHz REF IN connector on the back panel of each of the chassis, including the chassis where the M9300A module is placed. Each time the system is restarted open the M9300A SFP software to check the box **10 MHz Out Enabled** and uncheck the box **Drive BP 10 MHz Reference**. See the following screenshot clarification. For more information about the Keysight M9300A, see Keysight M9300A PXIe Frequency Reference.



Once the common 10 MHz reference source is setup, the Chassis SFP can be used to verify that each chassis is correctly receiving the common external reference signal. This can be done from the **Reference Clock** window shown in the screenshot below. Once you open the window please clear any Alarm that possibly occurred during the 10 MHz reference setup. After clearing **Alarm occurred** icon should stay idle (white color). Clock source shall set to **Rear 10 MHz Ref In**.



In the case of using a remote controller card, such as the M9023A PXI System Module, it is possible to see the backplane status LEDs that also indicate the correct clocking. On the chassis, backplane **REF** and **LOCK LED** lights are lit in green when the chassis is correctly locked to the external reference signal. By checking the LED lights on the backplane of each chassis you can ensure the 10 MHz reference is correctly shared among the different chassis.

The following image shows the LED lights on the chassis backplane, visible from the front panel by removing the panel in the chassis slot that is preceding chassis slot 1.

For more details on the Keysight PXIe Chassis Family see Keysight PXI Chassis.

Synchronization Resources and Clocks

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

HVI Synchronization Resources

For synchronization, synchronized execution and data sharing across instruments to work, you must specify the sync resources available for HVI to use in the System Definition. The sync resources consist of PXI triggers and are defined by the `keysight_hvi.TriggerResourceId` enumeration. the list must be specified in the `SyncResources` property of the `SystemDefintion` object:

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

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

- Triggered synchronization, for more information, see Chapter 7: HVI Time Management and Latency.
- Sync Register-sharing.
- Sync while.

As a guideline, Triggered synchronization requires 1 or 2 triggers, Sync Register-sharing as many triggers as bits shared, and Sync while requires 1 trigger. The HVI optimizes the use of triggers as much as possible and reuses the same triggers when possible for different operations, providing they are executed with sufficient time separation.

Synchronize The HVI Instrument's Engines

To correctly manage timing without jitter, the HVI needs information about all of the clocks in the 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.
- Non-HVI core 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 must be triggered from the HVI. 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 must be triggered from the HVI. Core clocks are used by the HVI to determine all but the Sync-Base period, that is Sync_CDC, Sync_ Fast, Sync_User<K>, etc.

HVI Synchronization signals and modes

HVI uses different periodic digital signals for synchronization purposes: `Sync_Base`, `Sync_CDC`, `Sync_Fast` and `Sync_User<K>`. 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 definitions listed below:

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

`Sync_User<K>` signals (where *K*= 0, 1, 2,.. N) , provide you some degree of freedom to define their frequency or phase so you can deploy them for synchronization purposes that you can define. The only constraint is that the frequency of `Sync_User<K>` signals must be an integer multiple of the frequency of the `Sync_Base` signal.

Sequencer

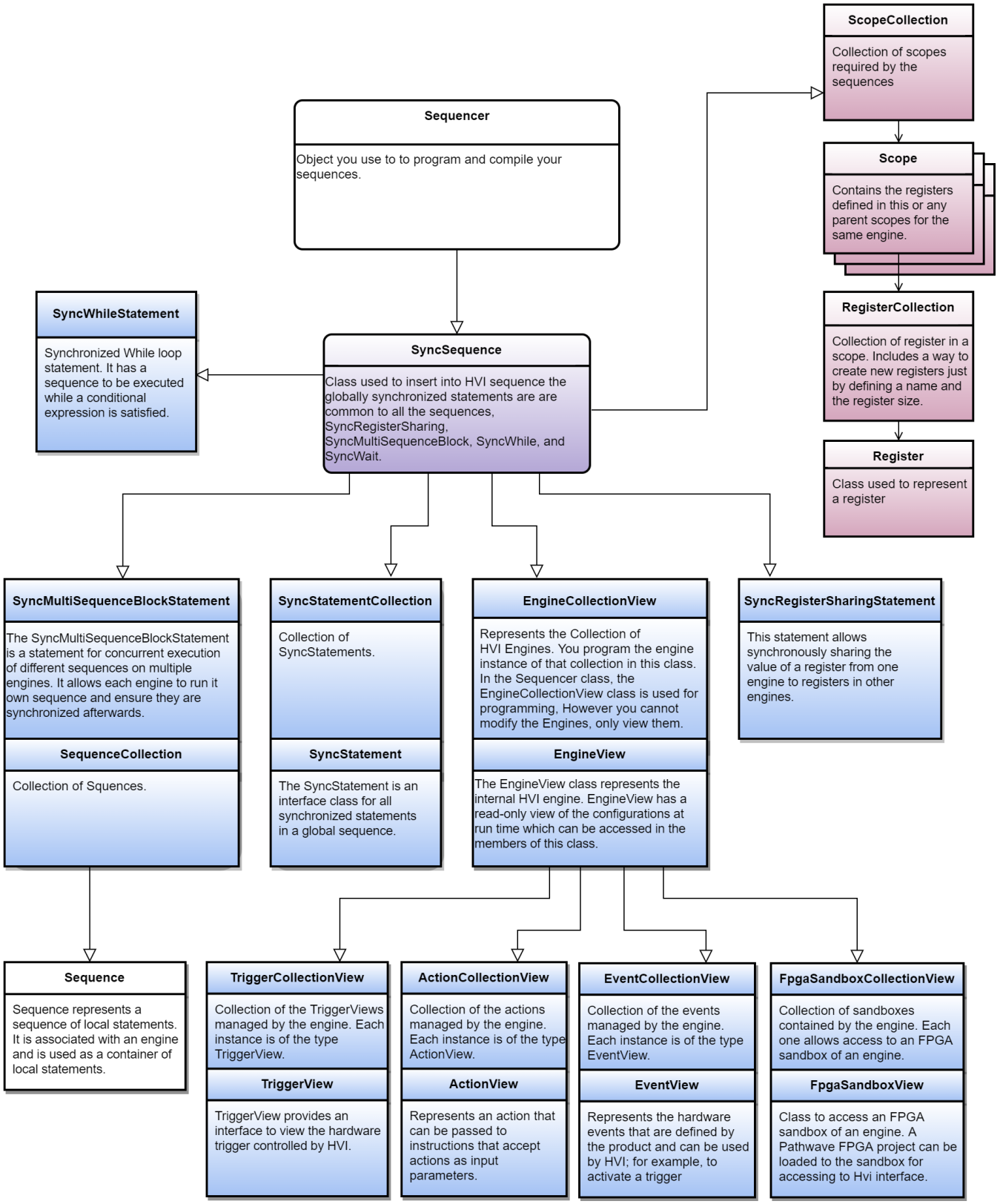
This section describes the Sequencer class, it contains the following sections:

About the Sequencer Class

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

- Program your sequences with the `SyncSequence` class.
- The `SystemDefinitionView` class enables you to see the system definition you have set up, but you cannot modify it.
- The HVI instance `Hvi`, is generated when you compile the sequencer.

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.

The `SyncSequence` and `Sequence` classes enable HVI sequence programming and usage. 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 that are added are collected in the `SyncStatement` class.

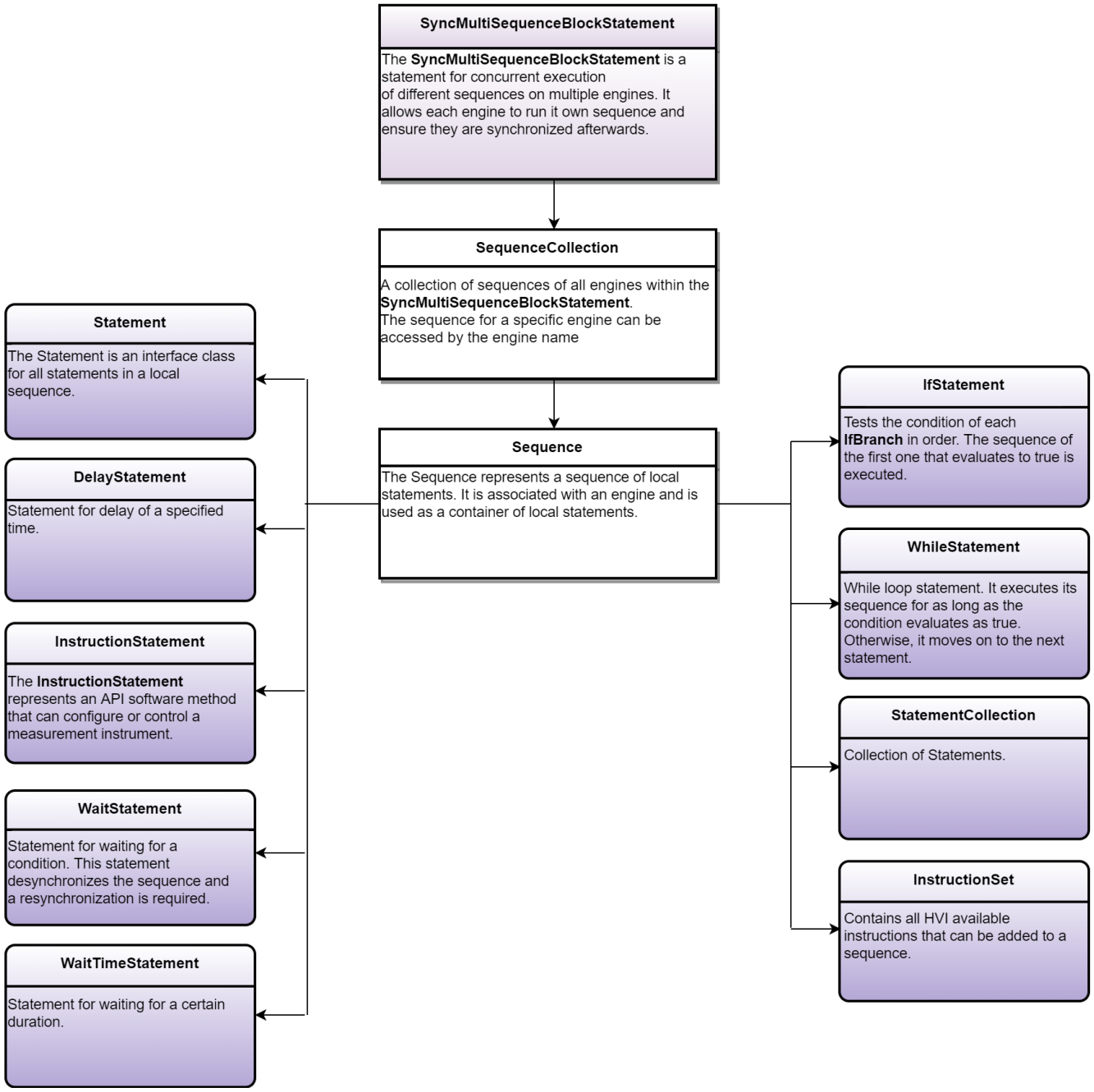
Synchronization and timing information are added within each synchronized statement so all sequences across the HVI are coordinated precisely. The `SyncMultiSequenceBlockStatement` exposes local flow control and instruction statements that are sent by the `Sequence` object. The other instructions are 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 to 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 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 together with Python code examples showing how to program the statements with the HVI Python API. The execution of each statement within a sequence is explained and shown with a corresponding HVI diagram.

Sync Statements

Sync statements are the building blocks used to program Sync sequences. The following types of Sync statement are available:

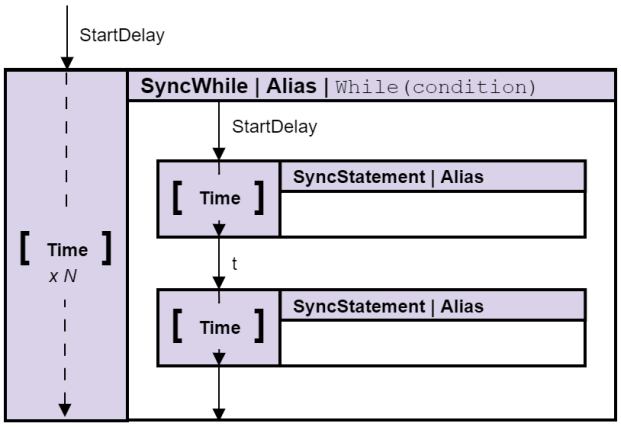
- Sync while.
- Sync multi-sequence block.
- Sync Register-sharing.

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.

Sync while 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.

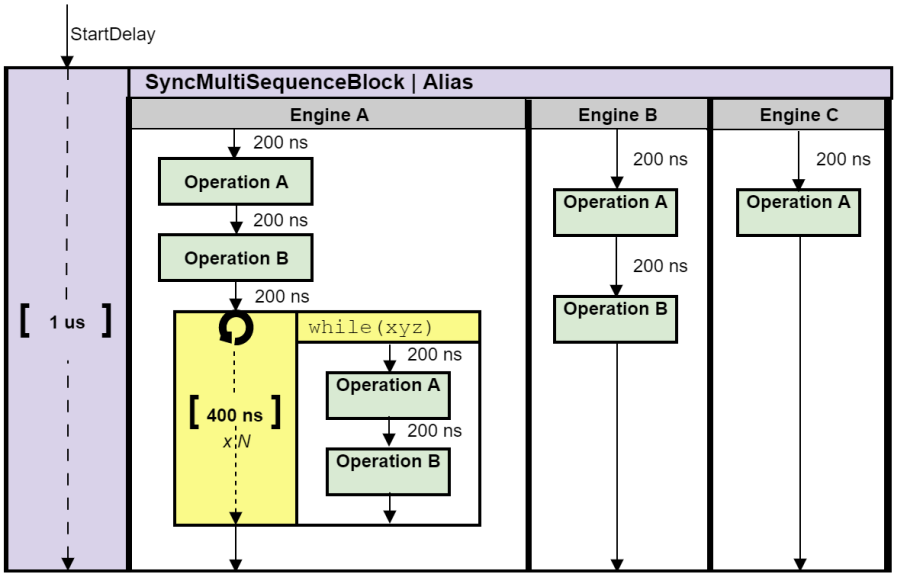
```
# Configure Sync While Condition
sync_while_condition = keysight_hvi.Condition.register_comparison(reg,
    keysight_hvi.ComparisonOperator.GREATER_THAN, 10)
#
# Add Sync While to a sync-sequence
sync_while = my_sync_seq.add_sync_while("sync_while", 10, sync_while_condition)
#
# Access the sync sequence 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 Local sequences start exactly at 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:



The following code snippet shows a Sync multi-sequence block being added with the call `add_sync_multi_sequence_block()`. The second line of code shows how a Local sequence is obtained and an instruction is 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. The execution of each statement within a sequence is explained and shown with a corresponding HVI diagram.

Programming Local Sequences

Local sequences can be programmed within a Sync multi-sequence block or within Local while or Local If, Local flow-control statements. 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. The example shows how to add an instrument specific HVI instruction to a Local sequence using the `add_instruction()` API method. The code also shows 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. These 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:

```
# 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)
```

FPGA Register Read

Instruction `fpga_register_read` is an HVI-native instruction that enables you read from an HVI Port Register. The value read from the HVI Port Register is written to a destination HVI Register.

The following code example shows an FPGA Register Read instruction:

```
# Read FPGA Register Register_Bank_HviAction4Cnt
sequence = sync_block_1.sequences["engine_Name"]
register_destination= hvi.sync_sequence.scopes["engine_Name"].registers.add("register_
destination", keysight_hvi.RegisterSize.SHORT)
hvi_register = hvi.sync_sequence.engines["engine_Name"].fpga_sandboxes["sandbox_Name"]
    .hvi_registers["hvi_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,
    register_destination)
readFpgaReg0.set_parameter(sequence.instruction_set.fpga_register_read.fpga_register.id,
    hvi_register)
```

FPGA Register Write

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.

The following code example shows an FPGA Register Write instruction:

```
# Write FPGA Register Register_Bank_HviPxiTrigOut
# NOTE: Please allow at least 50 ns between these instructions to ensure the HVI register
# action4_cnt is updated before writing its content to PXI lines
register_value= hvi.sync_sequence.scopes["engine_Name"].registers.add("register_value",
    keysight_hvi.RegisterSize.SHORT)
hvi_register = hvi.sync_sequence.engines["engine_Name"].fpga_sandboxes["sandbox_Name"]
    .hvi_registers["hvi_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,
    hvi_register)
writeFpgaReg0.set_parameter(seq.instruction_set.fpga_register_write.value.id,
    register_value)
```

FPGA Memory Map Write

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.

The following code example shows an FPGA Memory Map Write instruction:

```
# Register, Memory map objects
register = sync_sequence.scopes["engine_Name"].registers.add("register_value",
    keysight_hvi.RegisterSize.SHORT)
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 Memory Map Read

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.

The following code example shows an FPGA Memory Map Read instruction:

```
# Register, Memory map objects
register = sync_sequence.scopes["engine_Name"].registers.add("register_value",
    keysight_hvi.RegisterSize.SHORT)
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)
```

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 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)
```

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-Elseif-Else.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

The Wait-time flow control statement causes the sequence to wait for a certain time specified by an HVI Register. Once the time has elapsed, the sequence will continue.

Local Wait-For-Event

The Wait-event flow control statement causes the 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. In future releases, this will be extended to more complex conditions.

Local While

While flow-control is defined by executing the same sequence in a loop while the condition is met.

Local Delay

The Local Delay statement delays the sequence for a time you specify. The delay is specified in nanoseconds.

Local If-Elseif-Else

The If-Elseif-Else flow control statement is a flow-control statement that conditionally executes different possible Local sequences 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-Elseif-Else Statement can have multiple sequences. These statements have the following common characteristics:

- Sequences in flow-control statements can contain any 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.

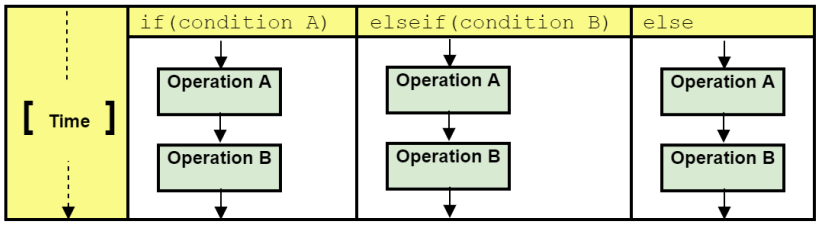
Local If-Elseif-Else Statement

The If-Elseif-Else Local flow-control statement conditionally executes one of a set of different possible Local sequences depending on the value of a defined condition.

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.
- 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-Elseif-Else 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 If-Elseif-Else statement contains one or more If branches and an Else branch. The instructions and statements contained in each If or Else branch are executed if the condition of each branch is met. The condition of each branch can be defined using the API class `ConditionalExpression`. The branch sequences can be programmed using the same API methods and classes used to program the main HVI sequence, using the classes `IfBranch` and `ElseBranch`.

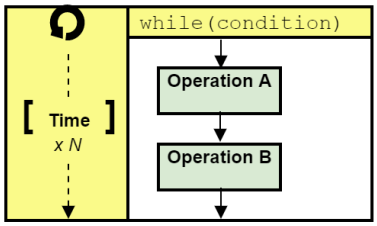
The following code is an example of an If-Elseif-Else statement:

```
# 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 the condition is met.

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.

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 by an HVI Register. Once the time is elapsed, the sequence will continue.

The following code is an example of a Wait-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 sequence for a time you specify. The delay is specified in nanoseconds.

The following code is 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)
```

InstructionSet

HVI instructions can be of two types: HVI-native instructions or instrument-specific instructions.

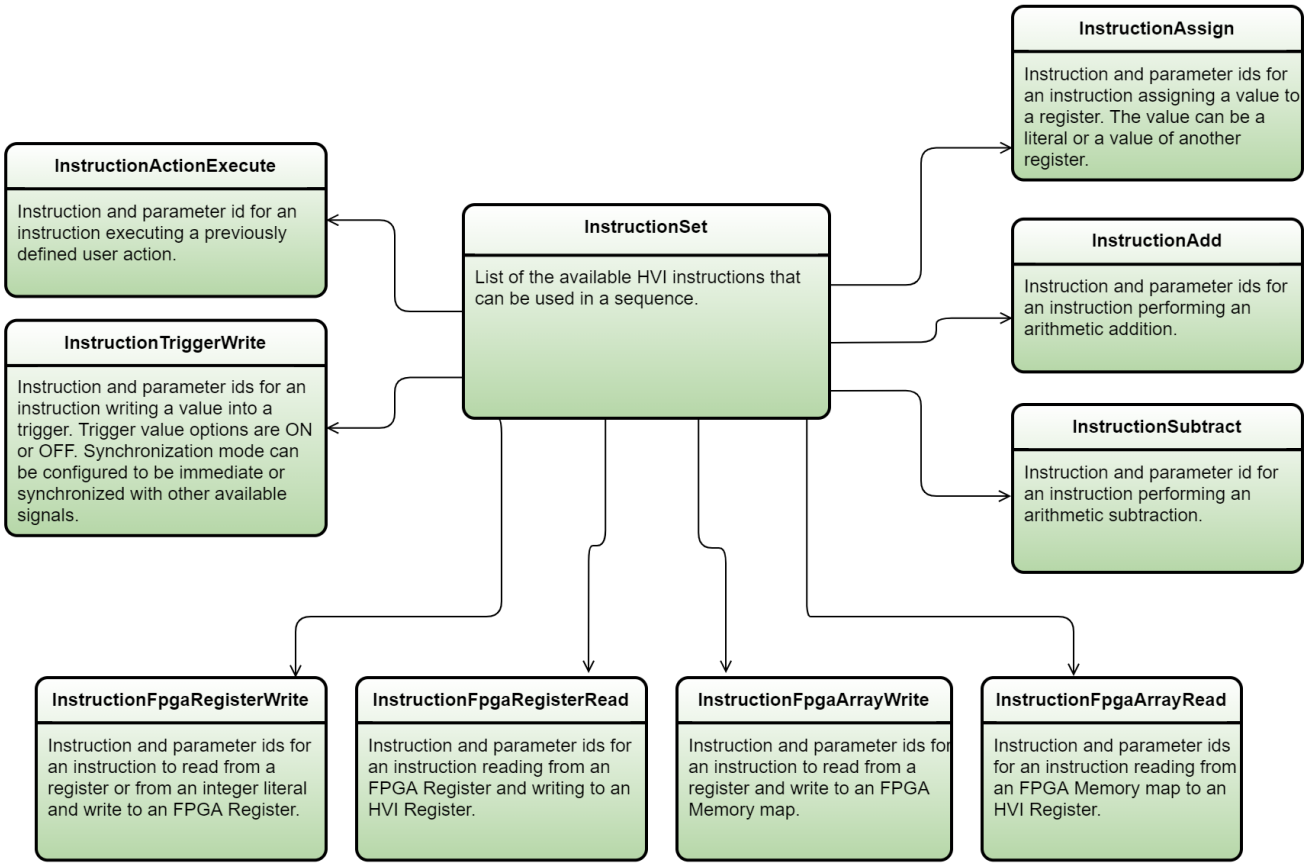
- Instrument-specific instructions are documented in instrument user guides.
- HVI-native instructions are part of the `InstructionSet` class.

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 `trigger_write` native instruction. For the meaning of each parameter value, see the HVI Python API help that is installed with PathWave Test Sync Executive. It is located at:

`C:\Program Files\Keysight\PathWave Test Sync Executive 2020 Update 1.0\api\python\Help`

Parameters for the HVI-native instruction: `sequence.instruction_set.trigger_write`

Parameter ID	Parameter Value
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)
```

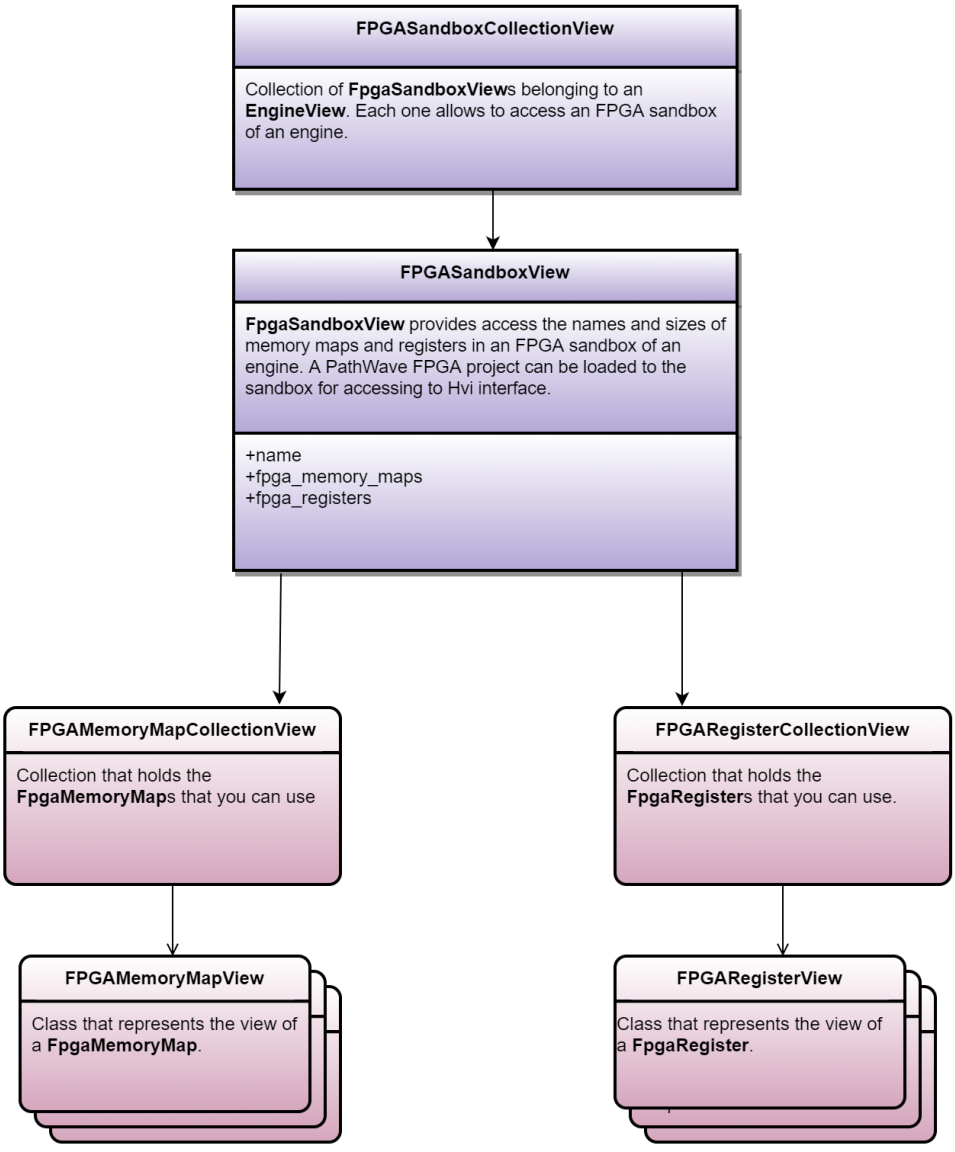
Instrument-specific HVI instructions

You program instrument-specific instructions into your HVI sequences using the same method 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 the user manual of your specific instruments. 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 MemoryMaps

The `FpgaSandboxView` object provides access to FPGA memory maps by providing handles to FPGA Registers and memory maps 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. You can then 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 the hardware Registers provided by HVI engines. Like Variables in a programming language, Registers can be used as parameters for instructions and statements and are modified in real-time during the sequence execution. The number of HVI Registers available is defined by the instrument (see HVI engine settings HviRegCount). Since HVI Registers belong to specific HVI engines, they cannot be accessed by other HVI engines. Explicit HVI Register sharing instructions are required to transfer data between HVI Registers. HVI Registers are defined by adding them to the HVI Register collection bound to HVI scopes.

HVI Scope

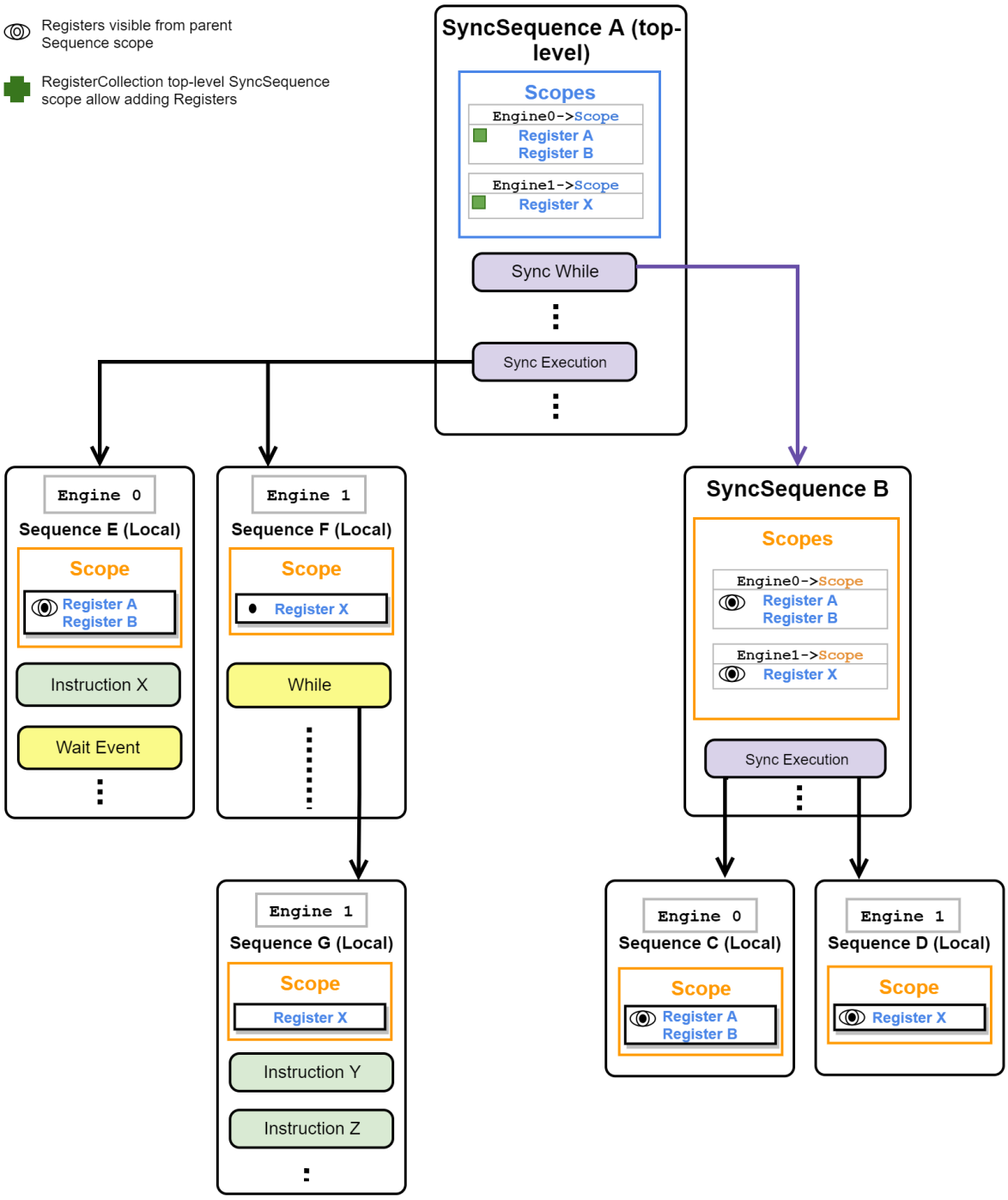
HVI Sync sequences and HVI Local sequences both include the concept of scope for Registers, similar to programming languages such as C. The concept of scope is necessary to define what HVI Registers, or memory resources can be used within each HVI Sequence, and when they can be used.

A specific scope is associated with a specific sequence and HVI engine. HVI Registers can be created only within the Global Sync sequence scope but they can be retrieved from any child sequence scope. The Registers created within the scope of a parent sequence are visible to its children sequences and can be accessed using their scope. HVI 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 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.

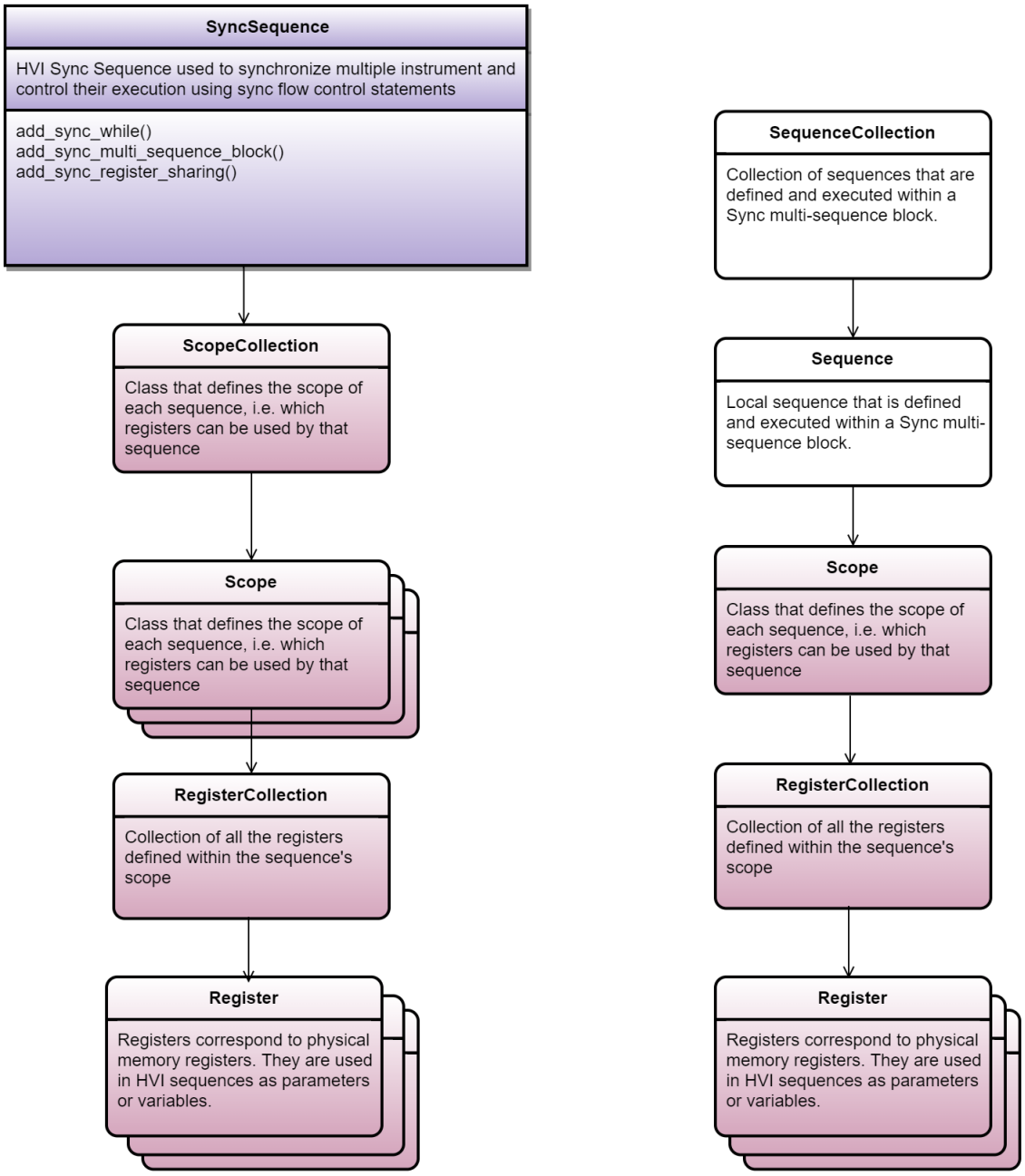
The following diagram shows the scope concepts. The eye icon is used to represent the visibility in child sequences of the Register belonging to the scope of parent sequences:



The scope of each HVI sequence is managed through the Scope class. Each HVI Local sequence is an instance of the Sequence class and 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 for each associated HVI Engine. The HVI Scope collection is an instance of the ScopeCollection class and contains objects instance of the Scope class, 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 HVI 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` and it lives in the Namespace `Time`. This class represents a time interval.

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 signatures 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, 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.

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:

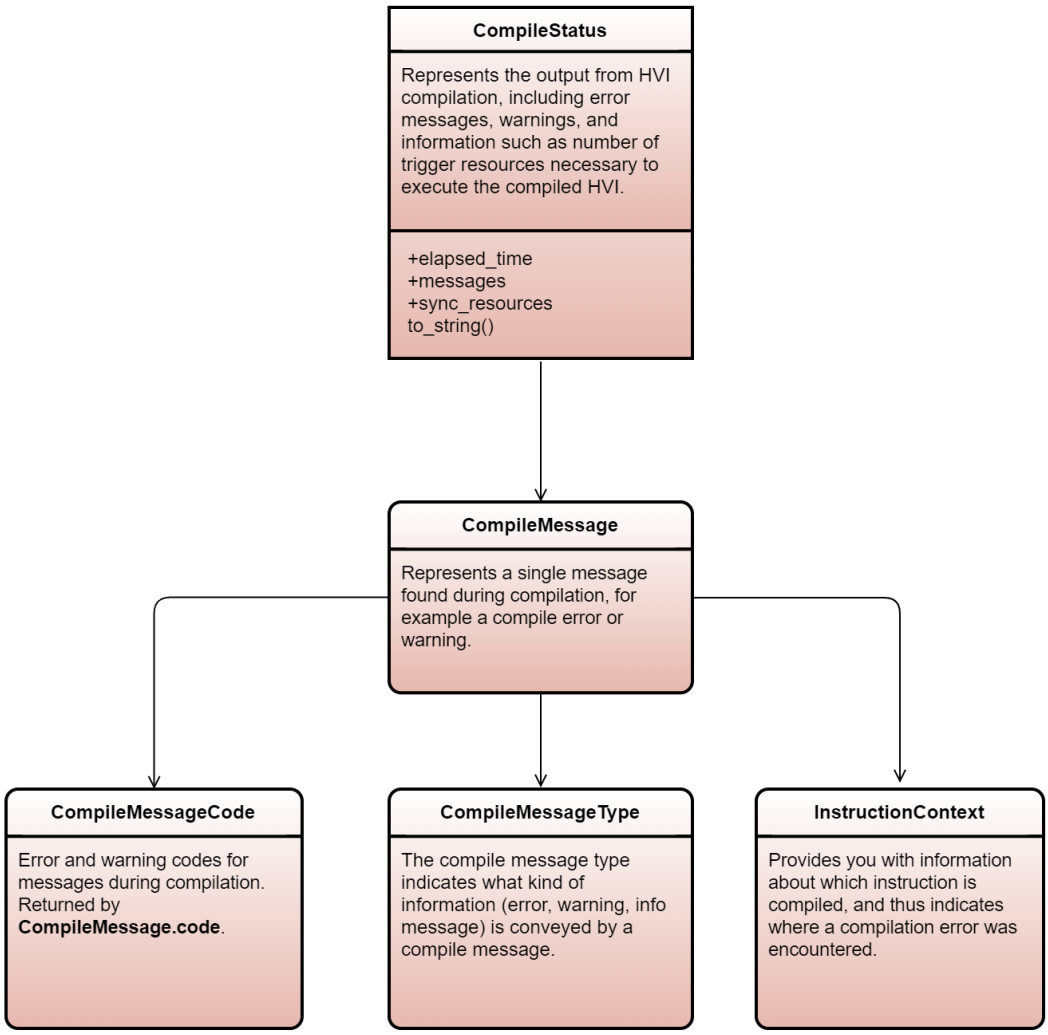
```
try:
   hvi = sequencer.compile()
      print('Compilation completed successfully!')
except kthvi.CompilationFailed as err: print('Compilation failed!')
      compile_status = err.compile_status
      print(compile_status.to_string()) # This line will print all the errors and warnings
collected during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

where:

- `ASIGNEE` 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. 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 `KtM360xA` 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:

```
+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"])
```

The Hvi Object

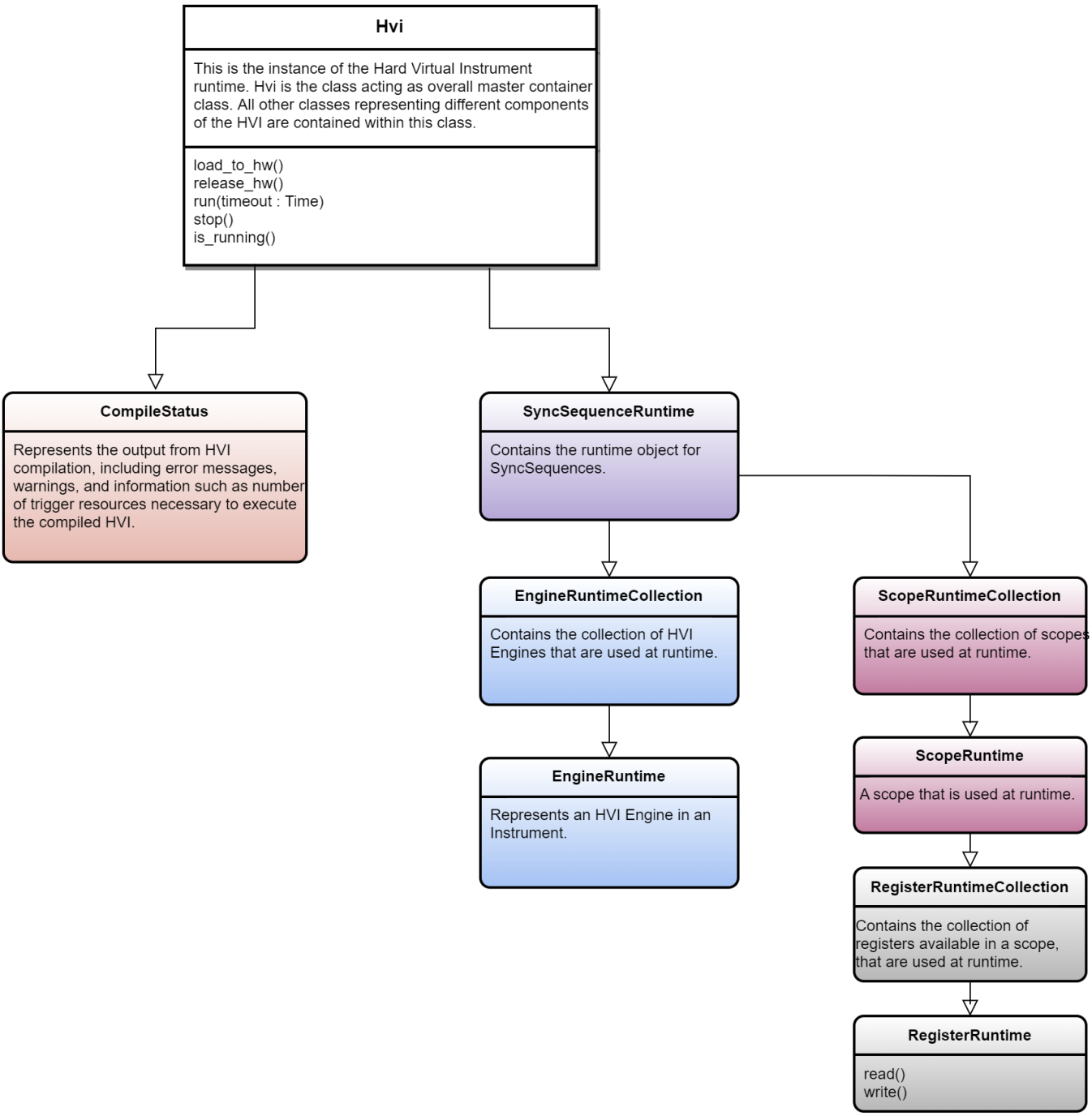
This section describes the Hvi object, it contains the following sections:

- EngineRuntime Components
- Load to Hardware and Run

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.

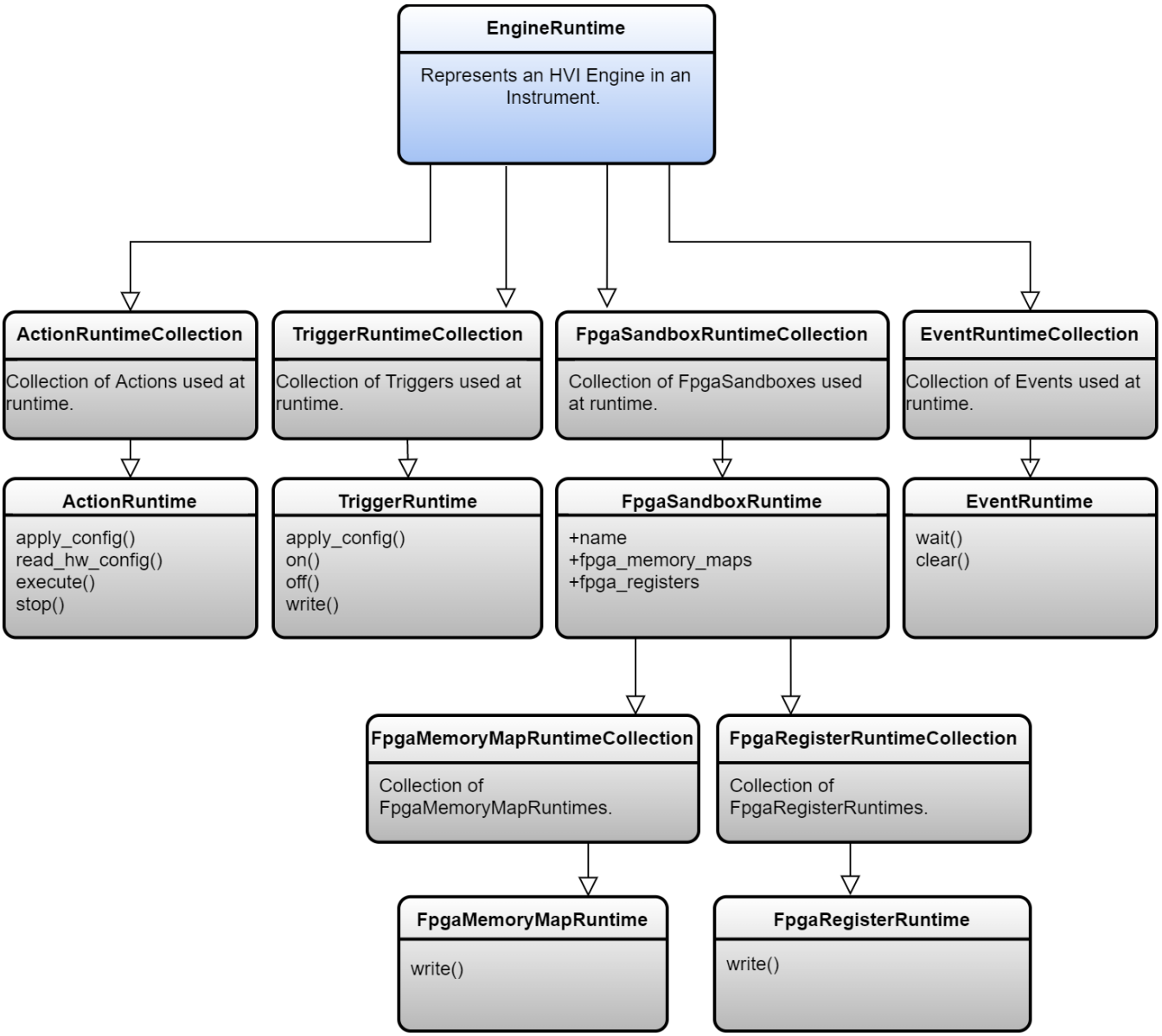
The following diagram shows the classes:



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 input parameters.

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 the product and can be used by HVI, for example, to activate `TriggerRuntime`.

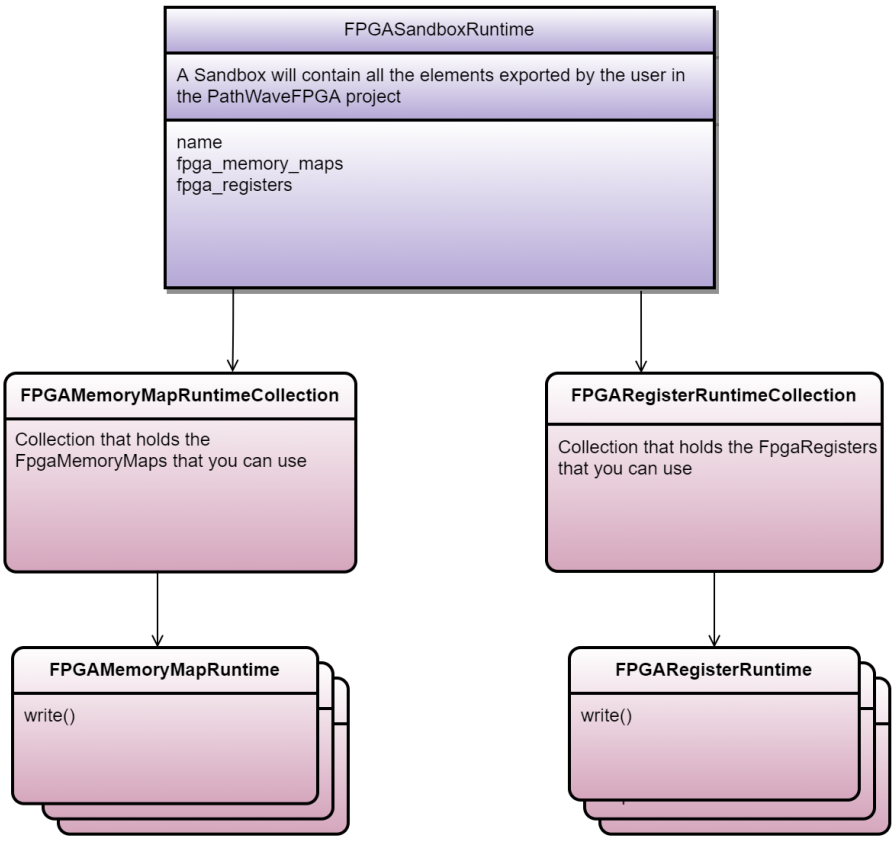
RegisterRuntime

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 between `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.

Chapter 6: 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.

NOTE The code examples provided in this chapter are in both Python and C#.

Planning an HVI

Programming an HVI requires some planning. You must assign and set up resources before you can use them in sequences. The resources you can use depend 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 assign the capabilities as resources in your application.

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.

When you have written your sequences, you call a compile command. After this, you upload the binaries to hardware and execute your sequences. Before running the HVI, you can redefine the initial values and configurations of the resources that are included in the HVI, such as HVI Registers for different Engines.

Programming steps

To program an HVI you must follow these steps:

1. Set up the HVI.
2. Write HVI sequences.
3. Compile your sequences.
4. Load to hardware.
5. Modify Initial Register Values (optional).
6. Execute sequences.
7. Release.

1 Set Up The HVI

Setting up the HVI requires a number of 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=M9018B,NoDriver=True")
```

C# code:

```
// Add chassis with number or options
sysDef.Chassis.AddWithOptions(1, "Simulate=True,DriverSetup=model=M9018A,NoDriver=True");
```

Define the chassis interconnects

Python code:

```
# Add interconnects
my_system.interconnects.add_M9031_modules(1, 1, 2, 1)
```

C# code:

```
// Add interconnects
sysDef.Interconnects.AddM9031Modules(1, 1, 2, 1);
```

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 from the engines you want to use them with. You must do this for the following types of resources:

- HVI Engines.
- Triggers.
- Actions.
- Events.
- 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_engine(module.hvi.engines.main_engine, "Receiver")
engine1 = my_system.engines.add_engine(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 to 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. The code examples below illustrate some example definitions.

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 instrument have a single sandbox region in their FPGA, but some instruments may have multiple. 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 Write HVI Sequences

Writing 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 Sandbox FPGA 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 Registers resource you require in each engine and u se the `add()` method to add them to the Register collection for that engine. 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.HVI Registers correspond to very fast access physical memory Registers in the HVI Engine located in the instrument hardware. You can use HVI Registers as parameters for operations and modify them during the sequence execution, the same as Variables in a programming language. 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 = self.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 about 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 Sandbox FPGA 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 7: 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 of statements. It also provides latency information for the different statements and instructions.

It 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.
- Latency Parameters.

Timing Concepts Overview

The following list describes the main concepts that apply to all statement types:

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 the actual execution. Depending on the statement or instruction characteristics, for instance, the number of parameters, a statement may take several HVI engine cycles to complete the fetch before the actual execution can start. The Fetch time consumes HVI engine execution cycles.

Start Delay

This is the user-defined delay value, in nanoseconds, from the Start-Time of the previous statement to the Start-Time of the current. The range of this value is as follows:

- The minimum boundary can be calculated by adding the Start-Latency of the current statement (or Entry-Latency in the case that the current statement is the first statement of a subsequence) and the End-Latency of the previous. If the previous statement is a local instruction, the End-Latency is considered equal to its Fetch-Time.
- The maximum boundary for the value is the maximum number of LCM clock cycles multiplied by the LCM clock period. LCM clock is the least common multiple clock frequency of all the clock frequencies of all the engines included in the Hvi. LCM clock cycles is a 64-bit signed integer.

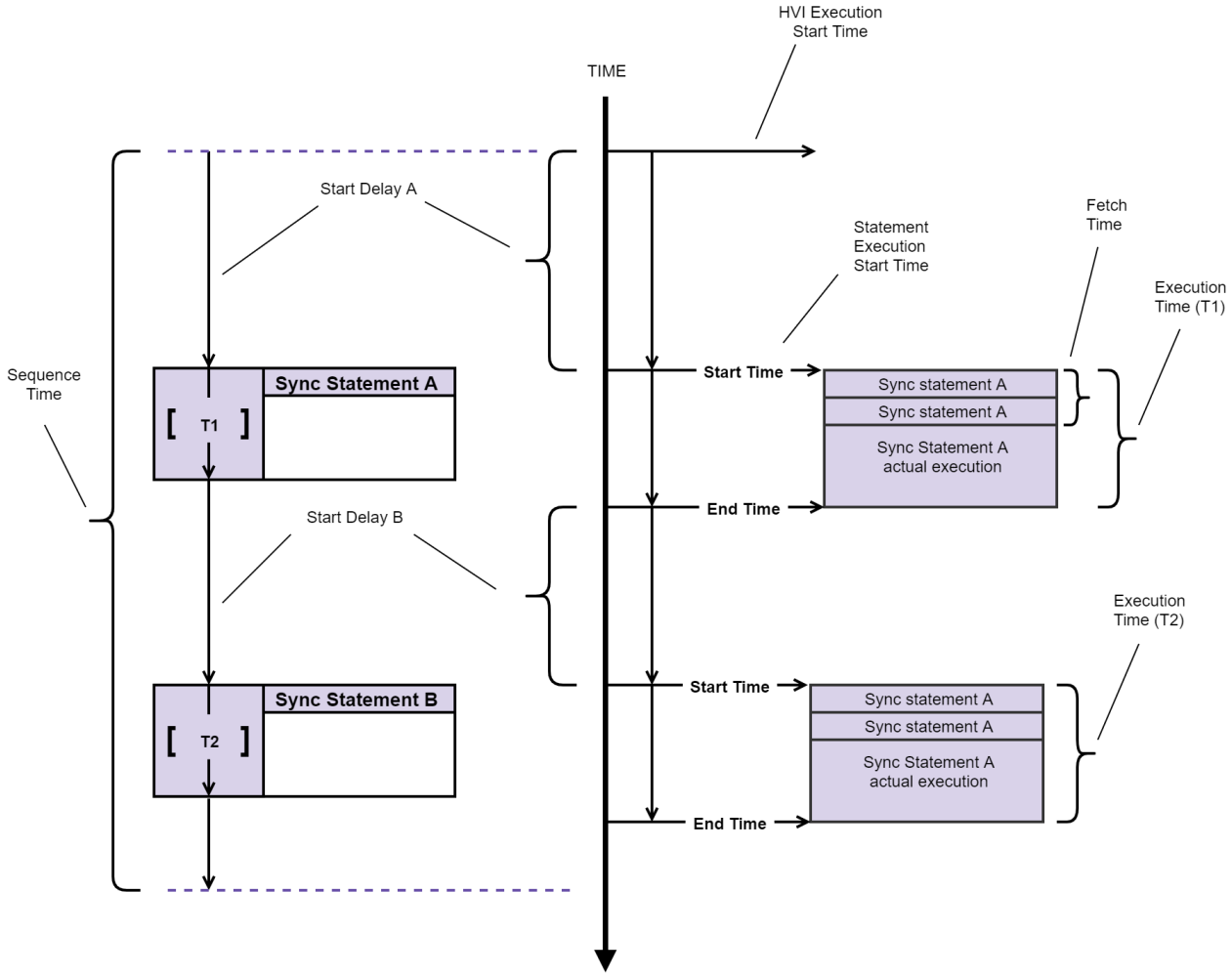
Execution Time

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. 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 only for the flow-control or sync statements.

The following diagram shows these concepts in an HVI diagram:



Latency Parameters

There are number of additional concepts and parameters you must be aware of to calculate timing, especially Start delays. Using incorrect values results in compilation errors.

Latency values are accounted for by HVI and included in start delays. They impose a minimum value to the start delays:

Start-Latency

This is the minimum number of clock cycles a flow-control statement requires to start the execution of the internal sequence(s). It is accounted by HVI as part of the Start delay of the statement and this imposes a limit on its minimum value.

Entry-Latency

This is the minimum number of cycles for the Start delay of the first statement of the internal sequence(s).

End-Latency

This is the minimum number of clock cycles a statement requires to exit the internal sequence(s) execution before another statement can be executed. It is accounted by HVI as part of the Start Delay of the next statement and this imposes a limit on its minimum value.

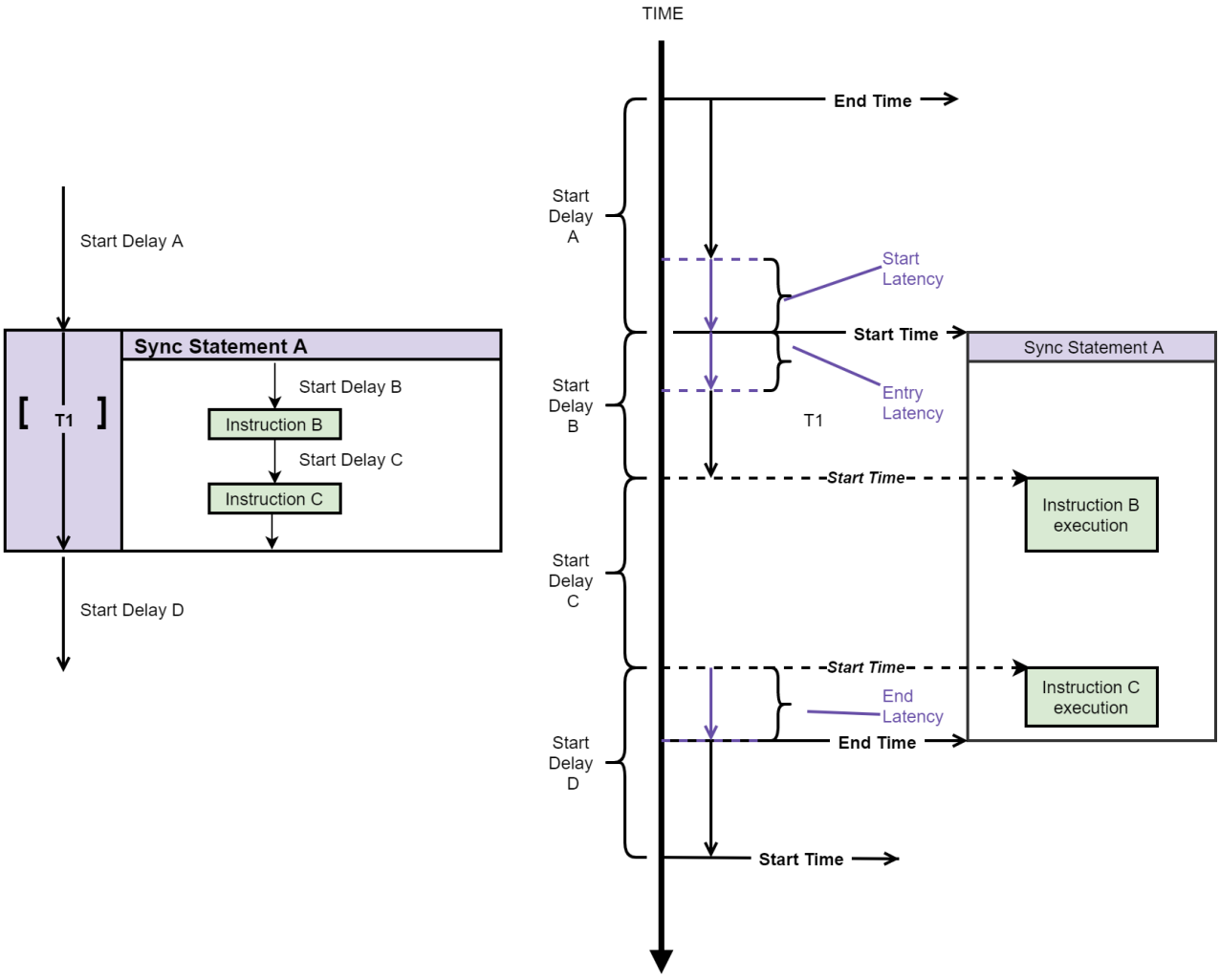
Iteration Latency (only for loop statements)

This is the minimum number of cycles for the Start delay of the first statement of the internal sequence(s) after completing an iteration of the loop.

NOTE The exact definitions of Start latency, Entry latency and End latency depend on the type of statement. See Statement Timing Tables.

Latency values are used in Sync Statement Timing and Local Statement Timing. The Latency values are listed in Statement Timing Tables.

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



Duration Property of Statements

Sync Statements and Local flow-control statements (If, While) have a user-configurable property called `duration`. The `duration` property enables you to specify the time interval a statement takes to execute.

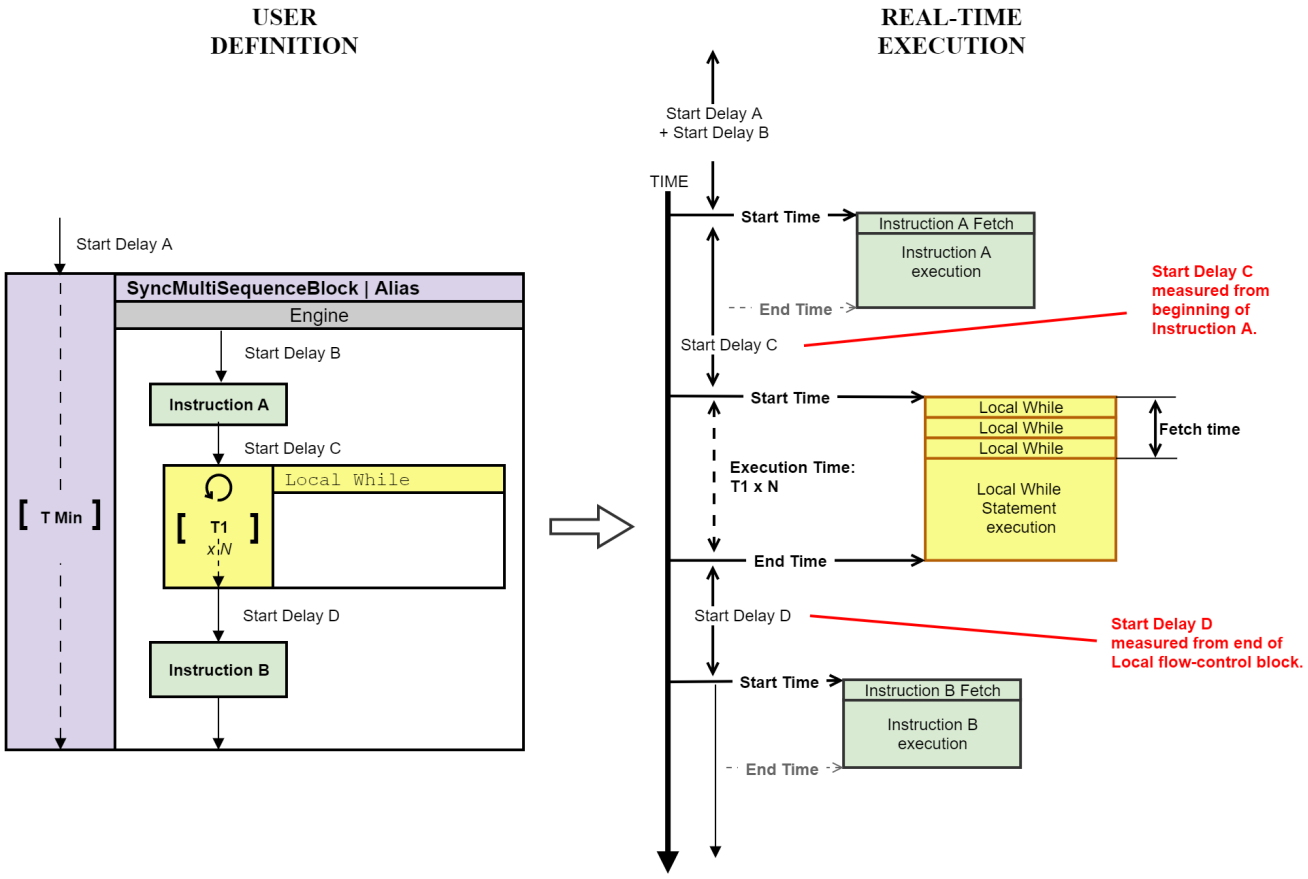
NOTE For the loop statements Local while and Sync-while, the `duration` property specifies the execution time of 1 iteration. This means that the overall execution time of a while statement depends on the number of iterations that executed. The total execution time is `duration` multiplied by the number 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. This can happen if, for example in an if-statement, more time is required to complete the statements inside a branch than the duration specified.

If the `duration` is set to a **minimum-time** interval, then execution time of a statement is the minimum possible given by the statements inside.

NOTE By default, if not specified, `duration` property is set to **minimum-time**.

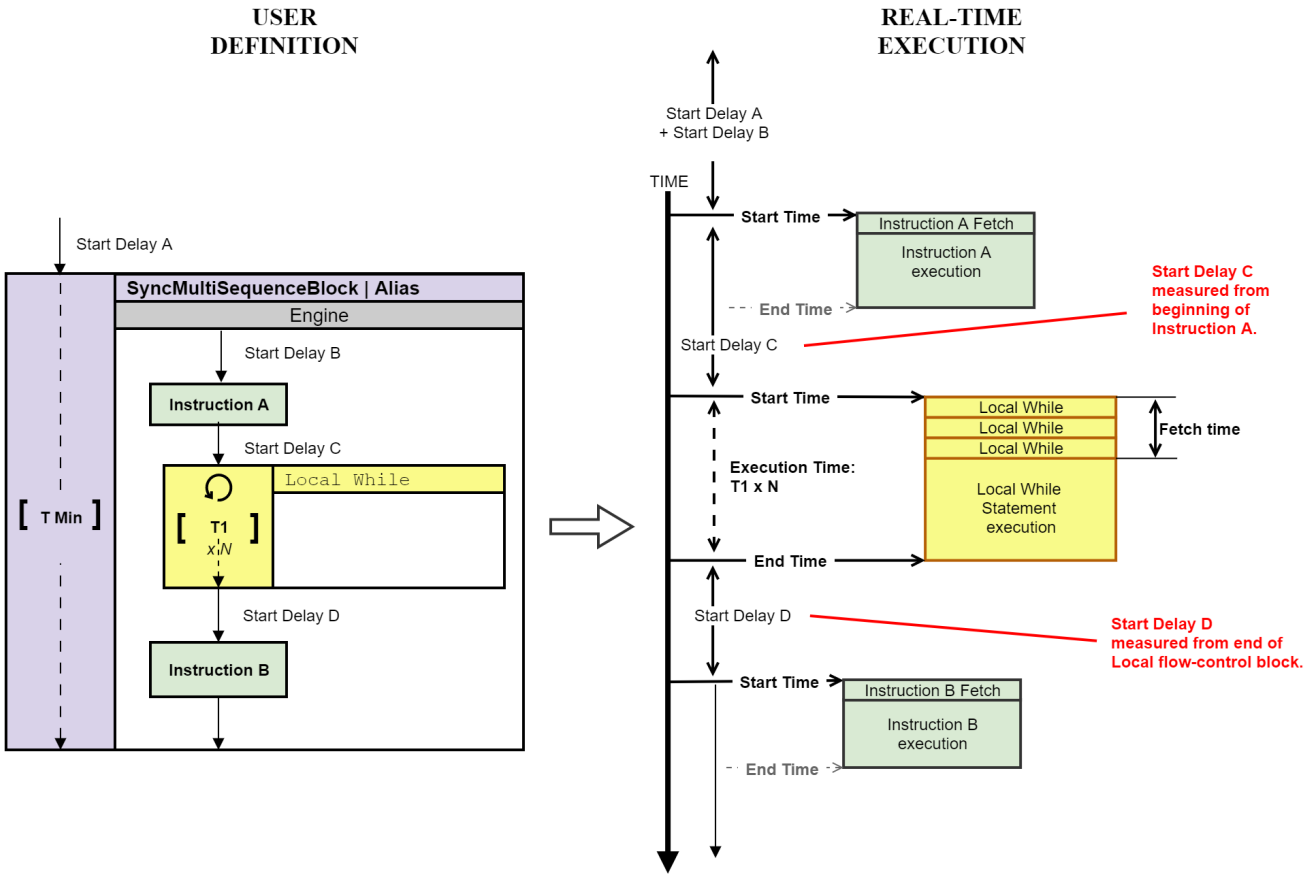
The following diagram shows how the `duration` property is applied to a Sync multi-sequence block.



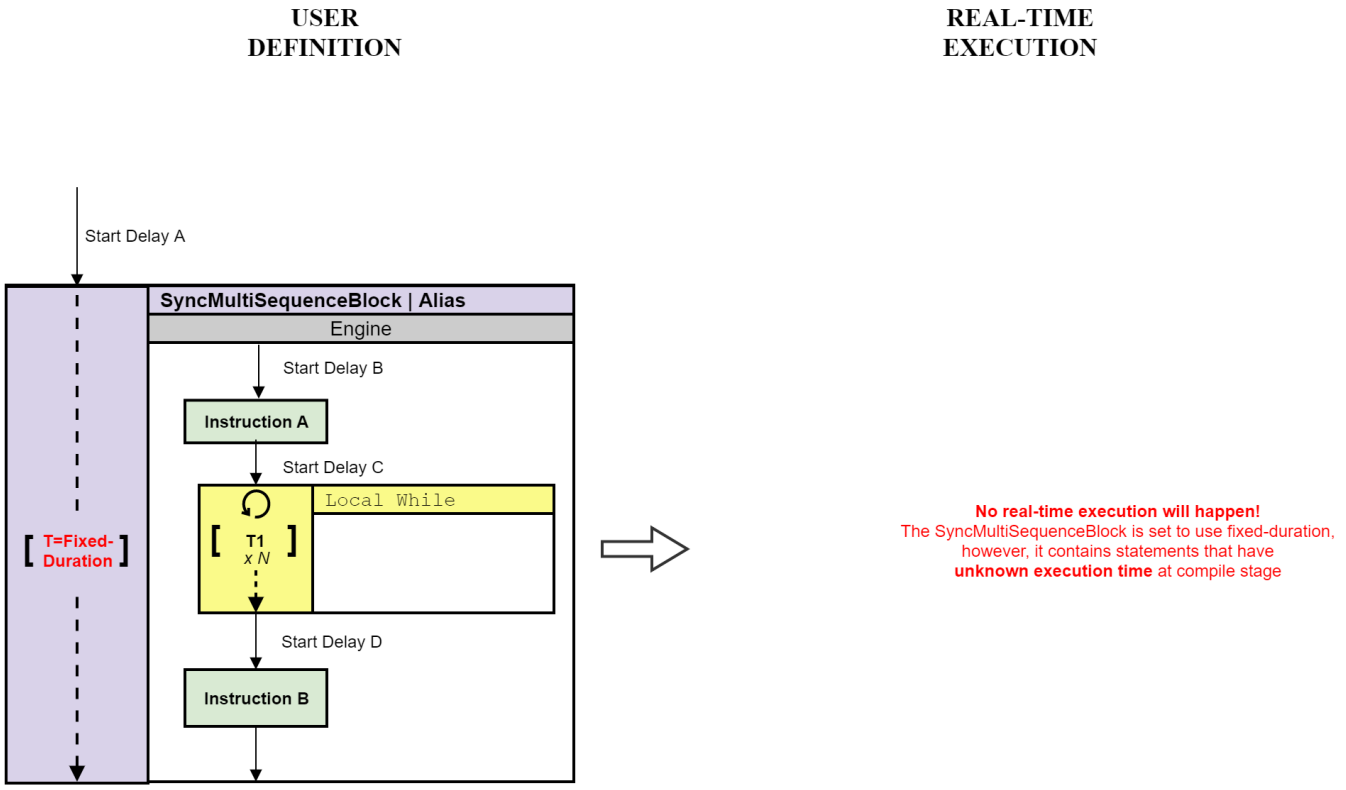
Python code for the preceding diagram:

```
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)
```

The following diagram shows a while loop:



The following diagram shows a while loop that generates an error:



Local Statement Timing

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

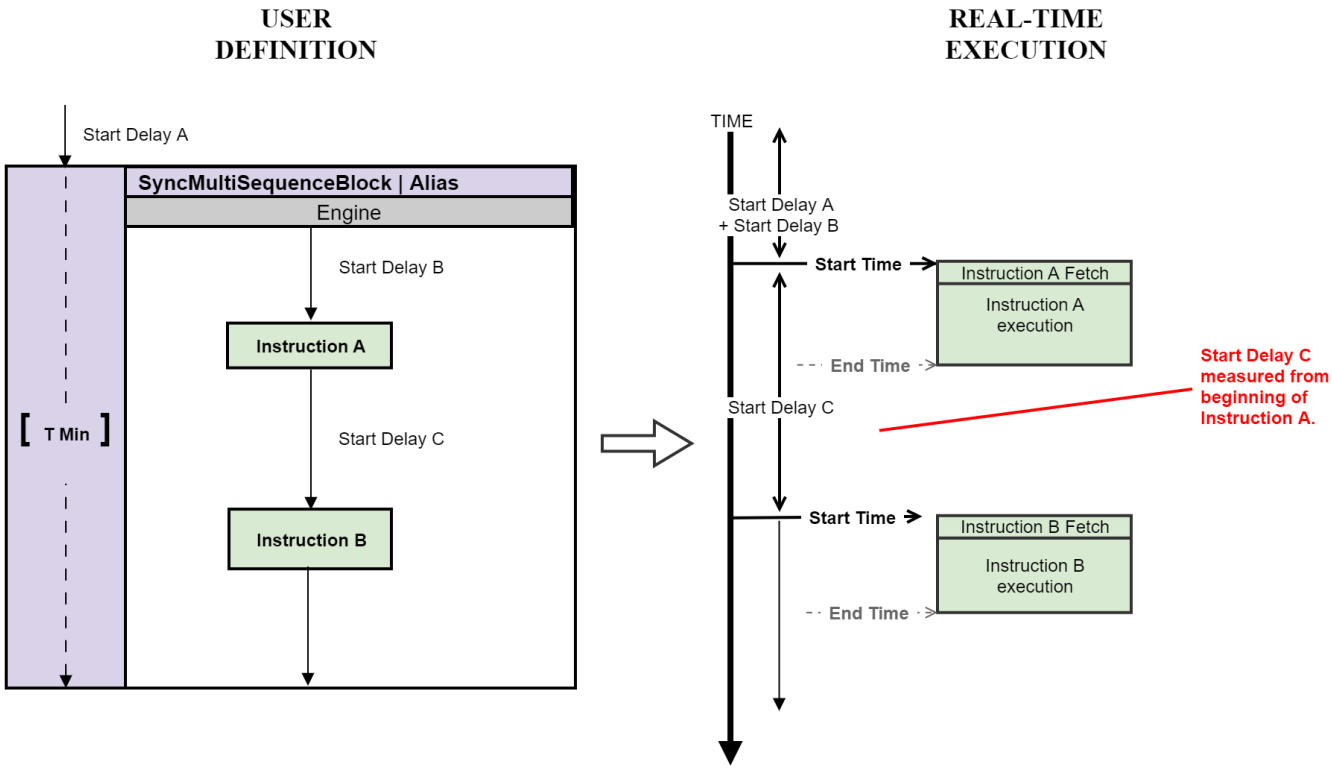
- Local Instruction Timing.
- Local Instruction Statement Parameters.
- Local Flow-Control Statement Timing.
- Calculating Local Flow-Control Statement Latency.
- Calculating Local Instruction Start Delay Requirements.

Local Instruction Timing

The following diagram shows the timing of Local instructions.

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

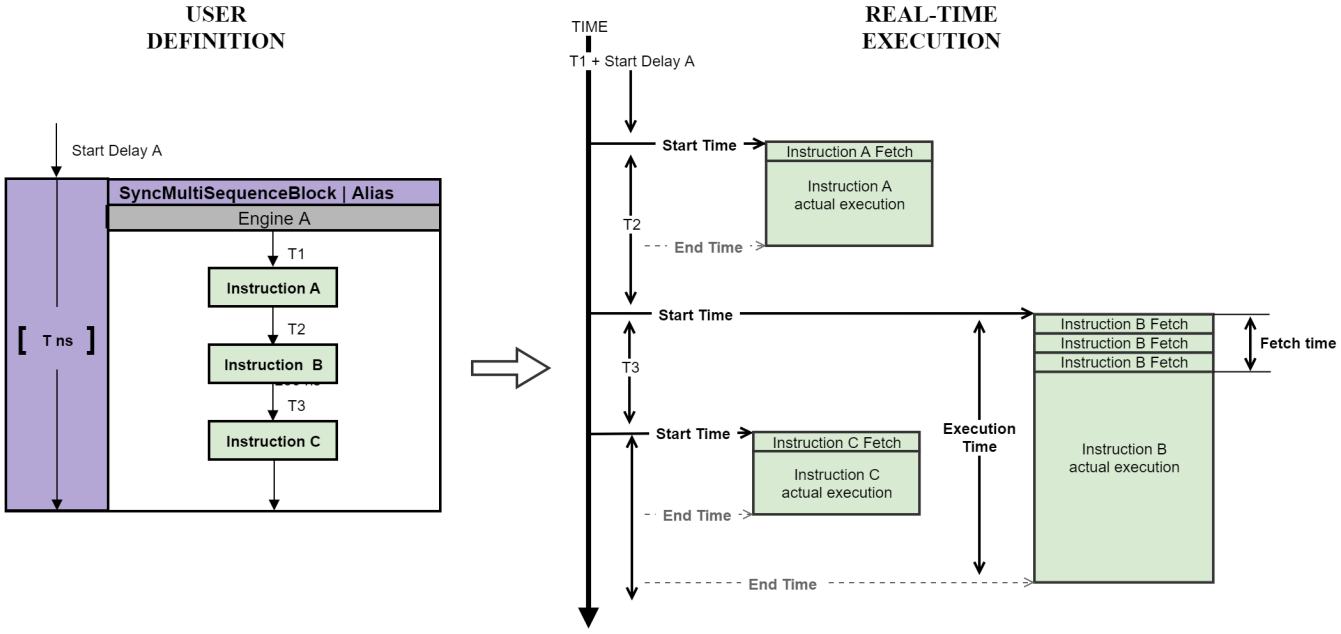
The following diagram shows two instructions and their timing:



Overlapping Instruction timing

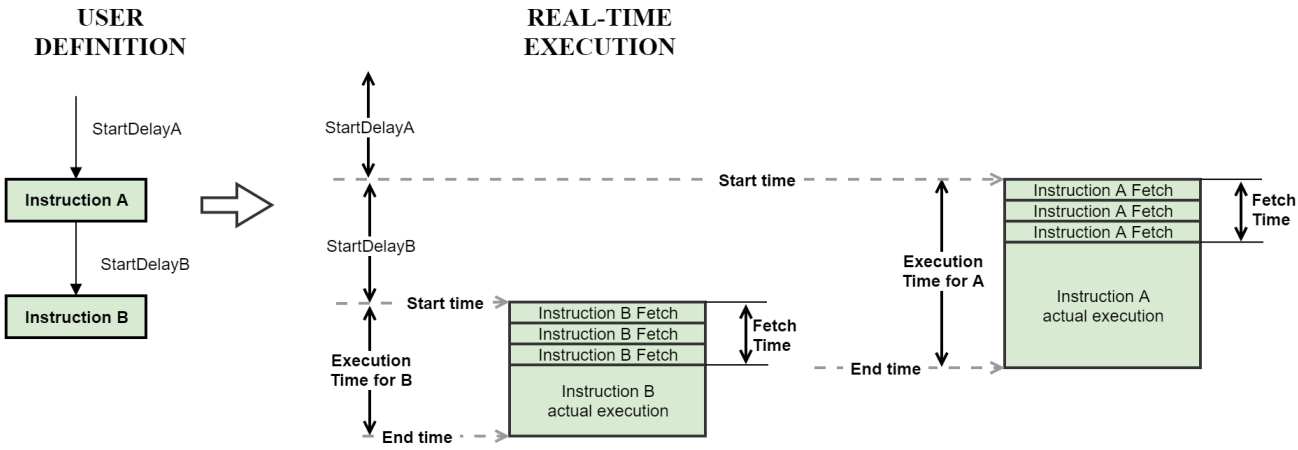
In some cases, Local instruction statements can overlap their execution. This is possible when one instruction has completed fetching data. A second instruction can then start. The fetches cannot overlap.

The following diagram shows an example. Instruction B starts execution, then after a delay of T3, instruction C starts execution while instruction B is still executing. Instruction B is still executing when instruction C finishes.



The following diagram shows an example with two local instruction statements and the timing when executed by the HVI engine.

Start delay B corresponds to the minimum time required before the result of instruction A is available to instruction B. The last execution cycle of instruction A overlaps with the last Fetch cycle of Instruction B. If the Start delay B is shorter than the result of instruction A, it will not be available by the time instruction B starts execution, for instance, in the case that instruction A modifies an HVI register, instruction B would use the previous value of the register.



NOTE It is important to consider the effects of overlapping instructions execution, because the result of the first instruction is not effective or available when the overlapping instruction finishes its fetch cycles and starts execution.

Local Instruction Statement Parameters

local instruction statements have a number of parameters and properties you must be aware of for calculating timing:

TriggerIO 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 TriggerIOs. Any number of TriggerIOs can be written at the same time.

ActionGroups

HVI actions are organized together in groups of up to 16 called ActionGroups. Any number of actions can be executed synchronously.

Instructions with dependencies

You need these to calculate the minimum start delay between instructions if there are dependencies, some instructions have a minimum start delay.

Overlapping Instructions

It is possible to overlap the execution of Local instructions. When the fetch of an instruction has completed, the next instruction can begin fetching. To calculate the the timing, in addition to the fetch time you must also know:

Actual execution time

The actual execution of a statement cannot begin until the fetch is completed. This is the actual execution time, after fetch has completed.

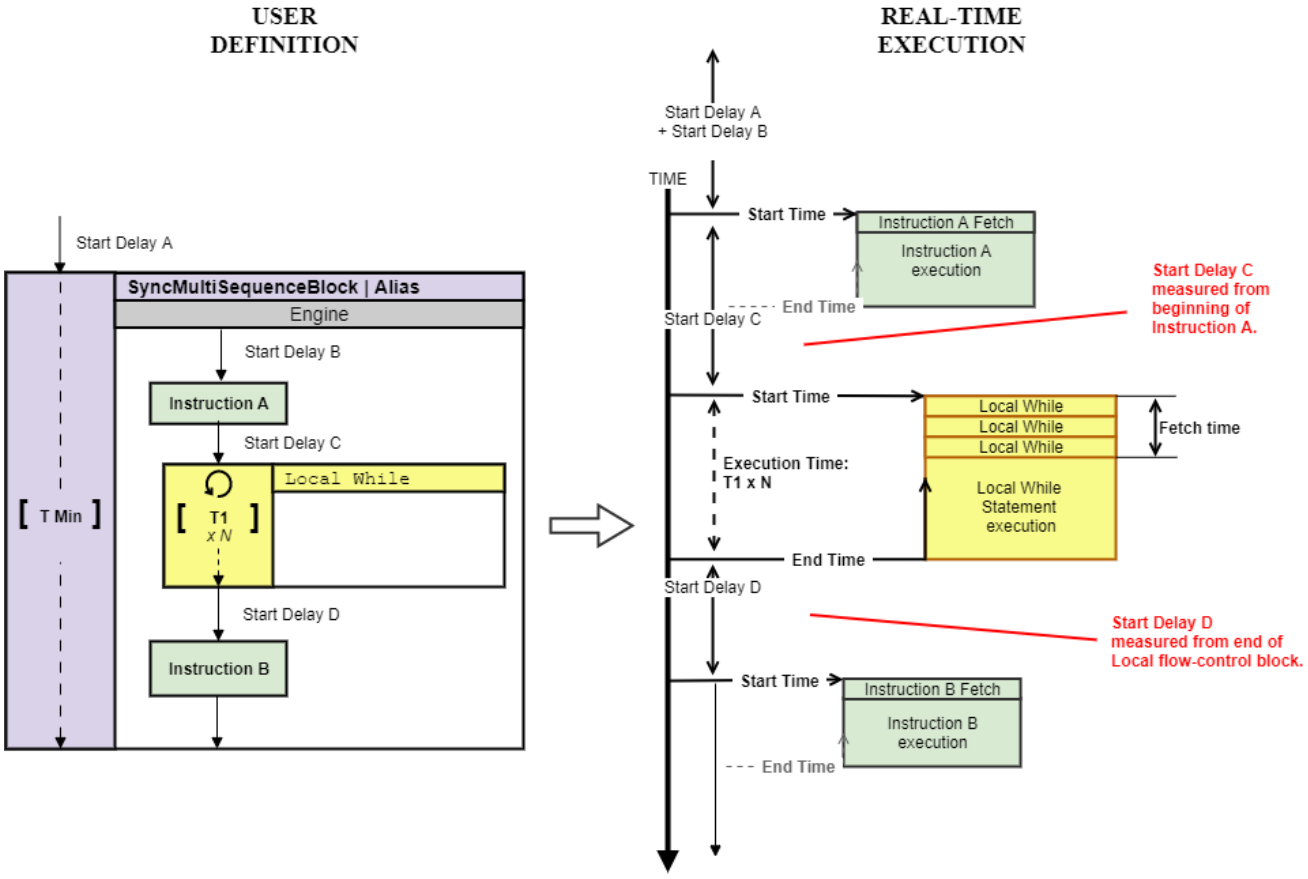
Branch matching

Branch matching is a concept used in Local If-elseif-else 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.

Local Flow-Control Statement Timing

Local flow-control 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

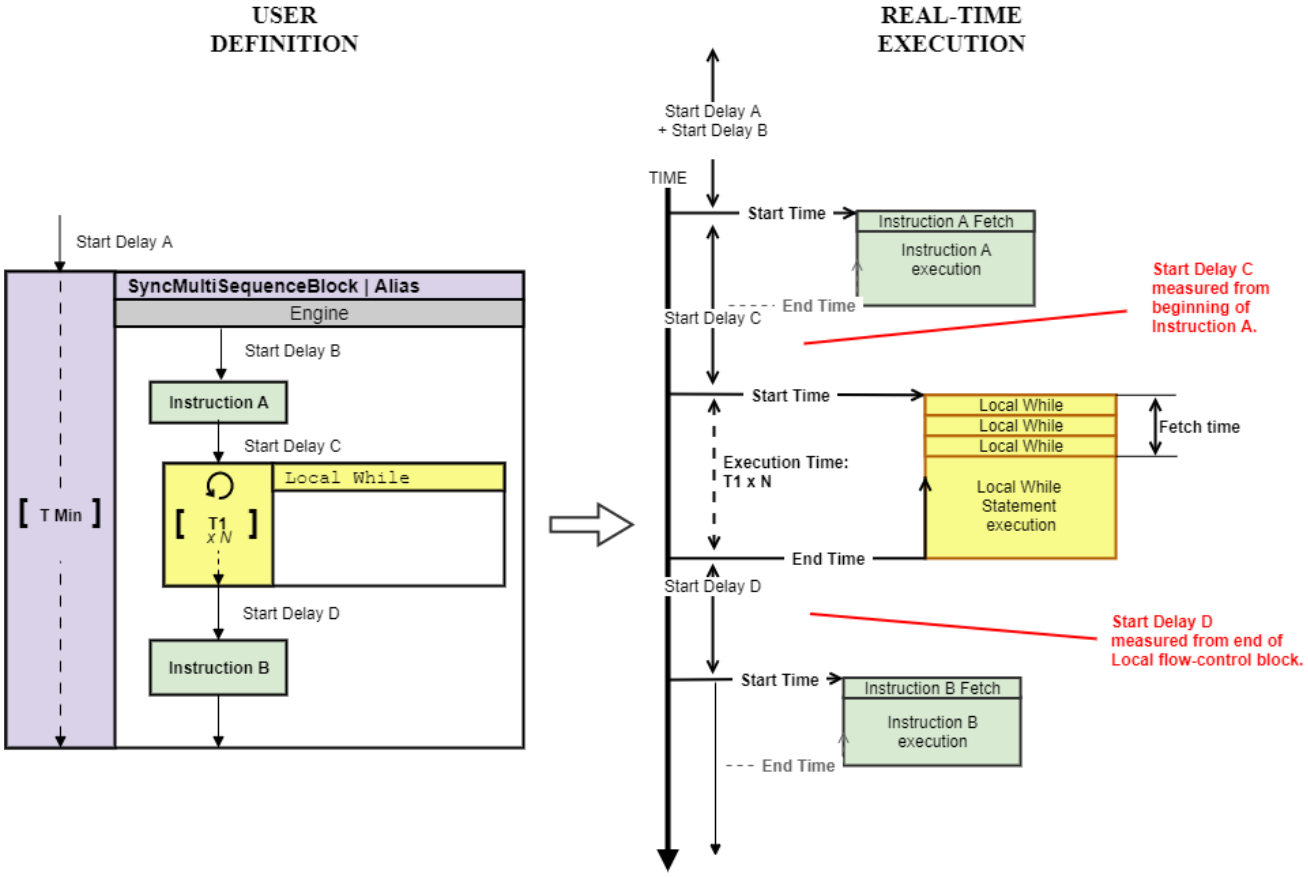
The Local while statement continues execution while a condition is met and finishes execution when the condition is no longer met. This is the same as timing of Sync while statements.

The following diagram show a Local while statement with other instructions.

The time for an iteration of 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 Local while statements, the following 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.



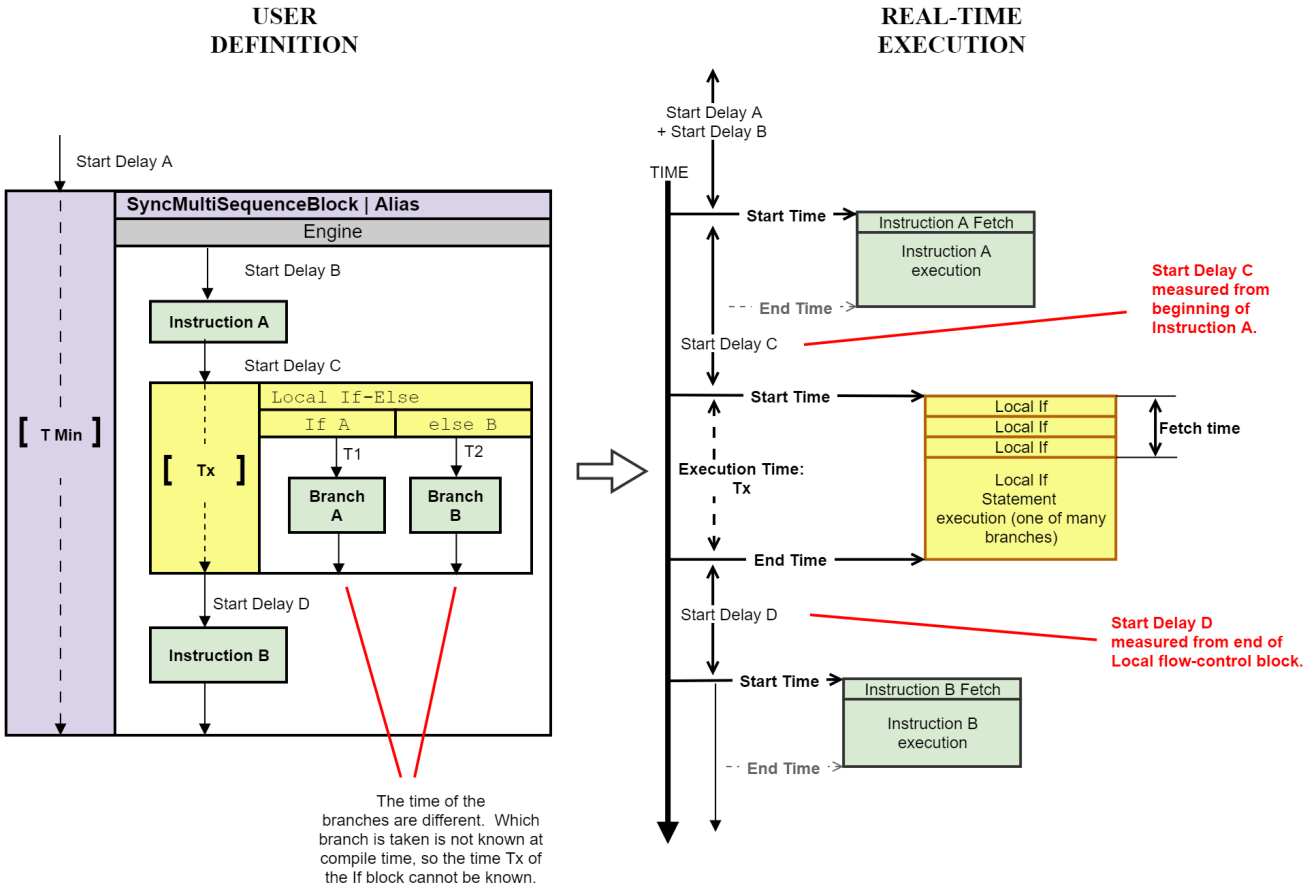
Local If

For Local if statements (also known as Local if-elseif-else), the following Start delay is measured from the end of the Local if 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.

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 TI 1 and TI 2. 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

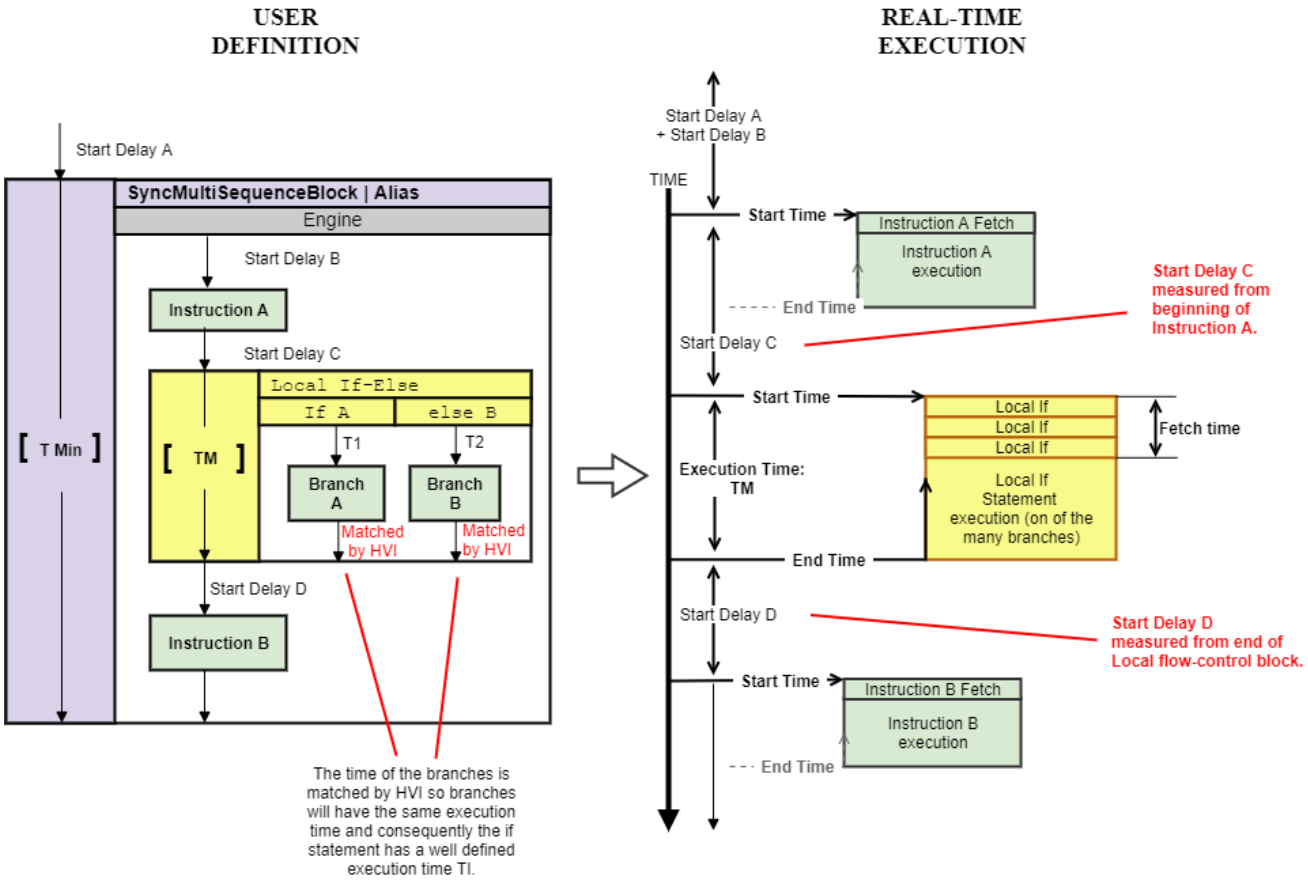
Unlike other flow-control options, the Local if statements can have different execution paths each with different times. The matched branches option enables you to control how the HVI deals with them.

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 duration of all branches is 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-else statement 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 and the containing Sync multi-sequence block are both solid.



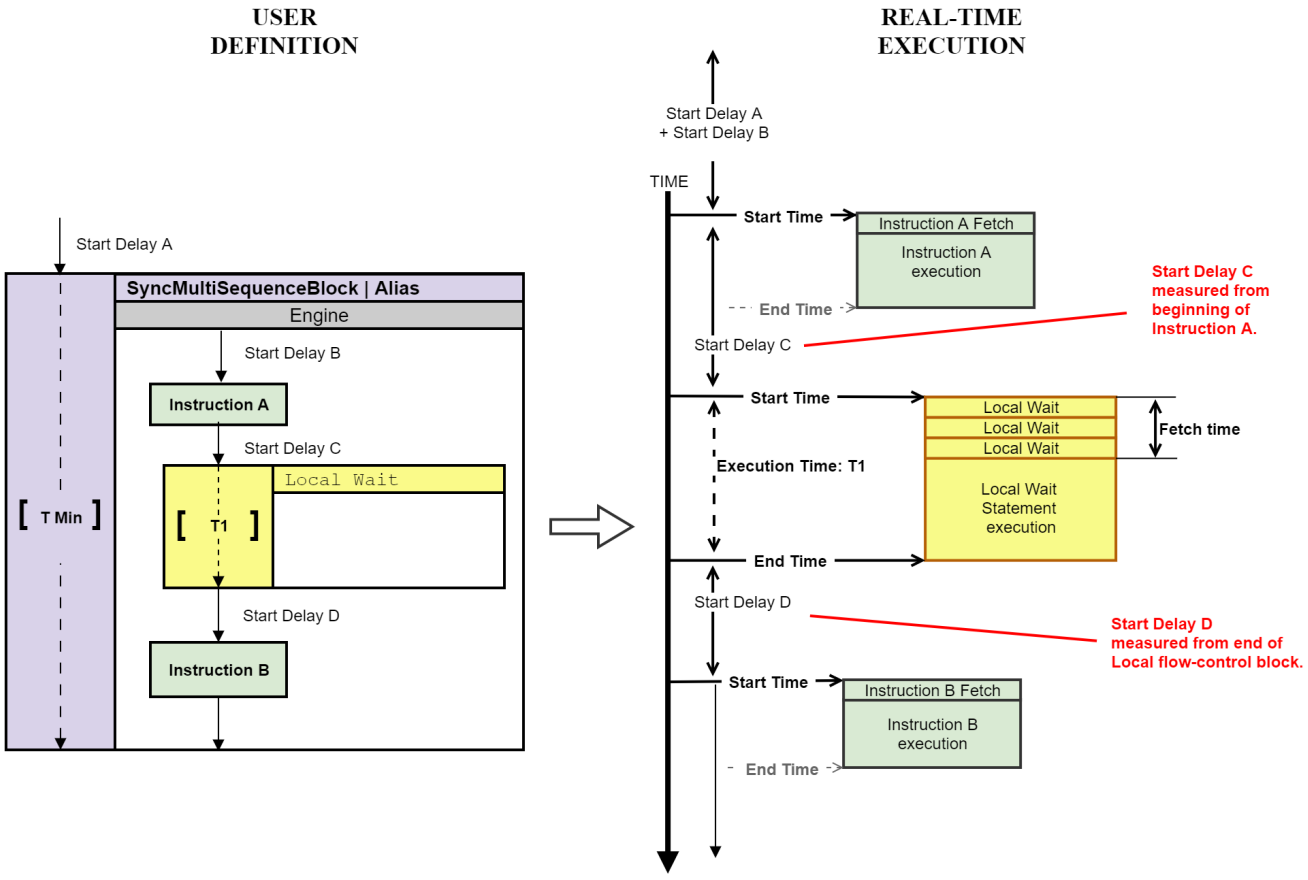
Local Wait (event or time in Register)

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.

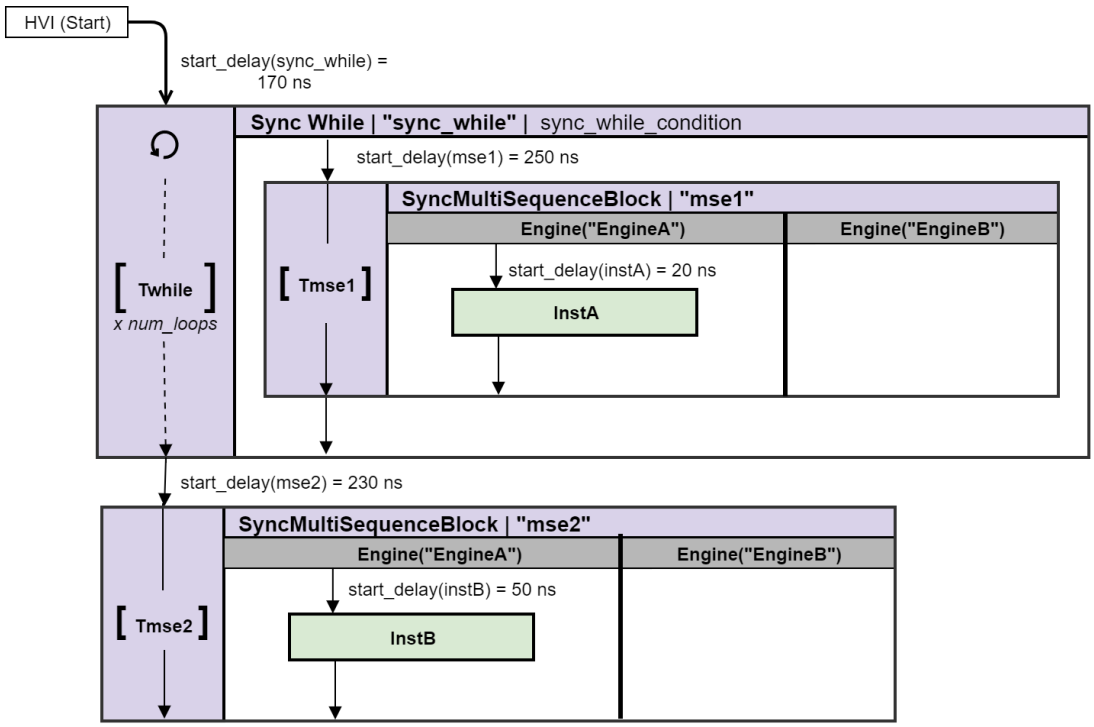
The time of the Sync multi-sequence block is indicated as T min. The dotted line indicates an unknown time.



Calculating Local Flow-Control Statement Latency

This example shows you how to calculate time 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 shows 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}  = 340ns
```

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.

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 Values* and in the instrument documentation.

When an instruction requires the result from a previous operation, the *minimum delay* (MinDelay) for the dependent, or 2nd instruction with respect to the 1st 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 maximum between the value from the equation above and the minimum start delay.

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 there is an `Add` instruction that writes 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 this value, we need to add the start-latency of the If, which, for this specific case, will be 6 cycles.

So, the minimum start-delay that the user should use to make sure that the result of the Add operation is used by the If statement will be:

- `MinDelay_If = 4 + 6 = 10 cycles`

Example 2: Add instruction inside a While Statement

In this example there is an `Add` instruction that writes a register, and the new value of the register is used for 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 specifying iteration_delay is not possible by the user, the user has to 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 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
```

Sync Statement Timing

This section describes non flow-control 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.
- Calculating Sync Flow-Control Statement Latency.
- 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 the Sync statement types Sync Register-sharing and Sync multi-sequence block is measured from the end of one Sync statement to the start of the next Sync statement.

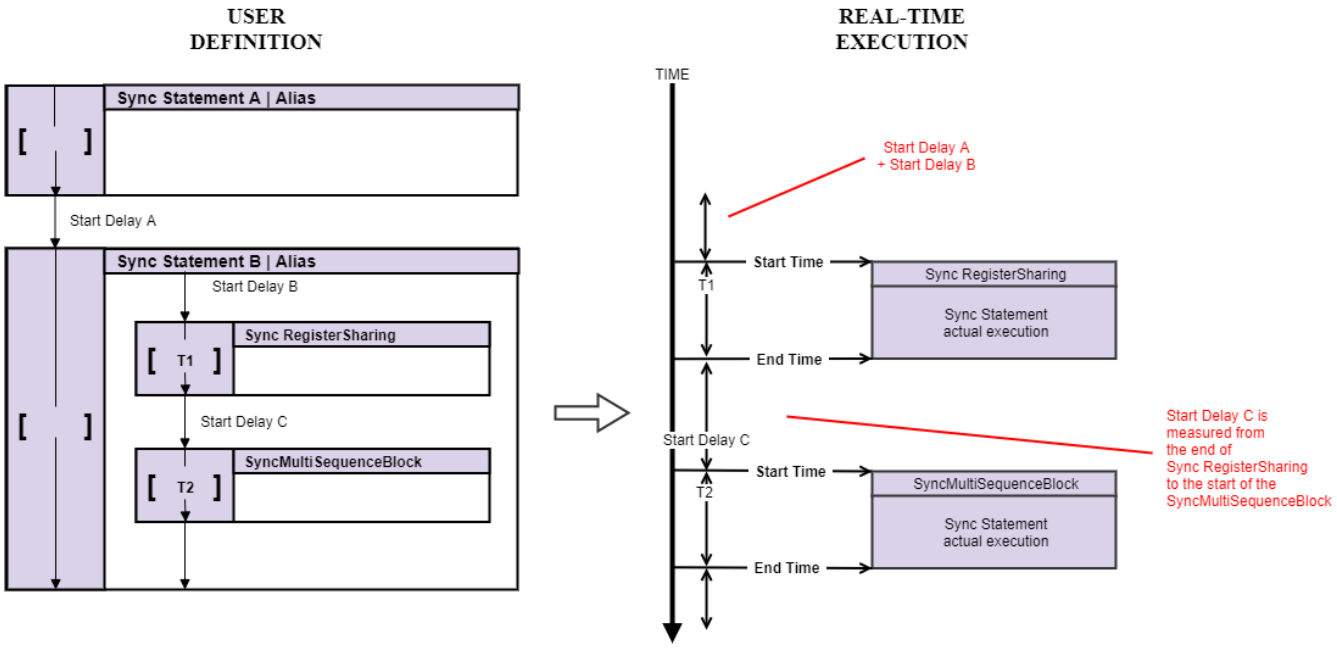
The following diagram shows the timing between a number of Sync Statements including a Sync Register-sharing statement and Sync multi-sequence block statement.

The diagram shows two Sync Statements A and B. Sync Statement B is a container for 2 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 RegisterSharing 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:



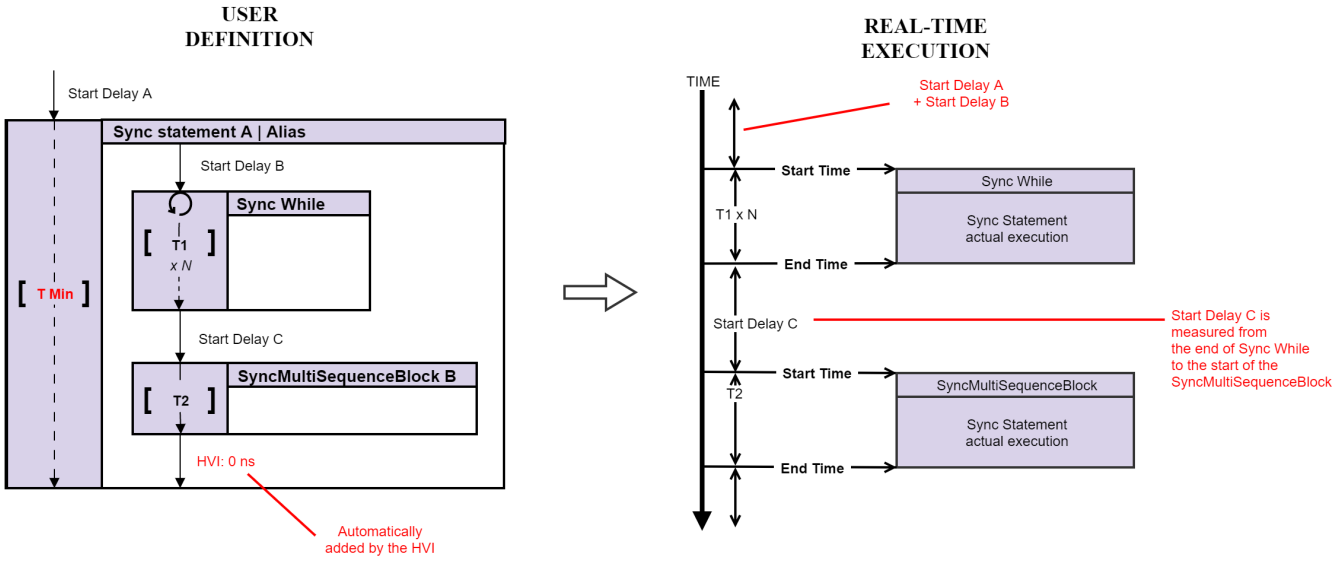
Sync While Timing

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 if it contains a flow-control statement. This is indicated by the dotted line and the time indicated as T min.

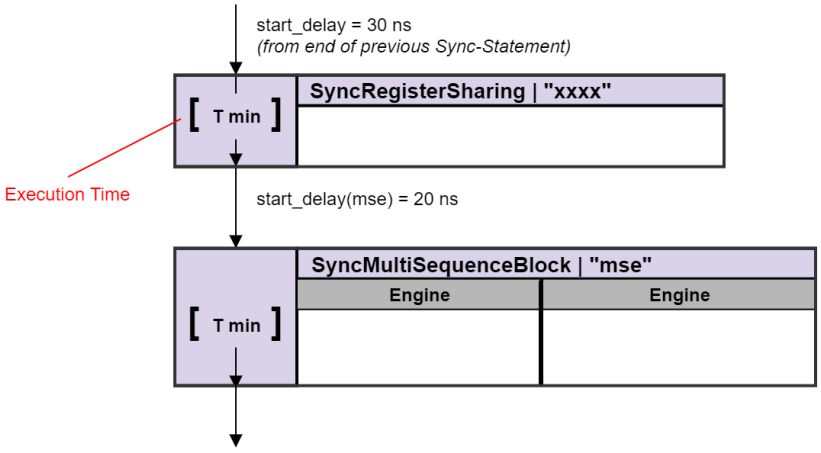
The time between the end of the Sync multi-sequence block B and the end of Sync statement A is 0 ns.



Sync Register-Sharing

The following diagram shows how the Sync register-sharing statement execution time must be accounted for when calculating the sync-sequence timing:

For the execution time see Statement Timing Tables.



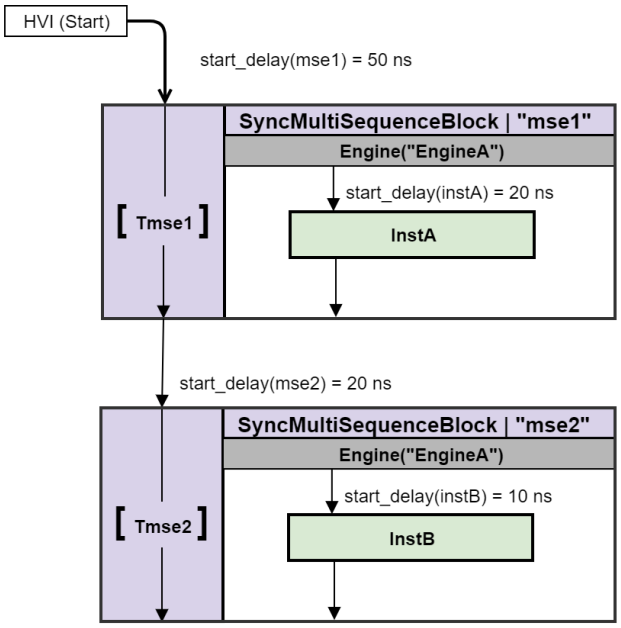
Basic Local Statement Timing Across Sync Multi-Sequence Blocks

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

Simple Local instruction timing calculation example

This example shows Local instructions within a pair of Sync Multi-sequence Blocks with the code and timing calculations:

Diagram:



Code:

```
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)
```

Timing calculations:

InstA Execution Start time from HVI-Start (`InstA_start`):

```
InstA_start = start_delay(mse1) + start_delay(instA) = 50ns + 20ns = 70ns
```

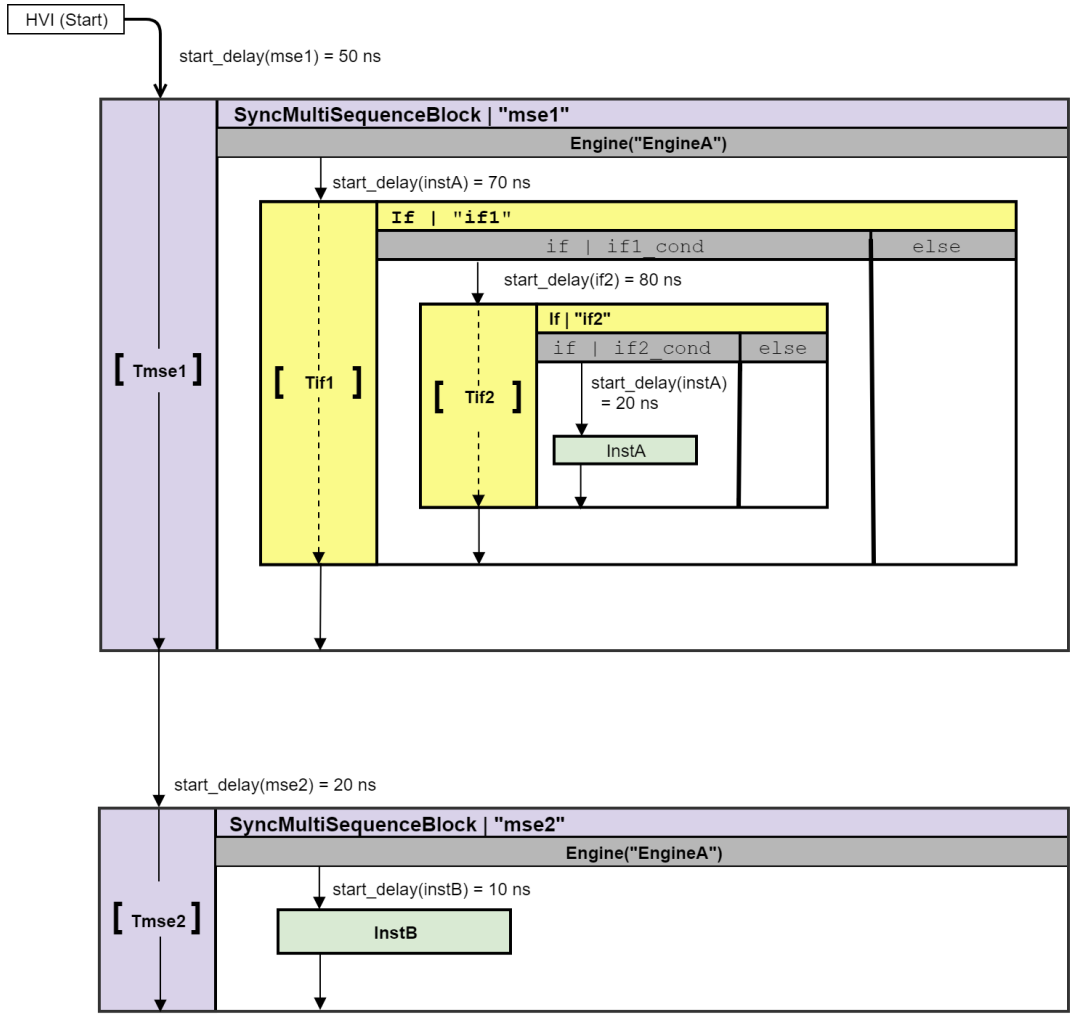
Time from InstA to InstB (`T_InstA_InstB`) :

```
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:

Diagram:



Code:

```
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)
```

Timing calculations:

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.

Sync Multi-Sequence Block Timing and Time Matching

In a synchronized multi-sequence block, you can define the statements that the HVI engines in instruments, execute in parallel with other engines.

Local sequences start and end their execution within the Sync multi-sequence block synchronously. You can define the exact start time of each Local statement after the previous one.

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 the sequences within the Sync multi-sequence block can all 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.

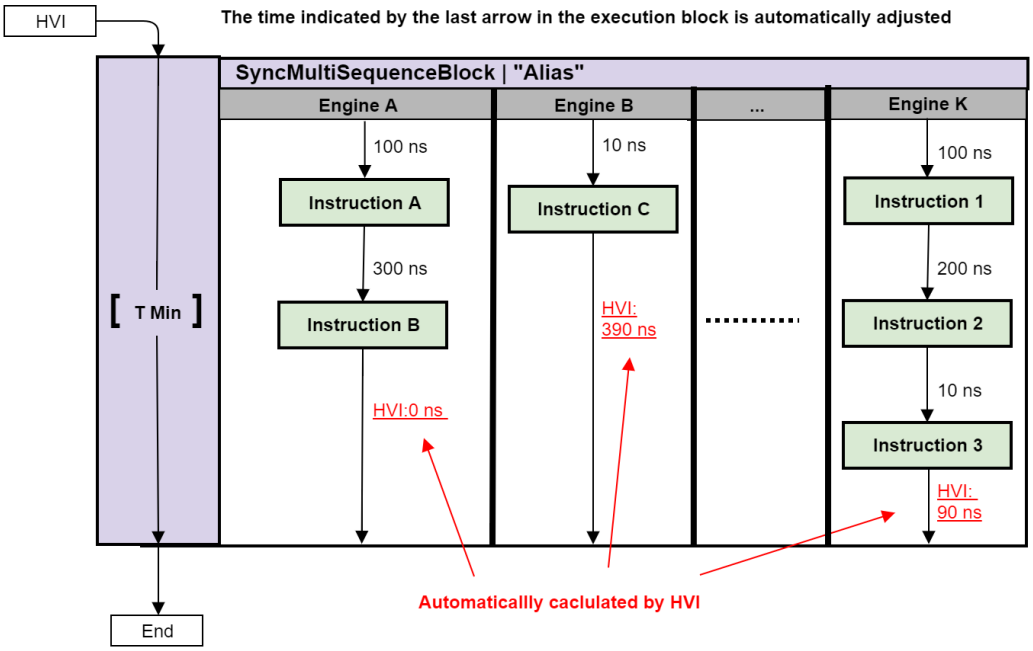
Known total local-sequence execution time

When all Local sequences contain instructions or flow-control statements with an execution time that is known at HVI compilation time, the HVI accounts for the different execution time of all local sequences during compilation and adjust the final times so each Local sequence reaches the end of the Sync multi-sequence block at the same time.

Sync multi-sequence block with minimum execution time

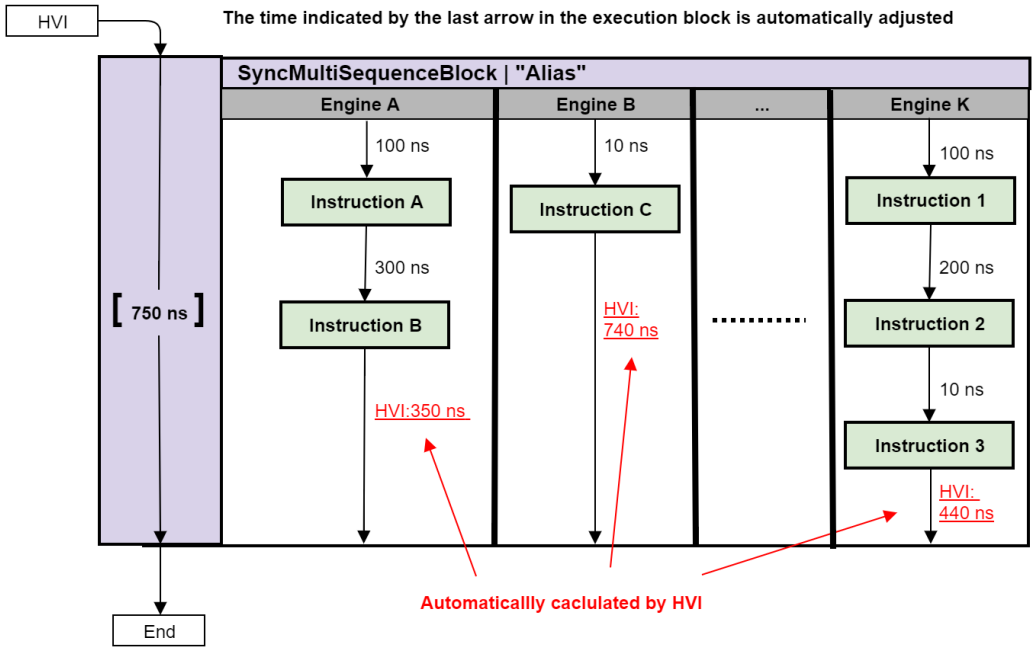
In the following diagram, the time of the Sync multi-sequence block is not specified, so the compiler will adjust the total execution time of all Local sequences to the longest one. The times of the instructions and the delays between them are known, so the timing between them and the timing of the entire 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 and 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)

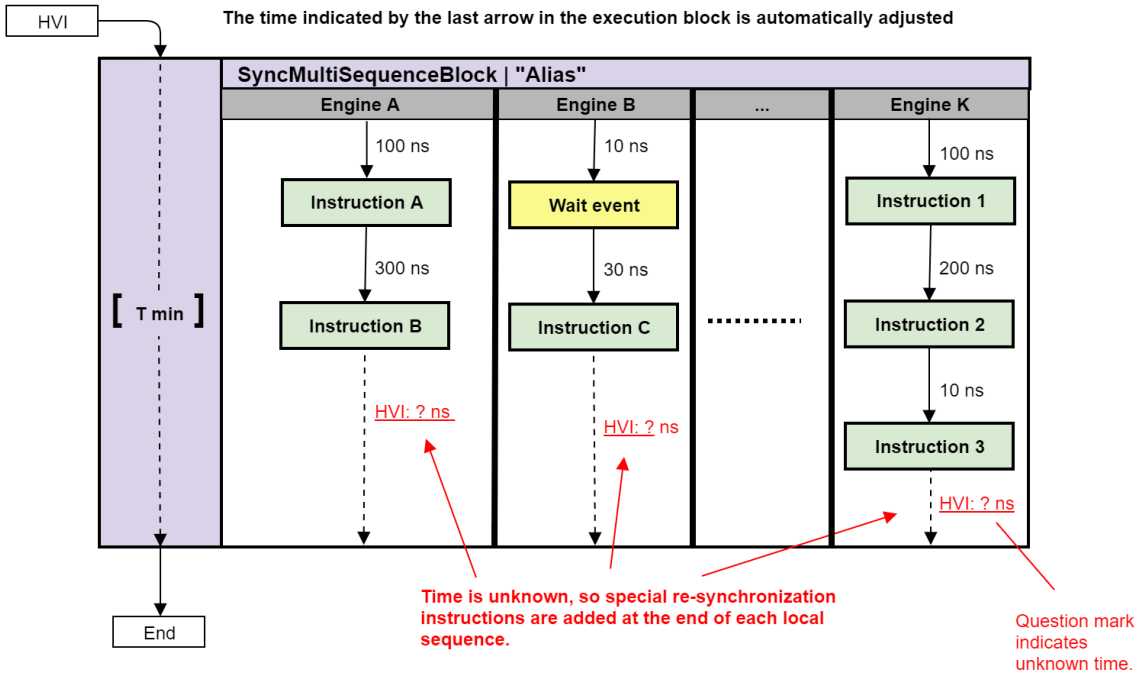
In the following diagram, the times of the instructions and the delays between them are known, so the timing between them and for the entire block can be calculated. In this case the total time is specified at 750 ns. The HVI calculates the times required for all the other engines to finish at the same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



Unknown total local-sequence execution time

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.

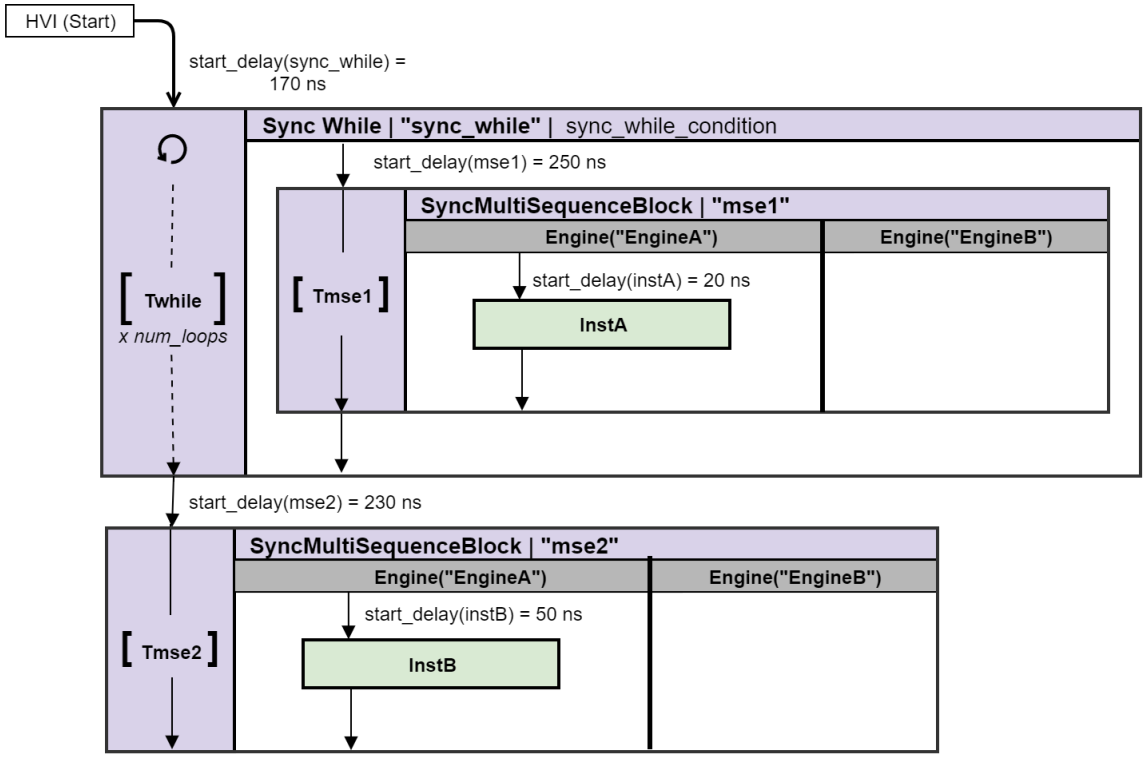
In the following diagram, the time of the instructions and the delays between them are known, except for the execution time of the Wait event. This means the execution time of the complete Sync multi-sequence block cannot be specified. HVI still enforces all Local sequences to end at the same time, but 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.



Calculating Sync Flow-Control Statement Latency

This example shows you how to calculate time 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 shows 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}  = 340ns
```

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.

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. This occurs in the following cases:

- At the start of the HVI Sequence (Main Sync Sequence).
- At the end of a Sync multi-sequence block statement, where there is one or more statements with unknown execution time at compile time, in at least one of the local sequences inside the Sync multi-sequence block. Possible cases are:
  - A WaitTime statement with a Register defining the wait time at runtime.
  - A Wait statement for an event.
  - A While loop statement.
  - An If statement, with unmatched branches that take different execution times.

Timing with Triggered-Sync Points

Triggered-Sync points require the use of trigger resources assigned in the `SyncResources` property in the `SystemDefinition` instance and the main Sync signal. This is to re-synchronize all HVI engines, and guarantee all engines then continue execution after the Sync Point at exactly the same point. This 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.

The Sync-Period uses this equation:

```
100ns * #chassis
```

Triggered-Sync Delay

The delay that a triggered-sync point adds to the sequence timing has four parts. Two of them are constant and the other two varies depending on the last statement and its position compared to the Sync Pulse time. The formula to calculate it is the following:

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

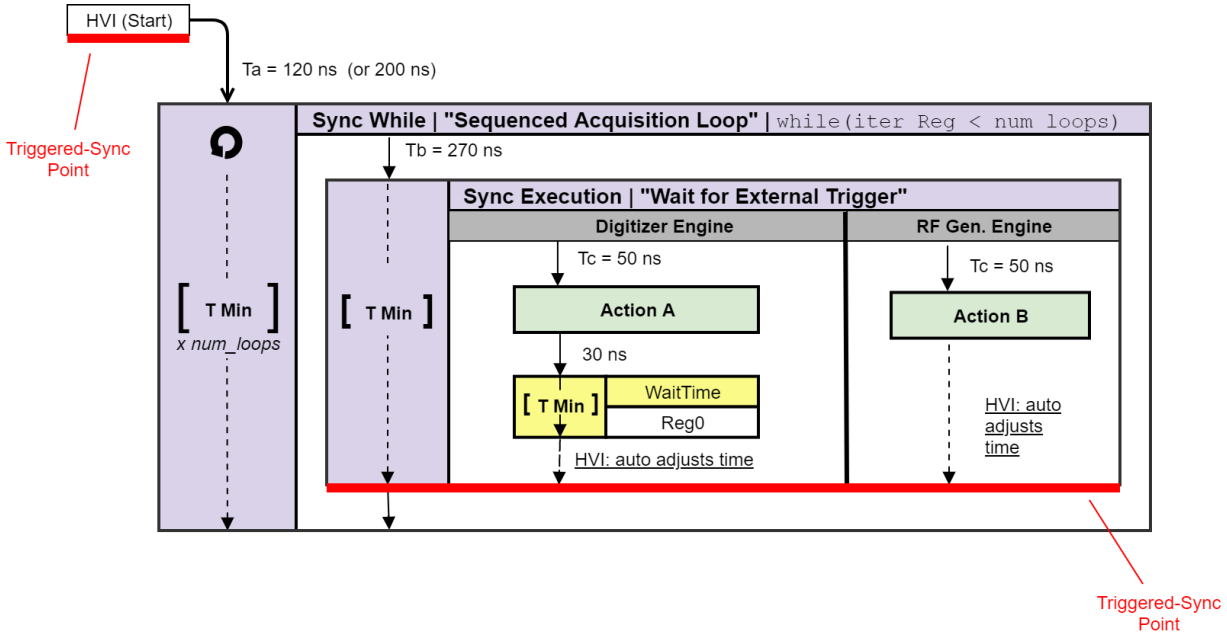
where:

- `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 their Fetch-Time.
- `sync_overhead` is constant per module. The value is typically 5 cycles.
- `edge_offset` is the time interval from the end of the `sync_overhead` to the sync-pulse rising 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 earlier.

Example of timing management with triggered-sync

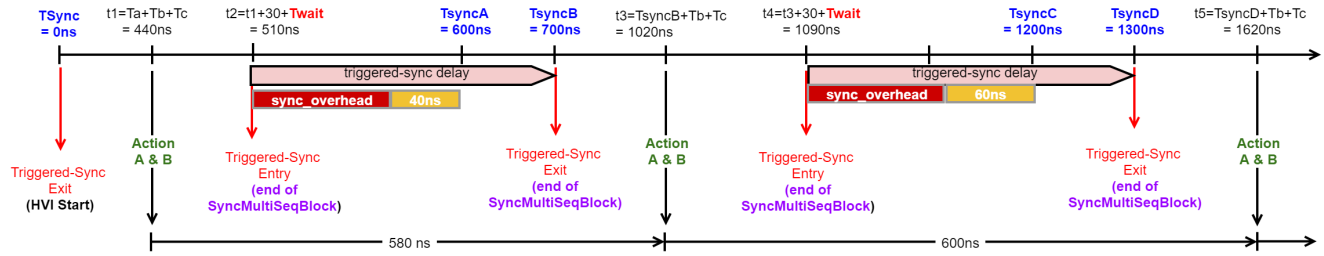
The example below is a simple sequence where the triggered-sync points has been marked in red. There are two triggered-sync points, the HVI start which is always present, and a second one at the end of the Sync multi-sequence block. The second sync point is required because there is a WaitTime with a register inside and the time for this cannot be determined at compile time.

In the following diagram there is 1 chassis, this means that the Sync period is 100 ns, based on the previous equation.

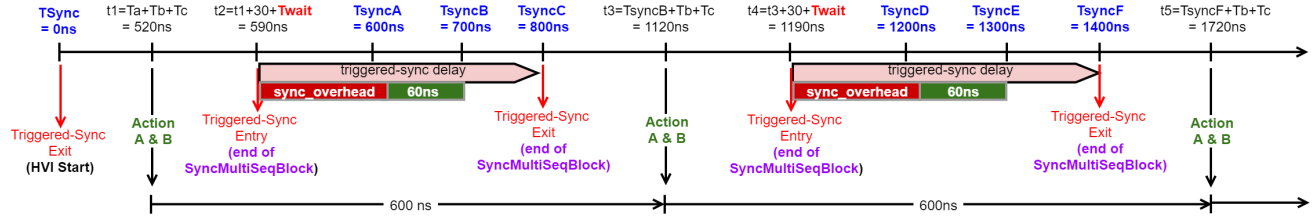


The following diagram shows the execution timeline for the first 3 iterations of the sequence shown in the previous diagram:

Ta = 120
Tb = 270
Tc = 50
Twait = 40ns = 10ns * [Reg0 Value] (assume Reg0=4)
sync_period = 100 ns
sync_overhead = 50 ns



Ta = 200 **<= Need to adjust it to match the triggered-sync delay in the first loop execution w.r.t the other loop iterations**
Tb = 270
Tc = 50
Twait = 40ns = 10ns * [Reg0 Value] (assume Reg0=4)
sync_period = 100 ns
sync_overhead = 50 ns



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:

- If 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.
- If 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.

Statement Timing Tables

This section lists the timing values, it contains the following sections:

- Sync Statement Timing Values
- Sync Flow-Control Statement Timing Values
- Local Flow-Control Statement Timing Values
- Local Instruction Statement Timing Values
- Instrument-Specific Local Instruction Statement Timing Values

Sync Statement Timing Values

This section provides timing values for Sync statements.

Timing values for Sync Register-Sharing

Sync Register-sharing latency does not depend on the number of bits shared. For more information on this functionality, see HVI Statements and HVI API Sync Statements.

Execution time (cycles) (1)	Fetch time (Primary Module, cycles)
`5 + Propagation_delay_cycles` (2)	1

(1) *The values provided here apply for the case that the duration property of the respecting statement is set to Minimum (default). If a fixed-duration has been set, then this is the Execution Time is equal to that value.*

(2) *Propagation_delay_cycles* = `100ns * clock_frequency * #chassis`

Latency values for Sync Register-sharing statement:

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

(1) *Propagation_delay_cycles* = `100ns * clock_frequency * #chassis`

Sync Multi-Sequence Blocks

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) + $End\text{-}Latency_{Last\text{-}statement}$ (2)	N/A

(1) *The values provided here apply for the case that the duration property of the respecting statement is set to Minimum (default). If a fixed-duration has been set, then this is the Execution Time is equal to that value.*

(2) *The values are calculated only for the branch that is being executed, when there are multiple branches available.*

Latency values for Sync Multi-Sequence Blocks:

Description		Time (cycles)
Start-Latency		0
Minimum start-delay for statement		
Entry- latency		1
Minimum start-delay for first statement inside any of the contained sequences		
End-Latency	timed-sync (1)	$End\text{-}Latency_{Last\text{-}statement\text{-}of\text{-}longest\text{-}branch}$ (1)
Minimum start-delay for the next statement)	**Minimum Duration**	(1) When sequence is empty, the value is constant 1
	timed-sync (1)	1
	Fixed Duration	
	triggered-sync (1)	1
Fixed-Duration		$[max_{for\_all\_Branches}[Branch\text{-}Duration] - 1]$ (1), where Branch-Duration is calculated as follows: $sum_{for\_all\_internal\_statements}(Start\text{-}Delay) + sum_{for\_all\_internal\_flow\_control\_statements}(Duration) + End\text{-}Latency_{Last\text{-}statement}$ (1) In the specific case where all the branches are empty, then the duration is equal to: 0

Sync Flow-Control Statement Timing Values

This section provides timing values for Sync Flow-control statements.

NOTE If the last statement contained in a flow-control statement is a Local instruction, the Fetch latency is used as the End-Latency.

HVI Start

HVI start basic timing value:

Parameter	Description	Time (cycles)
End-Latency	Minimum start-delay for first statement in HVI	3

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) + End-Latency$_{Last-statement}$]	3 + #Register_Conditions

(1) *The values provided here apply for the case that the duration property of the respecting statement is set to Minimum (default). If a fixed-duration has been set, then this is 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-Latency$_{Last-statement}$
		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-Latency$_{Last-statement}$
		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-Latency$_{Last-statement}$](3)

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

(2)*Propagation_delay_cycles* = `100ns * clock_frequency * #chassis`

(3) *In the specific case where the branch is empty, then the duration is equal to:* `8 + #Register_Conditions`

Local Flow-Control Statement Timing Values

This section provides timing values for Local Flow-control statements.

NOTE If the last statement contained in a flow-control statement is a Local instruction, the Fetch-latency is used as the End-Latency.

Local Wait Time

Local Wait Time statement blocks HVI execution in a Local sequence until a specific amount of time passes.

Local wait time timing value:

Instruction	Execution time (cycles) (1)	Fetch time (Primary Module, cycles)
Local Wait time (Register)	RegisterValue	1

(1) *The values provided here apply for the case that the duration property of the respecting 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 WaitTime latency values:

Parameter	Description	Time (cycles)
Start-Latency	Minimum start-delay for the statement	1
End-Latency		1

Local Wait (Event)

Wait statement blocks HVI execution in a Local sequence until an event occurs.

Local wait (event) timing value:

Execution time (cycles) (1)	Fetch time (Primary Module, cycles)
EventArrivalTime + 4	1

(1) *The values provided here apply for the case that the duration property of the respecting 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 Wait latency values:

Parameter	Description	Time (cycles)
Start-Latency	Minimum start-delay for the statement	0
End-Latency		1

Local If

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 branches has 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)
$[sum_{for\_all\_internal\_statements}\ (Start\text{-}Delay) + sum_{for\_all\_internal\_flow\_control\_statements}\ (Duration) + End\text{-}Latency_{Last\text{-}statement}]$ (3)	3 + #Register_Conditions

(1) *The values provided here apply for the case that the duration property of the respecting statement is set to Minimum (default). If a fixed-duration has been set, then this is the Execution Time is equal to that value.*

(2) *These values are calculated only for the branch that is executed, if there are multiple branches available.*

(3) *If the branch is empty, the execution time becomes* $Entry\text{-}Latency_{branch} - 1$.

The following tables shows Local If latency values:

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 Contribute to the minimum-possible start-delay of next statement outside the if statement	**Matching Branches** *disabled*	`3 + max`$_{for\_all\_Branches}$`[End-Latency`$_{Last\text{-}statement}$`]` (1) (1) In the specific case where the maximum end latency used in the equation above correspond to the if-branch, and the calculated latency is greater than 4 then the actual **End-Latency** is the calculated value minus 1.
	Matching Branches *enabled*	`3 + End-Latency`$_{Last\text{-}statement\text{-}of\text{-}longest\text{-}branch}$ (1) where longest branch, means the branch with longer execution time. (1) In the specific case where the longest branch is the if-branch, the actual **End-Latency** is the calculated value minus 1.
	Fixed-Duration	`1`
Fixed-Duration		`2 + m ax`$_{for\_all\_Branches}$`[Branch-Duration]` (1) where Branch-Duration is calculated as follows: `[sum`$_{for\_all\_internal\_statements}$ `(Start-Delay) + sum`$_{for\_all\_internal\_flow\_control\_statements}$ `(Duration) + End-`

$Latency_{Last\text{-}statement}$] (2)

(1) In the specific case where the maximum branch duration is used in the equation above correspond to the if-branch, then the actual Duration is the calculated value minus 1.

(2) In the specific case where a branch is empty, then the branch duration is equal to the Entry-Latency of the branch.

Local While

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) + End-$Latency_{Last\text{-}statement}$]	2 + #Register_Conditions

(1) *The values provided here apply for the case that the duration property of the respecting 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:

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) *In the specific case where the branch is empty, then the duration is equal to the Entry-Latency of the branch.*

Local Instruction Statement Timing Values

The following sub-sections list the fetch and execution latency for HVI-native Local instructions statements. Unless stated otherwise, all times are expressed 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 the instrument-specific documentation.

Write Trigger

Any number of TriggerIOs can be written at the same time. TriggerIOs are organized in groups of 16. The Fetch time of the instruction depends on the number of different TriggerIO groups included in the instruction (#TriggerIOGroups). See the instrument documentation for information about instrument TriggerIO definitions.

Write trigger basic timing values:

Instruction	Execution time (cycles)	Fetch time (cycles)
TriggerWrite	1 + Ceil(#TriggerIOGroups/2)	Ceil(#TriggerIOGroups/2)

NOTE Trigger execution time is instrument specific. For trigger execution timing information, see your instrument documentation.

Action Execute

The action-execute HVI instruction synchronously executes a list of HVI actions defined by the user. HVI actions are organized in groups 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 to which each action specified by the user belongs. 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.

Action execute basic timing values:

Instruction	Execution time (cycles)	Fetch time (cycles)
ActionExecute	2 + Ceil(#ActionGroups /2)	Ceil(#ActionGroups /2)

NOTE Action execution timing is instrument specific. For action execution timing information, see your instrument documentation.

Arithmetic Logic Unit Instructions

Arithmetic Logic Unit (ALU) instruction 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, see the instrument documentation for the specific value of `HVI_FPGA_ProductDelay`:

NOTE

- Consecutive FPGA read instructions must be issued with at least 1 cycle of delay between them.
- If an FPGA instruction that uses an HviRegister is issued before an FPGA instruction that does not use an HVI Register, the delay between both instructions must be at least 3 cycles.

FPGA user sandbox operations basic timing values:

Instruction	Execution time (cycles)	Fetch time (cycles)
fpga_array_read	2 * HVI_FPGA_ProductDelay + 4	1
fpga_array_read (Address from HviRegister)	2 * HVI_FPGA_ProductDelay + 6	1
fpga_array_write	HVI_FPGA_ProductDelay + 2	1
fpga_array_write (Address or data from HviRegister)	HVI_FPGA_ProductDelay + 4	1
fpga_register_read	2 * HVI_FPGA_ProductDelay + 4	1
fpga_register_write	HVI_FPGA_ProductDelay + 2	1
fpga_register_write (Address or data from HviRegister)	HVI_FPGA_ProductDelay + 4	1

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

This appendix lists the instruments that support PathWave Test Sync Executive release 2020.

PathWave Test Sync Executive is a new generation of Hard Virtual Instrument (HVI) technology and is not backward compatible with the previous generation. The previous generation of HVI technology is only programmable by M3601A Hard Virtual Instrument Design Environment and is not forward compatible with the new generation of HVI technology or PathWave Test Sync Executive.

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. The following table lists the firmware version requirements for the older M3601A, PathWave Test Sync Executive 2020 and PathWave Test Sync Executive 2020 Update 1.

M3000 Series Firmware Version Requirements

Instrument	M3601A	PathWave Test Sync Executive 2020	PathWave Test Sync Executive 2020 Update 1
M3100A Digitizer	< 2.00	2.00.30	≥ 2.01.02
M3102A Digitizer	< 2.00	2.01.40	≥ 2.01.60
M3201A AWG	< 4.00	4.02.65	≥ 4.02.85
M3202A AWG	< 4.00	4.00.95	≥ 4.01.20
M3300A AWG & Digitizer Combination	< 4.00	4.01.32	≥ 4.01.53
M3302A AWG & Digitizer Combination	< 4.00	4.00.31	≥ 4.01.02

NOTE To obtain the latest firmware version recommended for use with M3601A or PathWave Test Sync Executive 2020. Ensure you check the specific product pages at Keysight PXI Products.

M3000 Series Software Version Requirements

The M3000 series (SD1) software provides drivers, programming libraries and soft front panels for the M3000 series. There are similar version requirements that are also shown in the table below.

Description	M3601A	PathWave Test Sync Executive 2020	PathWave Test Sync Executive 2020 Update 1
All M3000 series instruments	< 3.00.00	3.00.95	≥ 3.01.10
listed above			

NOTE To obtain the latest firmware version recommended for use with M3601A or PathWave Test Sync Executive 2020. Ensure you check the specific product pages at Keysight PXI Products.

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 in the tables above. SD1 software is available at Keysight SD1 Software. Firmware is available at Keysight PXI Products, on the **Technical Support** page for your specific instruments, see the **Drivers, Firmware & Software** tab.

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 Ppnel 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 (AWGs) to synchronously generate mixed signals. Each instrument can be programmed to output either a Front Panel (FP) 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 of 10ns.

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 PathWave Test Sync Executive (KS2201A).