ASAM AE HIL

Application Programming Interface for ECU Testing via Hardware-in-the-Loop Simulation

Part 1 of 4

Programmers Guide

Version 1.0.0

Base Standard

Association for **S**tandardisation of **A**utomation and **M**easuring Systems

Dated:23.07.2009

Status of Document

Date:	23.07.2009
Author:	ASAM HIL workgroup
Version:	Version 1.0.0
Doc-ID:	
Status:	
Type	Base Standard

Revision History

This revision history shows only major modifications between release versions.

Date	Author	Filename	Comments

Table of contents

1 MOTIVATION

1.1 WHAT IS HARDWARE-IN-THE-LOOP SIMULATION ?

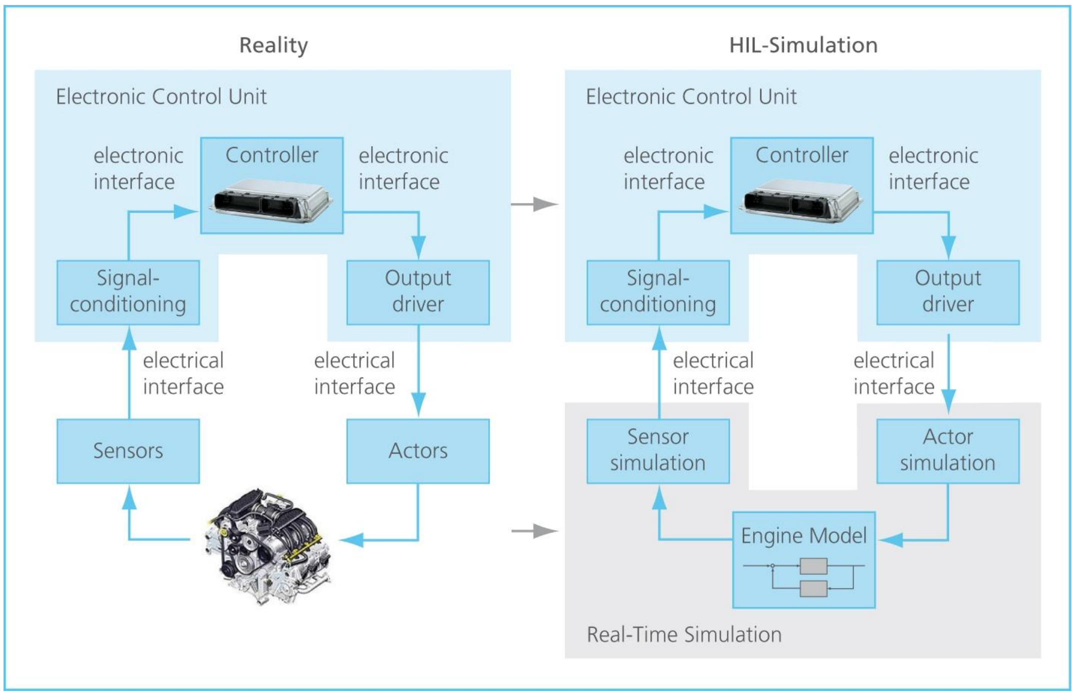


Figure 1 Principle of Hardware-in-the-Loop Simulation

Hardware-in-the-Loop (HIL) simulation has become a well-established verification technology applied in many ECU development projects today.
By means of HIL technology function tests can be shifted to earlier development stages to increase the maturity of new software and/or electronics components.
Cost and time expensive test drive cycles which have been performed in former times directly in vehicle or on conventional test benches can be substituted by simulation based operations.
Tests of failure situations or tests of dangerous maneuvers can be shifted into the computer, at least in parts of the complete test program.
The major advantage is the capability to automate these test benches. This allows to reproduce all test cycles and to operate these test benches 24 h per day.

A closed control loop of today's automotive electronic system as shown in the left part of Figure 1 (Controller, output driver, actors, plant, e.g. an engine, sensors and the input side signal conditioning) is substituted in parts. The electrical interfaces are retained. Sensors and actors are either replaced by full simulated versions or they are even attached as original physical load component in the test bench setup.
The plant part of the control loop, i. e. in this example the engine, is replaced completely by a simulation model, which can be calculated in the appropriate model precision in real-time.

1.2 A TYPICAL HIL TESTBENCH

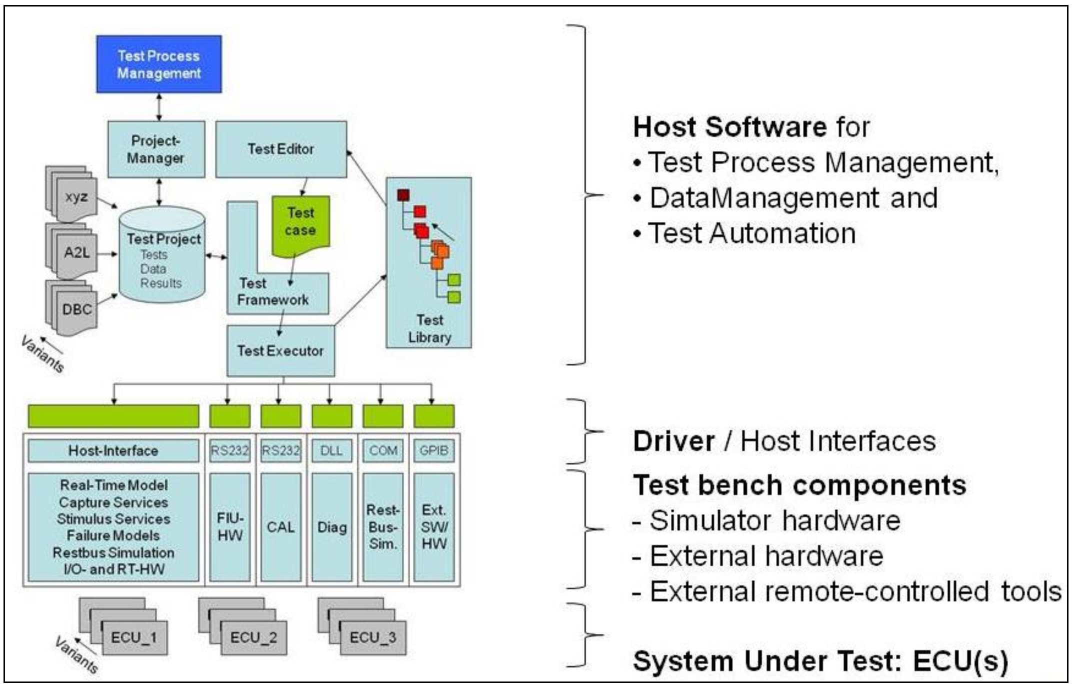


Figure 2 HIL Testbench Architecture

A typical HIL test bench consists of:

- **Host software** to interactively operate the different components connected to the test bench.
  Operation in this sense means configuration, status control, accessing data, … . To be able to perform this operations also in an unattended, i.e. automated manner also an appropriate test automation (TA) system is applied. These TA systems consist of a set of components, such as: test executors, test frameworks to manage the parameterization of test cases, specific graphical or tabular-based test editors, comprehensive test libraries etc. More and more it becomes inevitable to integrate these HIL host software tools into other engineering data processing tools, e.g. for data storage, test management, requirement management or bug tracking systems.
- **Driver**
  The host software accesses the connected test bench components via specific drivers. These drivers use very heterogeneous technologies, such as: RS232, DLLs, (D)COM-Interfaces, TCP/IP or even GPIB. Up to now only very few of these driver interfaces are standardized, e.g. [ASAM MCD-3] for calibration or diagnostic access to ECUs.

- **Test bench components**
  First there is the real-time simulator itself. Here the simulation parts are calculated in the required time accuracy. Via the input-/output cards resp. the network interfaces (e.g. for CAN, LIN, Flexray) the real-time process is connected on electrical level to the electronic control unit. Missing network components, e.g. missing ECUs might be simulated here (so-called rest bus simulation).
  A second very essential HIL test bench component is the so-called Failure Injection Unit (FIU) or Electrical Error Simulation (EES) system. Test cases may not only comprise checking the behavior of the System Under Tests (SUT) in a fully functional environment. It is also important to check the SUT in case of electrical errors on the input and output pins.
  To be able to access ECU internal variables external calibration or diagnostics systems need to be integrated into the test automation environment.
- **System under Test: ECU(s)**
  The last and most important part of HIL test benches is of course the system under test itself, i.e. the ECU or a complete network of ECUs. HIL technology is applied today on the one hand for component testing of single ECUs but very often also for integration testing of complete vehicle ECU networks.
  As ECUs are very often developed in different variants the equipment of HIL test benches is prepared to be adapted to these variants.

1.3 STANDARDIZATION POTENTIAL IN THE HIL WORKING AREA

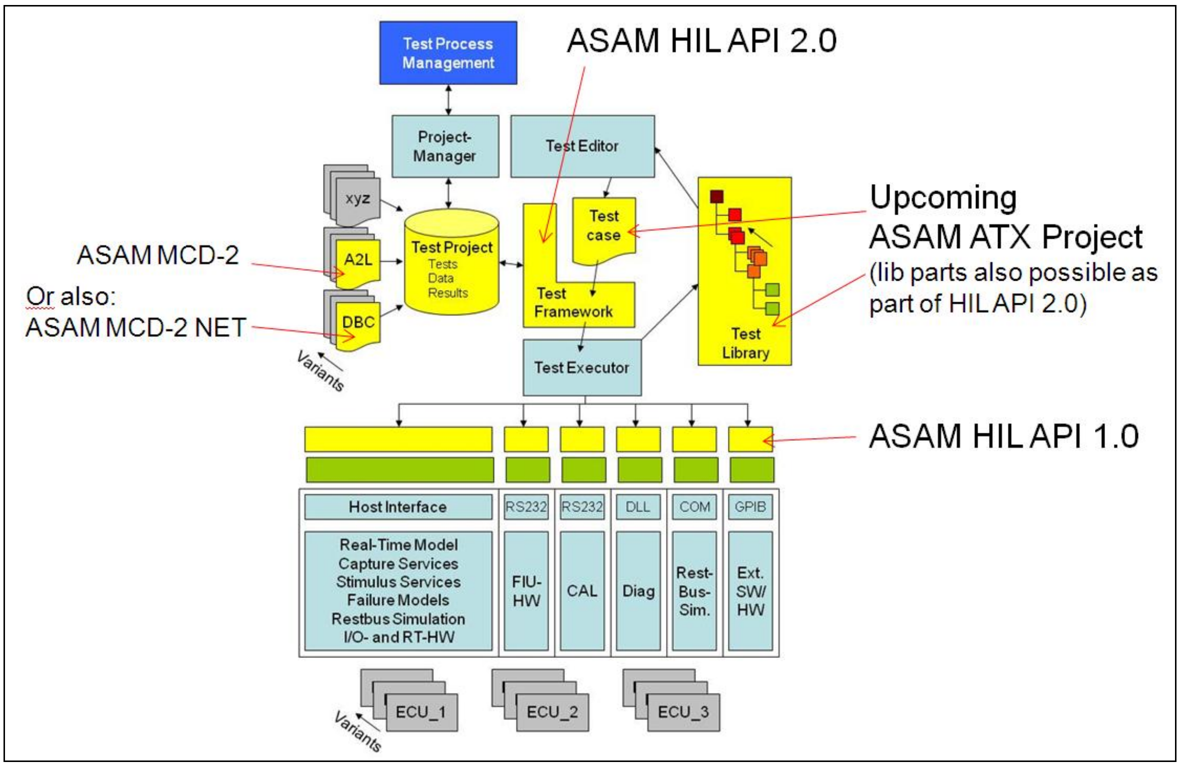


Figure 3 Standardization Potential around Hardware-in-the-Loop Simulation

Compared to other fields of activities in the area of Automotive Electronics (AE) the scope of Hardware-in-the-Loop simulation had not been addressed very intensively in the past.

Existing AE standards, such as [ASAM MCD-2 MC], [ASAM MCD-2 NET] or [ASAM MCD-3] are used naturally, but many other interfaces or sub-functionalities provide a huge potential for standardization.

The configuration and results of simulation test benches might be stored in data storage systems, which are comparable to other test bench areas. ASAM ODS might play a role here in future.

The exchange of test descriptions and entire test libraries via a standardized XML format would support many additional cases described. ASAM has just started another HIL technology project (ASAM Automotive Test Exchange Format 1.0.0) to work on these issues.

ASAM HIL API 1.0.0 addresses the standardization of the drivers of the most important HIL test bench components.

1.4 TODAY'S SITUATION

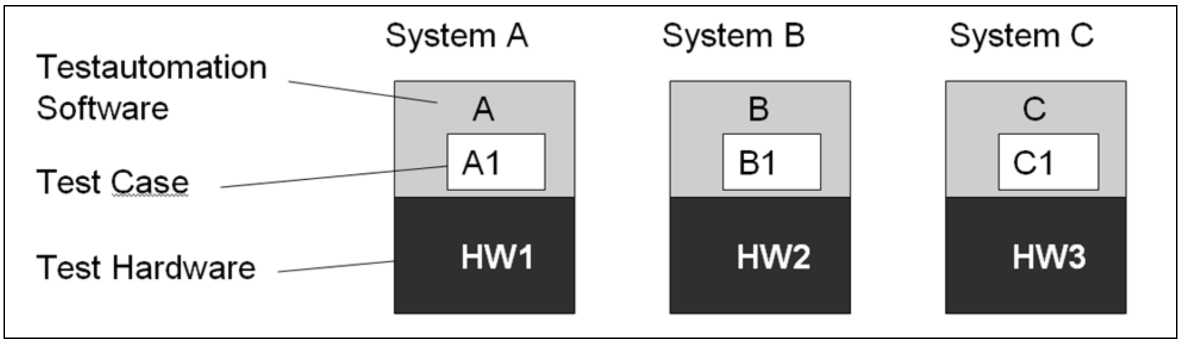


Figure 4 Today's Situation in HIL Testsystems

HIL technology had been developed over the years by only a few suppliers. Due to several reasons the architecture of these HIL systems was characterized by a direct rigid coupling of test automation software and used test hardware.
Therefore test cases directly depend on the used test hardware.
The end users perspective is, that not always the 'best' test software could be combined with the 'best' testing hardware.
Know-how could not be transferred from one test bench to the other. This resulted in additional training costs for employees.
Switching to the newest testing technology was difficult because of tool specific formats and test hardware compatibility issues.
This led to the consequence that the base pre-condition for an exchange of test cases, e.g. between OEM and supplier, was not fulfilled.

1.5 GOAL OF THE HIL API PROJECT

The major goal of all HIL technology standardization efforts is to allow for more reuse in test cases and to decouple test automation software from test hardware.

ASAM HIL API 1.0.0 only addresses the issue of decoupling. Therefore the reuse of test cases within the same test automation software on different test hardware systems should be achieved. This will lead to a reduction of effort for test hardware integration into test automation software.
Software investments and test case development efforts can be long-term protected. End users may decide on test automation software system on a perspective of many years without the coercion of being coupled to one test hardware supplier.

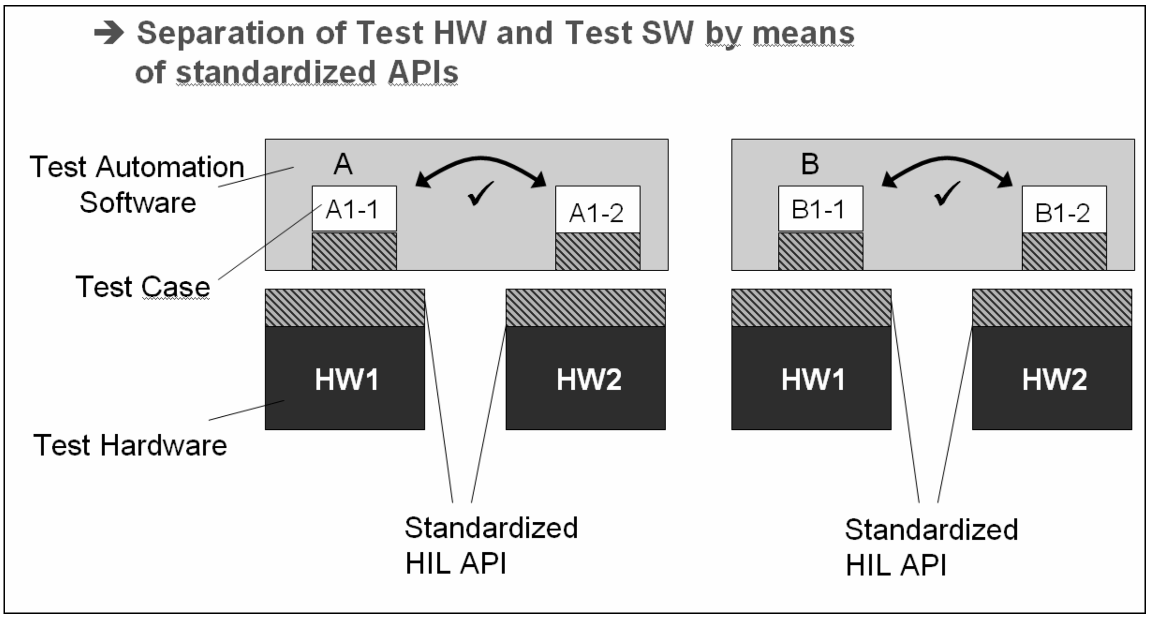


Figure 5 Solution Approach First Step: Standardisation of HIL API

1.6 TECHNOLOGICAL APPROACH

Today’s HIL test automation systems apply very different description technologies to define the test cases, e. g. the script language Python, C# or Java.
Graphical or tabular based notations might also be used but underneath transform to the mentioned languages.

ASAMs goal had always been to define technology independent standards. Following the approach which has been used in the [ASAM MCD-3] project first the ASAM HIL API 1.0.0 project team decided to develop the API as a generic UML-2 model, the so-called HIL API reference model.

Rules for the derivation of the market relevant Programming Language Versions for Python, C# and Java have been described and the so-called Technology References for these languages are equal work products of the standard.

The separation of UML based reference model also allows adding other technologies later without the need to modify the API model itself.

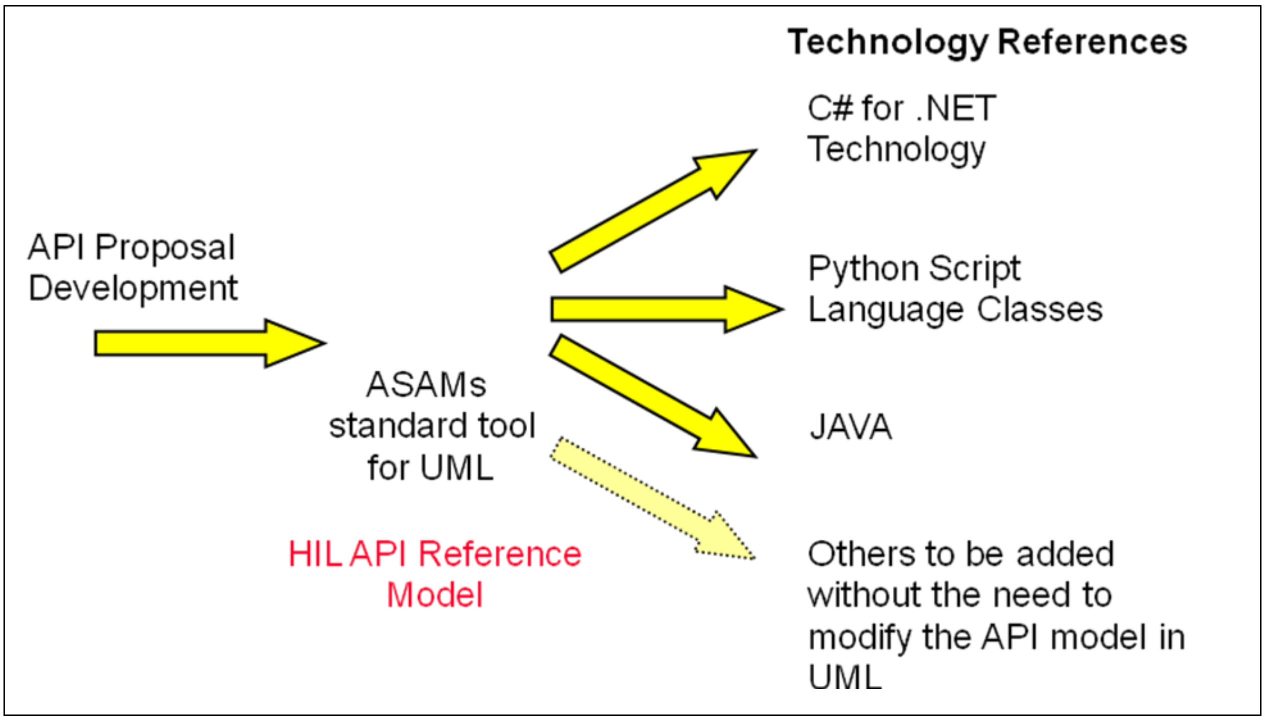

Technology References
C# for .NET Technology
API Proposal Development
Python Script Language Classes
ASAMs standard tool for UML
JAVA
HIL API Reference Model
Others to be added without the need to modify the API model in UML

Figure 6 Technological Approach via a generic HIL API Model

2 INTRODUCTION

2.1 ABBREVIATIONS

(D)COM	(Distributed) Component Object Model
AE	Automotive Electronics
API	Application Programming Interface
ASAM	Association for Standardisation of Automation and Measuring Systems
CAN	Controller Area Network
DLL	Dynamic Link Library
DTC	Diagnostic Trouble Code
ECU	Electronic Control Unit
EEPROM	Electrically Erasable Programmable Read Only Memory
EES	Electrical Error Simulation
ERD	Entity Relationship Diagram
FIU	Failure Injection Unit (see EES)
GPIB	General Purpose Interface Bus
HIL	Hardware In the Loop
HW	Hardware
LIN	Local Interconnect Network
PC	Personal Computer
RS232	Recommended Standard 232 (standard for serial binary data signals)
SEQ	Sequence Diagram
SUT	System Under Test
SW	Software
TA	Test Automation
TCP/IP	Transmission Control Protocol/Internet Protocol
UML2	Unified Modeling Language Version 2
XML	Extensible Markup Language

2.2 TECHNICAL APPROACH

2.2.1 TECHNOLOGY INDEPENDENCE

The object model of the HIL API is defined in UML. This UML model is mapped to different programming languages. As a result of the mapping process, all HIL API classes are available in each of the supported programming languages either as interface definitions or using native data types. A mapping guideline is available for each programming language which describes how the UML model is converted to the programming language.

Interface definitions are available for the following programming languages:

- C# [HIL C# Reference]
- Python [HIL Python Reference]
- Java [HIL Java Reference].

2.2.2 OBJECT CREATION

All instances of the HIL API classes are created either by a constructor or by an object factory.
If a constructor is used (which may have arguments), it is explicitly modeled in the UML model. These constructors always have the same name as the class; also if more than one constructor is defined (constructor overloading).
Classes which have no constructor must be created via an object factory. In that case, typically a method is available which returns an instance of the class (e.g. the CreateCapture method of the MAPort returns a new Capture instance).

Destructors are only modeled if they are explicitly needed to destroy the object.

3 Standardized Functionality of the HIL API

3.1 Overview

Hil API provides manufacturer independent access to the functionalities of a HIL simulator. It consists of interface definitions. Each tool vendor can provide an implementation of these interfaces which is specific for his tool set. Thus, the user of the HIL API gains standardized access to the tools of different vendors.

The HIL API covers the functional areas

- Model access,
- ECU access,
- Diagnostics access, and
- Electrical error simulation.

Each of these functional areas is represented by one or several *ports*. As the initialization of the tools is not part of the HIL API, the initialization of the HIL system has to be done by the vender-specific functions.

3.2 INITIALIZATION

A typical proceeding for using the HIL API for the access to a HIL simulator is depicted in Figure 7:

1. The HIL simulator is configured and started by using the vendor-specific functions.
2. Using the vendor-specific implementation of the HIL API, a port instance is created.
3. After this, the standardized HIL API functions are used to access the simulator.

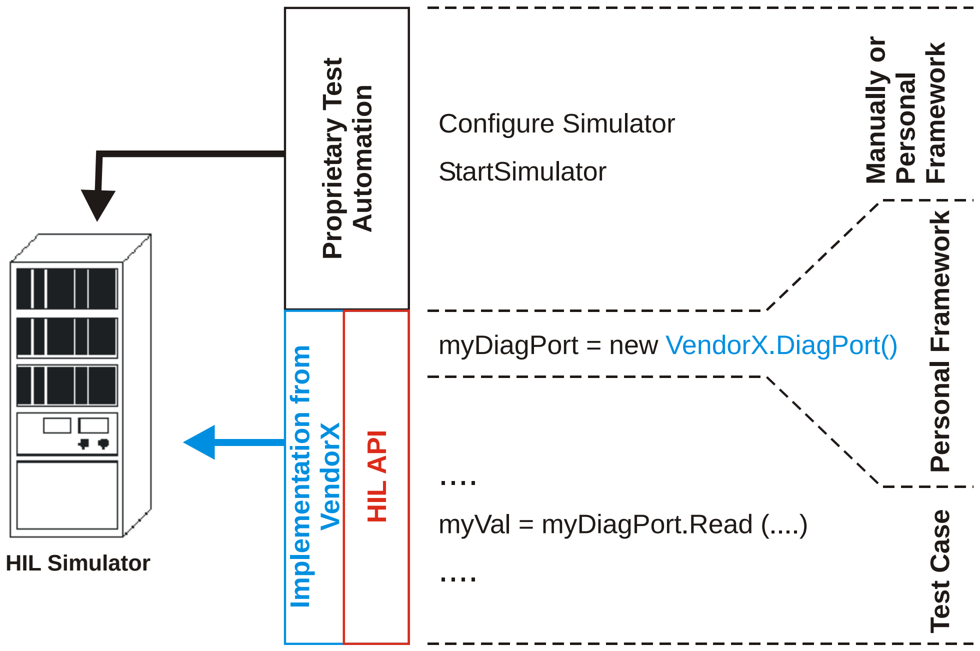

HIL Simulator
Proprietary Test Automation
Implementation from VendorX
HIL API
Configure Simulator
StartSimulator
myDiagPort = new VendorX.DiagPort()
....
myVal = myDiagPort.Read (....)
....
Manually or Personal Framework
Personal Framework
Test Case

Figure 7 Getting Access to HIL API

3.3 PORTS OF THE HIL API

All ports of the HIL API are derived from a common base interface `Port`. Figure 8 shows all ports, defined in this standard.

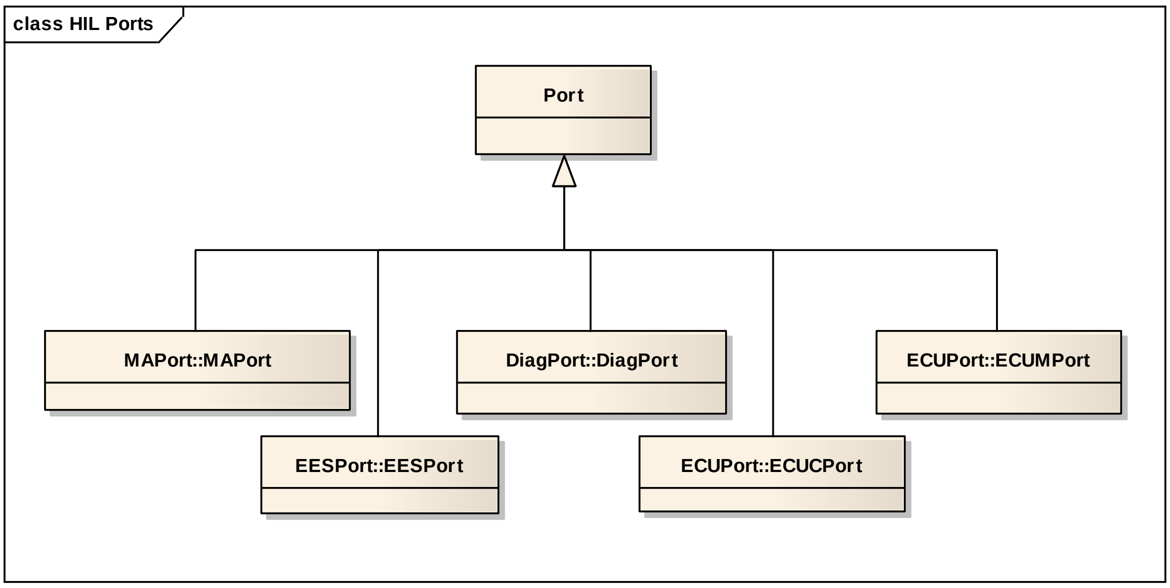


Figure 8 Ports in HIL

Table 1 gives an overview of these ports.

Table 1 Ports

Port	description
MAPort	The Model Access port provides access to the simulation model. It is possible to read and to write parameters and to capture and to generate signals.
DiagPort	The Diagnostic port communicates with a diagnostic system to read data via diagnostic services from an ECU or Functional Group.
EESPort	The EES port controls electrical error simulation hardware. It allows setting different types of errors.
ECUPort	The ECU ports communicate with an MC system and thus provide access to ECU internal values. The ECU M port allows to capture and to read measurement variables. The ECU C port is used for calibration.

4 COMMON FUNCTIONALITIES

This chapter describes functionalities of HIL API which are not specific for one port.

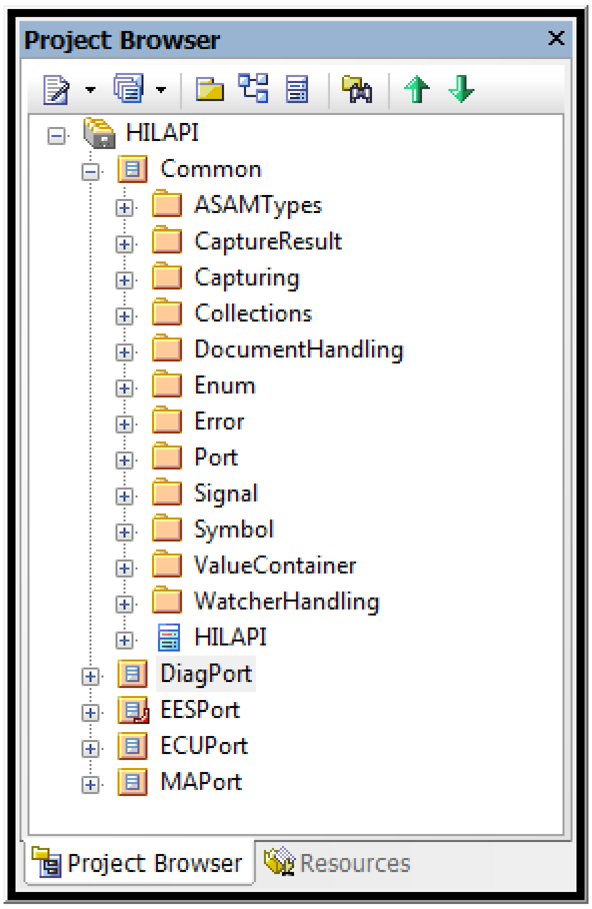


Figure 9 Packages in Common part

This Figure 9 shows all top level packages and the sub packages of the package Common of the HILAPI UML model. The most important parts of the package Common are described in the following sections.

Table 2 Packages of Common part

Sub package name	**Description**
ASAMTypes	Contains all basic data types which are used in the UML model, e.g. Boolean, string or int.
CaptureResult	Contains classes which handle the result of Capturings.
Capturing	Contains classes which do Capturing.
Collections	Contains all collection classes used in the model.
DocumentHandling	Gives an overview of all classes which are used to read or write content from/to the file system in different file formats.
Error	Contains common classes used for error handling. Error codes are defined in the sub package Enum.
Port	Gives an overview of all available ports and defines the base class for all ports.
Signal, Symbol	Contain classes for describing signal waveforms. Such signals are used for signal generation.
ValueContainer	A set of classes which are designed to store values of different types, e.g. scalar, matrix or map values. Together with the ASAMTypes and the Collection classes, the value container classes are the fundamental type system which is used in the entire UML model.
WatcherHandling	Classes being used by the capturing classes for defining trigger conditions.

4.1 VERSIONING

Versioning of the HILAPI is done using three numbers: major version, minor version and revision number. The major number is the actual version, the minor number the actual maintenance of the version. The revision number is the revision of the maintenance. These numbers define the interface version of the HIL API.

In addition, a build number allows a tool manufacturer to clearly version his HIL API implementation, independently from the HIL API version. It is of type string in order to provide more flexibility.

The version information can be retrieved using the HILAPI class. The standard implementation of the HILAPI class returns the following version numbers:

Table 3 Version number

Number	**Value**
Major version	1
Minor version	0
Revision number	0
Build number	empty

Tool manufacturers must override the base implementation and return a valid build identifier.

4.2 ASAM DATA TYPES

ASAM data types are used in the entire UML model. These define the type system for all scalar basic data types. All complex data types use these base types (see e.g. package Common/ ValueContainer at chapter 4.4). More information about the ASAM data types is available in [ASAM Data Types].

The following basic data types are used in the HIL API UML model:

- A_ASCIISTRING
- A_BOOLEAN
- A_BYTEFIELD
- A_FLOAT64
- A_INT64
- A_UINT64
- A_UNICODE2STRING

The ASAM data types are included in the model in the sub package ASAMTypes:

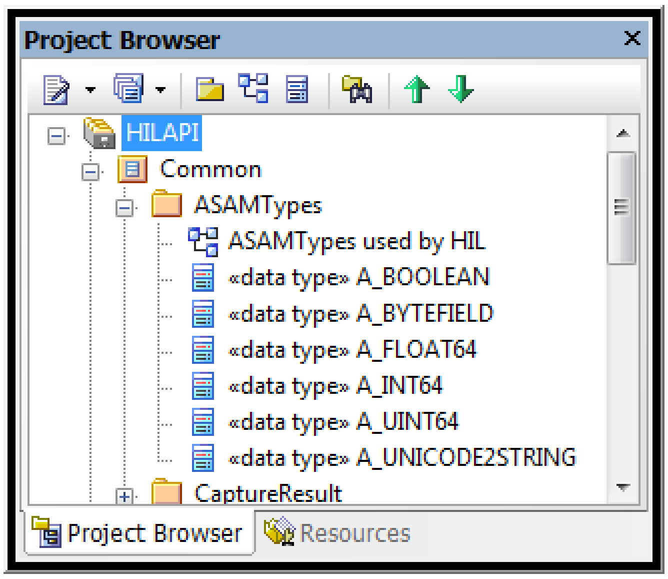


Figure 10 ASAM data types

When the UML model is transformed to different programming languages (e.g. Python, C#, Java), the ASAM data types are mapped to native, language specific data types.

4.3 Collections

A lot of HIL API classes need collections like arrays, lists and dictionaries. The sub package Collections defines all collection classes used in the HIL API. All collection classes are derived from the base class Collection.

Figure 11 gives an overview of the collection classes.

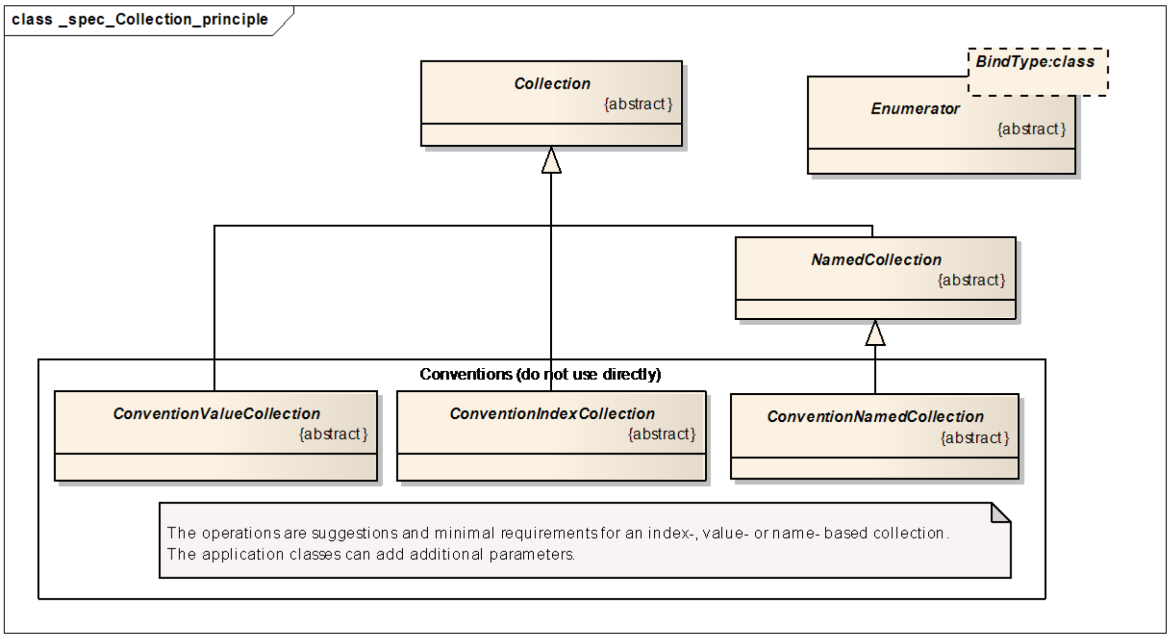

class _spec_Collection_principle
Collection {abstract}
BindType:class
Enumerator {abstract}
NamedCollection {abstract}
Conventions (do not use directly)
ConventionValueCollection {abstract}
ConventionIndexCollection {abstract}
ConventionNamedCollection {abstract}
The operations are suggestions and minimal requirements for an index-, value- or name- based collection.
The application classes can add additional parameters.

Figure 11 Collection classes

Collection is the base class for all collection classes. It provides functions to get the number of elements in the collection and to access all elements using an enumerator.

Enumerator: Enumerators allow iterating over all elements of a collection. They provide a `next()` method to move to the next element in the collection. Enumerators become invalid if the underlying collection has been modified, e.g. because an element has been added or removed.

The following collection classes are meant to be understood as a set of pattern classes. They represent different types of collection, e.g. index based collections or dictionaries. They are not used directly in the UML model. Instead, the concretely typed versions of these classes are used. These pattern classes are:

- ConventionIndexedCollection represents a collection with a fixed order of elements. Consequently, calling the `GetByIndex()` method several times using the same index yields the same element.

- ConventionNamedCollection: The NamedCollection class is the template for all dictionaries used in the HIL API whose key is a string name. A method `Contains()` is available which allows to check if the `NamedCollection` contains an element with the given name.

- ConventionValueCollection is the template for all dictionaries used in the HIL API whose key is an integer number.

In order to avoid multiple inheritances, the template collection classes are modeled using a „Realize"-association (see UML-2). The methods are defined in the base classes of the collections, but the implementation of a derived collection type explicitly contains all methods of the collection class that it realizes. Figure 12 shows this using the `SegmentSignalDescription` class. The `getCount()` and `GetEnumerator()` methods are modeled in the collection base calls and in the `SegmentSignalDescription` class as well.

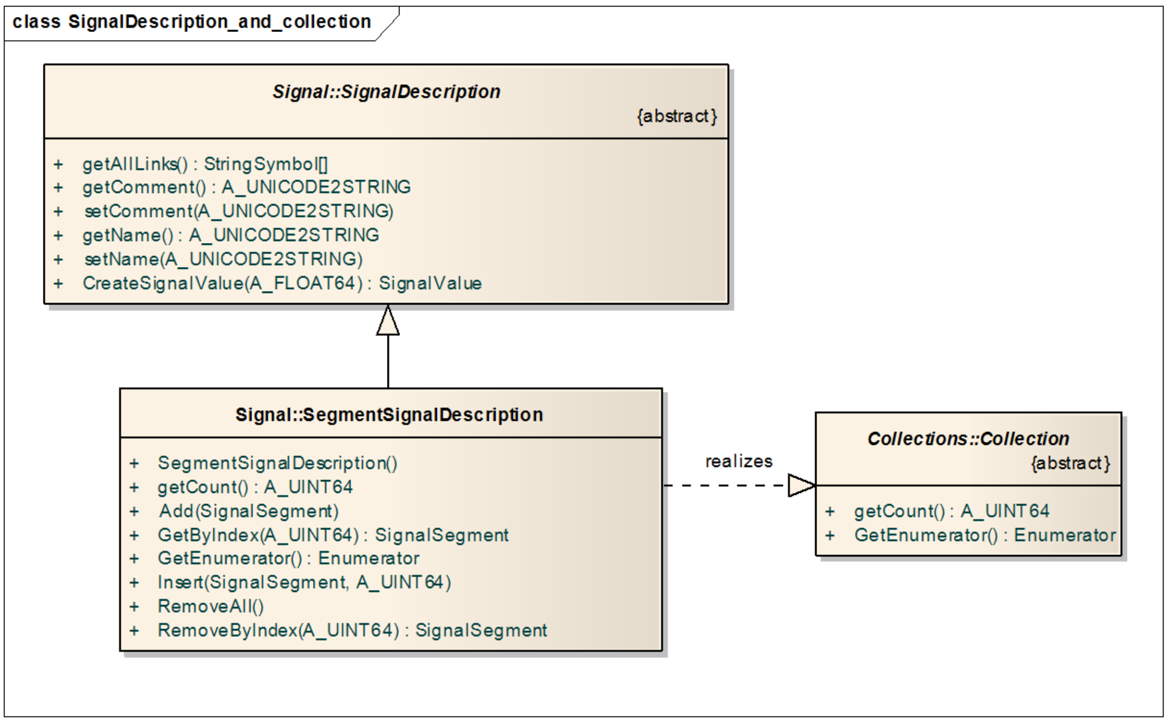

class SignalDescription_and_collection
Signal::SignalDescription
{abstract}
+ getAllLinks() : StringSymbol[]
+ getComment() : A_UNICODE2STRING
+ setComment(A_UNICODE2STRING)
+ getName() : A_UNICODE2STRING
+ setName(A_UNICODE2STRING)
+ CreateSignalValue(A_FLOAT64) : SignalValue
Signal::SegmentSignalDescription
+ SegmentSignalDescription()
+ getCount() : A_UINT64
+ Add(SignalSegment)
+ GetByIndex(A_UINT64) : SignalSegment
+ GetEnumerator() : Enumerator
+ Insert(SignalSegment, A_UINT64)
+ RemoveAll()
+ RemoveByIndex(A_UINT64) : SignalSegment
realizes
Collections::Collection
{abstract}
+ getCount() : A_UINT64
+ GetEnumerator() : Enumerator

Figure 12 Collection example: SegmentSignalDescription

Besides the base and pattern classes there are collection classes which are concrete realizations of the above described patterns. A `StringNamedCollection` maps a string name to string value, a `UintNamedCollection` maps string name to an unsigned integer value and a `FloatNamedCollection` a string name to a float value. The `AnyObjectNamedCollection` maps string names to any kind of object, e.g. to an HIL API object or to a native data type. Figure 13 illustrates this.

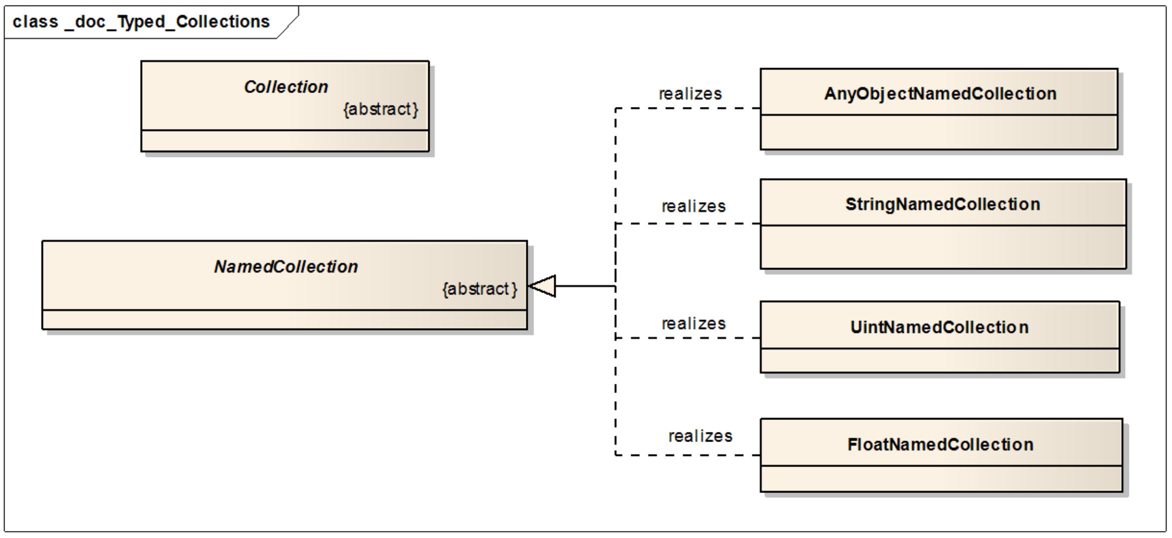


Figure 13 Typed Collections

4.4 VALUECONTAINER

4.4.1 OVERVIEW

The ValueContainer package provides a set of container classes, which are used to store data values. These container classes are divided into three categories:

1. The first category comprises all general container classes which are either scalars or which contain elements being accessed by integer based indices, e.g. vectors and matrices. Concrete sub classes are available for the most important data types like Boolean, integer, float and string. Some examples are `ScalarFloatValue`, `StringVectorValue` and `BooleanMatrixValue`.
2. In addition, there are more application oriented classes, which are used for calibration access, for capturing or for signal generation. Examples are the classes `CurveValue`, `MapValue` and `SignalGroupValue`.
3. The third category consists of named collections. These are explained in more detail in section 4.3.

All container classes are derived from a common base class named `BaseValue`. Its method `getType()` allows to retrieve the concrete data type of a value container instance as specified by the enumeration type `DataTypes`.

It is possible to attach meta information to a value container instance. Examples for such meta information are the name of the variable or the unit of the value. More detailed information about meta data can be found in chapter 4.4.4.

The `getValue` methods return copies (not references) of the internal data objects, e.g. a new instance of `VectorValue` is returned when using the `XVector` property on a `MapValue` object. So the value itself cannot be changed by altering the returned instances.

In the following chapters explain the different value container categories and the concept of meta data information in more detail.

4.4.2 General Value Container Classes

General ValueContainer classes represent scalar, vector and matrix values (Figure 14).

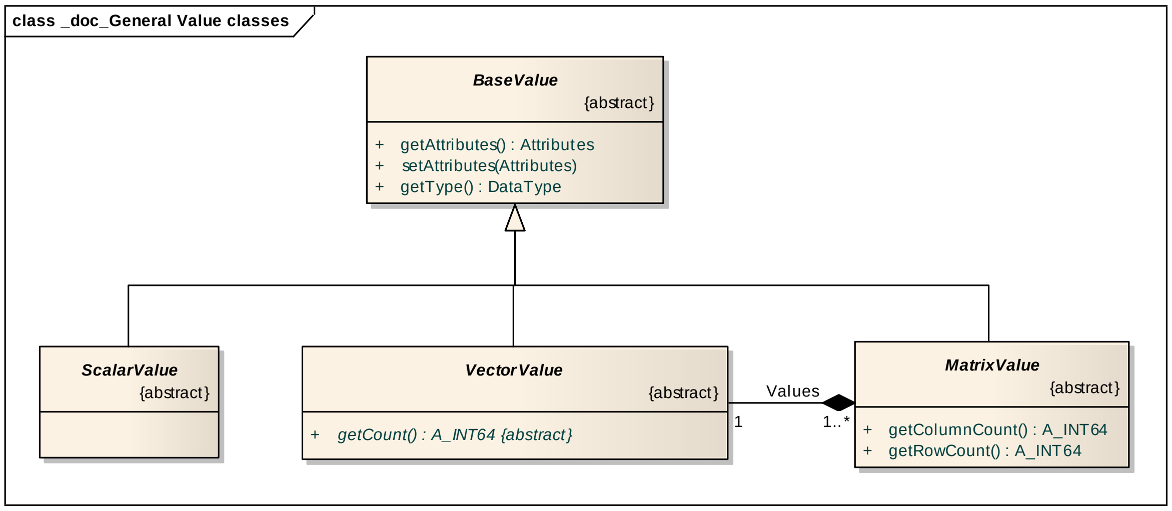

class _doc_General Value classes
BaseValue
{abstract}
+ getAttributes() : Attributes
+ setAttributes(Attributes)
+ getType() : DataType
ScalarValue
{abstract}
VectorValue
{abstract}
+ getCount() : A_INT64 {abstract}
Values
1
1..*
MatrixValue
{abstract}
+ getColumnCount() : A_INT64
+ getRowCount() : A_INT64

Figure 14 General Value classes

All elements inside a composite ValueContainer class (e.g. VectorValue or MatrixValue) are homogenous, meaning all elements must be of the same type, which is specified by the class. Class names are prefixed with *Int, Uint, Float, String* and *Boolean* corresponding to type of the managed elements (Figure 15).

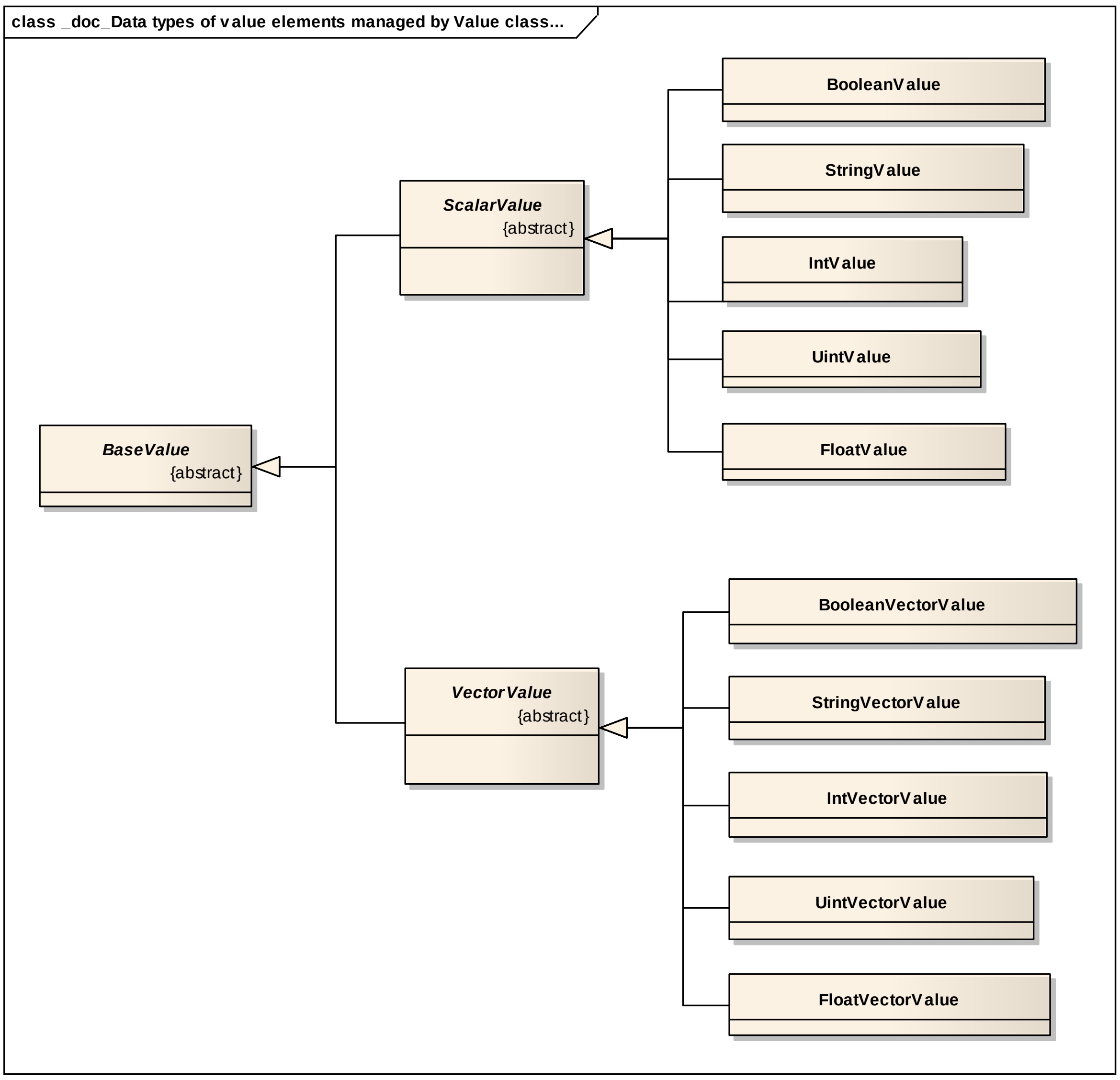


Figure 15 Data types of value elements managed by Value classes ScalarValue and VectorValue

The `ScalarValue` classes `IntValue`, `FloatValue`, `StringValue` and `BooleanValue` represent a single value of the particular data type.

The `VectorValue` classes `IntVectorValue`, `FloatVectorValue`, `StringVectorValue` and `BooleanVectorValue` represent an ordered sequence of values. The `Count` property returns the number of values in the collection.

The `MatrixValue` classes `IntMatrixValue`, `FloatMatrixValue`, `StringMatrixValue` and `BooleanMatrixValue` represent a two dimensional array of values. The `ColumnCount` and `RowCount` properties return the number of values of each dimension inside the matrix.

4.4.3 APPLICATION ORIENTED VALUE CONTAINER CLASSES

Application oriented `ValueContainer` classes (Figure 16) are used for more specialized applications like calibration and capturing.

`MapValue` and `CurveValue` *classes* are widely used for calibration of curve (1D table) and map (2D table) values. Their X and Y vectors must be either monotonously increasing or decreasing and the number of rows / columns of the function values must be equal to the length of the Y / X vector.

`SignalValue` and `SignalGroupValue` are used to represent captured signal data.

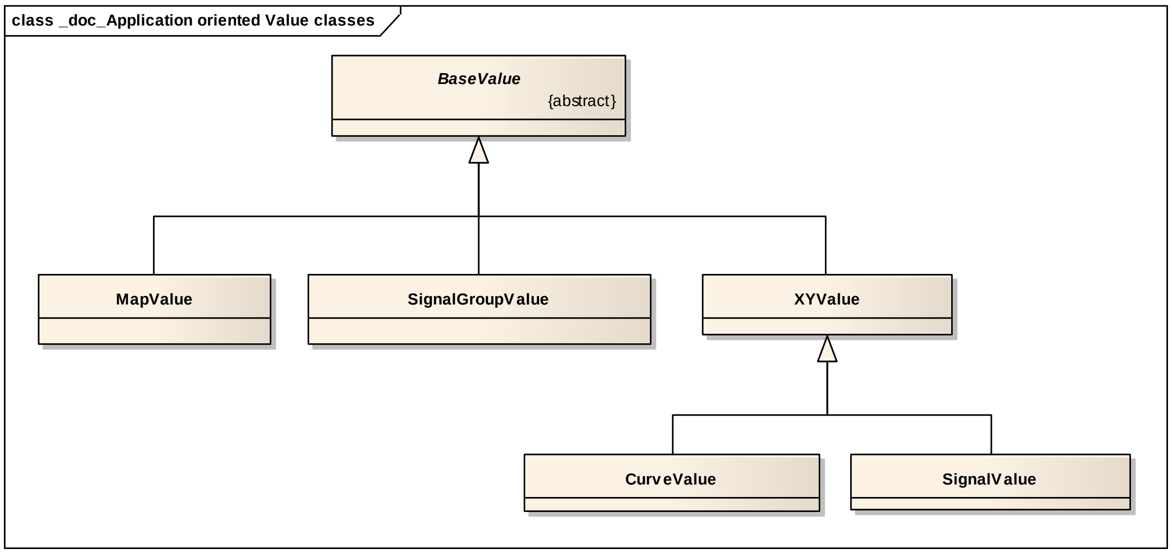


Figure 16 Application oriented Value classes

4.4.4 ATTRIBUTES

Instances of the `Attributes` class are used to attach meta data to `ValueContainer` objects. The information consists of a list of attribute names their values. The name and the value of an attribute are strings (`A_UNICODE2STRING`*).*

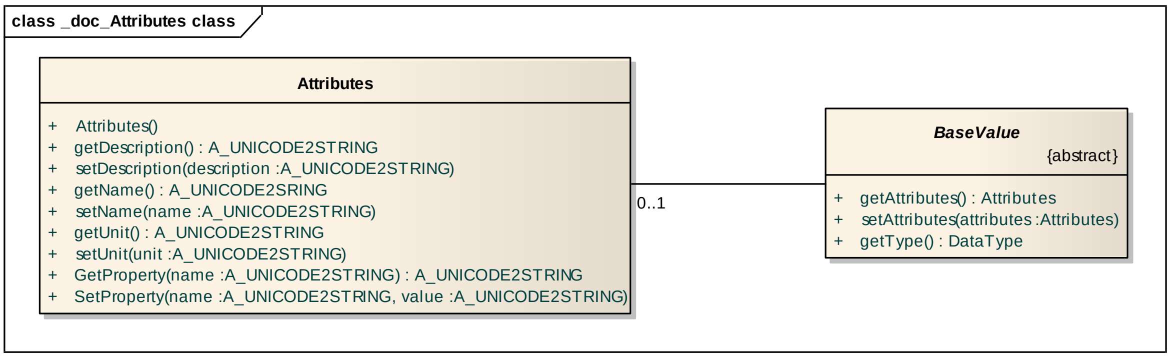

class _doc_Attributes class
Attributes
+ Attributes()
+ getDescription() : A_UNICODE2STRING
+ setDescription(description :A_UNICODE2STRING)
+ getName() : A_UNICODE2SRING
+ setName(name :A_UNICODE2STRING)
+ getUnit() : A_UNICODE2STRING
+ setUnit(unit :A_UNICODE2STRING)
+ GetProperty(name :A_UNICODE2STRING) : A_UNICODE2STRING
+ SetProperty(name :A_UNICODE2STRING, value :A_UNICODE2STRING)
0..1
BaseValue
{abstract}
+ getAttributes() : Attributes
+ setAttributes(attributes :Attributes)
+ getType() : DataType

Figure 17 Attributes class

Some commonly used attributes are predefined. These are:

- Name
- Description
- Unit

It is also possible to add user-defined attributes using the methods `SetProperty()` and `GetProperty()`.

4.4.5 LINK TO SAMPLES

The sample code for this chapter will be found at

C#	C#\SampleCode\Common\ValueContainer\ValueContainerExample.cs
Python	Python\SampleCode\Common\ValueContainerExample.py
Java	JAVA\SampleCode\Common\ValueContainer\ValueContainerExample.java

4.5 ERROR HANDLING

4.5.1 EXCEPTION CLASSES

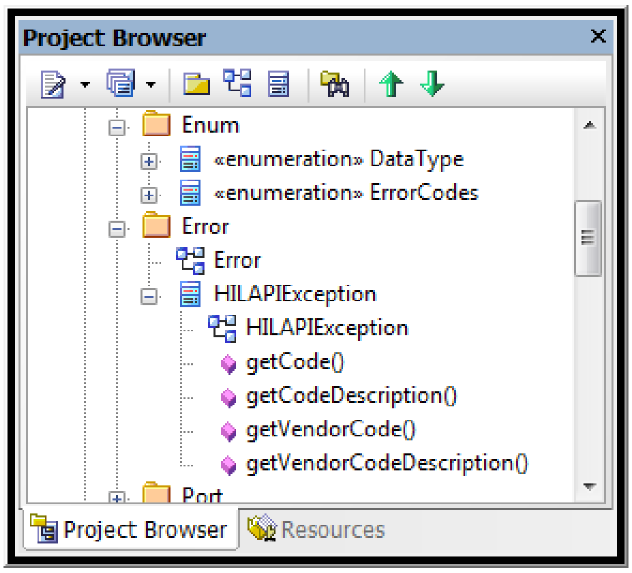


Figure 18 Exceptions

Errors are handled using exceptions. `HILAPIException` is the base class for all HIL API specific exception types. `HILAPIException` is mapped to the language native exception mechanism (e.g. in C#, `HILAPIException` is derived from `System.Exception`).
Application oriented exception classes are derived from `HILAPIException`, e.g. `MAPortException`, `DiagPortException` (see Figure 18). These are defined in a separate sub-package named Error and named <portname>Exception, e.g. `MAPortException`.

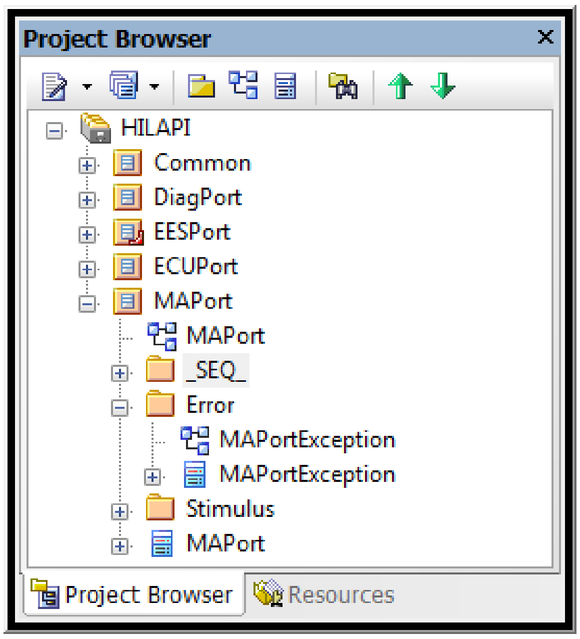


Figure 19 Package structure

4.5.2 ERROR CODES

Enum Error Codes

Error codes are used to uniquely identify each type of error which occurs. For each exception class a range of error codes is defined.

The enumeration “ErrorCodes” in the sub package Common/Enum contains all available error codes. Each error code is defined by a unique unsigned error number and a unique name. The following ranges of error codes are available:

Table 4 ErrorCode PreFixes and ErrorValue Range

PreFix of ErrorCode	Range of ErrorValue
eCOMMON _	1000 - 1999
eDIAG_	2000 - 2999
eEES _	3000 - 3999
eMA _	4000 - 4999
eECU _	5000 - 5999
eNW_ (reserved for further version)	6000 - 6999
eFW_ (reserved for further version)	7000 - 7999

The complete list of error codes along with their descriptions and messages can be found in the file ASAM_AE_HIL_BS_ErrorOverview_V1-0-0.xls (part of the version 1.0.0 deliverables).

4.6 DOCUMENT HANDLING

The classes derived from the abstract class `DocumentManager` are designed to save and load data to/from files. Each class derived from the `DocumentManager` provides a `Load()` and a `Save()` function to store data in a particular file format. E.g. sub classes are defined for reading and writing signal descriptions, signal generator properties, capture results, and EES Port configurations.

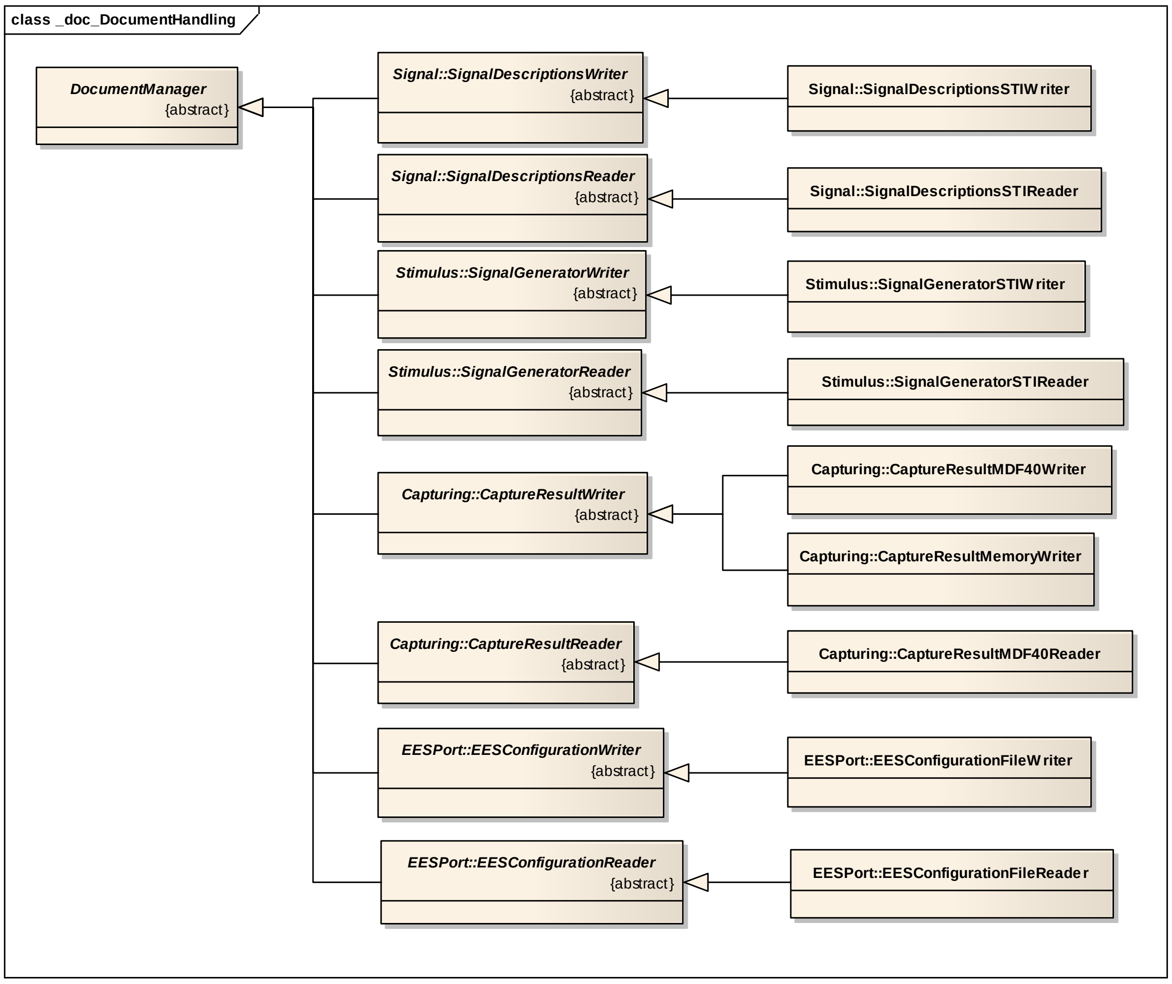

class _doc_DocumentHandling
DocumentManager {abstract}
Signal::SignalDescriptionsWriter {abstract}
Signal::SignalDescriptionsSTIWriter
Signal::SignalDescriptionsReader {abstract}
Signal::SignalDescriptionsSTIReader
Stimulus::SignalGeneratorWriter {abstract}
Stimulus::SignalGeneratorSTIWriter
Stimulus::SignalGeneratorReader {abstract}
Stimulus::SignalGeneratorSTIReader
Capturing::CaptureResultWriter {abstract}
Capturing::CaptureResultMDF40Writer
Capturing::CaptureResultMemoryWriter
Capturing::CaptureResultReader {abstract}
Capturing::CaptureResultMDF40Reader
EESPort::EESConfigurationWriter {abstract}
EESPort::EESConfigurationFileWriter
EESPort::EESConfigurationReader {abstract}
EESPort::EESConfigurationFileReader

Figure 20 DocumentHandling in HIL

4.7 SIGNAL DESCRIPTION

When testing ECUs via HIL simulation, signals play an important role in different use cases. In many test cases, model variables are stimulated. In other tests, variables are captured and the captured data has to be compared with reference signals. For these use cases, the HIL API introduced the classes `SignalDescription`, `SignalDescriptionSet` and `SignalGenerator`, as shown in Figure 21.

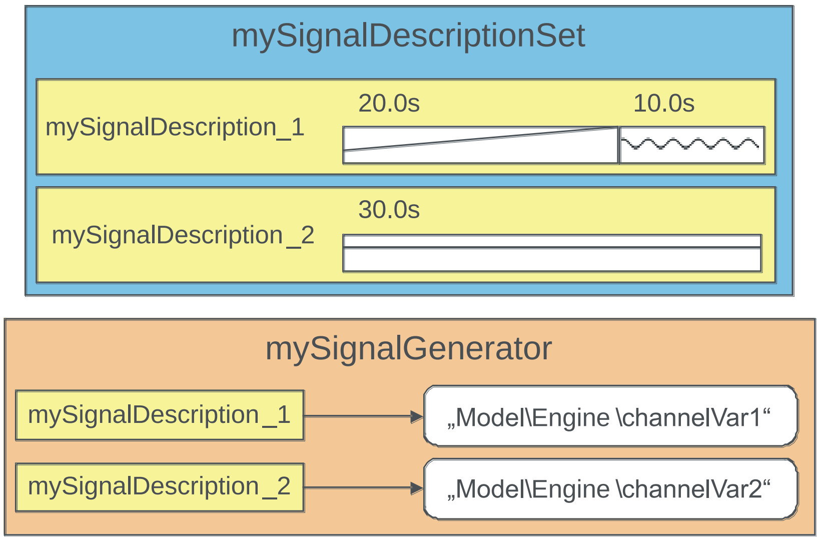


Figure 21 SignalDescriptions and SignalGenerator

A signal description consists of one or multiple segments, e.g. a ramp, followed by sine, which is denoted as "mySignalDescription_1" in the figure, or simply a constant signal denoted as "mySignalDescription_2". Many other segment types are also defined by the HIL API (see below). Such a signal description does not have any relation to variables of the simulation model. It can be used e.g. as a reference signal. Multiple signals are aggregated in a signal description set.

In order to use signals for stimulation, a signal generator is used. A signal generator relates signals to model variables and controls the signal generation process.

When modeling signals, an advanced specification is possible: Figure 22 shows a ramp signal, denoted as "modulateSignal" and a sine signal ("mySignalDescription_1") whose amplitude is specified by the ramp. The resulting signal is depicted besides the signal generator. All parameters of all segment types can be specified by other signals.

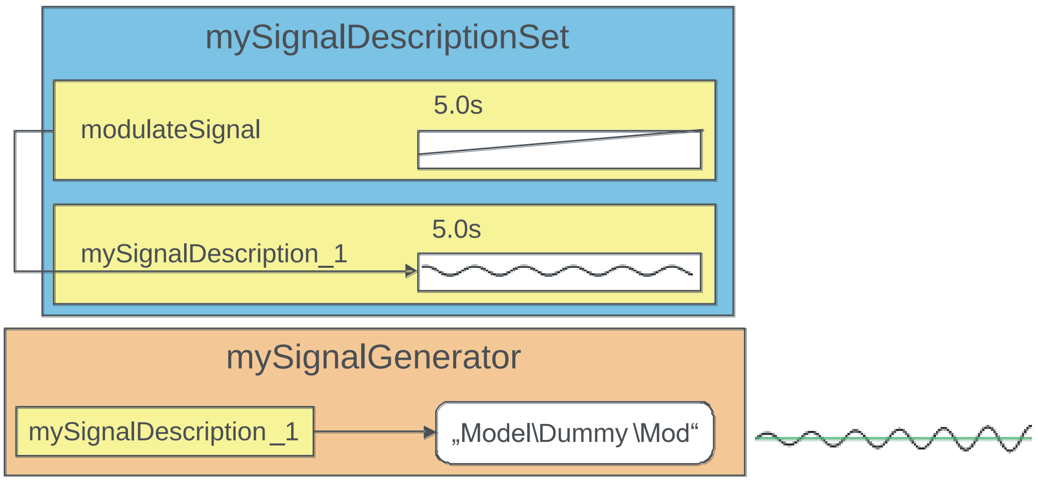


Figure 22 Modulate Signal Parameter by further Signals

Another possibility to describe signals are operational signal descriptions: An operational signal adds or multiplies two signals, as shown in Figure 23.

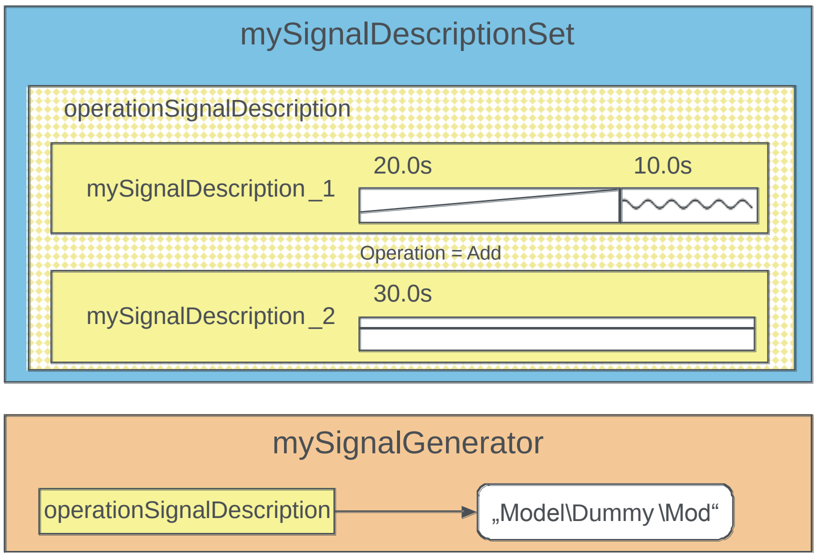


Figure 23 SignalDescriptions and SignalGenerator

In order to compare a signal description for example with sample data, i.e. signals that are defined by a couple of points in time and corresponding functional values, it is helpful to transform the signal description into an equivalent format (see Figure 24). Calling the method `CreateSignalValue()` on a signal description with the sample time as parameter, creates an according signal value (see Chapter 4.4). Calling method `CreateSignalGroupValue()` on `SignalDescriptionSet` creates a signal group value.

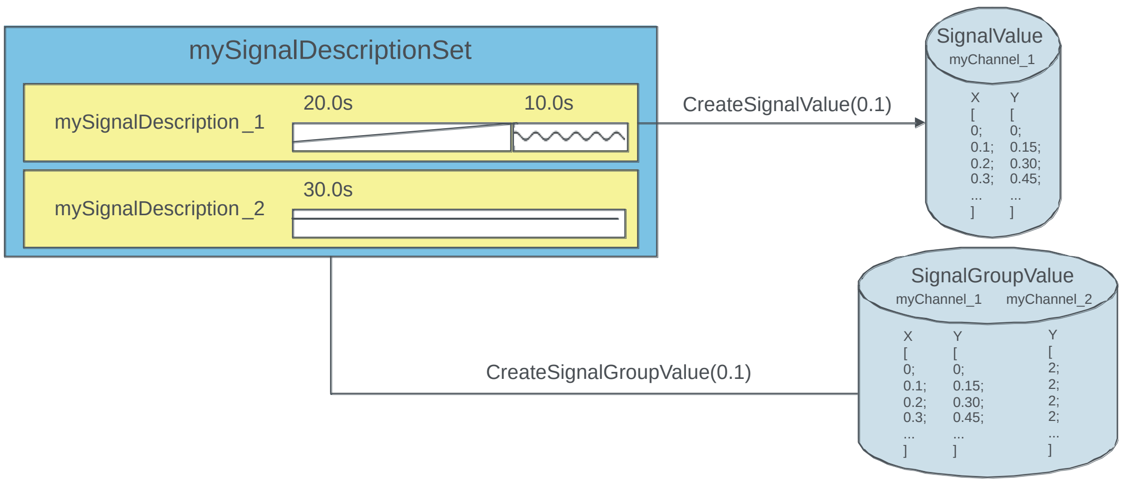


Figure 24 SignalDescriptions and SignalGenerator (data transformation)

In general the signal description is used to describe a signal for general purpose usage. A signal can be described by using synthetic waveform elements like ramp or sine and/or with elements which contain the signal points in form of numerical data.
The entry point is the class `SignalDescriptionSet` which acts as a container for signals to group several signals to one signal-set.
The `SignalDescription` is the abstract base class of `OperationSignalDescription` and SegmentSignalDescription.
The class `OperationSignalDescription` adds or multiplies (depends on operation property) 2 signals (left and right signal). The limitation of these 2 signals was explicitly done for version 1.0.0 of HIL. The schema of Stimulus already allows 0..n operands.
The `SegmentSignalDescription` is used to define a signal waveform based on a temporal sequence of different segments. Thus the `SegmentSignalDescription` is an indexed collection of signal-segments.

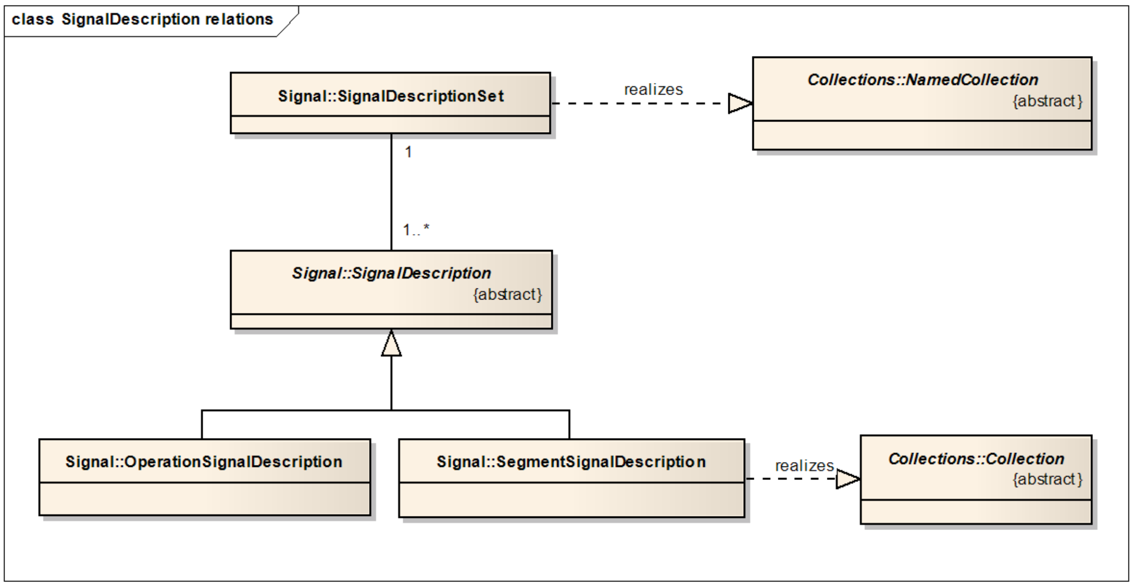

class SignalDescription relations
Signal::SignalDescriptionSet
realizes
Collections::NamedCollection {abstract}
1
1..*
Signal::SignalDescription {abstract}
Signal::OperationSignalDescription
Signal::SegmentSignalDescription
realizes
Collections::Collection {abstract}

Figure 25 SignalDescription relations

4.7.1 SIGNAL FILE READING AND WRITING

To save the whole content of a `SignalDescriptionSet` or to load a complete set of signals into a `SignalDescriptionSet` there are two classes: The `SignalDescriptionWriter` and the `SignalDescriptionReader`. This concept allows it to load and save data in different formats. One format is already standardized in HILAPI 1.0.0, the STI-Format (see SignalDescriptionFormat.xsd).

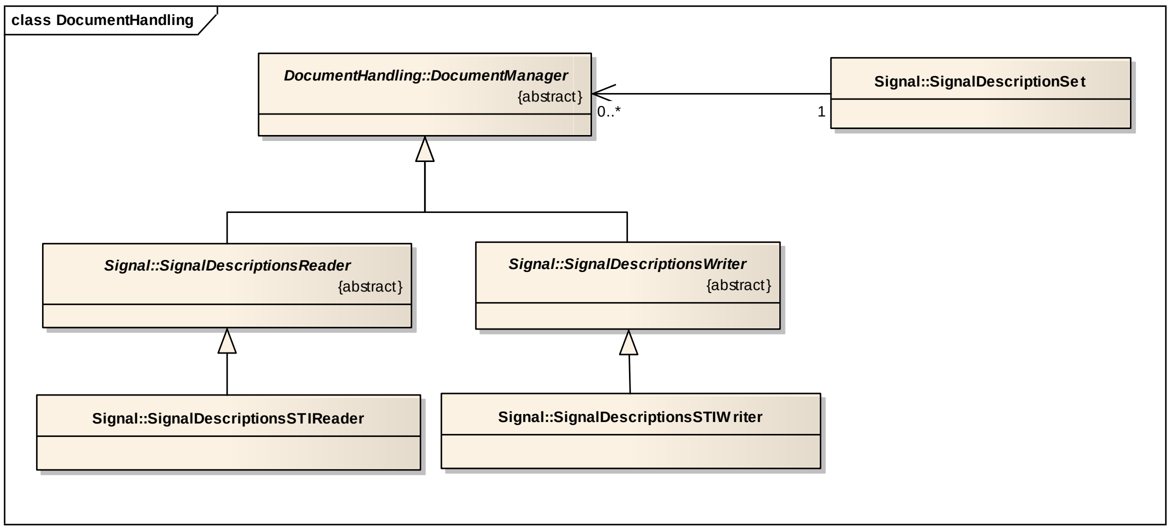

class DocumentHandling
DocumentHandling::DocumentManager {abstract}
0..*
1
Signal::SignalDescriptionSet
Signal::SignalDescriptionsReader {abstract}
Signal::SignalDescriptionsWriter {abstract}
Signal::SignalDescriptionsSTIReader
Signal::SignalDescriptionsSTIWriter

Figure 26 SignalDescription Reader and Writer

4.7.2 General Remarks about segment-based Signals

A segment is the smallest unit that describes the signal form completely for a defined time period. Properties of a segment are:

Type: Each segment has a read-only property type that indicates the kind of the segment for post-analysis (`SignalSegment.getType(): SegmentTypes`).

Comment: Each segment has an optional property `comment` that can be used by the tester to write a description linked to the segment definition, for example to help to understand the complete signal definition.

Duration: Most of the segments have the property `duration` that specifies the length in time. The unit of duration is second. The `SignalValueSegment` and the `OperationSegment` have no duration property.

The other segment parameters/properties are segment specific. For example the `SineSegment` has the parameters amplitude, offset, period and phase.
All segment parameters use the Symbolic Mapping. That means the parameters accept a numeric value (`ConstSymbol`) or another channel (`SignalSymbol`) that is used to modulate a segment parameter. An example for this is the amplitude modulation of a `SineSegment`. It is not possible to modulate the duration of the segment by another signal, thus the duration property only accepts the `ConstSymbol`.

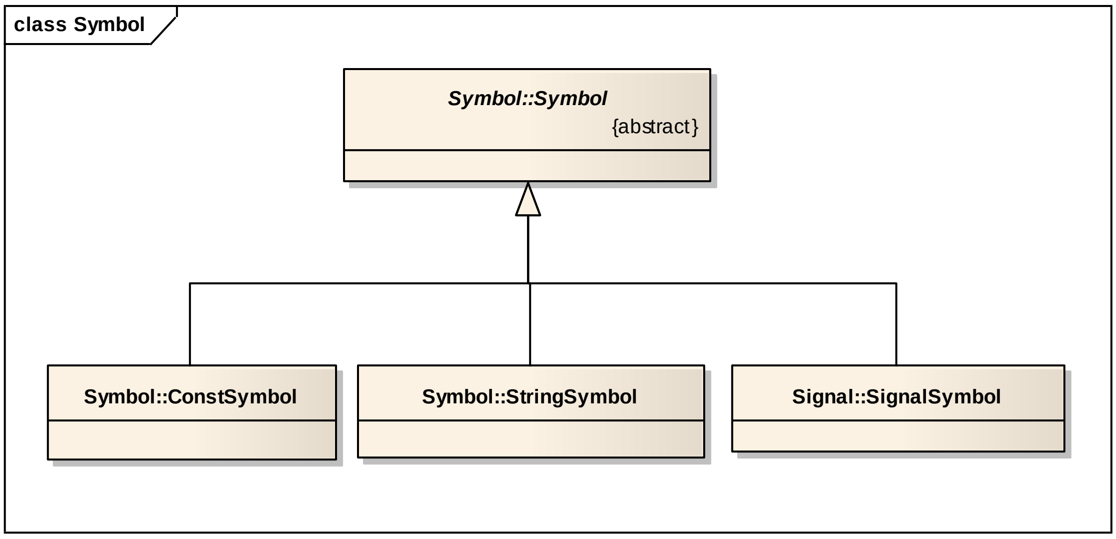


Figure 27 Symbol

Additionally segments can be combined together by operations. So you can for example add a ramp signal to a noise signal. This operation on can be done by the `OperationSegment` that can be used in the same way as the native segments.

List of segments:

Synthetic Waveform Segments:

- `ConstSegment`
- `RampSegment`
- `IdleSegment`
- `NoiseSegment`
- `SlopeRampSegment`
- `SineSegment`
- `SawSegment`
- `PulseSegment`
- `ExpSegment`

Data Oriented Segments:

- `SignalValueSegment`

Complex Segments:

- `OperationSegment`

4.7.3 Signal Segments

4.7.3.1 ConstSegment

The `ConstSegment` is used to generate a part (segment) of the signal with a constant signal flow. The amplitude of the signal is on a constant value during the whole duration of the segment.

<u>Mathematical description</u>

$f(t) = A$

A: Amplitude of the signal

<u>HILAPI – Description</u>

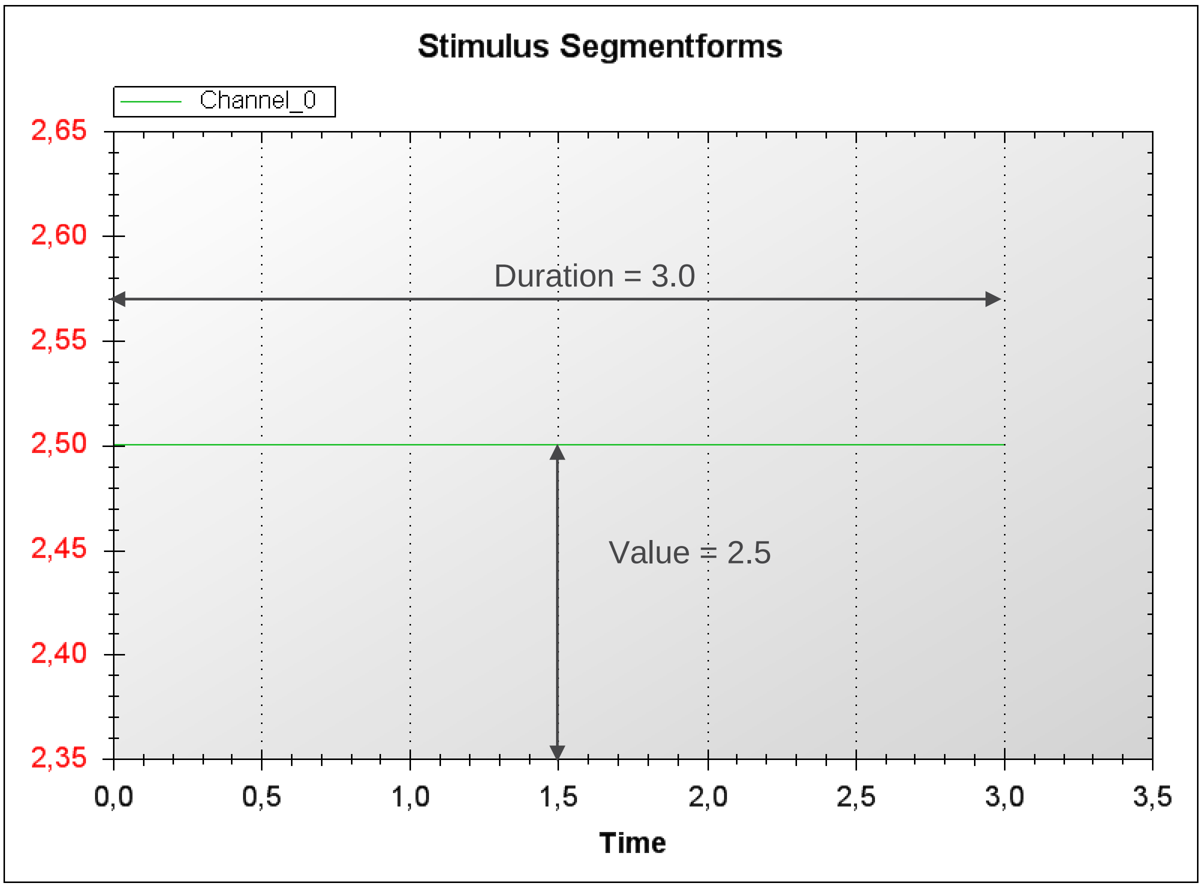


Figure 28 ConstSegment

Parameters:

`Duration`:	Duration / run time of the segment	
	Unit:	Seconds [s]
	Range:	[0 < Duration <= MAX(A_FLOAT64)]
`Value`:	Value which is used as signal amplitude.	
	Unit:	-
	Range:	[MIN(A_FLOAT64) <= Value <= MAX(A_FLOAT64)]

4.7.3.2 RampSegment

The `RampSegment` is used to generate a part (segment) of the signal with a ramp-shaped signal flow. The amplitude of the signal follows a straight line according to a linear equation.
The slope of the line is calculated from the given start- and stop-amplitude of the ramp and the duration of the segment (Δy/Δx).

Mathematical description

$$f(t) = \frac{y_2 - y_1}{T_D} \cdot t + y_1$$

y_1: Start amplitude

y_2: Stop amplitude

T_D: Duration

HILAPI-Description

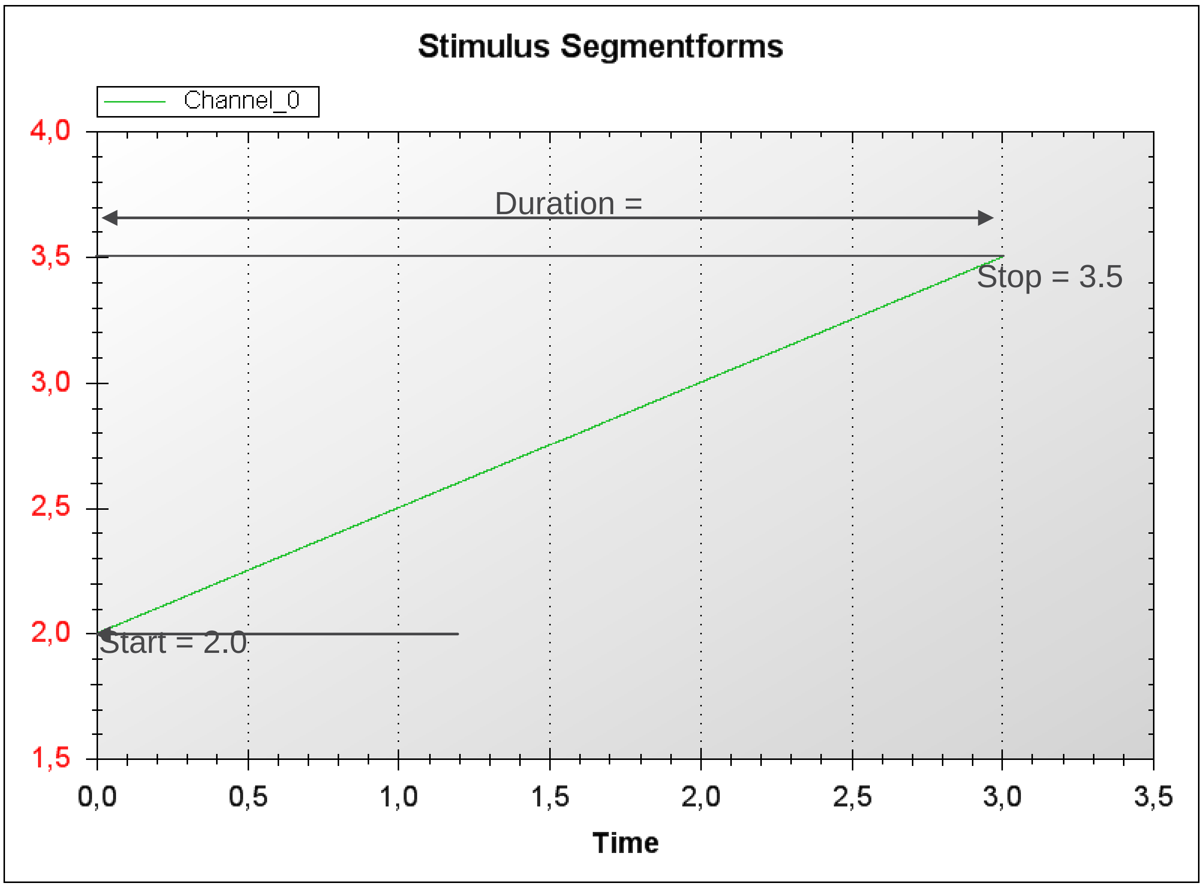


Figure 29 RampSegment

Parameters:

`Duration`:	Duration / run time of the segment	
	Unit:	Seconds [s]
	Range:	[0 < Duration <= MAX(A_FLOAT64)]
`Start`:	Start value of the amplitude	
	Unit:	-
	Range:	[MIN(A_FLOAT64) <= Start <= MAX(A_FLOAT64)]
`Stop`:	Stop value of the amplitude	
	Unit:	-
	Range:	[MIN(A_FLOAT64) <= Stop <= MAX(A_FLOAT64)]

4.7.3.3 IdleSegment

The `IdleSegment` sets the signal generation into idle-mode for the given duration. During this idle time the signal generator will not write to the corresponding model variable, respectively the memory location of the model variable.
The `IdleSegment` is normally used to allow other parts of the model to write to the variable (eg. model-i/o or user interaction).
If the variable was not written during the idle time by some other parts of the model, the variable is left untouched and will keep its value.

Mathematical description

-

HILAPI-Description

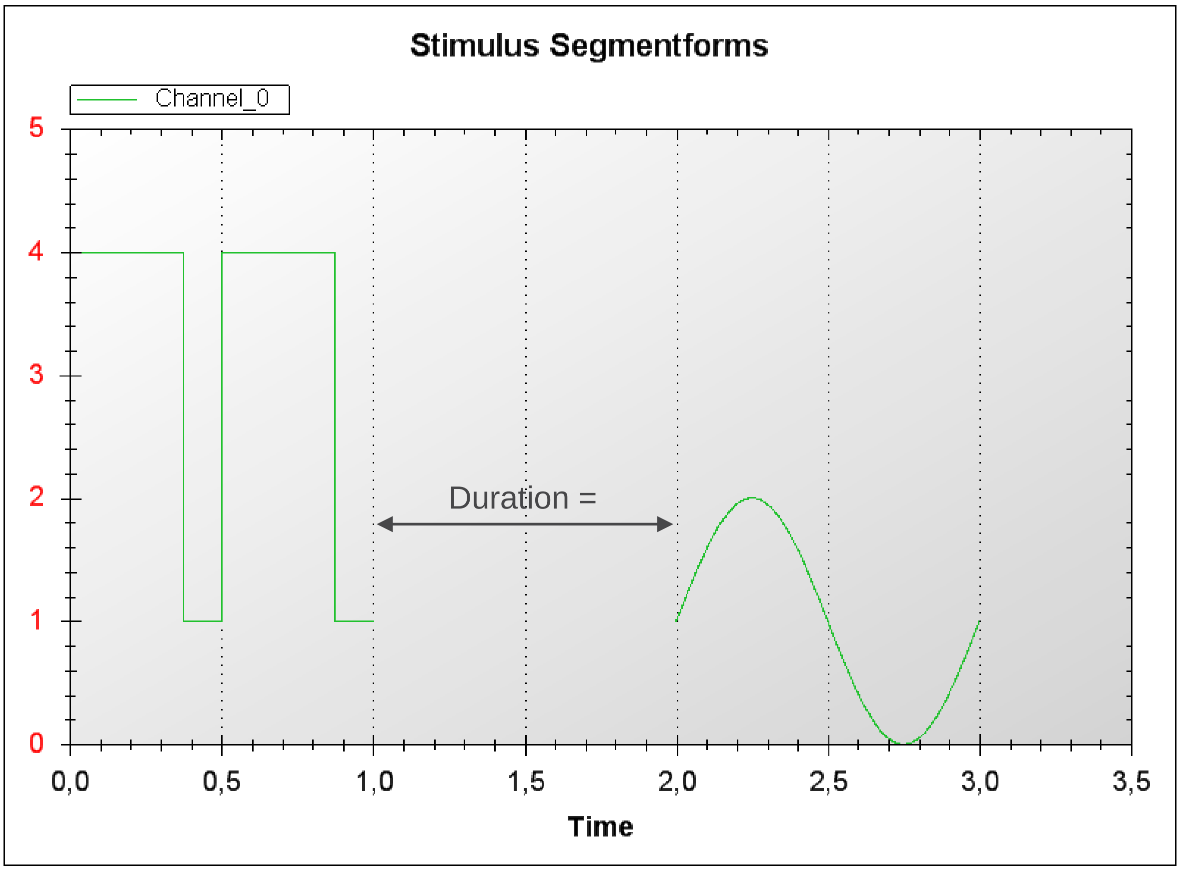


Figure 30 IdleSegment

Parameters:

`Duration`: Duration / run time of the segment
Unit: Seconds [s]
Range: [0 < Duration <= MAX(A_FLOAT64)]

4.7.3.4 NoiseSegment

The `NoiseSegment` is used to generate a part (segment) of the signal with gaussian noise. That means that the amplitude of the signal is gaussian distributed.
In each model step one noise value is calculated by using a random generator. The generated random value is than applied against the gaussian distribution to get amplitude values according to the gaussian bell-shaped curve.

Mathematical description

Gaussian Distribution:

$$f(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

Box-Muller-Method:
From two standard independent random numbers u_1 and u_2 in the range 0..1 (e.g. generated via random()) two standard normal-distributed and independent random numbers z_1 and z_2 will be created.

$$z_1 = \sqrt{-2 \cdot \ln(1-u_1)} \cdot \cos(2\pi \cdot u_2)$$

and

$$z_2 = \sqrt{-2 \cdot \ln(1-u_1)} \cdot \sin(2\pi \cdot u_2)$$

With

$$x_i = \mu + \sigma \cdot z_i$$

It is possible to generate normal distributed random numbers x_i with any mean and sigma parameters you need.
μ: Mean value
σ: Standard deviation

Note: The Box-Muller-Method is used by the Python function *random.gauss(mu, sigma)*.

HILAPI-Description

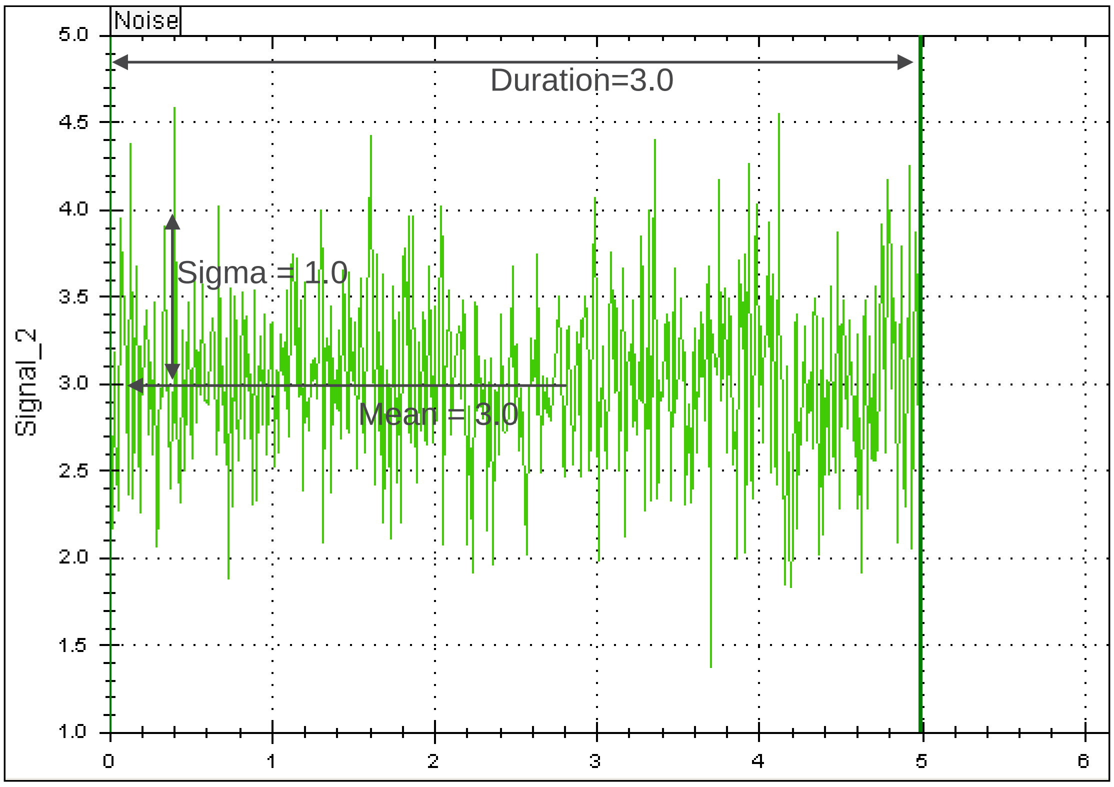


Figure 31 NoiseSegment

Parameters:

`Duration`:	Duration / run time of the segment Unit: Seconds [s] Range: [0 < Duration <= MAX (A_FLOAT64)]
`Mean`:	Mean value, where the Gaussian distribution is moving Unit: - Range: [MIN (A_FLOAT64) <= Mean <= MAX (A_FLOAT64)]
`Sigma`:	Standard deviation of the signal amplitude against the mean value Unit: - Range: [MIN (A_FLOAT64) <= Sigma <= MAX (A_FLOAT64)]
`Seed`:	Start value of the random generator Unit: - Range: [-2147483646 <= Seed <= +2147483645]

4.7.3.5 RampSlopeSegment

The `RampSlopeSegment` is used to generate a part (segment) of the signal with a ramp-shaped signal flow. The amplitude of the signal follows a straight line according to a linear equation.
The segment form is similar to `RampSegment`. Only the parameters are different.

<u>Mathematical description</u>

$f(t) = m \cdot t + b$

m : Slope of the line

b : Offset of the line

<u>HILAPI-Description</u>

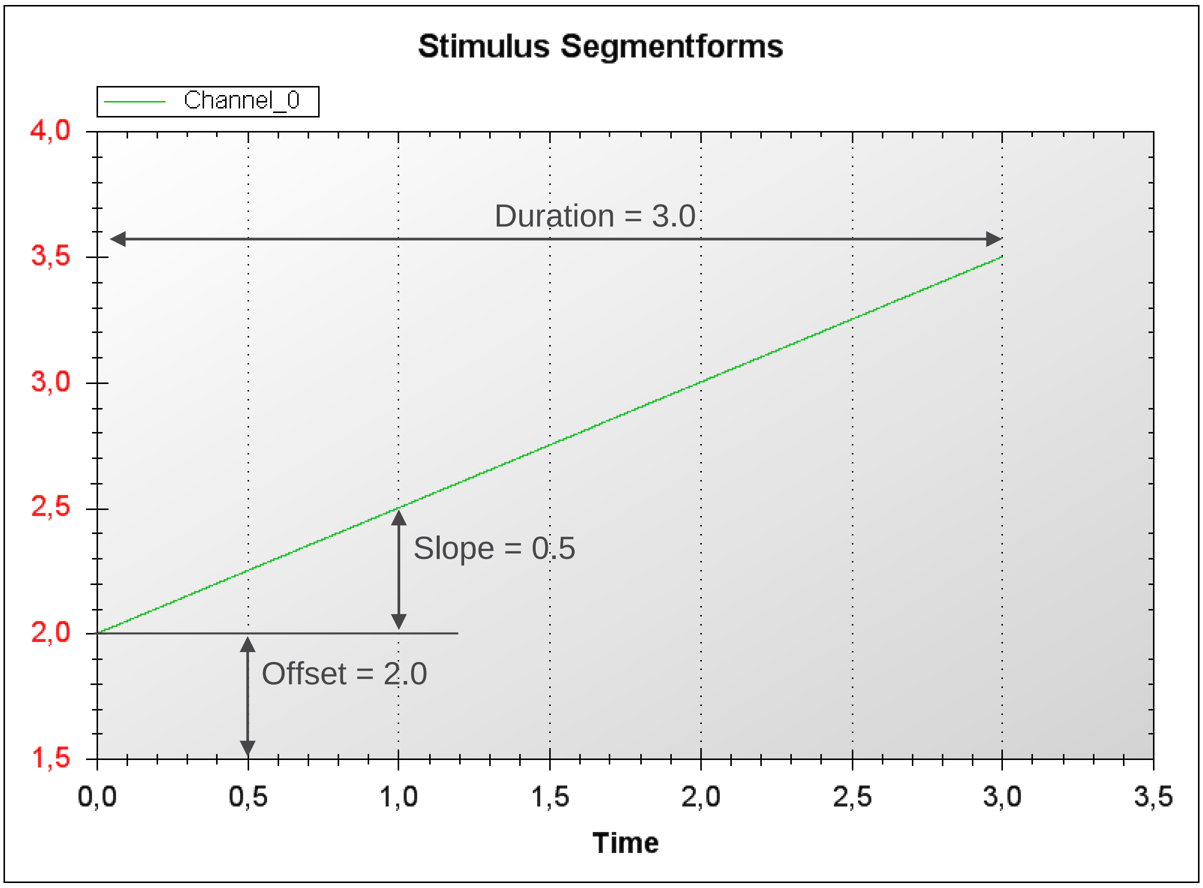


Figure 32 RampSlopeSegment

Parameters:

`Duration`:	Duration / run time of the segment	
	Unit:	Seconds [s]
	Range:	[0 < Duration <= MAX (A_FLOAT64)]
`Offset`:	Offset of the ramp	
	Unit:	-
	Range:	[MIN (A_FLOAT64) <= Offset <= MAX (A_FLOAT64)]
`Slope`:	Slope of the ramp	
	Unit:	-
	Range:	[MIN (A_FLOAT64) <= Slope <= MAX (A_FLOAT64)]

4.7.3.6 SINESEGMENT

The `SineSegment` is used to generate a part (segment) of the signal with a sine-shaped signal flow. The amplitude of the signal follows a periodical sine-waveform.

<u>Mathematical description</u>

$$f(t) = A \cdot \sin(\frac{2\pi}{T} \cdot t + \varphi) + b$$

A: Amplitude of the Signal
T: Cycle time
φ: Initial phase shift
b: Offset of the Signal

<u>HILAPI-Description</u>

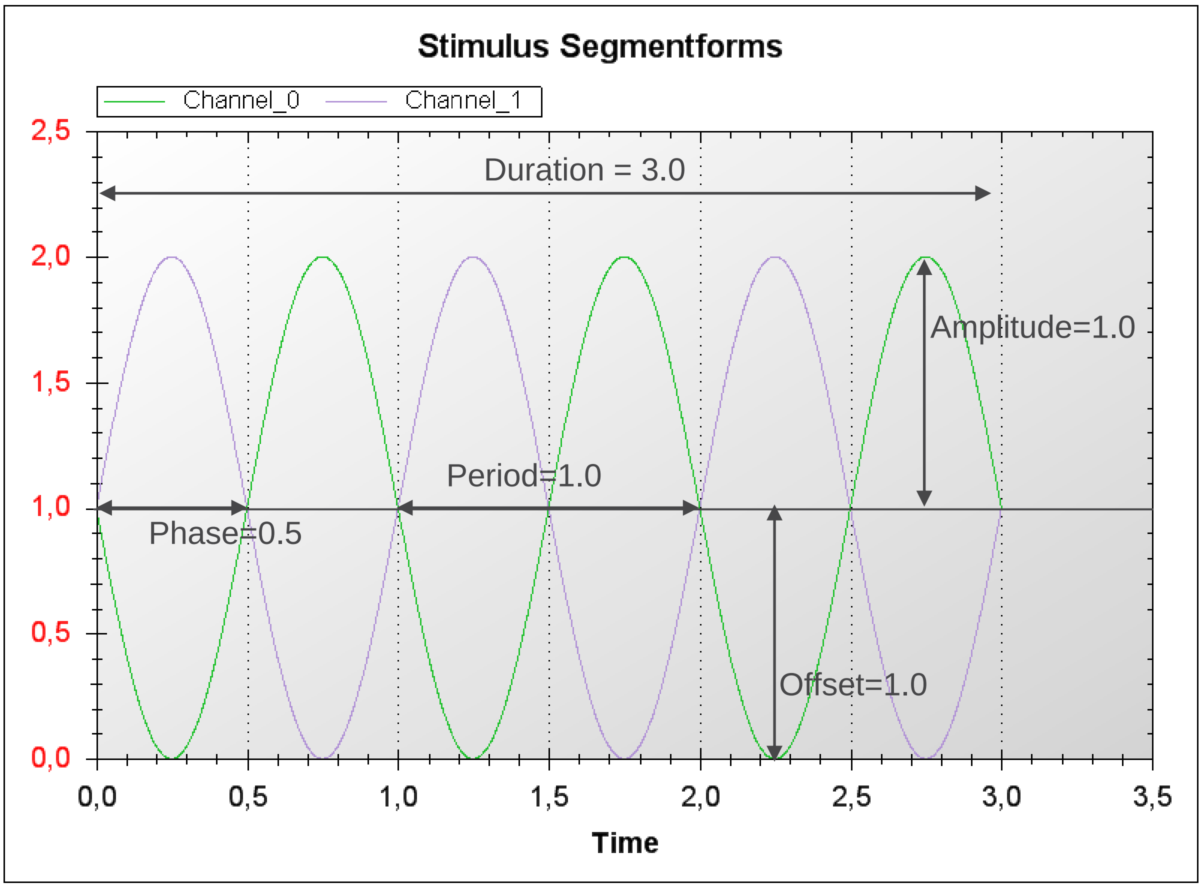


Figure 33 SineSegment

Parameters:

`Duration`:	Duration / run time of the segment	
	Unit:	Seconds [s]
	Range:	[0 < Duration <= MAX (A_FLOAT64)]
`Offset`:	Offset of the sine waveform	
	Unit:	-
	Range:	[MIN (A_FLOAT64) <= Offset <= MAX (A_FLOAT64)]
`Period`:	Cycle time of the sine waveform	
	Unit:	-
	Range:	[MIN (A_FLOAT64) <= Period <= MAX (A_FLOAT64)]
`Amplitude`:		Amplitude of the sine waveform
	Unit:	-
	Range:	[MIN (A_FLOAT64) <= Amplitude <= MAX (A_FLOAT64)]
`Phase`:	Initial phase shift as positive or negative factor of the cycle time	
	Unit:	-
	Range:	[-1.0 <= Phase <= +1.0]
	(0.25 is equal to 90° phase shift, -0.33 is equal to -120° phase shift)	

4.7.3.7 SAWSEGMENT

The `SawSegment` is used to generate a part (segment) of the signal with a saw tooth shaped or triangle shaped signal flow. The amplitude of the signal follows a periodical saw tooth waveform.

Mathematical description

$$f(t) = \begin{cases} \frac{A}{t_r} t + b & 0 < t + \varphi < t_r \\ -\frac{A}{t_f} t + b & t_r < t + \varphi < t_f \end{cases}, \; t_r = T \cdot \delta, \; t_f = T - t_r, \; t_r \neq 0, \; t_f \neq 0$$

A: Amplitude of the Signal
T: Cycle time
δ: Duty cycle (ratio of rise-time to cycle-time)
t_r: Rise time
t_f: Fall time
φ: Initial phase shift
b: Offset of the Signal

HILAPI-Description

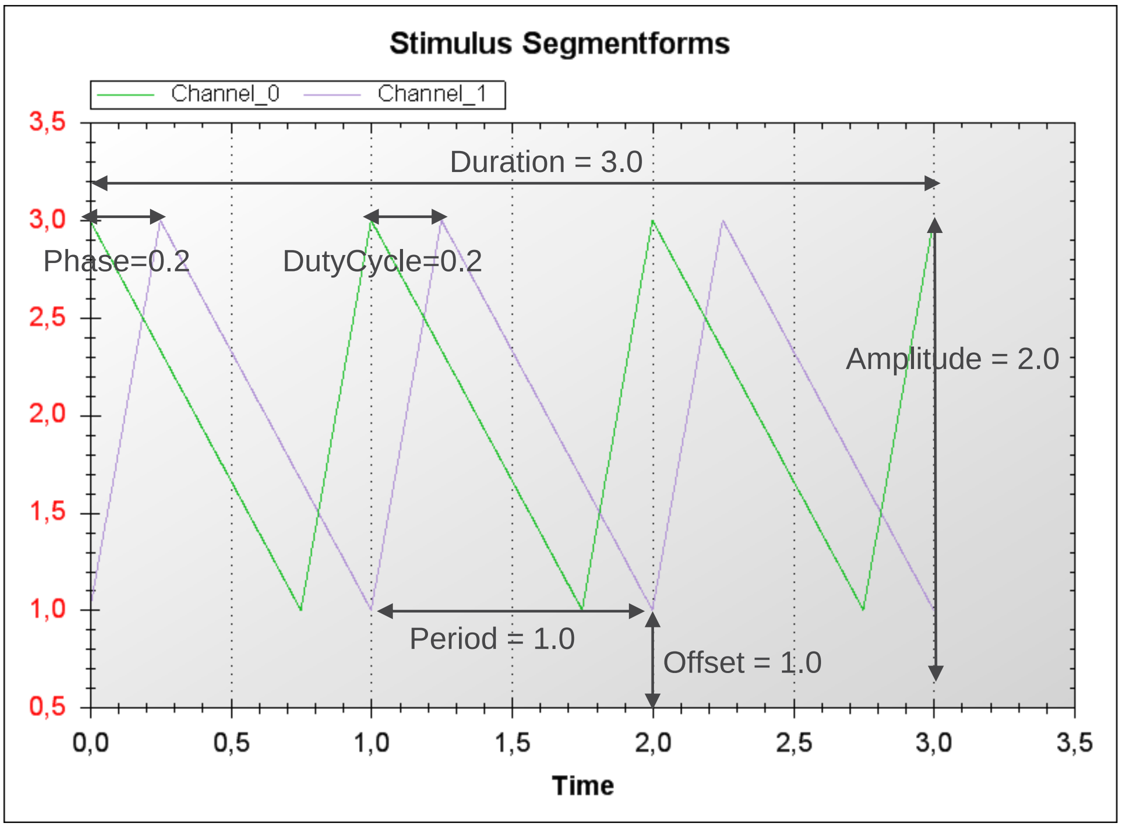


Figure 34 SawSegment

Parameters:

`Duration`:	Duration / run time of the segment Unit: Seconds [s] Range: [0 < Duration <= MAX (A_FLOAT64)]
`Offset`:	Offset of the saw tooth waveform Unit: - Range: [MIN (A_FLOAT64) <= Offset <= MAX (A_FLOAT64)]
`Period`:	Cycle time of the saw tooth waveform Unit: - Range: [MIN (A_FLOAT64) <= Period <= MAX (A_FLOAT64)]
`Amplitude`:	Amplitude of the saw tooth waveform Unit: - Range: [MIN (A_FLOAT64) <= Amplitude <= MAX (A_FLOAT64)]
`Phase`:	Initial phase shift as positive or negative factor of the cycle time Unit: - Range: [-1.0 <= Phase <= +1.0] (0.25 is equal to 90° phase shift, -0.33 is equal to -120° phase shift)
`DutyCycle`	Ratio of raise-time to cycle-time as a positive factor Unit: - Range: [0.0 <= DutyCycle <= 1.0] (use 0.5 to get a triangular shaped signal)

4.7.3.8 PULSESEGMENT

The `PulseSegment` is used to generate a part (segment) of the signal with a rectangular-shaped signal flow. The amplitude of the signal follows a periodical rectangle-waveform.

<u>Mathematical description</u>

$$f(t) = \begin{cases} A + b & 0 < t + \varphi < t_h \\ b & t_h < t + \varphi < T \end{cases}, \; t_h = T \cdot \delta$$

A: Amplitude of the Signal
T: Cycle time
t_h: High-time
b: Offset of the Signal

<u>HILAPI-Description</u>

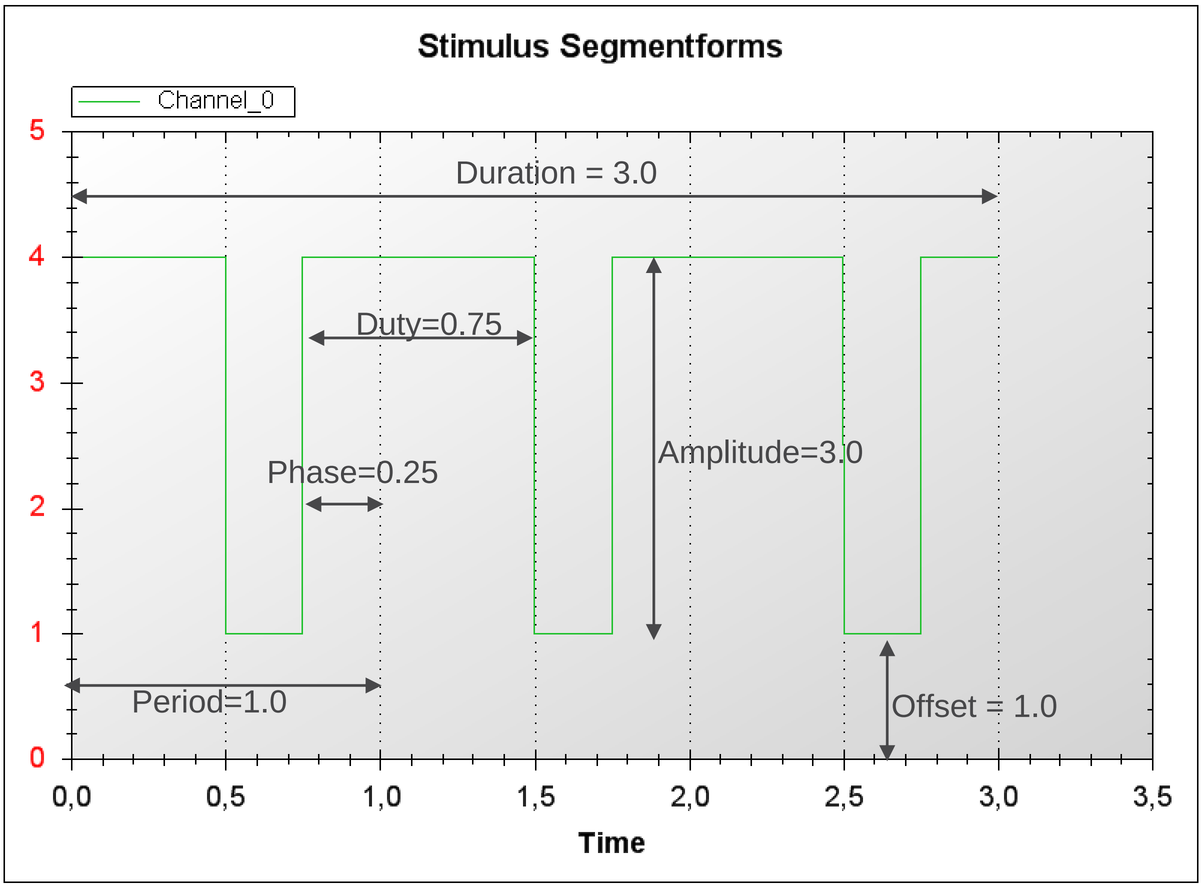


Figure 35 PulseSegment

Parameters:

`Duration`:	Duration / run time of the segment
	Unit: Seconds [s]
	Range: [0 < Duration <= MAX (A_FLOAT64)]
`Offset`:	Offset of the rectangle waveform
	Unit: -
	Range: [MIN (A_FLOAT64) <= Offset <= MAX (A_FLOAT64)]
`Period`:	Cycle time of the rectangle waveform
	Unit: -
	Range: [MIN (A_FLOAT64) <= Period <= MAX (A_FLOAT64)]
`Amplitude`:	Amplitude of the rectangle waveform
	Unit: -
	Range: [MIN (A_FLOAT64) <= Amplitude <= MAX (A_FLOAT64)]
`Phase`:	Initial phase shift as positive or negative factor of the cycle time
	Unit: -
	Range: [-1.0 <= Phase <= +1.0]
	(0.25 is equal to 90° phase shift, -0.33 is equal to -120° phase shift)
`DutyCycle`:	Ratio of high-time to cycle-time as a positive factor
	Unit: -
	Range: [0.0 <= DutyCycle <= 1.0]
	(use 0.5 to get a symmetric rectangular shaped signal, use 1.0 to get a constant value)

4.7.3.9 EXPSEGMENT

The `ExpSegment` is used to generate a part (segment) of the signal with an exponential-shaped signal flow. The amplitude of the signal follows an exponential curve.

<u>Mathematical description</u>

$$f(t) = A \cdot (1 - e^{-\frac{t}{\tau}}) + b$$

A: Amplitude of the Signal
τ : Time constant (tau)
b : Offset of the Signal

<u>HILAPI-Description</u>

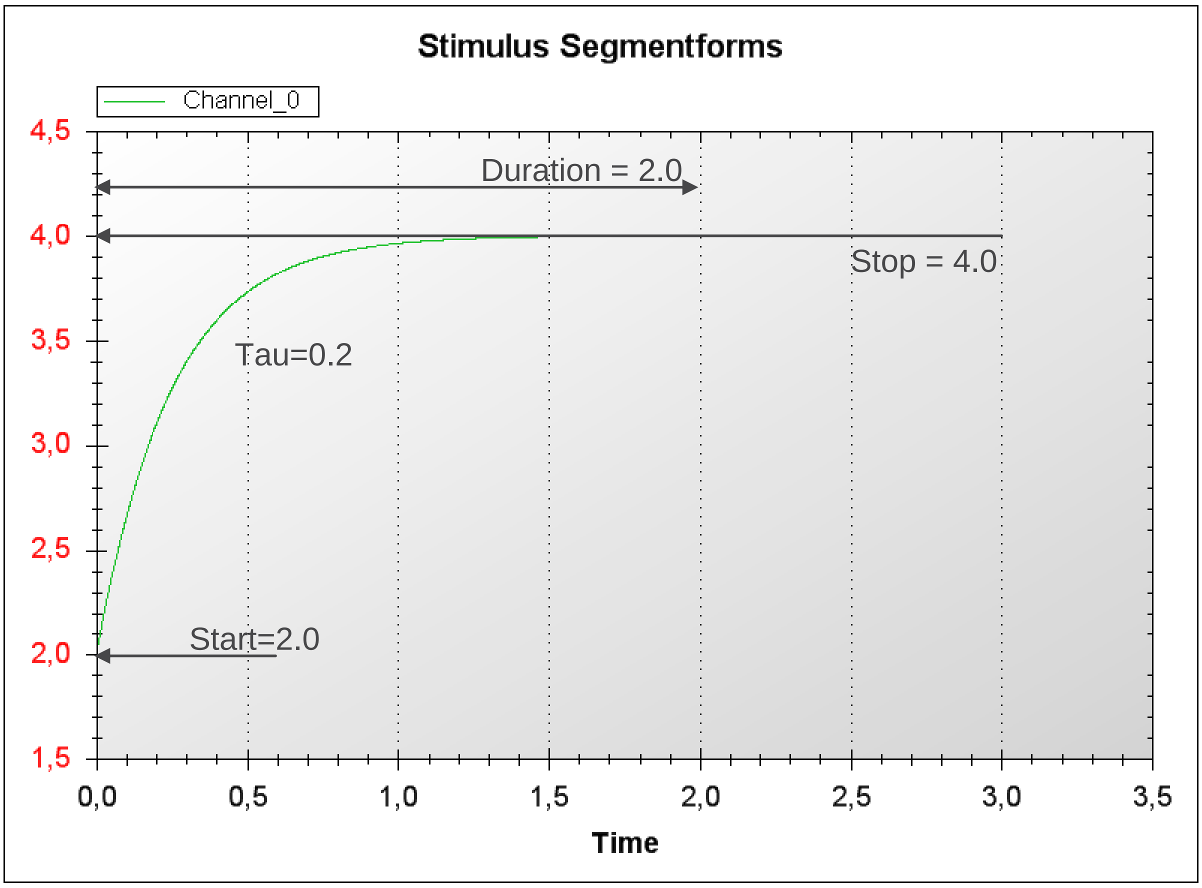


Figure 36 ExpSegment

Parameters:

`Duration`:	Duration / run time of the segment	
	Unit:	Seconds [s]
	Range:	[0 < Duration <= MAX (A_FLOAT64)]
`Start`:	Start amplitude (Offset of the Signal)	
	Unit:	-
	Range:	[MIN (A_FLOAT64) <= Start <= MAX (A_FLOAT64)]
`Stop`:	Stop amplitude	
	(Note: Amplitude of the Signal *A* = Stop – Start)	
	Unit:	-
	Range:	[MIN (A_FLOAT64) <= Stop <= MAX (A_FLOAT64)]
`Tau`:	Time constant of the e-curve	
	Unit:	Seconds [s]
	Range:	[MIN (A_FLOAT64) <= Tau <= MAX (A_FLOAT64)]

4.7.3.10 SIGNALVALUESEGMENT

The `SignalValueSegment` is used to generate a part (segment) of the signal which directly uses numerical data. The amplitude of the signal follows directly the given data-points.
Normally this segment is used to replay measured data.
The numerical (respective measured) data is stored in a `SignalValue` object (see chapter 4.4) which is given during creation of the segment or during configuration of the segment. The duration of the segment is derived from the time vector.
The serialization of the numerical data (e.g. `SignalDescriptionSet.Save()`) is done by generating a flat MAT-File with two vectors of type double. The first vector describes the time vector, and the second vector describes the corresponding signal amplitude values.
The duration of the segment is implicitly derived from the time vector.

Mathematical description

$f(t) = N[t]$

N : Array with numerical data

HILAPI-Description

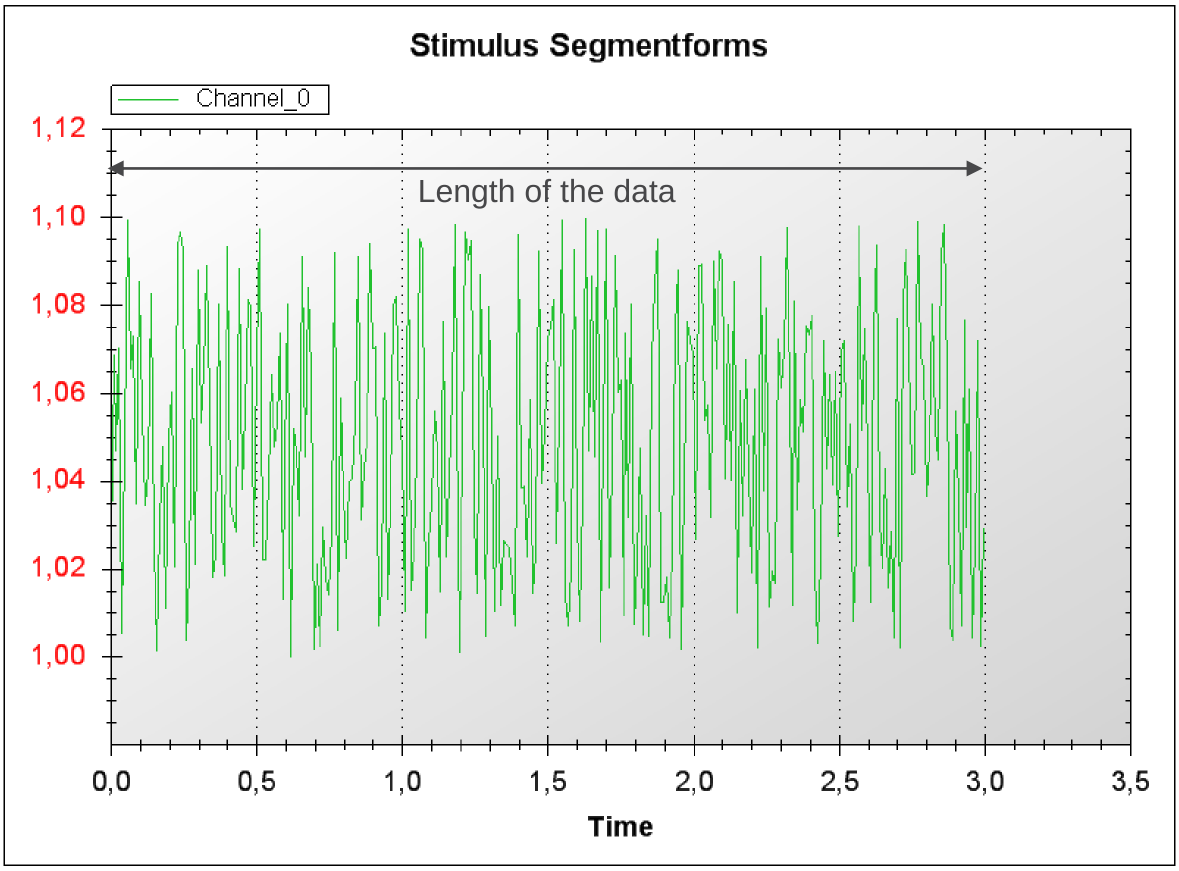


Figure 37 SignalValueSegment

Parameters:

`SignalValue`: SignalValue object which contains the time-vector and the data-vector
Unit: time-vector: Seconds [s], data-vector: -
Range: [MIN (A_FLOAT64) <= time, data <= MAX (A_FLOAT64)]

`Interpolation`: Interpolation method
Unit: -
Range: enum InterpolationTypes

eFORWARD:	Next data point will be used immediately (staircase forward)
eBACKWARD:	Actual data point will be used until next data point (staircase backward)
eLINEAR:	Linear interpolation

4.7.3.11 OperationSegment (OperationTypes)

The `OperationSegment` is used to generate a part (segment) of the signal which is a combination of two other segments. The two segments are combined by a mathematical operation like addition or multiplication. The amplitude of the signal follows the calculated result. The duration of the resulting segment is derived from the shorter segment.

Mathematical description

$$f(t) = S_1(t)\ op\ S_2(t) \qquad , op = operation$$

S_1 : First segment / first operand

S_2 : Second segment / second operand

HILAPI-Description

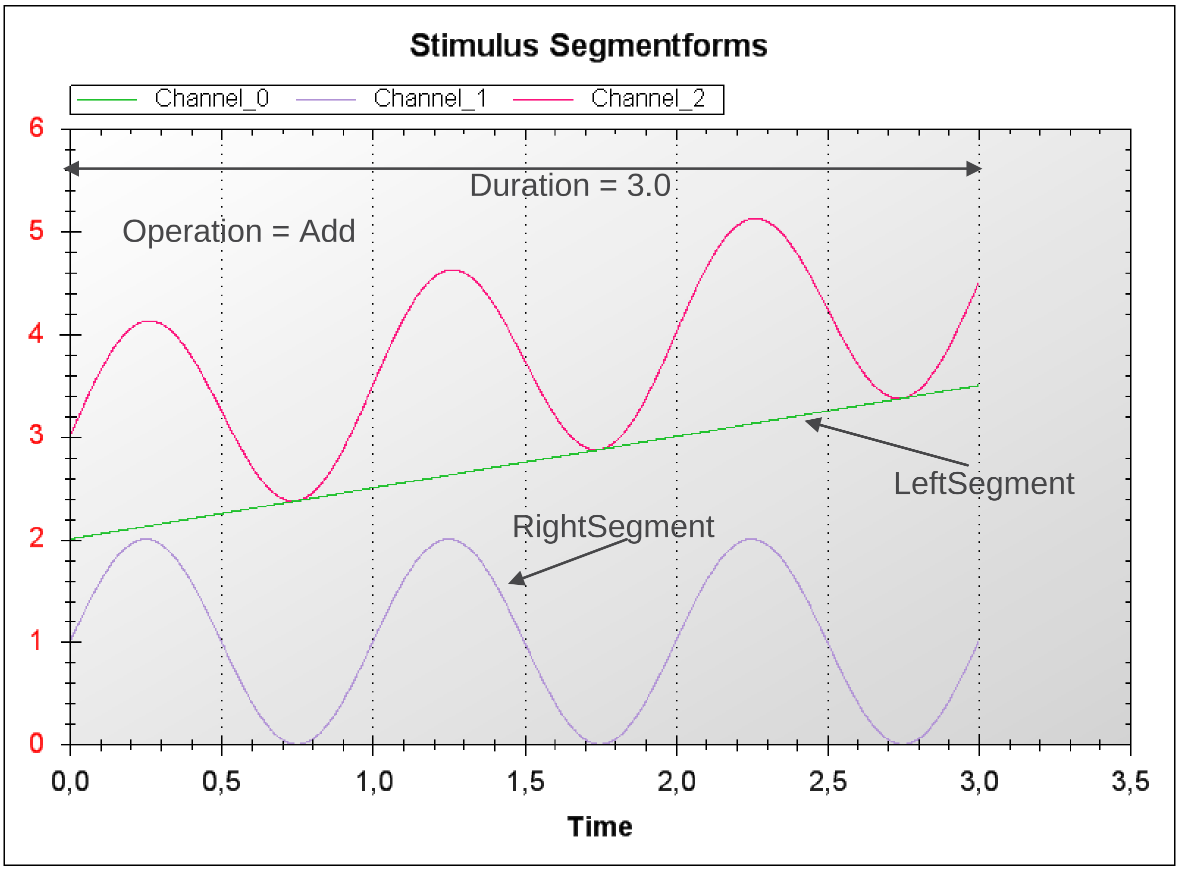


Figure 38 OperationSegment

Parameters:

`leftSegment`:	left segment object (left operand s1)		
	Unit:	-	
	Range: -		
`rightSegment`:	right segment object (right operand s2)		
	Unit:	-	
	Range: -		
`Operation`:	Operation which is used to calculate the corresponding signal		
	Unit:	-	
	Range:	enum OperationTypes	
		eADD:	Addition (y(t) = s1(t) + s2(t))
		eMULT:	Multiplication (y(t) = s1(t) * s2(t))

4.7.4 USAGE OF SIGNAL DESCRIPTION

The sample code for this example will be found at

C#	C#\SampleCode\Common\Signal\SignalExample.cs
Python	Python\SampleCode\Common\Signal\SignalExample.py
Java	JAVA\SampleCode\Common\Signal\SignalExample.java

4.7.4.1 USING DIFFERENT SEGMENTS

Each `SegmentSignalDescription` consists of one or more segments. The following sequence diagrams (Figure 39, Figure 40 and Figure 41) show the creation of all signal types.

1. ConstSegment
2. RampSegment
3. RampSlopeSegment
4. SineSegment
5. SawSegment
6. PulseSegment
7. ExpSegment
8. IdleSegment
9. Operationsegment
10. SignalValueSegment

After creating instances of the segments, these instances are added to the `SegmentSignalDescription` object.

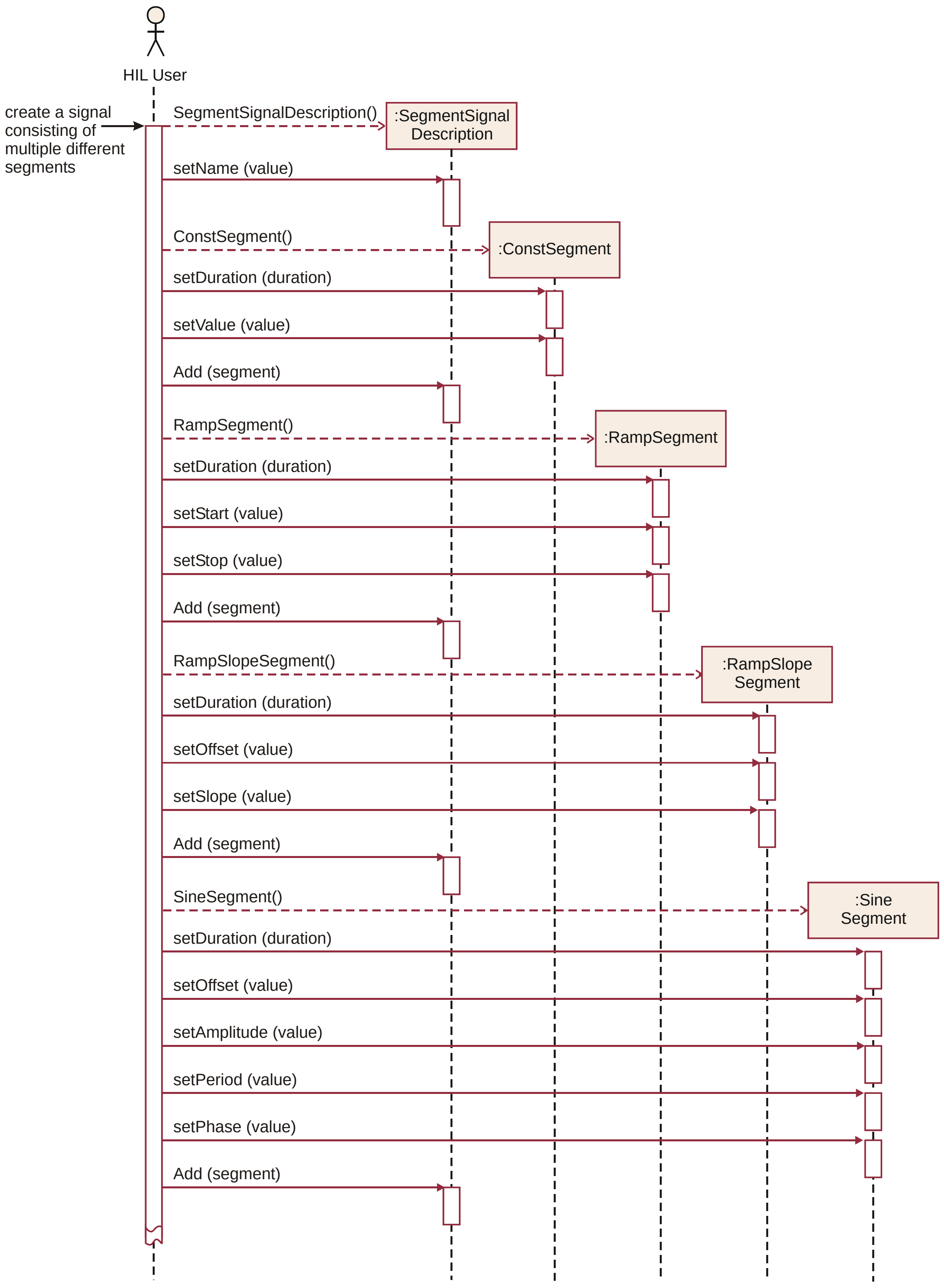


Figure 39 Signal creation: Const-, Ramp- RampSlope- and SineSegment (part 1)

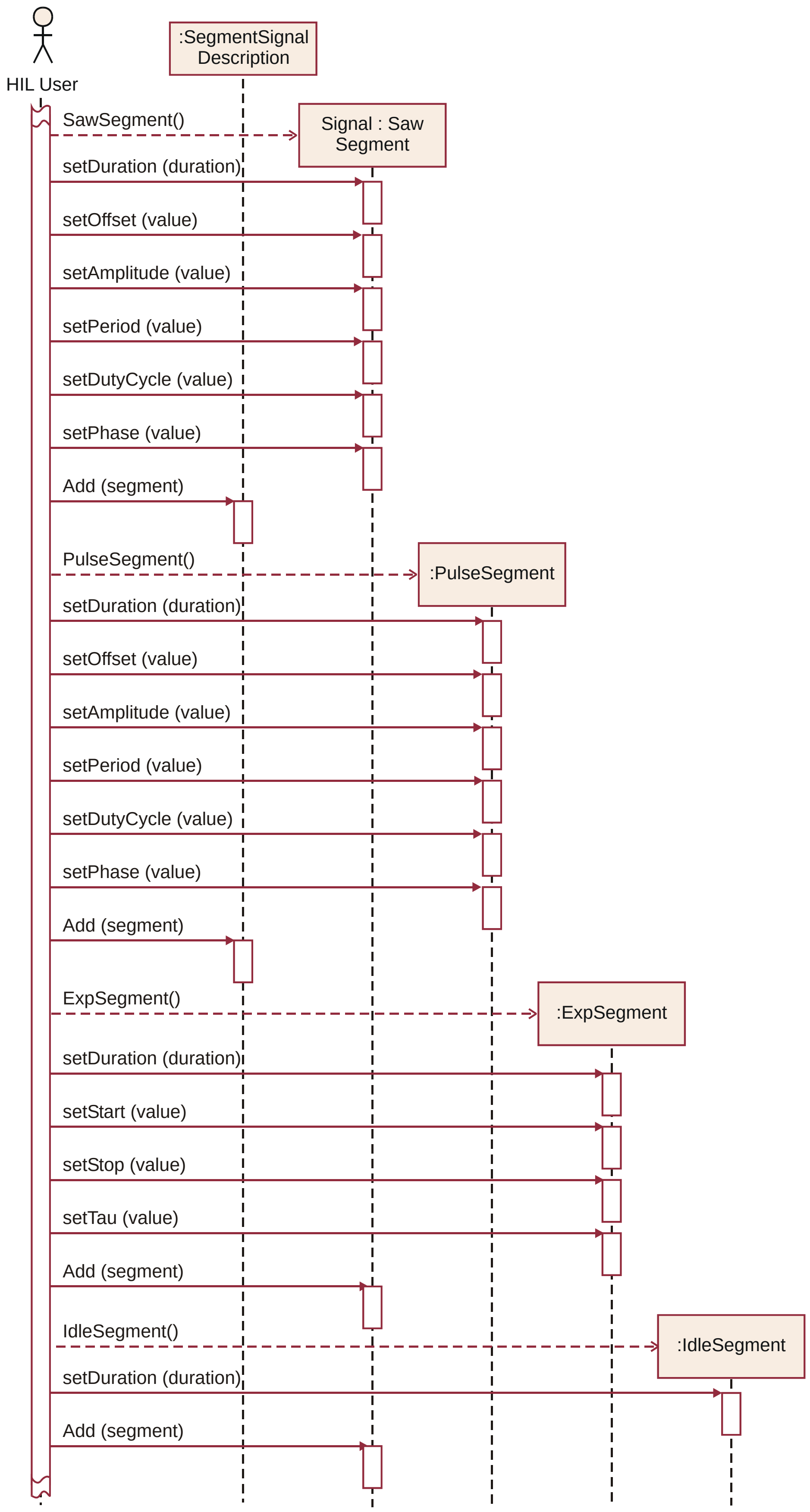


Figure 40 Signal creation: Saw-, Pulse-, Exp- and IdleSegment (part 2)

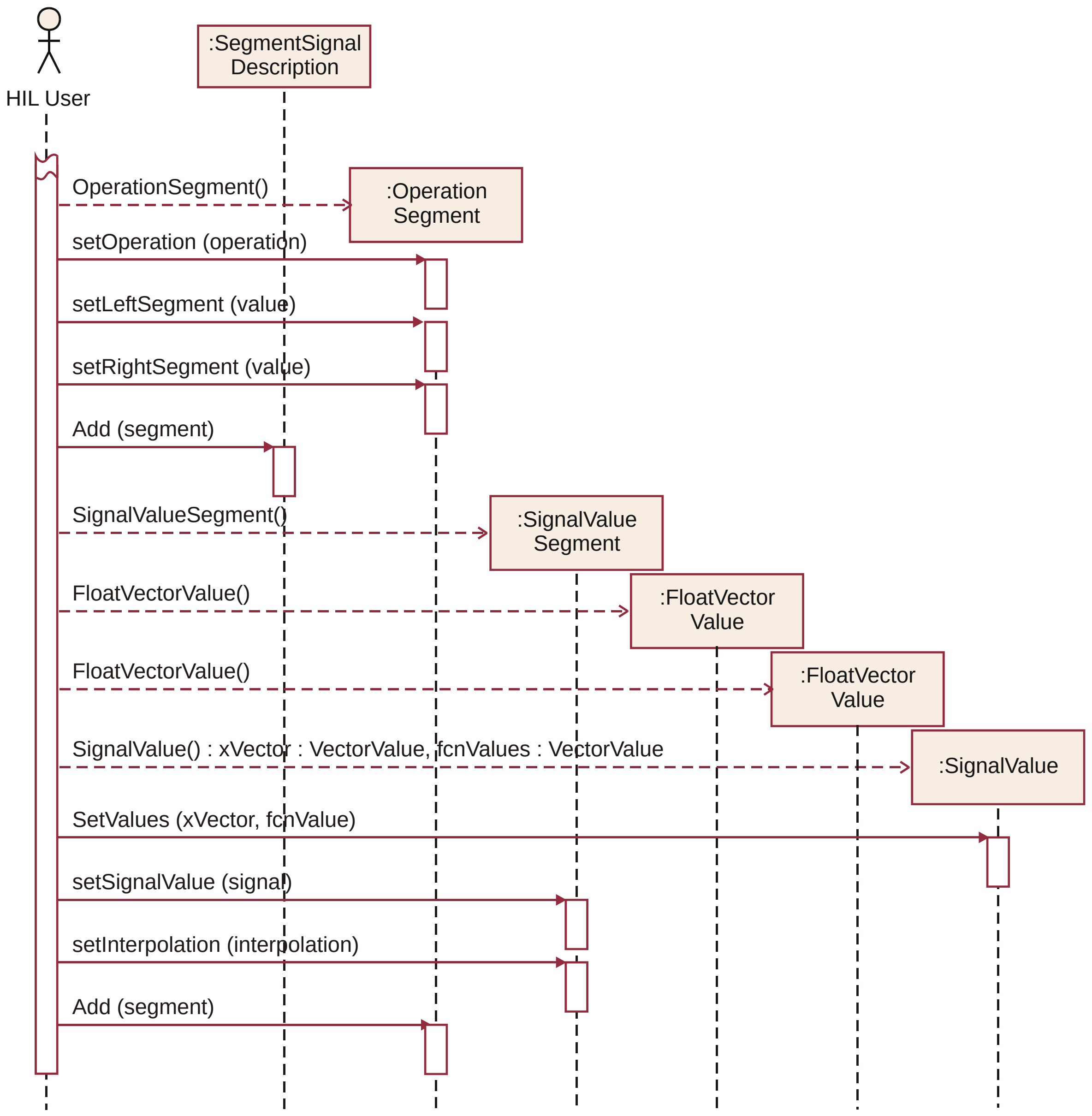


Figure 41 Signal creation: Operation- and SignalValueSegment (part 3)

4.7.4.2 CREATING AN OPERATION SIGNAL

The next sequence diagrams (Figure 42 and Figure 43) are describing the creation of an `OperationSignal` in detail. It consists of two `SegmentSignalDescriptions` which are combined by the given operation. The `SignalDescriptions` itself can have more than one signal segment inside. In this case the first has 2 signal segments (`RampSegment` and `SawSegment`) and the second has only one signal segment (`SineSegment`). The operation in this example is Multiplication.

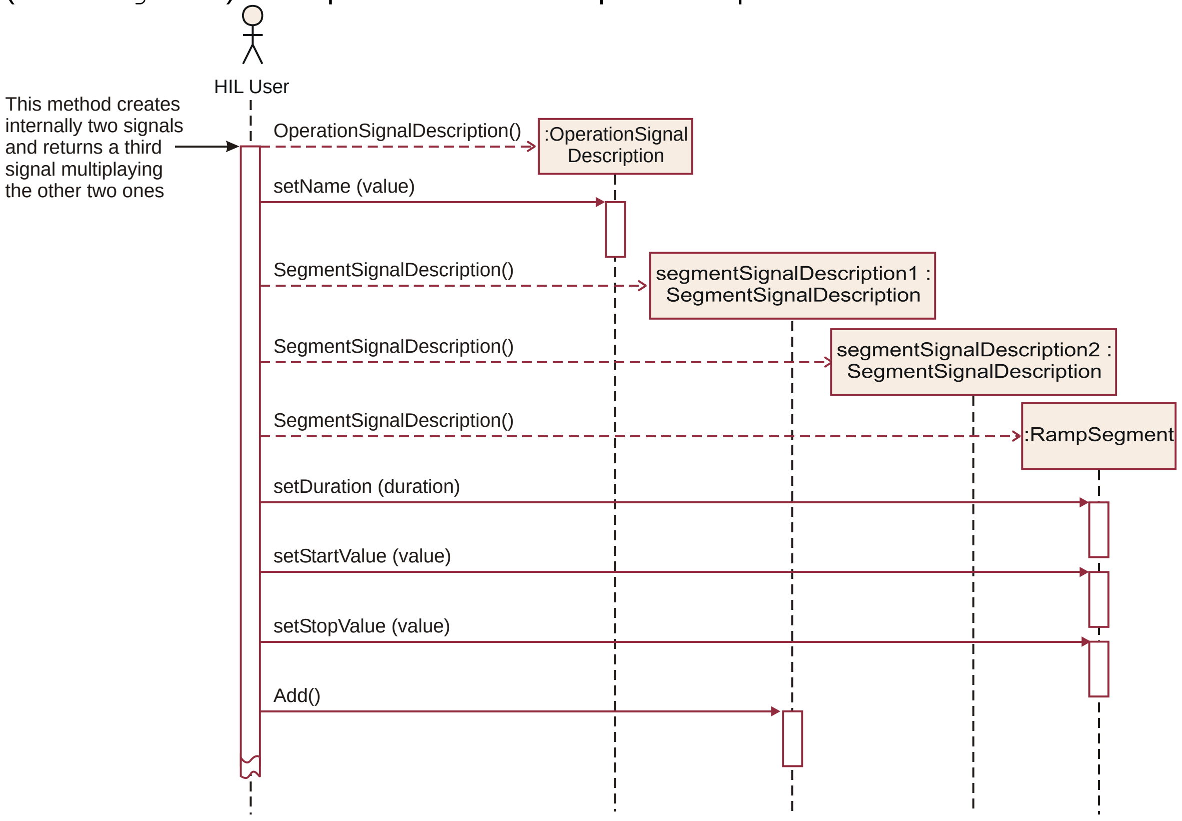

HIL User
This method creates internally two signals and returns a third signal multiplaying the other two ones
OperationSignalDescription()
:OperationSignal Description
setName (value)
SegmentSignalDescription()
segmentSignalDescription1 : SegmentSignalDescription
SegmentSignalDescription()
segmentSignalDescription2 : SegmentSignalDescription
SegmentSignalDescription()
:RampSegment
setDuration (duration)
setStartValue (value)
setStopValue (value)
Add()

Figure 42 Create an OperationSignal (part 1)

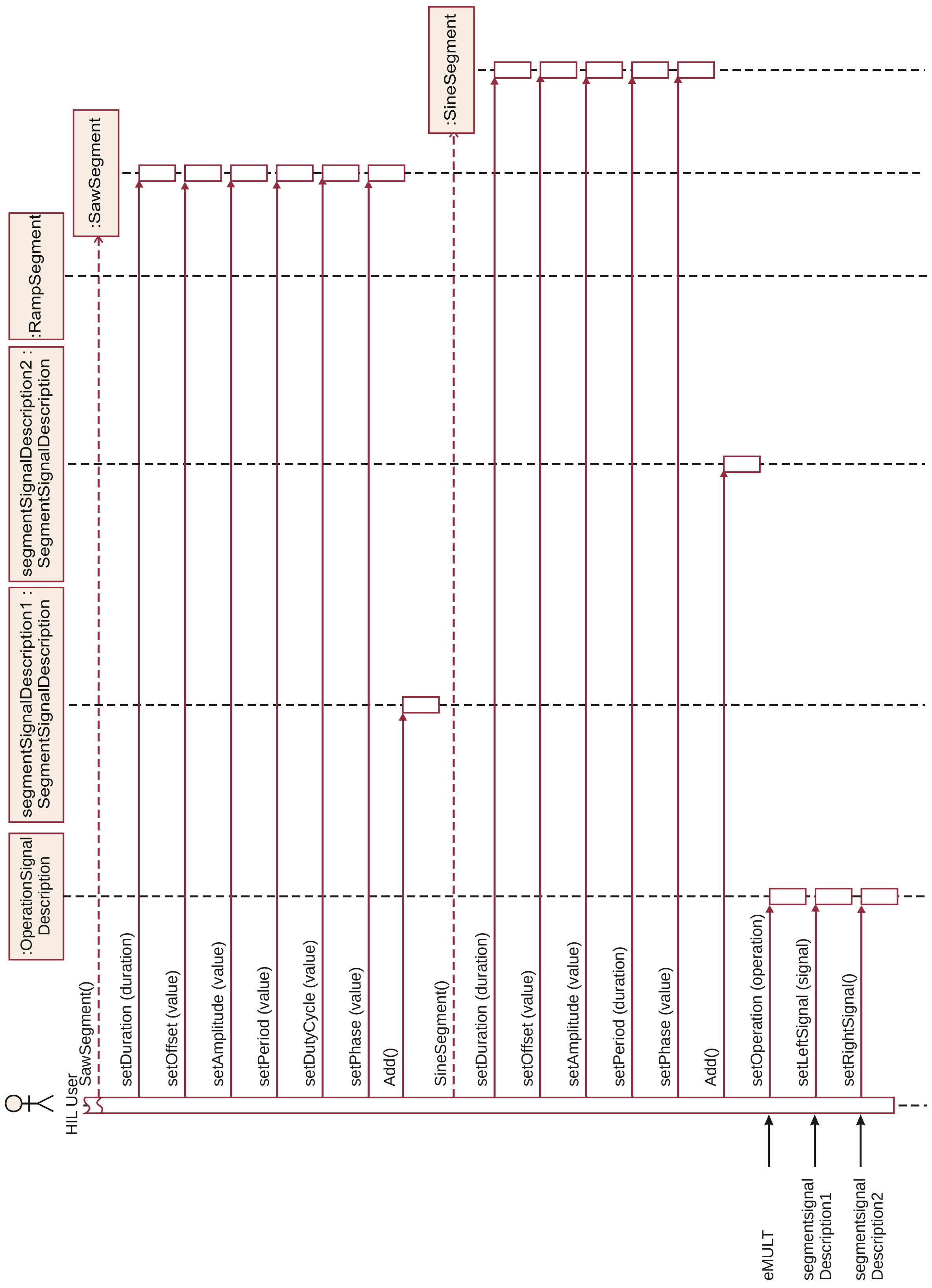


Figure 43 Create an OperationSignal (part 2)

4.7.4.3 CREATING A WOBBLE SIGNAL

In this example (Figure 44) a periodic signal is created. The frequency property is described by a saw signal, so that the sine signal is wobbling.

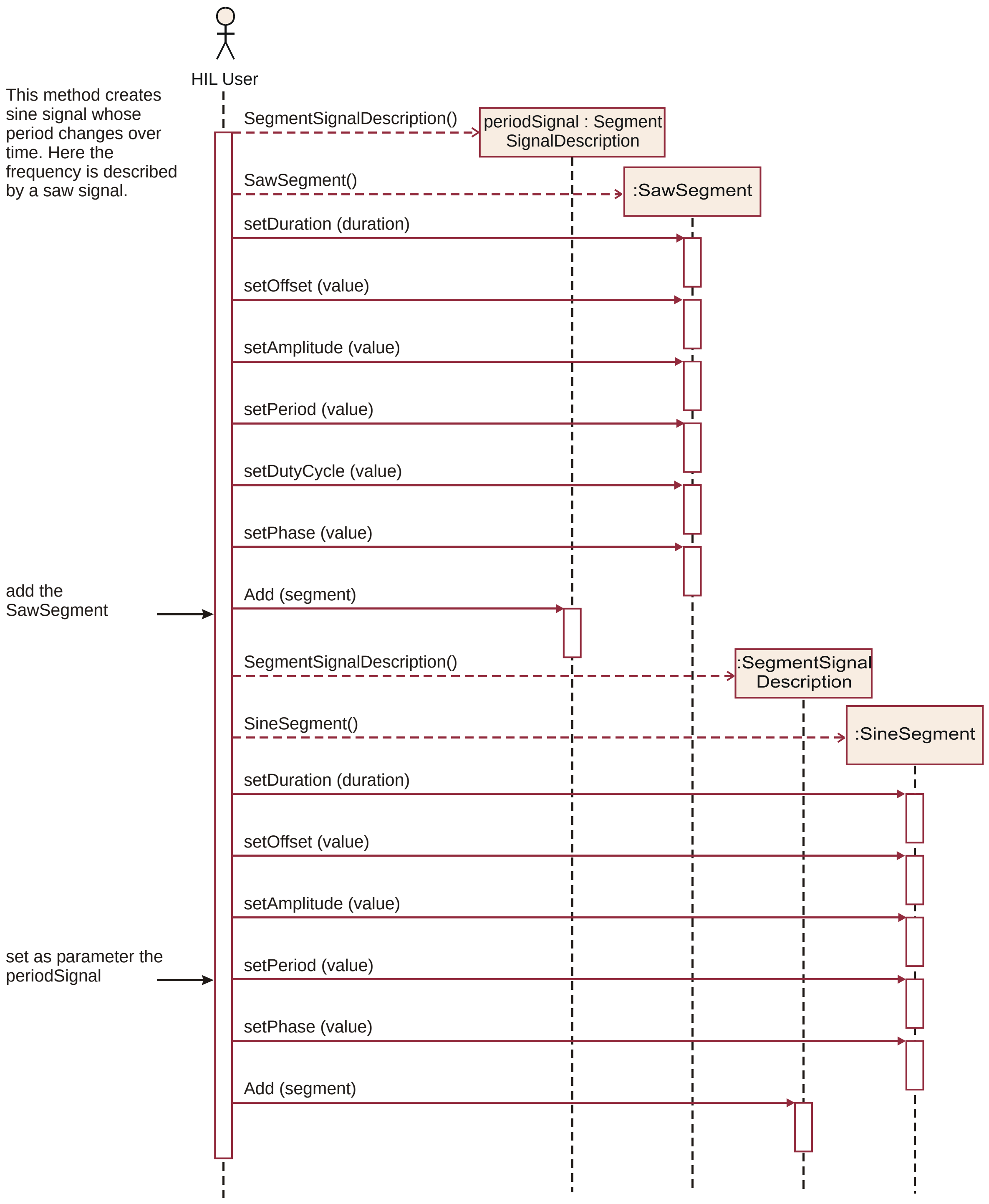


Figure 44 Create a wobbling signal

4.7.4.4 Signal Description Set

In this example (Figure 45) the access to a signal description set is shown. Two signal descriptions, already created before, are added to the signal description set. Then the set is queried for the names and the contained descriptions. Each of the descriptions is converted into a `SignalValue` object.

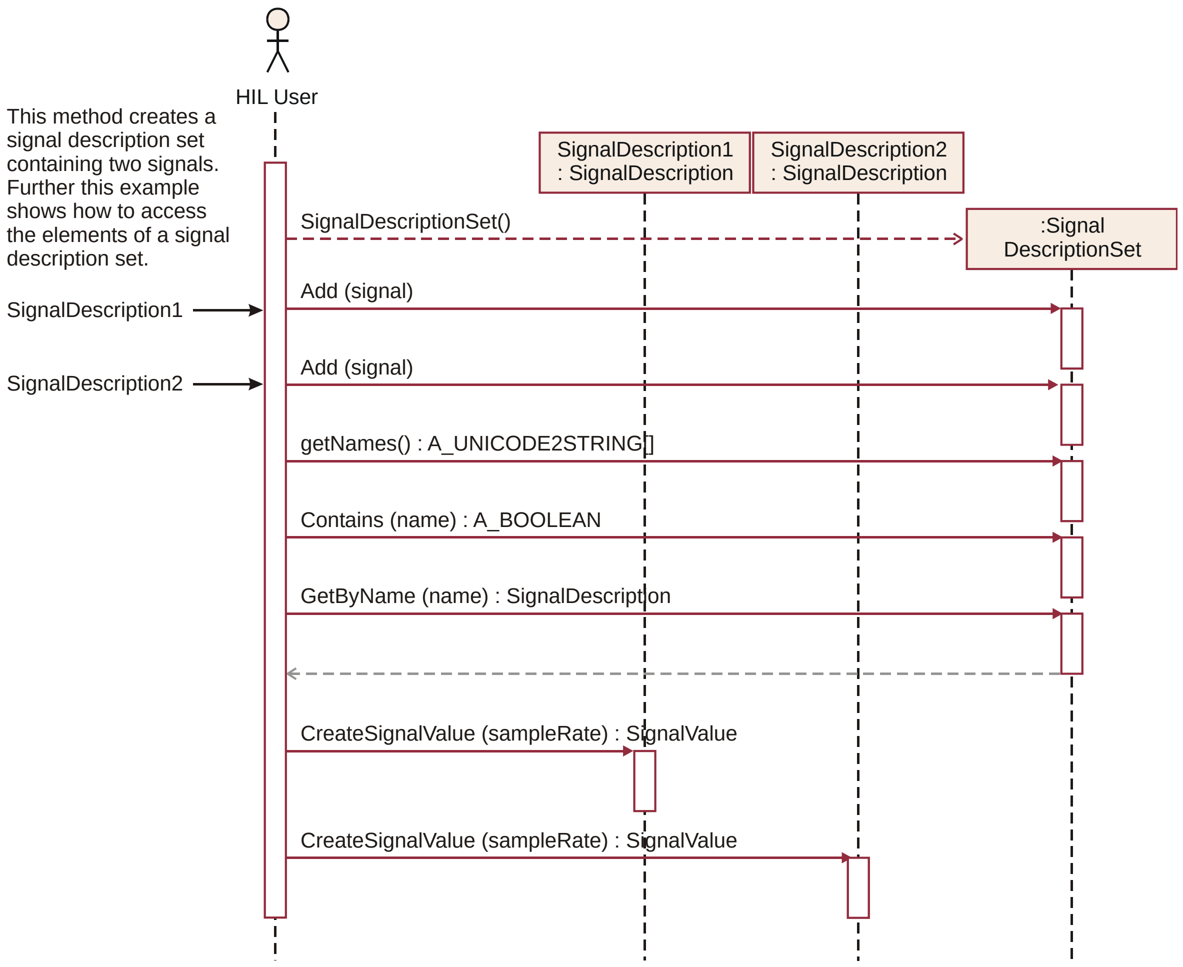

HIL User
This method creates a signal description set containing two signals. Further this example shows how to access the elements of a signal description set.
SignalDescription1
SignalDescription2
SignalDescription1 : SignalDescription
SignalDescription2 : SignalDescription
:Signal DescriptionSet
SignalDescriptionSet()
Add (signal)
Add (signal)
getNames() : A_UNICODE2STRING[]
Contains (name) : A_BOOLEAN
GetByName (name) : SignalDescription
CreateSignalValue (sampleRate) : SignalValue
CreateSignalValue (sampleRate) : SignalValue

Figure 45 Create and query a SignalDescriptionSet

4.7.4.5 LOADING THE SIGNAL DESCRIPTION

The Figure 46 shows the alternative way to get a signal description set: via loading an existing set from a STI file. Please, see SignalDescriptionFormat.xsd and SchemaDoc for the definition and a description of the STI file format. The output of this example is written to file SignalDescriptionSet.sti. From this file, the MAT file VectorData.mat is referenced, containing sample data.

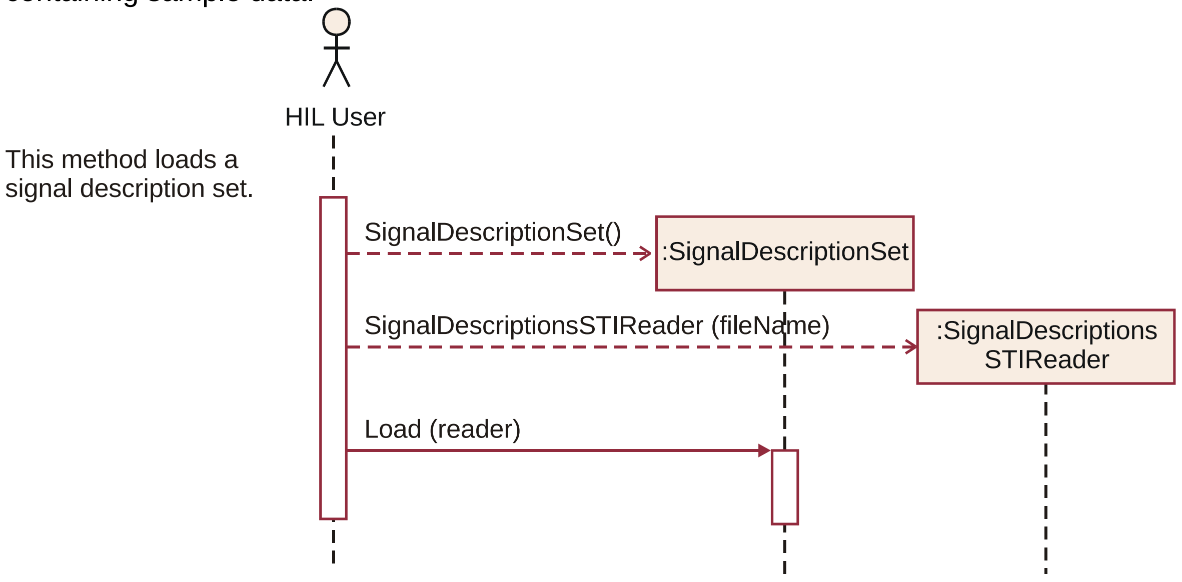

HIL User
This method loads a signal description set.
SignalDescriptionSet()
:SignalDescriptionSet
SignalDescriptionsSTIReader (fileName)
:SignalDescriptions STIReader
Load (reader)

Figure 46 Load a SignalDescriptionSet

4.7.4.6 SAVING THE SIGNAL DESCRIPTION

Figure 47 shows how to save a signal description set to a file for further reuse. Again, see SignalDescriptionFormat.xsd and SchemaDoc for the definition and a description of the STI file format.

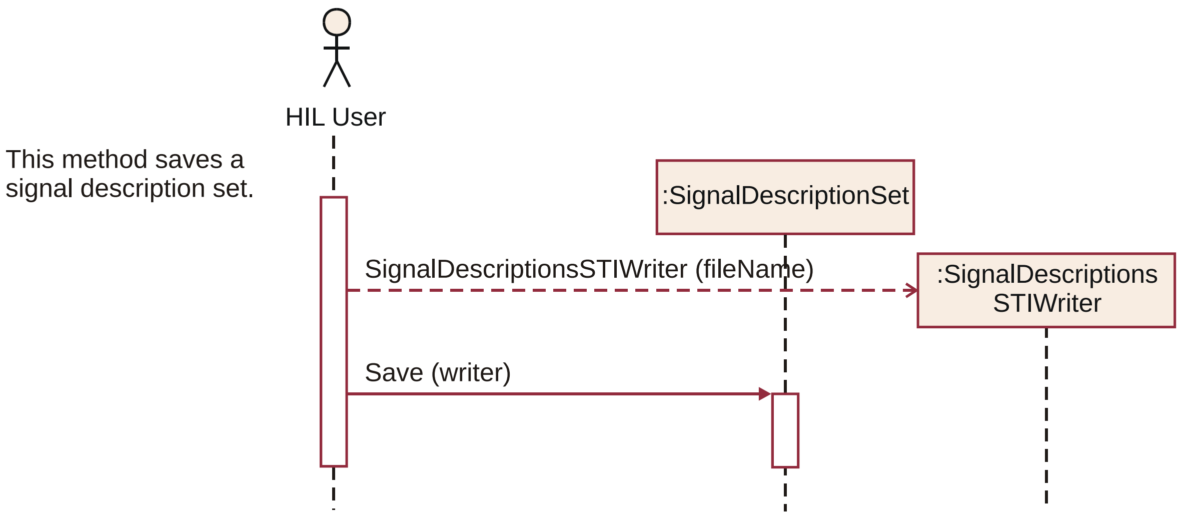

HIL User
This method saves a signal description set.
:SignalDescriptionSet
SignalDescriptionsSTIWriter (fileName)
:SignalDescriptions STIWriter
Save (writer)

Figure 47 Save signal description set

4.8 WATCHER

4.8.1 GENERAL

The `Watcher` is conceptually designed as a generic event generator. It can be used e.g. for the trigger definition of captures.

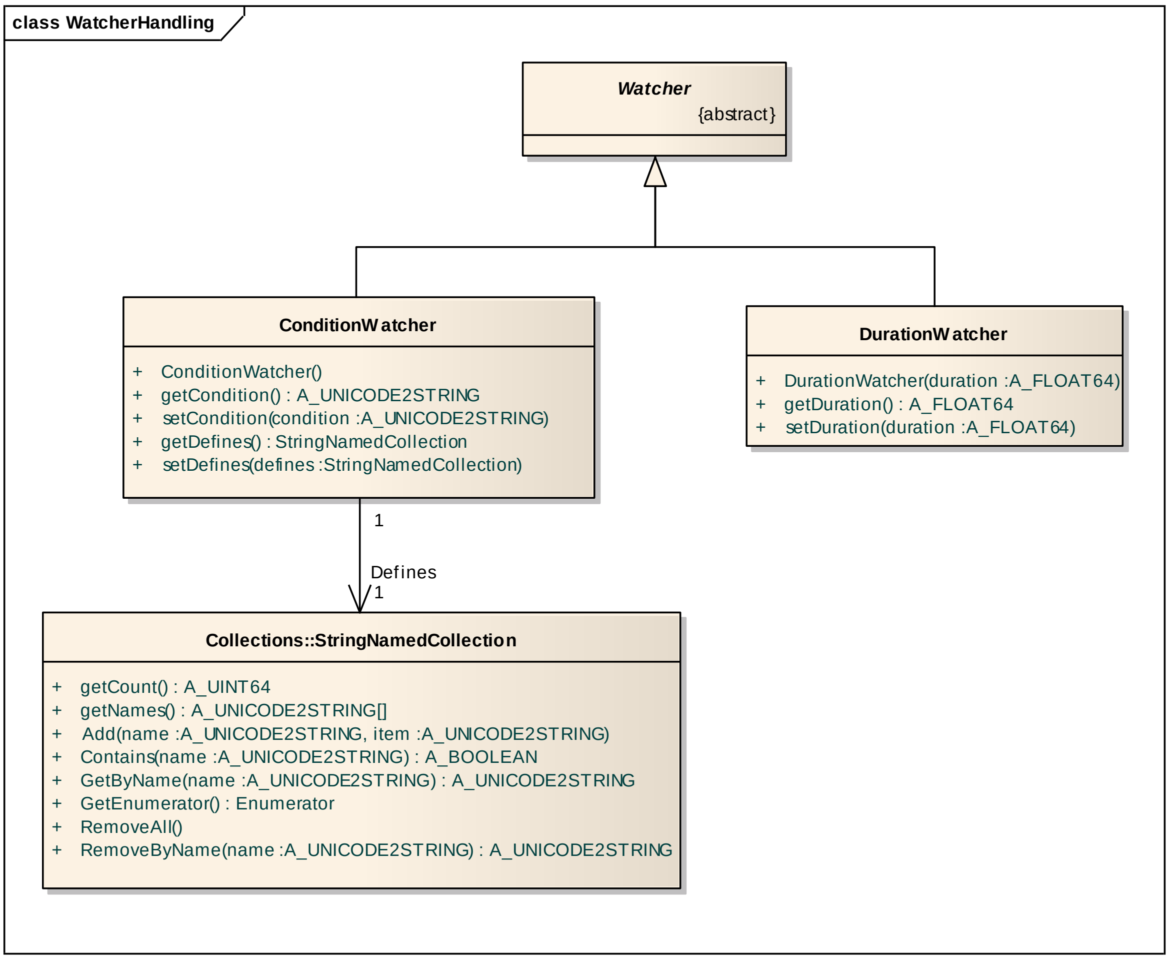

class WatcherHandling
Watcher
{abstract}
ConditionWatcher
+ ConditionWatcher()
+ getCondition() : A_UNICODE2STRING
+ setCondition(condition :A_UNICODE2STRING)
+ getDefines() : StringNamedCollection
+ setDefines(defines :StringNamedCollection)
DurationWatcher
+ DurationWatcher(duration :A_FLOAT64)
+ getDuration() : A_FLOAT64
+ setDuration(duration :A_FLOAT64)
1
Defines
1
Collections::StringNamedCollection
+ getCount() : A_UINT64
+ getNames() : A_UNICODE2STRING[]
+ Add(name :A_UNICODE2STRING, item :A_UNICODE2STRING)
+ Contains(name :A_UNICODE2STRING) : A_BOOLEAN
+ GetByName(name :A_UNICODE2STRING) : A_UNICODE2STRING
+ GetEnumerator() : Enumerator
+ RemoveAll()
+ RemoveByName(name :A_UNICODE2STRING) : A_UNICODE2STRING

Figure 48 Watcher

HIL API distinguishes between 2 watcher types: `ConditionWatcher` and `DurationWatcher`.

DurationWatcher

The `DurationWatcher` fires after a specified duration relative to the start of capture. No matter which type of start occurs, manual or triggered.
This watcher type can only be used in Capture StopTrigger, because it is relative to the start of capturing.

<u>ConditionWatcher</u>

For the definition of a `ConditionWatcher`, a condition and zero, one, or multiple defines are set.
A define maps a name to a model path. The advantage of using defines is that these names can be used in the definition of the condition. This makes a condition more readable for humans and further leads to a decoupling of the test from the model. All defines must be added to a String collection and then added as one object to the `ConditionWatcher`. The condition and all defines are described as strings.

e.g. "velocity" mapped to "Model Root/Subsystem/Vel/Value"

A condition defines when a `ConditionWatcher` fires. The syntax of the condition is defined in [ASAM Expression]. The condition syntax will be validated inside the `setCondition` method. The condition is cyclic examined after start of the Watcher. If the condition is true the `ConditionWatcher` fires.

e.g. velocity > 100

The Watcher itself has no states. It only triggers the Capture with an event (see Capture state diagram in the chapter above).

4.8.2 Usage In Capture

For capturing, the Start and Stop trigger can be set via a Watcher.
The Start trigger can only be a `ConditionWatcher`.
The Stop trigger can be of both types, because the `DurationWatcher` needs a defined start.

The Start and Stop trigger can be set in Capture State `eCONFIGURED`. After starting the capturing it stays in `eACTIVATED` until the Start trigger fires. To stop the capturing also a trigger can be set.

4.9 DATA CAPTURING

4.9.1 GENERAL APPROACH

Capturing is a process of acquiring data in a continous data stream. It guarantees that all process data can be retrieved as they occur related to the real-time service resp. to the real-time task.
The classes in the Capturing and in the Capture Result package are used to define captures, to control the execution of capturing and to obtain the measured data as results. They are located in the Common package as they are used for the Model Access Port as well as for the ECU M Port.

4.9.2 CAPTURING

The main class of the Capturing package is the class `Capture` (Figure 49). It is used to define captures and to control the execution of capturing.

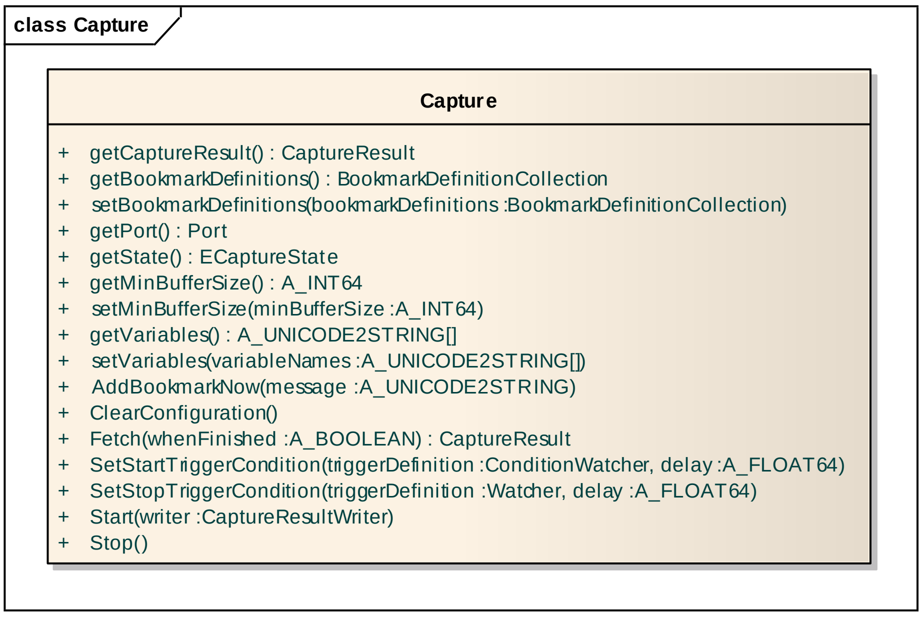

class Capture
Capture
+ getCaptureResult() : CaptureResult
+ getBookmarkDefinitions() : BookmarkDefinitionCollection
+ setBookmarkDefinitions(bookmarkDefinitions :BookmarkDefinitionCollection)
+ getPort() : Port
+ getState() : ECaptureState
+ getMinBufferSize() : A_INT64
+ setMinBufferSize(minBufferSize :A_INT64)
+ getVariables() : A_UNICODE2STRING[]
+ setVariables(variableNames :A_UNICODE2STRING[])
+ AddBookmarkNow(message :A_UNICODE2STRING)
+ ClearConfiguration()
+ Fetch(whenFinished :A_BOOLEAN) : CaptureResult
+ SetStartTriggerCondition(triggerDefinition :ConditionWatcher, delay :A_FLOAT64)
+ SetStopTriggerCondition(triggerDefinition :Watcher, delay :A_FLOAT64)
+ Start(writer :CaptureResultWriter)
+ Stop()

Figure 49 The class Capture

4.9.2.1 CAPTURE

An instance of class `Capture` represents a capture definition. It does not have any constructors. A capture is defined by the port for which a capture shall be defined (`MAPort` or `ECUMPort`). The capture is defined by setting

- the variables to be captured,
- bookmark definitions,
- the minimal buffer size,
- the start trigger condition including a delay, and
- the stop trigger condition including a delay.

After configuration of the `Capture` object, the method `Start()` is called to activate the start trigger. After activating the `Capture` object, the capturing starts at the moment the trigger condition becomes true. In case no start trigger is set, the capturing starts immediately after calling method `Start()`. Note that the method `Start()` is an asynchronous (non-blocking) method. It returns immediately after being called – even if the capturing has not been started yet.

The capturing is stopped either when the stop trigger fires or by calling the method `Stop()`. Note that the start trigger is defined by a `ConditionWatcher` object, the stop trigger is defined either by a `ConditionWatcher` or by a `DurationWatcher` object (see Chapter 4.8). This allows stopping a capturing after a specific amount of time or in dependence of a specific boolean condition. In case no stop trigger is set, the capturing runs until the method `Stop()` is called.

If the trigger condition of the start trigger becomes true again during the capturing or after ending the capturing, this does not have any influence to the capturing or its result. The same holds true for the stop trigger: If it becomes true again after the capture ended, this does not influence the capturing or its result.

Furthermore, the class `Capture` provides methods to obtain the captured data and to observe the current state of the capturing, e.g. to check if the start trigger occurred already.

Special cases: Delayed Triggering

When setting a start or a stop trigger for a `Capture` object, it is possible to set a delay. In case the delay is not zero, this leads to the following behavior, as depicted in the following Figure 50:

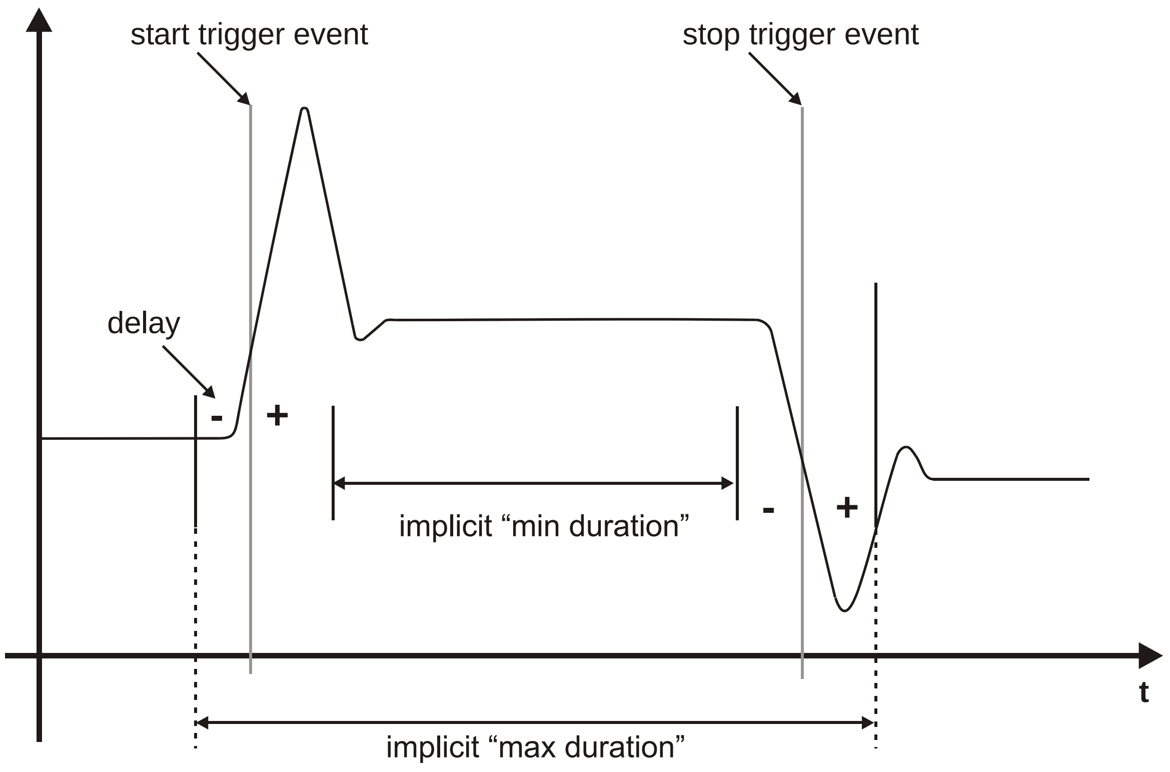


Figure 50 Start and Stop Trigger used

If the delay for the start trigger is positive, the capturing starts the specified amount of time after the start trigger became true – or in case no start trigger has been specified, the capturing starts the specified amount of time after the `Start()` method has been called. If the delay for the start trigger is negative, the capturing starts the specified amount of time before the start trigger occurred, i.e. the capture result will contain even values before the start trigger became true. Obviously, this case is limited: It is not possible to obtain measured values which occurred before the call of the `Start()` method.

If the delay for the stop trigger is positive, the capturing stops the specified amount of time after the stop trigger occurred (or `Stop()` is called resp.). If it is negative, it stops the specified amount of time before, i.e. the capture result will not contain the measured values that occurred during the delay time before the stop trigger occurred (or `Stop()` is called resp.).

4.9.3 CAPTURE RESULT

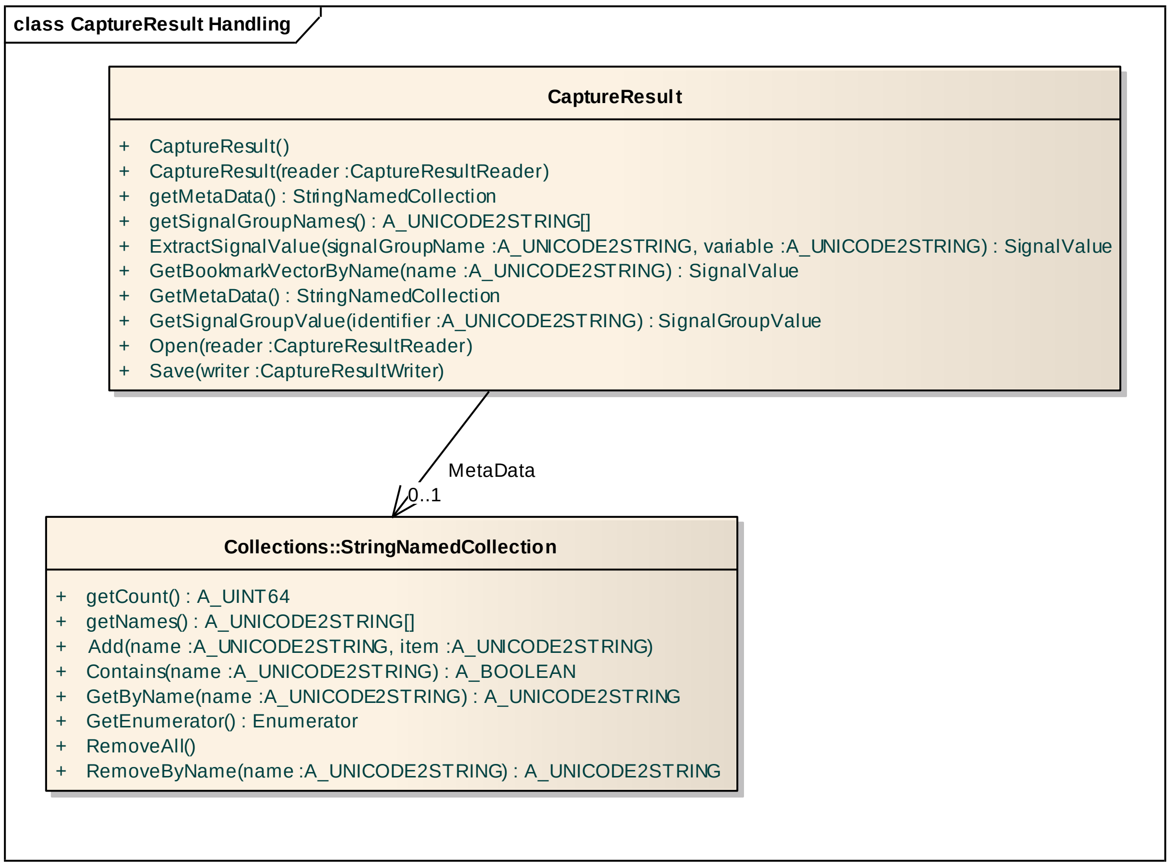

class CaptureResult Handling
CaptureResult
+ CaptureResult()
+ CaptureResult(reader :CaptureResultReader)
+ getMetaData() : StringNamedCollection
+ getSignalGroupNames() : A_UNICODE2STRING[]
+ ExtractSignalValue(signalGroupName :A_UNICODE2STRING, variable :A_UNICODE2STRING) : SignalValue
+ GetBookmarkVectorByName(name :A_UNICODE2STRING) : SignalValue
+ GetMetaData() : StringNamedCollection
+ GetSignalGroupValue(identifier :A_UNICODE2STRING) : SignalGroupValue
+ Open(reader :CaptureResultReader)
+ Save(writer :CaptureResultWriter)
MetaData
0..1
Collections::StringNamedCollection
+ getCount() : A_UINT64
+ getNames() : A_UNICODE2STRING[]
+ Add(name :A_UNICODE2STRING, item :A_UNICODE2STRING)
+ Contains(name :A_UNICODE2STRING) : A_BOOLEAN
+ GetByName(name :A_UNICODE2STRING) : A_UNICODE2STRING
+ GetEnumerator() : Enumerator
+ RemoveAll()
+ RemoveByName(name :A_UNICODE2STRING) : A_UNICODE2STRING

Figure 51 Capture results

<u>CaptureResult</u>

A `CaptureResult` object holds the data captured by a `Capture` object. It provides access to objects of type `ValueContainer::SignalGroupValue` which holds the sampled data.

<u>MetaData</u>

Via the MetaData association, additional information can be added for the `CaptureResult`.

4.9.4 BOOKMARK HANDLING

The Capturing package contains also the class `BookmarkDefinition` to define bookmarks. Bookmarks are special marks for captured data, when specific conditions become true.

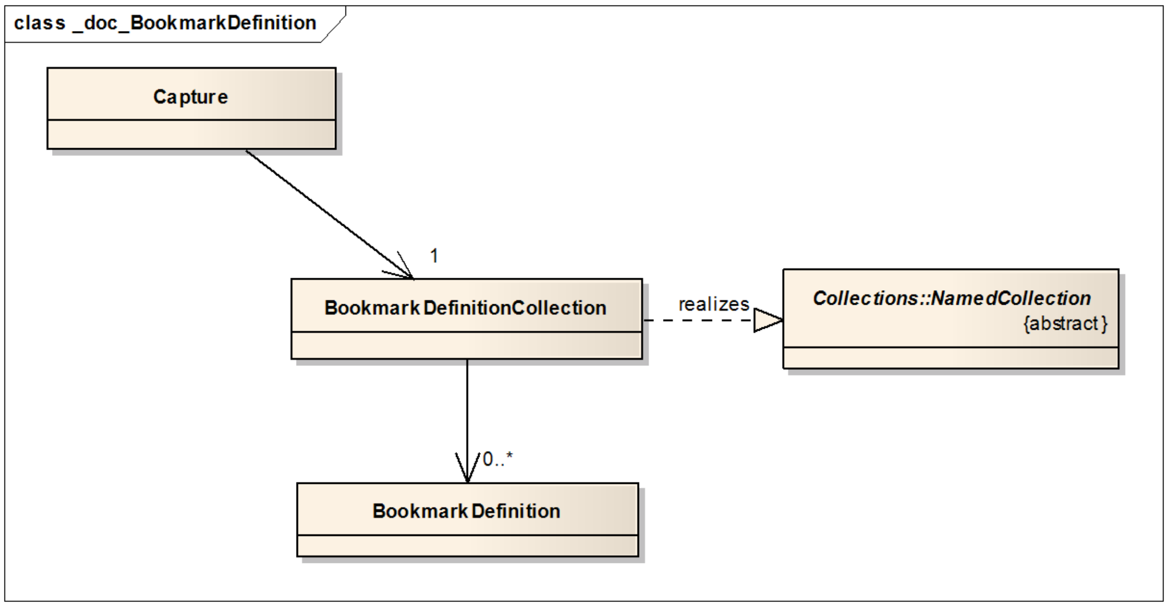

class _doc_BookmarkDefinition
Capture
1
BookmarkDefinitionCollection
realizes
Collections::NamedCollection
{abstract}
0..*
BookmarkDefinition

Figure 52 Bookmark handling

<u>Capture::BookmarkDefinitionCollection</u>

The class `BookmarkDefinitionCollection` is the collection to hold the `BookmarkDefinition` objects, defined for a `Capture` object.

<u>Capture::BookmarkDefinition</u>

`BookmarkDefinition` objects define specific conditions to be observed during capturing. In case such a condition becomes true during capturing, a mark is set in the `CaptureResult` object. Besides the `ConditionWatcher` object that defines the condition to be observed, a bookmark definition consists of a name and a message. Both are used by the capture result object (see below).

CaptureResultHandling::Bookmarks Association

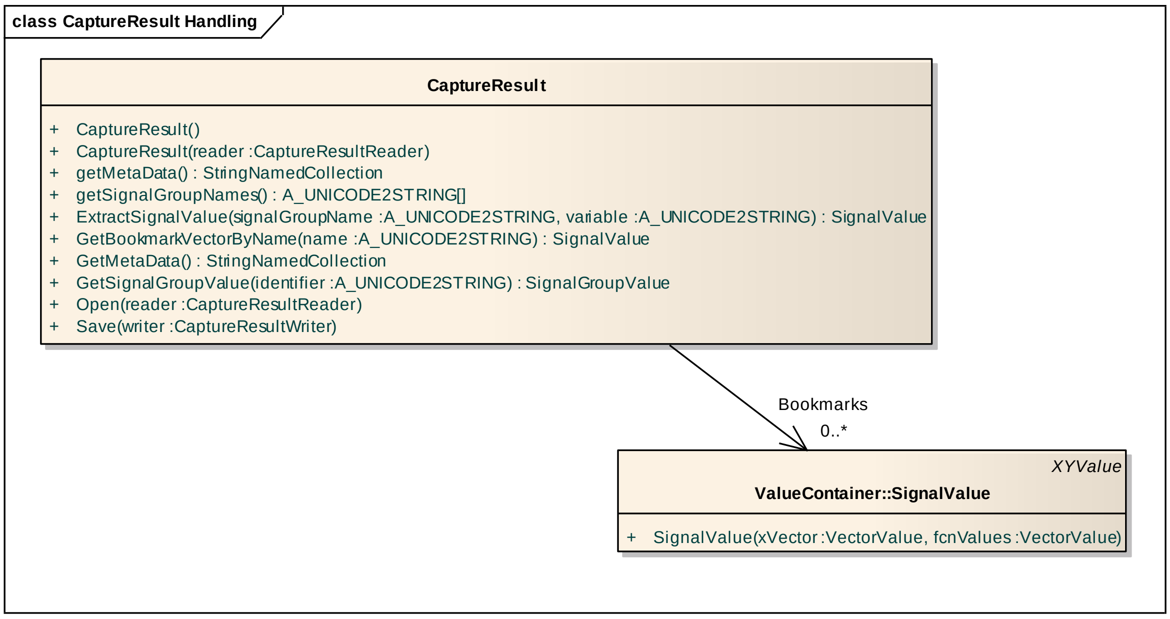

class CaptureResult Handling
CaptureResult
+ CaptureResult()
+ CaptureResult(reader :CaptureResultReader)
+ getMetaData() : StringNamedCollection
+ getSignalGroupNames() : A_UNICODE2STRING[]
+ ExtractSignalValue(signalGroupName :A_UNICODE2STRING, variable :A_UNICODE2STRING) : SignalValue
+ GetBookmarkVectorByName(name :A_UNICODE2STRING) : SignalValue
+ GetMetaData() : StringNamedCollection
+ GetSignalGroupValue(identifier :A_UNICODE2STRING) : SignalGroupValue
+ Open(reader :CaptureResultReader)
+ Save(writer :CaptureResultWriter)
Bookmarks
0..*
XYValue
ValueContainer::SignalValue
+ SignalValue(xVector :VectorValue, fcnValues :VectorValue)

Figure 53 Bookmark Association

In case `BookmarkDefinition` objects have been created and referenced by a `Capture` object, the capturing process observes if the bookmark conditions become true during the capturing. For each bookmark definition whose condition became true at least once during the capturing, a `SignalValue` object is created and referenced via the Bookmarks association from `CaptureResult`. In this `SignalValue`, the time stamps when the bookmark occurred, is stored in the xVector, the message from the bookmark definition is stored as fcnValue. The `SignalValue` object is accessible via the `CaptureResult` object and its Bookmarks are referenced using the name, specified in the bookmark definition.

Furthermore, it is possible to set bookmarks manually by calling the `Capture` object's method `AddBookmarkNow()`. This leads to another `SignalValue` object, referenced by the `CaptureResult` object via the Boomarks association. It can be accessed using the default name "ManualBookmarks". The message can be set for each call of the `AddBookmarkNow()` method.

Figure 54 shows a schematic overview of a possible capture result: It shows the `CaptureResult` object, containing the measured data, and 3 `SignalValue` objects containing bookmarks. The first bookmarks has been defined with the name "name1" and the message "temp > 80". The second bookmarks has been defined with the name "name2" and the message "Attention!". Furthermore some manual bookmarks have been fired: The first one at point of time 3 with the message "msg1", the second one at point of time 5 with the message "msg2". Note that the `SignalValue` objects, containing the bookmarks, do only hold the points of time, when bookmarks occurred. Their xVector is usually different from the xVector of the `CaptureResult` object.

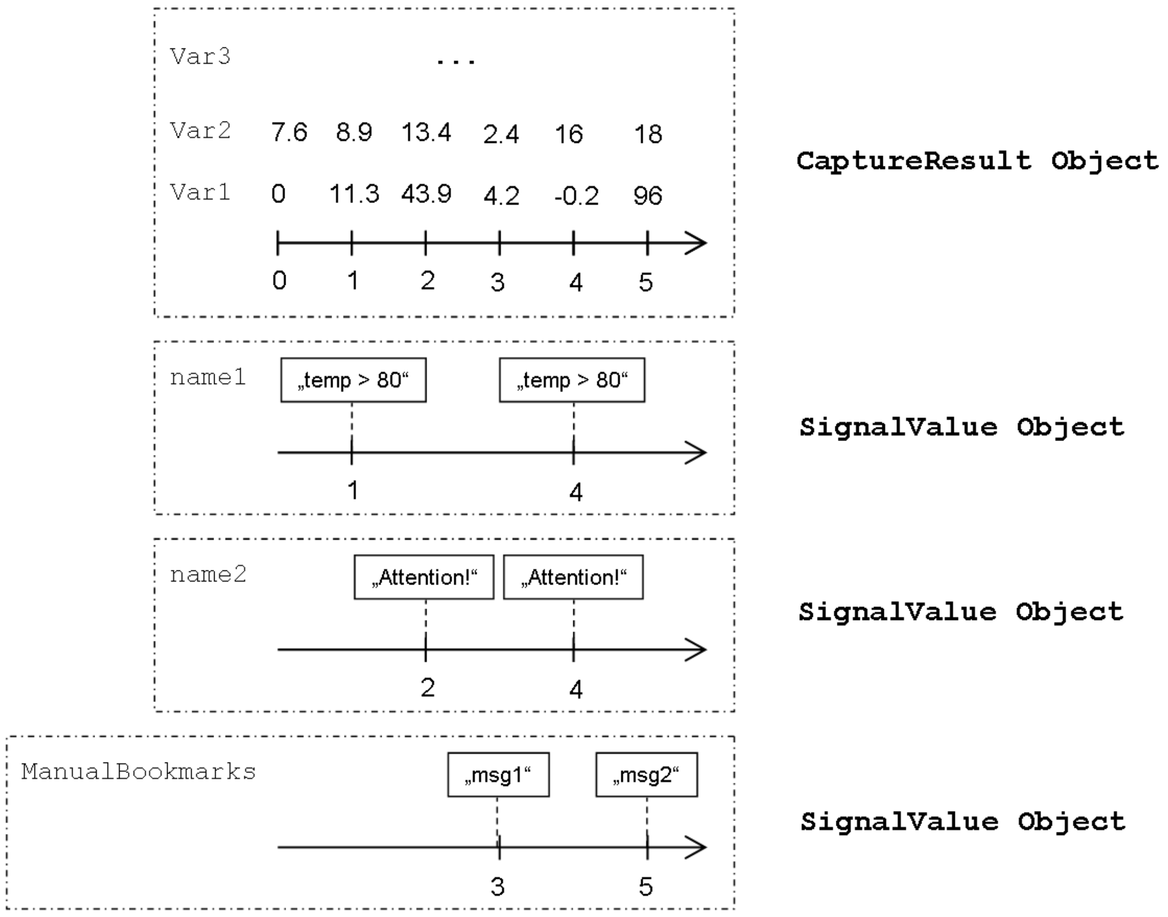


Figure 54 Bookmarks and Capture Results

4.9.5 DOCUMENT HANDLING FOR CAPTURE DATA

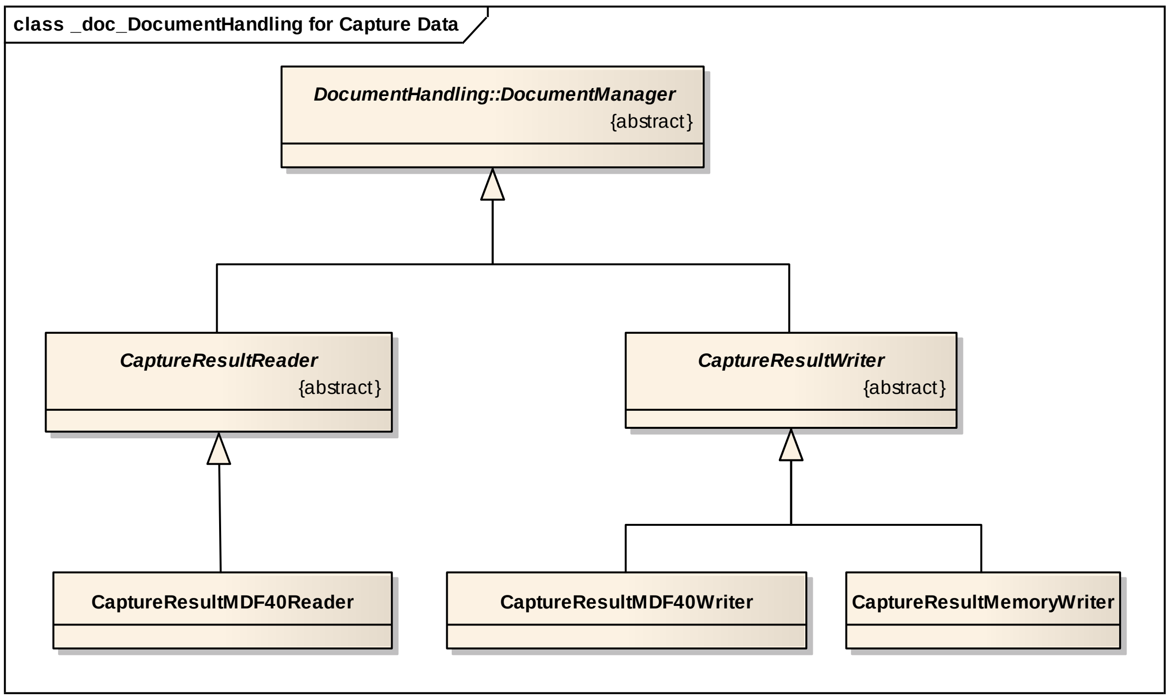

class _doc_DocumentHandling for Capture Data
DocumentHandling::DocumentManager {abstract}
CaptureResultReader {abstract}
CaptureResultWriter {abstract}
CaptureResultMDF40Reader
CaptureResultMDF40Writer
CaptureResultMemoryWriter

Figure 55 Document Handling

<u>CaptureResultReader & CaptureResultWriter</u>

Abstract super classes for the concrete reader and writer classes. These classes provide a `Load` or a `Save` method resp. to load and to save `CaptureResult` objects.

<u>CaptureResultMDF40Reader</u>

This class handles the loading of MDF 4.0.0 files [ASAM MDF]. The loaded data structure is stored in a `CaptureResult` object.

<u>CaptureResultMDF40Writer</u>

This class handles the saving of `CaptureResult` objects compliant to the MDF 4.0.0 format.

<u>CaptureResultMemoryWriter</u>

In case a `CaptureResult` is not stored in the file system during the capturing process, an object of the `CaptureResultMemoryWriter` class is used as writer instance. Instead of streaming the capture to disk, the `CaptureResult` is held in the RAM.

4.9.6 STATE DIAGRAM OF CAPTURING

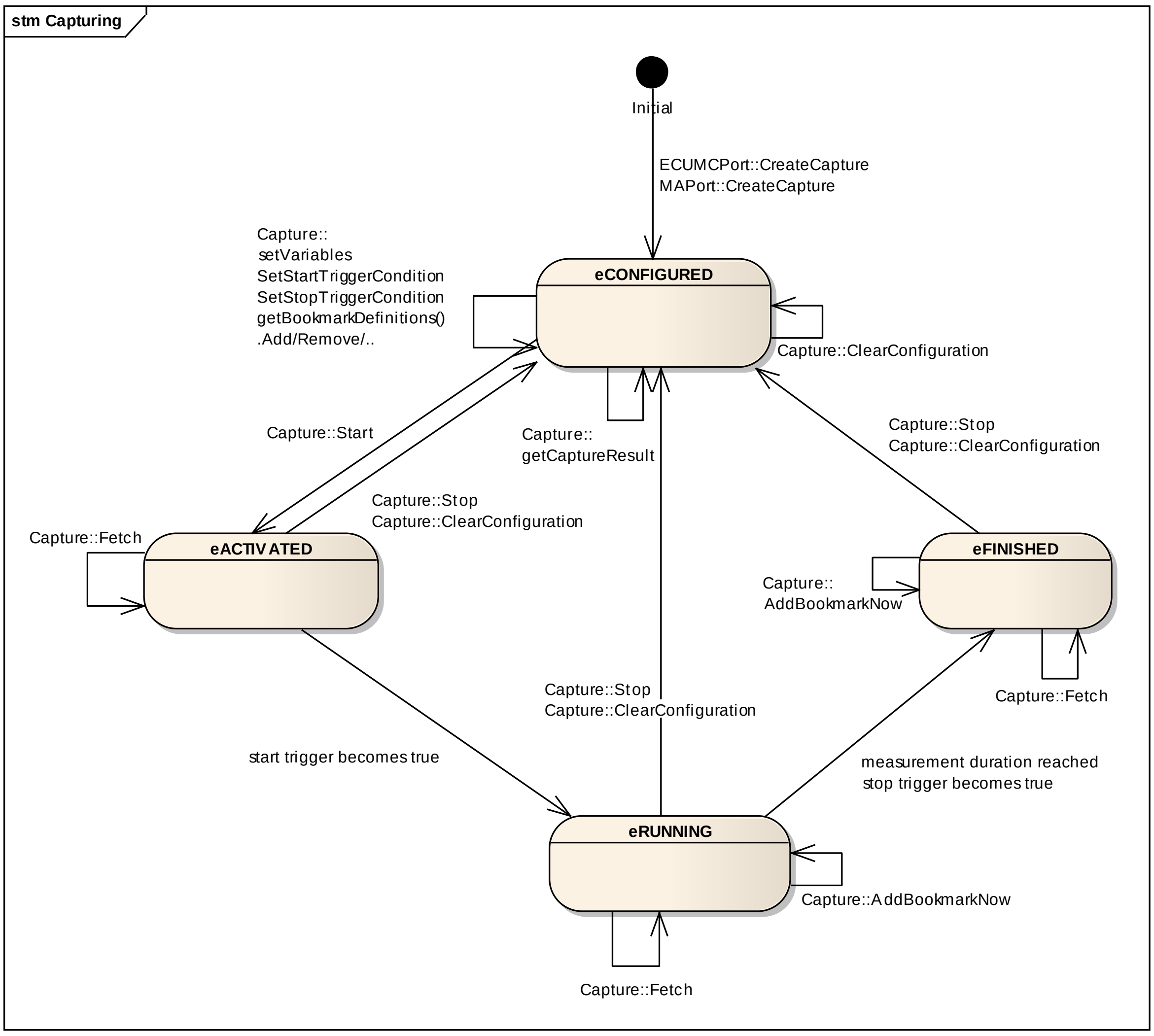


Figure 56 Capturing state diagram

<u>eCONFIGURED</u>

After creation, a `Capture` object is in state `eCONFIGURED`. In this state, the capturing is defined / configured. Usually, it is started when configuration has been done.

<u>eACTIVATED</u>

In this state, the capture waits for the start trigger to become true. When this happens, the capturing switches to state `eRUNNING`.

<u>eRUNNING</u>

While residing in this state, data is captured until the stop trigger holds or until `Stop()` is called manually.

<u>eFINISHED</u>

In this state, the capturing is finished. The captured data is still available to be fetched.

4.9.7 USAGE OF CAPTURING

The sample code for this example will be found at

C#	C#\SampleCode\Common\Capturing\CaptureExample.cs
Python	Python\SampleCode\Common\CaptureExample.py
Java	JAVA\SampleCode\Common\Capturing\CaptureExample.java

Capturing with Watcher

Figure 57, Figure 58 and Figure 59 show how to use capturing. First, a `Capture` object has to be configured: Here, the instance of the capture object is created by an instance of the `MAPort`. Then, all variables that shall be captured are added to the capture's list of variables. To define the beginning and the end of the capture, two watcher objects are created. For a simple human understanding of the trigger conditions, defines are created. A define relates a name to the path of a model variable. These names are used in the conditions of the watcher objects. Finally, the watcher objects are set as start and stop triggers for the `Capture` object.

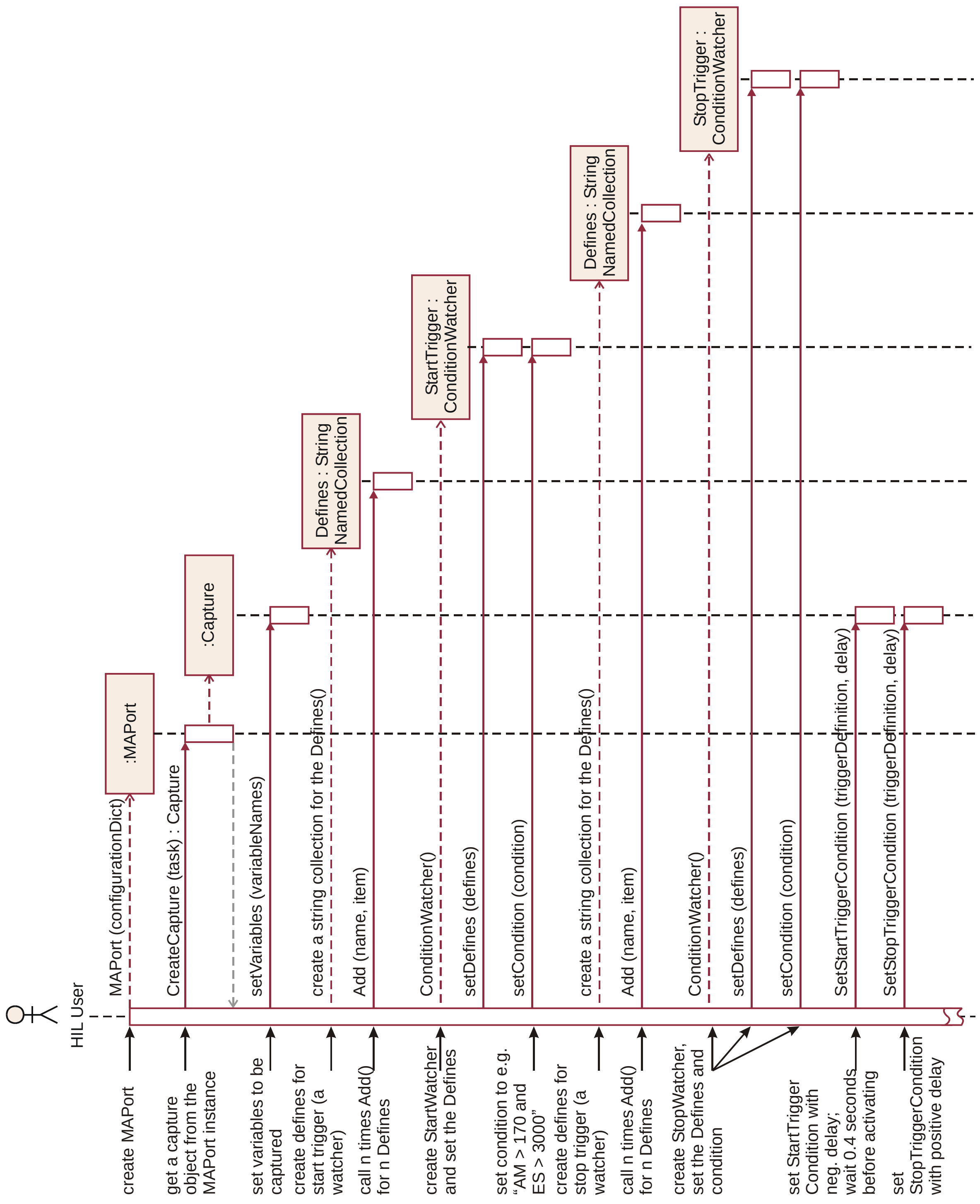

HIL User
:MAPort
:Capture
Defines : String NamedCollection
StartTrigger : ConditionWatcher
Defines : String NamedCollection
StopTrigger : ConditionWatcher
MAPort (configurationDict)
CreateCapture (task) : Capture
setVariables (variableNames)
create a string collection for the Defines()
Add (name, item)
ConditionWatcher()
setDefines (defines)
setCondition (condition)
create a string collection for the Defines()
Add (name, item)
ConditionWatcher()
setDefines (defines)
setCondition (condition)
SetStartTriggerCondition (triggerDefinition, delay)
SetStopTriggerCondition (triggerDefinition, delay)
create MAPort
get a capture object from the MAPort instance
set variables to be captured
create defines for start trigger (a watcher)
call n times Add() for n Defines
create StartWatcher and set the Defines
set condition to e.g. “AM > 170 and ES > 3000”
create defines for stop trigger (a watcher)
call n times Add() for n Defines
create StopWatcher, set the Defines and condition
set StartTrigger Condition with neg. delay; wait 0.4 seconds before activating
set StopTriggerCondition with positive delay

Figure 57 Usage of capturing with Watcher (part 1)

At the beginning of Figure 57, bookmarks are defined for the capturing: First, two other `ConditionWatcher` objects are created. Then, each of these `ConditionWatcher` objects plus a name and a message are used to create a `BookmarkDefinition` object. Finally, the instances of `BookmarkDefinition` added to the Capture's collection of bookmark definitions.

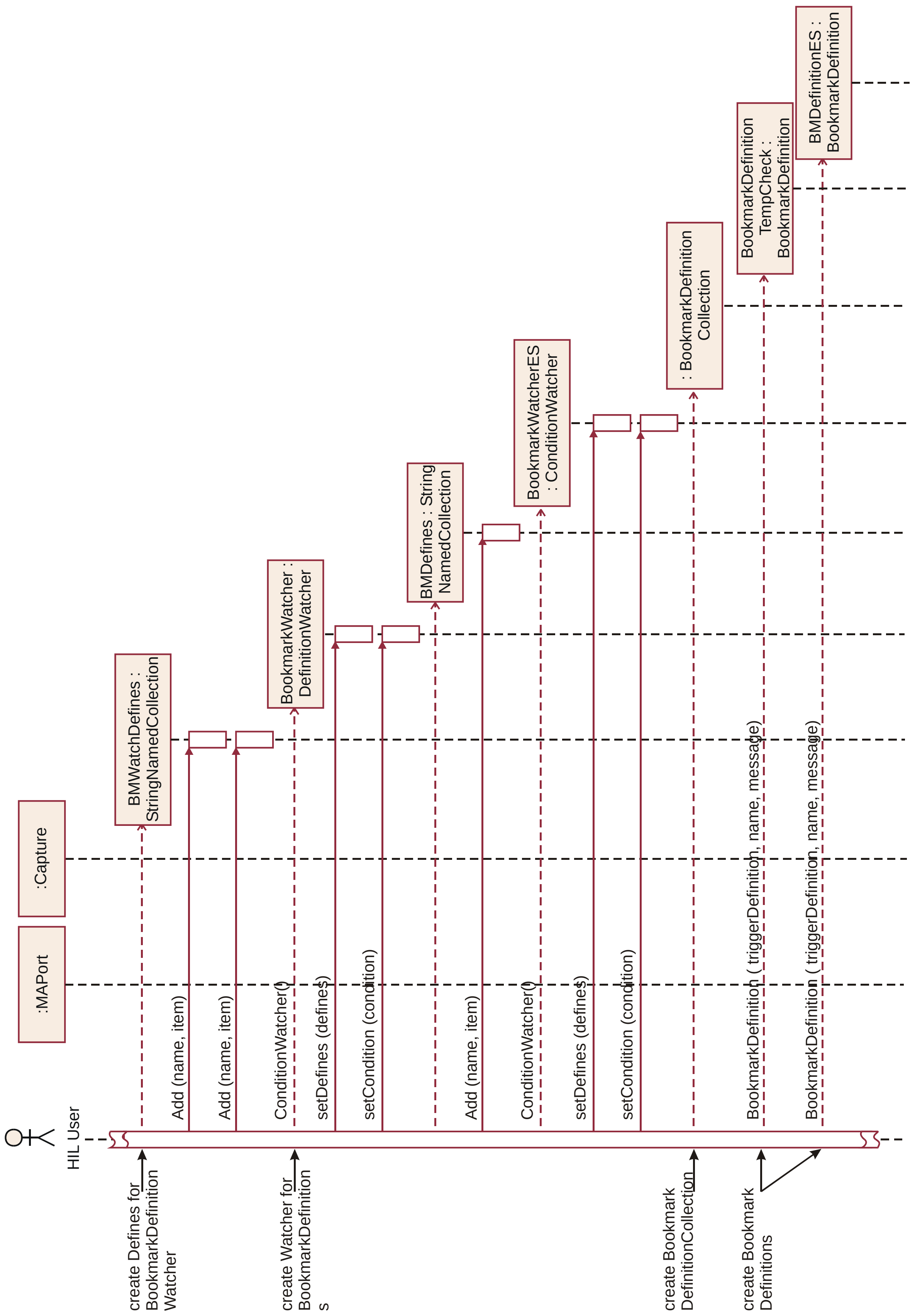

HIL User
:MAPort
:Capture
BMWatchDefines : StringNamedCollection
BookmarkWatcher : DefinitionWatcher
BMDefines : String NamedCollection
BookmarkWatcherES : ConditionWatcher
: BookmarkDefinition Collection
BookmarkDefinition TempCheck : BookmarkDefinition
BMDefinitionES : BookmarkDefinition
Add (name, item)
Add (name, item)
ConditionWatcher()
setDefines (defines)
setCondition (condition)
Add (name, item)
ConditionWatcher()
setDefines (defines)
setCondition (condition)
BookmarkDefinition (triggerDefinition, name, message)
BookmarkDefinition (triggerDefinition, name, message)
create Defines for BookmarkDefinition Watcher
create Watcher for BookmarkDefinitions
create Bookmark DefinitionCollection
create Bookmark Definitions

Figure 58 Usage of capturing with Watcher (part 2)

After setting the minimal buffer size, an MDF 4.0.0 writer for capture results is created. Starting the capture with this writer causes the capture to stream the captured data directly to disk using the MDF 4.0.0 format [ASAM MDF]. During capturing, data is fetched and manual bookmarks are added. The capture is ended manually, i.e. independent of the stop trigger, by calling the Capture's `Stop()` method. Finally, the `Capture` provides the capture result and its configuration is cleared.

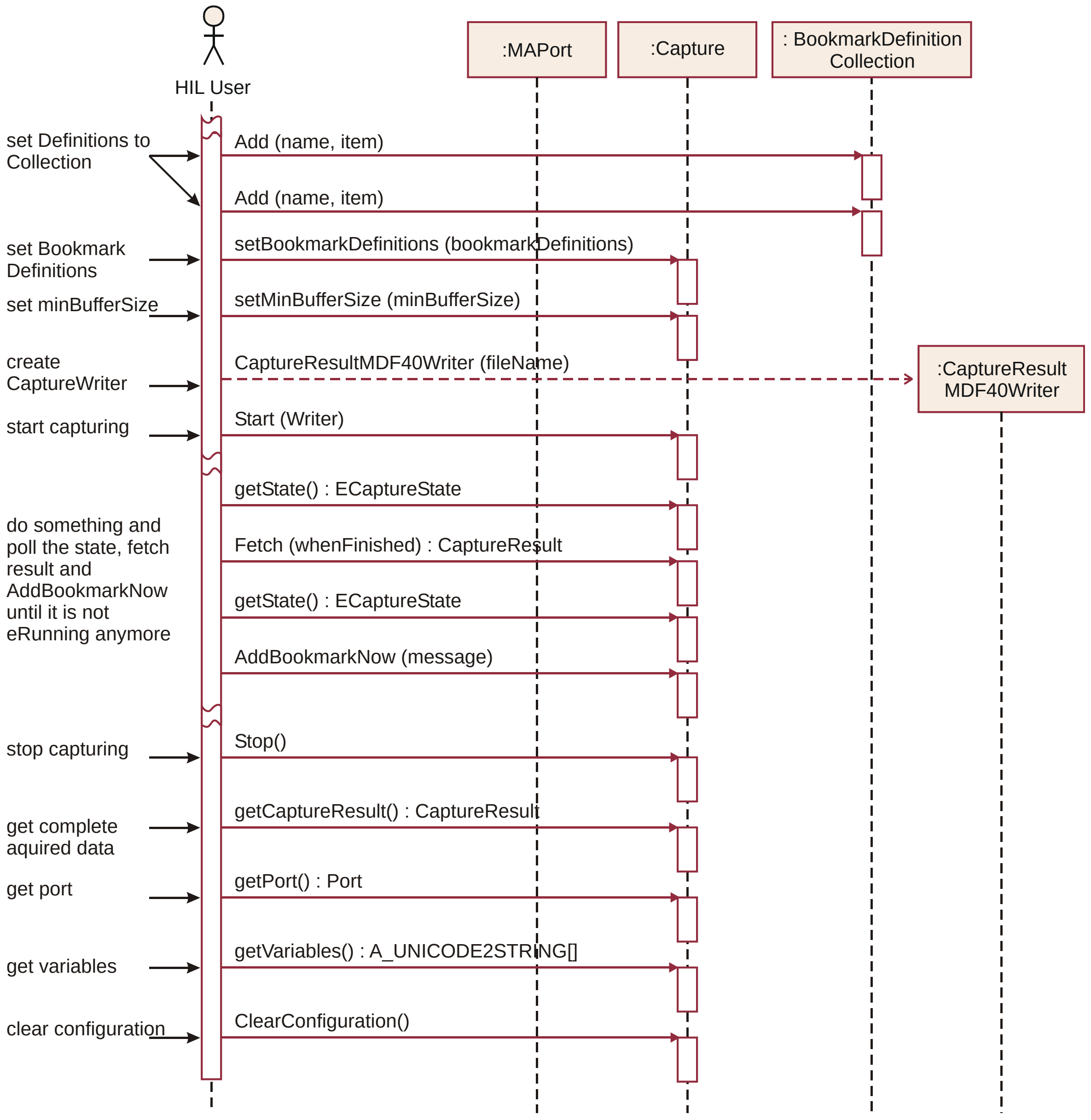

HIL User
:MAPort
:Capture
: BookmarkDefinition Collection
:CaptureResult MDF40Writer
set Definitions to Collection
Add (name, item)
Add (name, item)
set Bookmark Definitions
setBookmarkDefinitions (bookmarkDefinitions)
set minBufferSize
setMinBufferSize (minBufferSize)
create CaptureWriter
CaptureResultMDF40Writer (fileName)
start capturing
Start (Writer)
getState() : ECaptureState
do something and poll the state, fetch result and AddBookmarkNow until it is not eRunning anymore
Fetch (whenFinished) : CaptureResult
getState() : ECaptureState
AddBookmarkNow (message)
stop capturing
Stop()
get complete aquired data
getCaptureResult() : CaptureResult
get port
getPort() : Port
get variables
getVariables() : A_UNICODE2STRING[]
clear configuration
ClearConfiguration()

Figure 59 Usage of capturing with Watcher (part 3)

5 MODEL ACCESS PORT

5.1 USER CONCEPT

5.1.1 GENERAL

The Model Access port is the central point for managing access to the model, simulated on the HIL simulator. This port provides functionality for read- and write-access to the model, to set up capturings and stimuli, and to manage model variables.

When using this port, it is required that all initialization of the HIL simulator, like download and start of the model, has been done previously.

The ModelAccessPort-package is related to its sub-package "Stimulus" and to the packages "Common:Capturing" and "Common:CaptureResult". The two latter ones are not sub-packages of ModelAccess as they are also used by the ECUPort.

5.1.2 MODEL ACCESS PORT

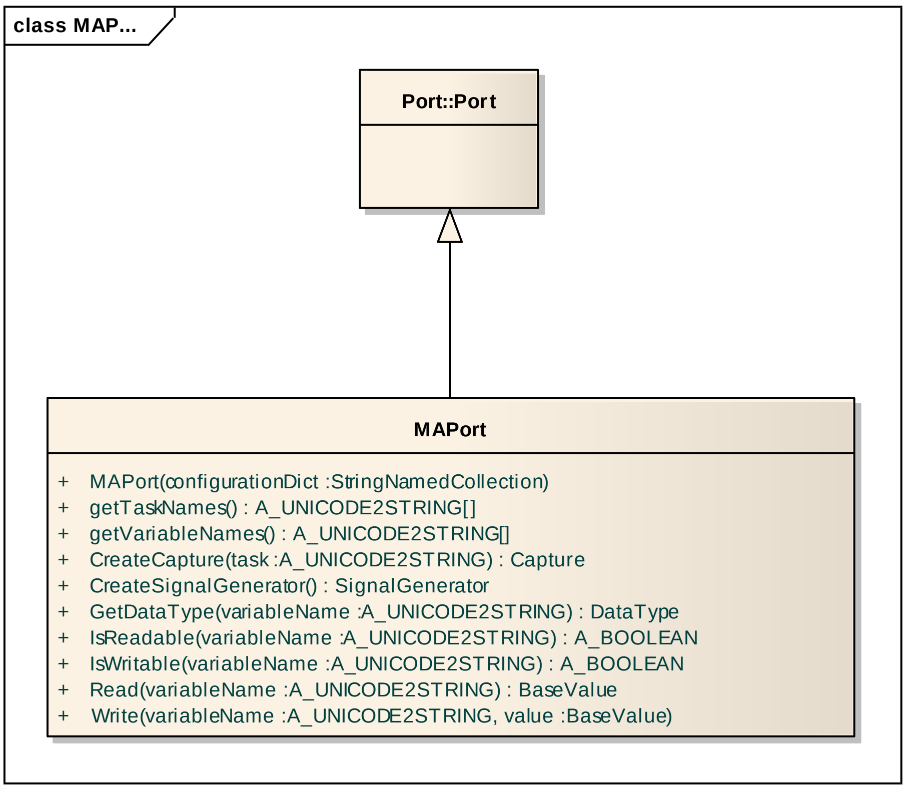

class MAP...
Port::Port
MAPort
+ MAPort(configurationDict :StringNamedCollection)
+ getTaskNames() : A_UNICODE2STRING[]
+ getVariableNames() : A_UNICODE2STRING[]
+ CreateCapture(task :A_UNICODE2STRING) : Capture
+ CreateSignalGenerator() : SignalGenerator
+ GetDataType(variableName :A_UNICODE2STRING) : DataType
+ IsReadable(variableName :A_UNICODE2STRING) : A_BOOLEAN
+ IsWritable(variableName :A_UNICODE2STRING) : A_BOOLEAN
+ Read(variableName :A_UNICODE2STRING) : BaseValue
+ Write(variableName :A_UNICODE2STRING, value :BaseValue)

Figure 60 Model Access Port

<u>Class MAPort</u>

On the one hand, this class provides general functionality like for example functionality to get information about available model variables, their readability and writeability and to read and write model variables. On the other hand, it provides initialization functionality to create Capture and SignalGenerator instances.

5.1.3 STIMULUS

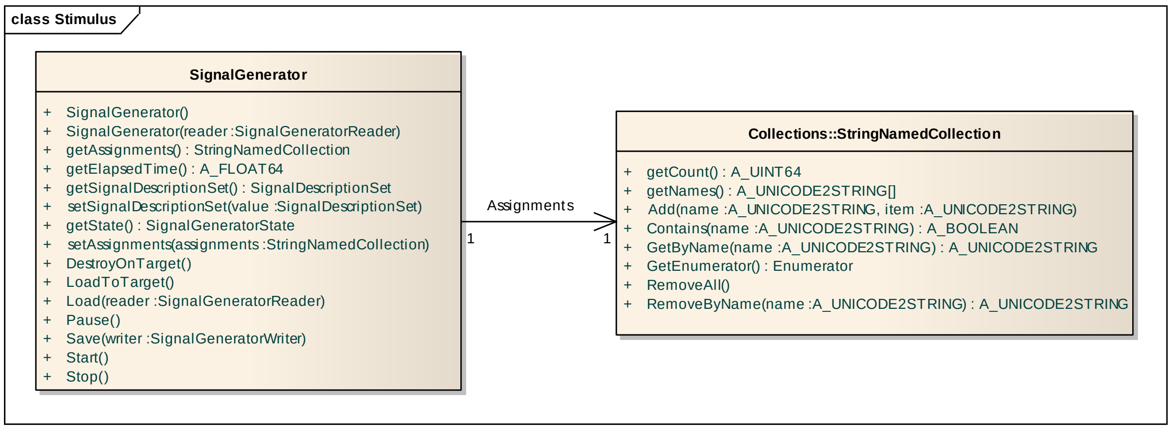

class Stimulus
SignalGenerator
+ SignalGenerator()
+ SignalGenerator(reader :SignalGeneratorReader)
+ getAssignments() : StringNamedCollection
+ getElapsedTime() : A_FLOAT64
+ getSignalDescriptionSet() : SignalDescriptionSet
+ setSignalDescriptionSet(value :SignalDescriptionSet)
+ getState() : SignalGeneratorState
+ setAssignments(assignments :StringNamedCollection)
+ DestroyOnTarget()
+ LoadToTarget()
+ Load(reader :SignalGeneratorReader)
+ Pause()
+ Save(writer :SignalGeneratorWriter)
+ Start()
+ Stop()
Assignments
1
1
Collections::StringNamedCollection
+ getCount() : A_UINT64
+ getNames() : A_UNICODE2STRING[]
+ Add(name :A_UNICODE2STRING, item :A_UNICODE2STRING)
+ Contains(name :A_UNICODE2STRING) : A_BOOLEAN
+ GetByName(name :A_UNICODE2STRING) : A_UNICODE2STRING
+ GetEnumerator() : Enumerator
+ RemoveAll()
+ RemoveByName(name :A_UNICODE2STRING) : A_UNICODE2STRING

Figure 61 SignalGenerator

<u>Class SignalGenerator</u>

A `SignalGenerator` defines stimuli and manages their execution. For the definition of a stimulus, a `SignalDescriptionSet` is referenced by the `SignalGenerator`. The signals from the `SignalDescriptionSet` are assigned with model variables in the "Assignments" collection. For the management of the stimulus, functionality is provided for downloading the stimulus to the HIL simulator, for starting, stopping, and pausing it and for observing its current state.

5.1.4 DOCUMENT HANDLING

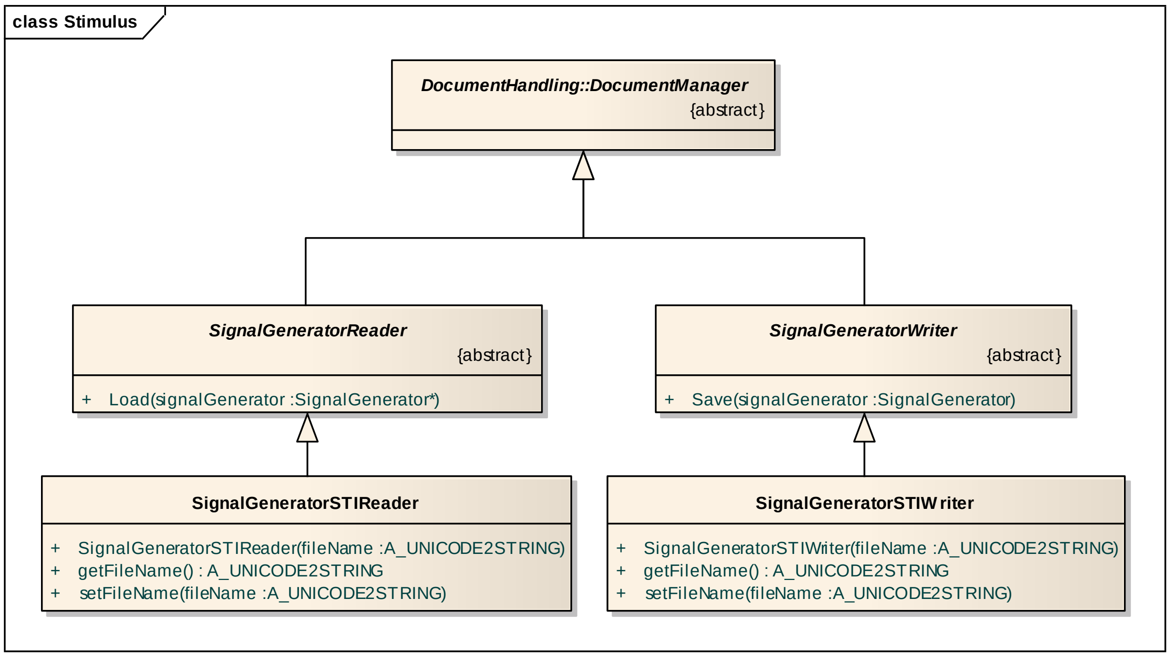

class Stimulus
DocumentHandling::DocumentManager
{abstract}
SignalGeneratorReader
{abstract}
+ Load(signalGenerator :SignalGenerator*)
SignalGeneratorWriter
{abstract}
+ Save(signalGenerator :SignalGenerator)
SignalGeneratorSTIReader
+ SignalGeneratorSTIReader(fileName :A_UNICODE2STRING)
+ getFileName() : A_UNICODE2STRING
+ setFileName(fileName :A_UNICODE2STRING)
SignalGeneratorSTIWriter
+ SignalGeneratorSTIWriter(fileName :A_UNICODE2STRING)
+ getFileName() : A_UNICODE2STRING
+ setFileName(fileName :A_UNICODE2STRING)

Figure 62 Document Handling

SignalGeneratorReader & SignalGeneratorWriter

These classes are abstract super classes for the concrete reader and writer classes. These classes provide a `Load` or a `Save` method resp. to load and to save `SignalGenerator` objects.

SignalGeneratorSTIReader

This class handles the loading of a `SignalGenerator` object stored in a STI files. STI is a file format for `SignalGenerator` objects which is also part of the HIL API standard. The loaded data structure is stored in a `SignalGenerator` object.

SignalGeneratorSTIWriter

This class handles the saving of `SignalGenerator` objects in STI format. STI is a file format for `SignalGenerator` objects which is also part of the HIL API standard.

5.1.5 State Diagram of Signal Generator

The state of the `SignalGenerator` class can be queried at any time by the method `SignalGenerator.getState()`.

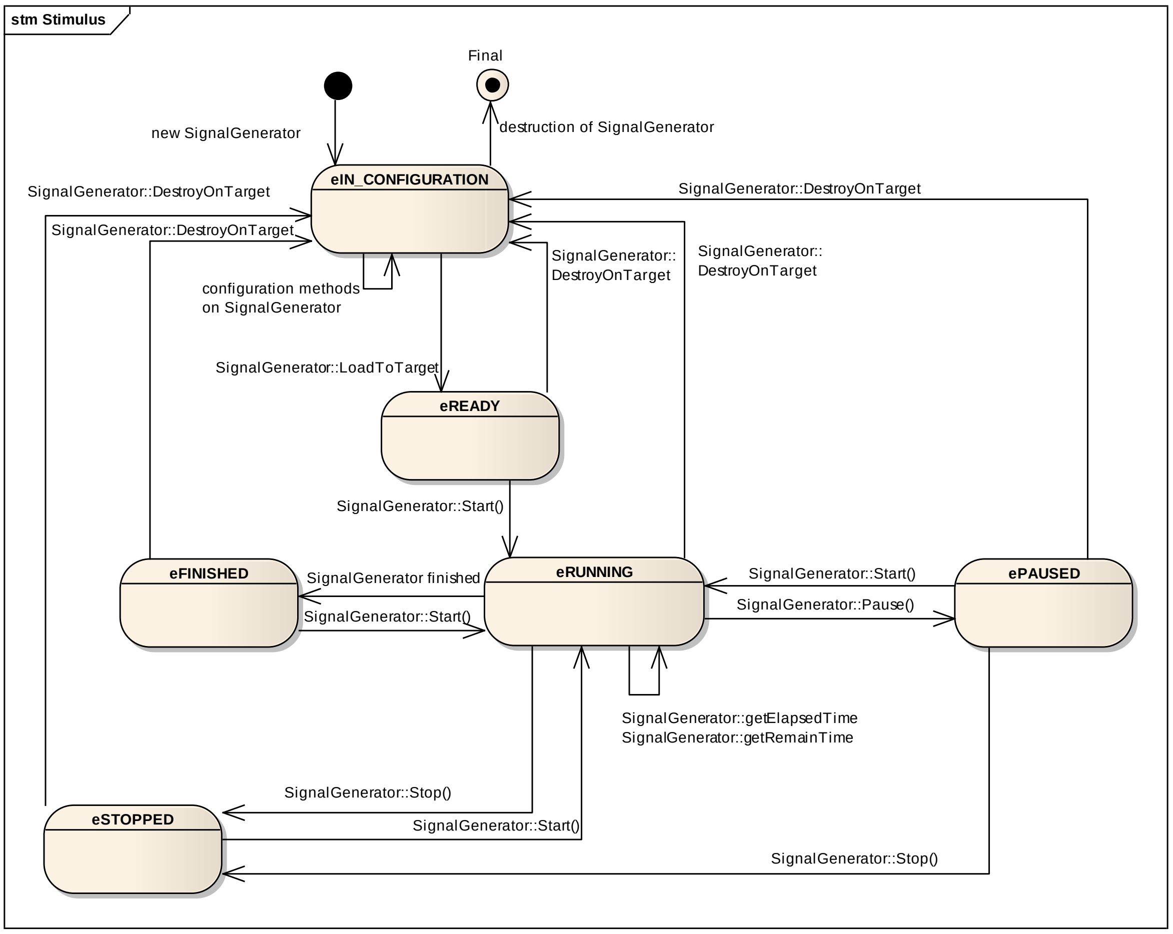

stm Stimulus
Final
new SignalGenerator
destruction of SignalGenerator
eIN_CONFIGURATION
SignalGenerator::DestroyOnTarget
SignalGenerator::DestroyOnTarget
SignalGenerator::DestroyOnTarget
SignalGenerator:: DestroyOnTarget
SignalGenerator:: DestroyOnTarget
configuration methods on SignalGenerator
SignalGenerator::LoadToTarget
eREADY
SignalGenerator::Start()
eFINISHED
SignalGenerator finished
SignalGenerator::Start()
eRUNNING
SignalGenerator::Start()
SignalGenerator::Pause()
ePAUSED
SignalGenerator::getElapsedTime
SignalGenerator::getRemainTime
SignalGenerator::Stop()
eSTOPPED
SignalGenerator::Start()
SignalGenerator::Stop()

Figure 63 Signal generator state diagram

<u>eIN_CONFIGURATION</u>

After creation, a `SignalGenerator` object is in state `eIN_CONFIGURATION`. In this state, the signal generation is defined / configured. Usually, it is loaded to the HIL simulator target, when configuration has been done.

<u>eREADY</u>

After loading a defined signal description set to the HIL Simulator target, the SignalGenerator object is in state `eREADY`. In this state, it waits for being started.

<u>eRUNNING</u>

After starting the signal generation, the SignalGenerator object is in state `eRUNNING`. In this state, the model variables are stimulated by the actual signals as defined.

<u>eFINISHED</u>

If signal generation is finished, this state is entered.

<u>ePAUSED</u>

Signal generation can be paused. In this case, state `ePAUSED` is entered. Leaving this state the signal generation resumes, and does not start at beginning.

<u>eSTOPPED</u>

If the signal generation is stopped, this state is entered.

5.2 USAGE OF THIS PORT

In this chapter, the usage of this port is described by means of some examples. It is shown how to read and write model variables and how to set up a signal generator or a capture. How to use captures is shown in Chapter 4.9.

5.2.1 READING & WRITING MODEL VARIABLES

The sequence diagram in Figure 64 depicts how to handle and how to access model variables: First, an instance of the model access port is created. When such an instance has been created, it is assumed that the HIL simulator has been initialized and a simulation model is running. The instance of the `MAPort` is used to request all available model variables and all tasks (timing raster) existing in the simulation. A `Capture` object is created by the `MAPort` instance with a raster specified by one of the existing tasks.

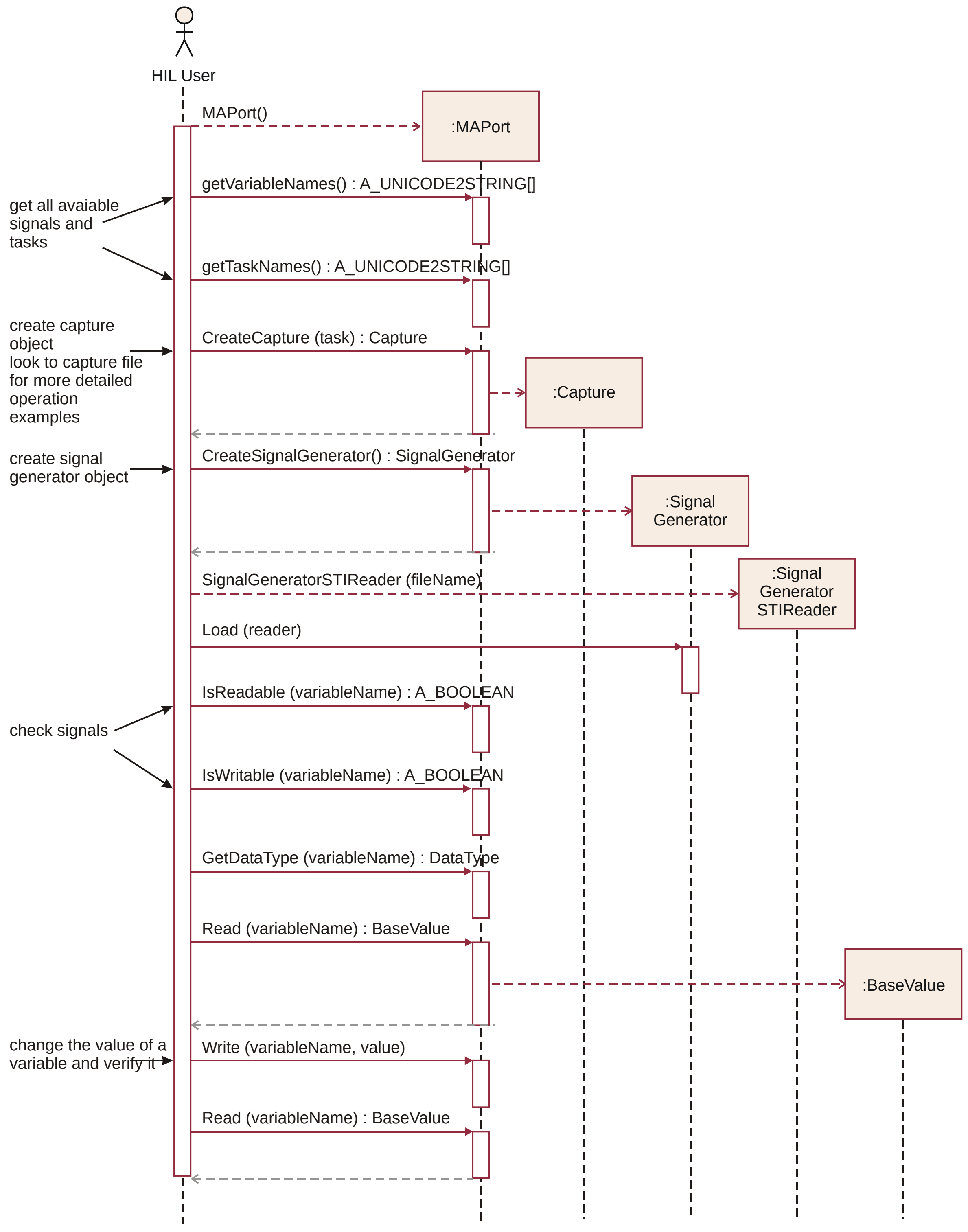

HIL User
MAPort()
:MAPort
getVariableNames() : A_UNICODE2STRING[]
get all avaiable signals and tasks
getTaskNames() : A_UNICODE2STRING[]
create capture object look to capture file for more detailed operation examples
CreateCapture (task) : Capture
:Capture
create signal generator object
CreateSignalGenerator() : SignalGenerator
:Signal Generator
SignalGeneratorSTIReader (fileName)
:Signal Generator STIReader
Load (reader)
check signals
IsReadable (variableName) : A_BOOLEAN
IsWritable (variableName) : A_BOOLEAN
GetDataType (variableName) : DataType
Read (variableName) : BaseValue
:BaseValue
change the value of a variable and verify it
Write (variableName, value)
Read (variableName) : BaseValue

Figure 64 Model AccessPort example

In order to stimulate model variables by signals, a `SignalGenerator` instance is required that is also constructed by the `MAPort` object. Existing signal descriptions can be loaded (see Chapter 5.2.2 for details). The usage of the `Capture` and the `SignalGenerator` instances is described in Chapters 4.9 and 5.2.2.

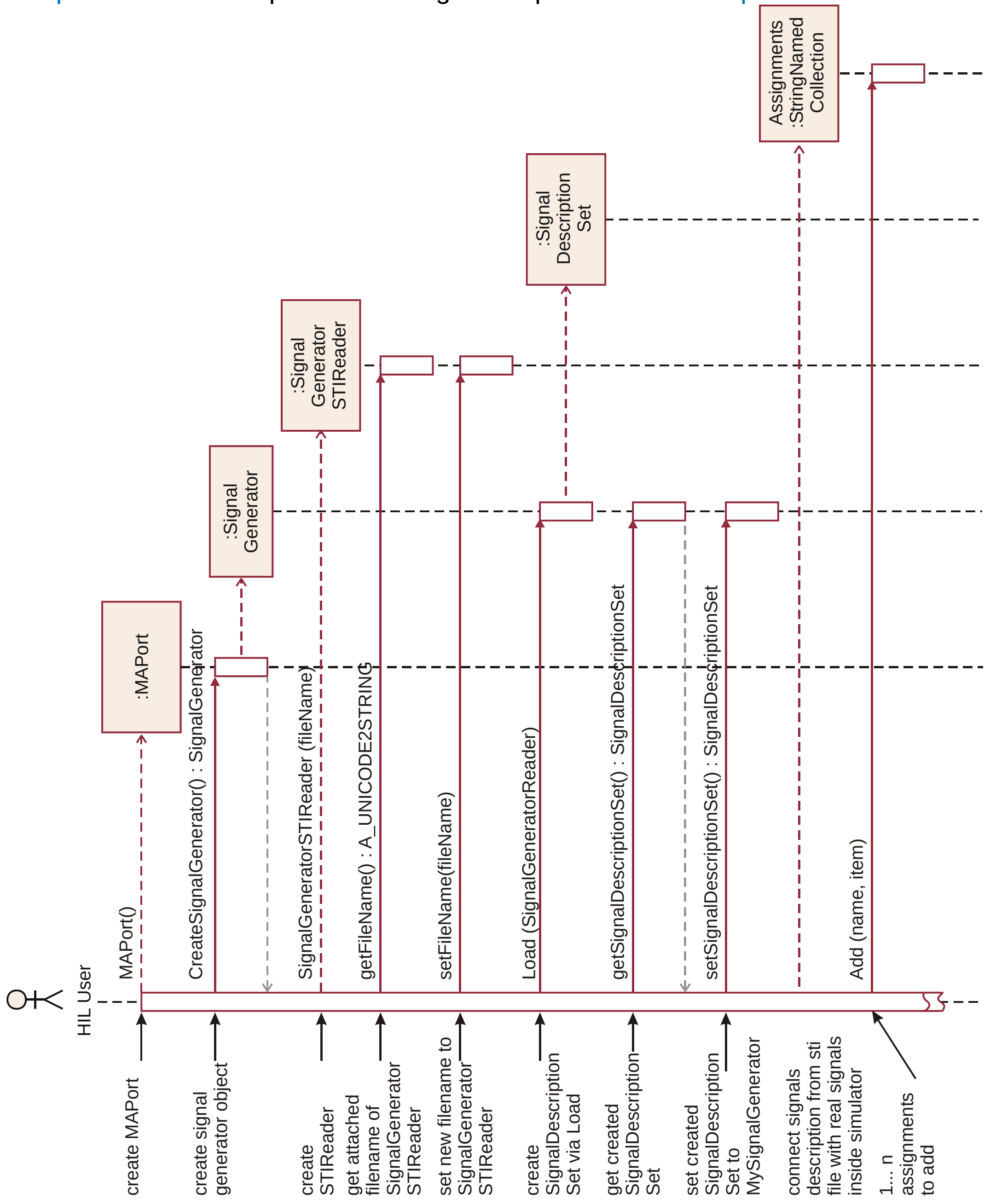
Before accessing a model variable, the `MAPort` instance can check if the variable is readable or writeable (or both) and of which data type it is. Finally the variable is accessed by the `Read()` and the `Write()` method of the `MAPort` object.

The sample code for this example will be found at

C#	C#\ASAM.HILAPI\SampleCode\MAPort\MAPortSample.cs
Python	Python\SampleCode\MAPort\MAPortExample.py
Java	JAVA\SampleCode\MAPort\MAPortExample.java

5.2.2 STIMULATING MODEL VARIABLES

How to stimulate model variables by signals is depicted in Figure 65 and Figure 66. As described in the previous chapter, instances of class `MAPort` and of class `SignalGenerator` need to be created beforehand. Using a `SignalGeneratorSTIReader` object, existing signals are loaded as described in Chapter 4.6. An example for such signals is presented in Chapter 4.7.



:MAPort
:Signal Generator
:Signal Generator STIReader
:Signal Description Set
Assignments :StringNamed Collection
HIL User
MAPort()
CreateSignalGenerator() : SignalGenerator
SignalGeneratorSTIReader (fileName)
getFileName() : A_UNICODE2STRING
setFileName(fileName)
Load (SignalGeneratorReader)
getSignalDescriptionSet() : SignalDescriptionSet
setSignalDescriptionSet() : SignalDescriptionSet
Add (name, item)
create MAPort
create signal generator object
create STIReader
get attached filename of SignalGenerator STIReader
set new filename to SignalGenerator STIReader
create SignalDescription Set via Load
get created SignalDescription Set
set created SignalDescription Set to MySignalGenerator
connect signals description from sti file with real signals inside simulator
1... n assignments to add

Figure 65 SignalGenerator example (part 1)

After executing the `Load()` method, signal descriptions are referenced by the `SignalGenerator` via a `SignalDescriptionSet` object. It may be that the file contains already information that assigns the signals to model variables. In this case the signal generator is configured after loading. Otherwise, these assignments are specified by adding name-item pairs to the Assignments-Collection of the `SignalGenerator` object. The name is one of the signal names, the item is the model path of the variable to be stimulated.

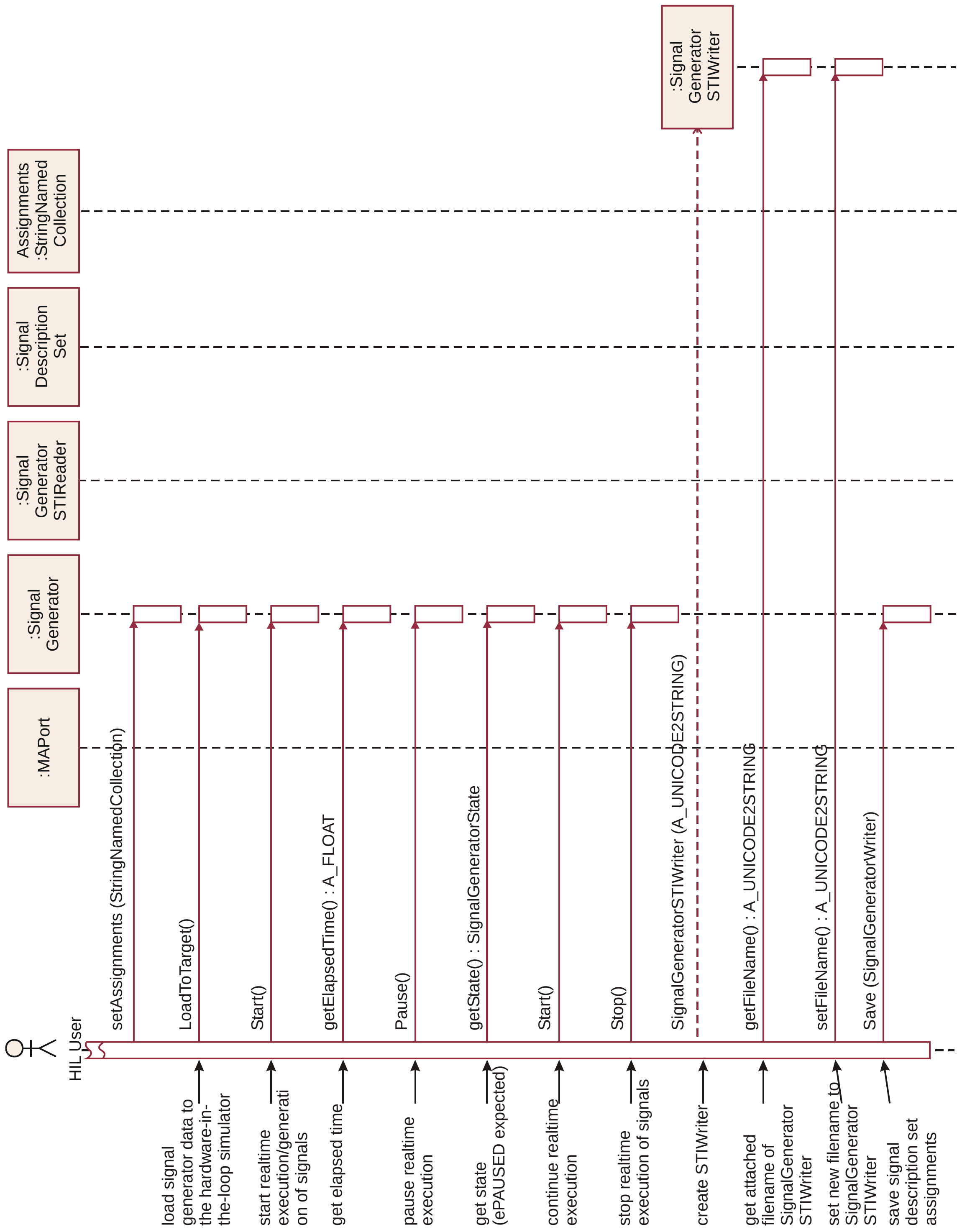

HIL User
:MAPort
:Signal Generator
:Signal Generator STIReader
:Signal Description Set
Assignments :StringNamed Collection
:Signal Generator STIWriter
setAssignments (StringNamedCollection)
LoadToTarget()
Start()
getElapsedTime() : A_FLOAT
Pause()
getState() : SignalGeneratorState
Start()
Stop()
SignalGeneratorSTIWriter (A_UNICODE2STRING)
getFileName() : A_UNICODE2STRING
setFileName() : A_UNICODE2STRING
Save (SignalGeneratorWriter)
load signal generator data to the hardware-in-the-loop simulator
start realtime execution/generation of signals
get elapsed time
pause realtime execution
get state (ePAUSED expected)
continue realtime execution
stop realtime execution of signals
create STIWriter
get attached filename of SignalGenerator STIWriter
set new filename to SignalGenerator STIWriter
save signal description set assignments

Figure 66 SignalGenerator example (part 2)

After setting these assignments, the stimulus is configured. The next step is to load the stimulus down to the target which is usually the HIL simulator. Then, the stimulus can be started, paused, and stopped by calling the corresponding methods. Further, the user can ask for the current state of the signal generator using the `getState()` method. Finally, the signal generator object can be saved including the new assignments, using a `SignalGeneratorSTIWriter` as described in Chapter 4.6.

The sample code for this example will be found at

C#	C#\ASAM.HILAPI\SampleCode\MAPort\SignalGeneratorSample.cs
Python	Python/SampleCode/MAPort/StimulusExample.py
Java	JAVA/SampleCode/MAPort/SignalGeneratorExample.java

6 DIAGNOSTIC PORT

6.1 USER CONCEPT

6.1.1 GENERAL

The Diagnostic Port facilitates the integration of diagnostic tools within the hardware-in-the-loop test automation setup. A diagnostic tool may consist of hardware, software or both that allow for ECU diagnostics. Figure 67 outlines the interaction of the participating components. The test automation system acts as a client of the Diagnostic Port API.

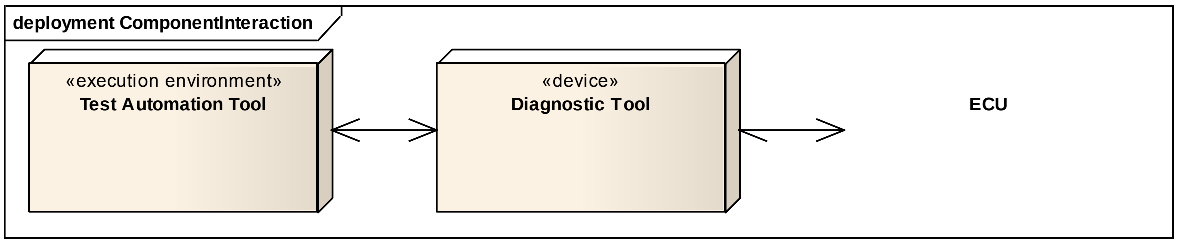


Figure 67 Component Interaction

The Diagnostic Port enables not only the integration of diagnostic tools within the test automation system but also the unification and standardization of the diagnostic tools' functional interface by defining a standardized API. By implementing the standardized Diagnostic Port API, diagnostic tool providers make sure that the diagnostic tools are interchangeable.
The Diagnostic Port API is not a replacement or an extension of existing diagnostic standards or diagnostic protocol standards. Rather the Diagnostic Port API forms a programming interface that reflects the client's requirements related to a diagnostic tool in the context of hardware-in-the-loop test automation. As a general rule, existing standards for diagnostics may be applied underneath the Diagnostic Port or rather by the diagnostic tool providers.
Underneath the Diagnostic Port there is the diagnostic tool and one or more ECUs that may be connected to form a network.

6.1.1.1 COMMUNICATION MODES

The Diagnostic Port communicates directly with the diagnostic tool and indirectly with one or more ECUs. The client has to consider that there is hardware underneath the diagnostic tool that - depending upon the state of development of the hardware - is more or less robust and reliable. That is why the client may anticipate hardware and communication failures underneath the diagnostic tool. The system and communication failures that are detected by the diagnostic tool are delivered to the Diagnostic Port's client by exceptions.
The Diagnostic Port API allows for setting the communication mode between the diagnostic tool and the ECU. The communication mode is either automatic or explicit. In automatic communication mode the client does not need to call the methods for starting and stopping the communication with the ECU explicitly as this is handled by the

diagnostic tool. In explicit communication mode however, the client has to call the methods for starting and stopping the communication explicitly. In automatic communication mode the diagnostic tool has to check whether the communication with the ECU has already been started whenever a client calls an method of the ECU class, the FunctionalGroup class or one of the BaseController classes. If the communication has not been already started, the diagnostic tool has to start and stop the communication on its own responsibility so that the communication status is preserved regardless of the invoked method. The automatic communication mode is the default communication mode. Chapter 6.2.11 shows how to send a HEX service with explicit communication.

6.1.1.2 MACROS

The Diagnostic Port API supports the execution of macros. Macros are a convenient way to group and execute a bunch of diagnostic commands. Chapter 6.1.2.2 shows the classes that are provided for macro commands by the Diagnostic Port API. Additionally, chapter 6.2.3 demonstrates how to execute macro commands with the Diagnostic Port API.

6.1.1.3 FUNCTIONAL GROUPS

A functional group is a group of ECUs that have equal or similar functionality. Mostly, these ECUs can be addressed by a functional address, i.e. a symbolic name for the functional group. For example, all door controlling ECUs of a vehicle may form a functional group. Chapter 6.1.2.3 shows the classes that are provided for functional groups by the Diagnostic Port API.

6.1.1.4 SUPPORTED USE CASES

The Diagnostic Port API provides functions for the following use cases:

Diagnostic Tool

- Configuring the diagnostic tool
- Setting the communication mode

ECU

- Starting and stopping the communication explicitly
- Reading and clearing the fault memory
- Reading identification data
- Reading measurement data by short names
- Reading and writing variant coding data
- Reading and writing data from and to the EEPROM by address
- Reading and writing data from and to the EEPROM by alias names
- Executing diagnostic jobs
- Executing diagnostic job macros
- Sending hex services

Functional Group

- Starting and stopping the communication explicitly
- Reading and clearing the fault memory
- Reading measurement data by short names
- Sending hex services

6.1.2 API

The Diagnostic Port API consists of several classes that represent logical components used in the diagnostic use cases–such as the diagnostic tool, the ECU, the fault memory etc.–in an abstract manner. The following chapters describe the classes of the Diagnostic Port API and the relations between them. Furthermore the context and the usage of these classes is illustrated.

6.1.2.1 ECU

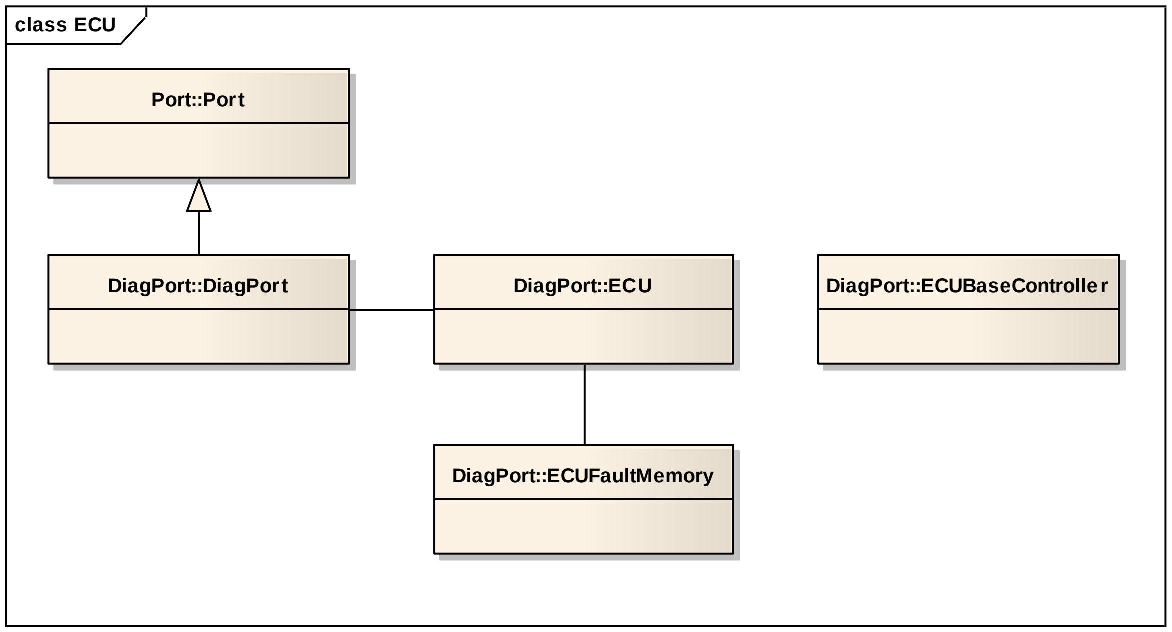


Figure 68 ECU classes

The main classes of the Diagnostic Port API are outlined in the class diagram in Figure 68. The `DiagPort` class is a subclass of the generic Port class. A `DiagPort` object can be used to obtain an `ECU` object that is a representative of a real ECU. Therefore the `ECU` object contains methods for high level use cases such as executing diagnostic jobs and macros, reading variant data or the fault memory for example. The `ECU` object can also be used to receive an `ECUBaseController` object for more sophisticated tasks such as sending HEX services or reading from the EEPROM.

6.1.2.2 MACROS

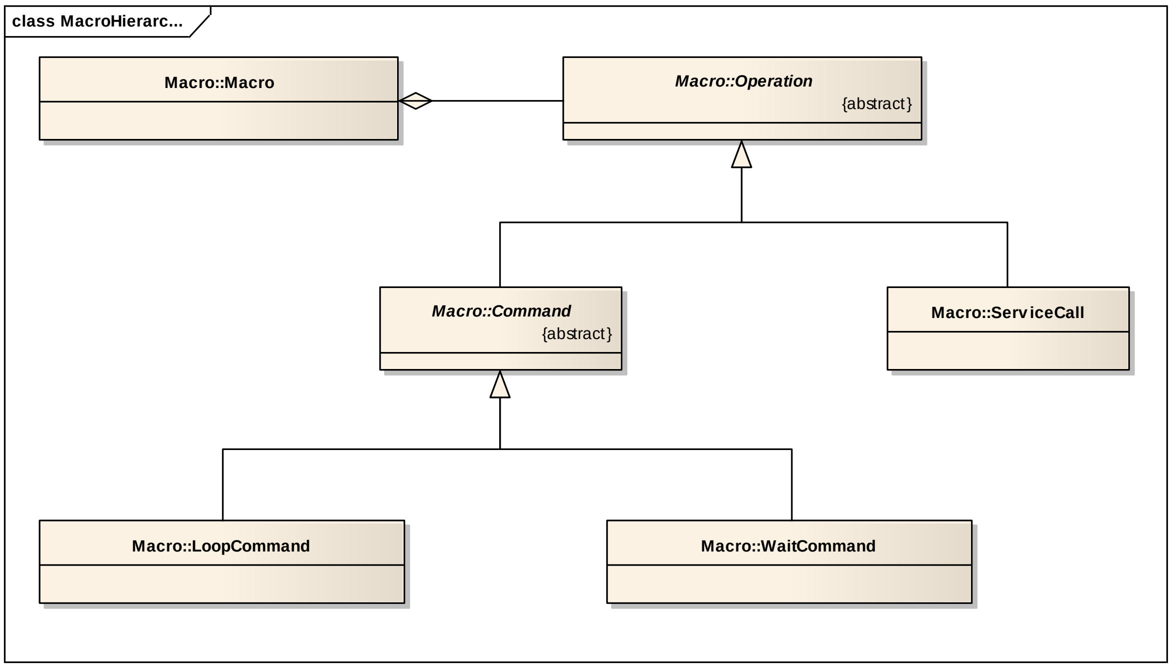


Figure 69 Class hierarchy of the macro classes

The Diagnostic Port API provides classes for the execution of service calls using macro commands. Macros are compositions of diagnostic service calls. Additional commands allow for using loop commands and wait commands within macro definitions. Figure 69 depicts the associations between the macro classes. A `Macro` object contains a list of methods. An method is either a service call, a loop command or a wait command. A loop is a kind of macro since it also aggregates several methods.

6.1.2.3 FUNCTIONAL GROUPS

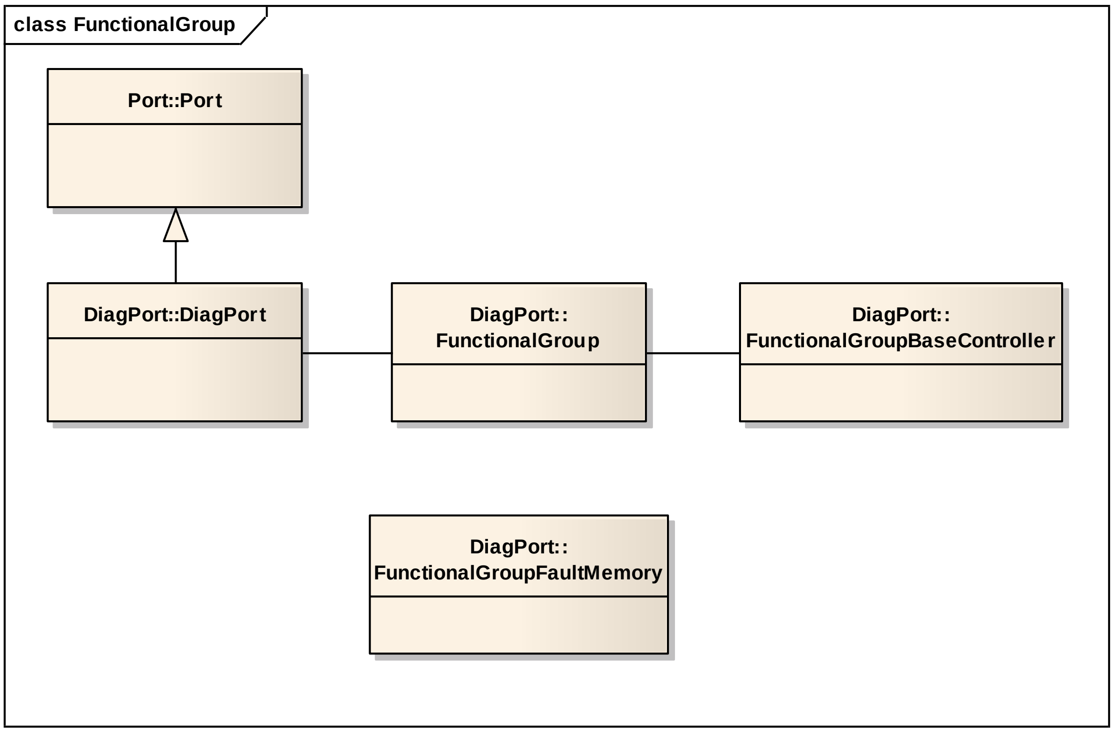

class FunctionalGroup
Port::Port
DiagPort::DiagPort
DiagPort::FunctionalGroup
DiagPort::FunctionalGroupBaseController
DiagPort::FunctionalGroupFaultMemory

Figure 70 Functional group classes

As illustrated in chapter 6.1.1.3 ECUs may be logically grouped to form functional groups. The Diagnostic Port API contains several classes in order to meet the demands for functional groups. See Figure 70 for an overview of the functional group classes. A `DiagPort` object can be used to obtain a `FunctionalGroup` object that is a representative of a functional group of real ECUs. The `FunctionalGroup` object contains methods for high level use cases such as reading measurement data or the fault memory for example. In order to read or clear the fault memory a `FunctionalGroupFaultMemory` object is used. The `FunctionalGroup` object can also be used to receive a `FunctionalGroupBaseController` object for more sophisticated tasks such as sending HEX services.

6.2 USAGE OF THIS PORT

This chapter depicts the usage of the Diagnostic Port API. The example use cases are based on popular tasks in the context of hardware-in-the-loop test automation.

6.2.1 GETTING THE ECU OBJECT

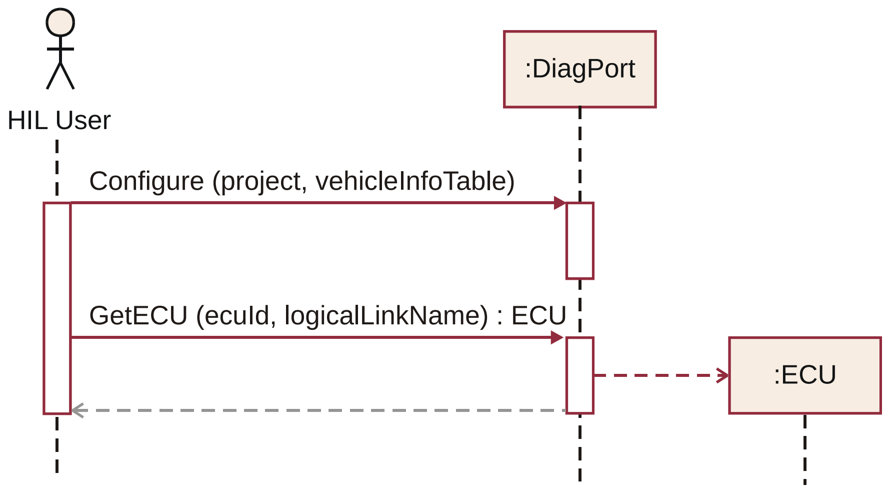

HIL User
:DiagPort
Configure (project, vehicleInfoTable)
GetECU (ecuId, logicalLinkName) : ECU
:ECU

Figure 71 Getting the ECU object

In the following chapters it is assumed that the client has a valid `DiagPort` instance. See the documentation of your HiL-API implementation provider how to obtain this instance. Before getting the ECU object the client needs to configure the `DiagPort` instance. This is done by invoking the `Configure()` method of the `DiagPort` object. The `Configure()` method takes two parameters: one for the project and one for the vehicle information table. The diagnostic tool has to know how to deal with the given parameters in order to access ECUs. Then the ECU object can be obtained by invoking the `GetECU()` method of the `DiagPort` object with parameters for the ECU's ID and the name of the logical link the ECU is connected to. The sequence diagram in Figure 71 depicts the steps needed to obtain the `ECU` object.

6.2.2 READING AND CLEARING THE FAULT MEMORY

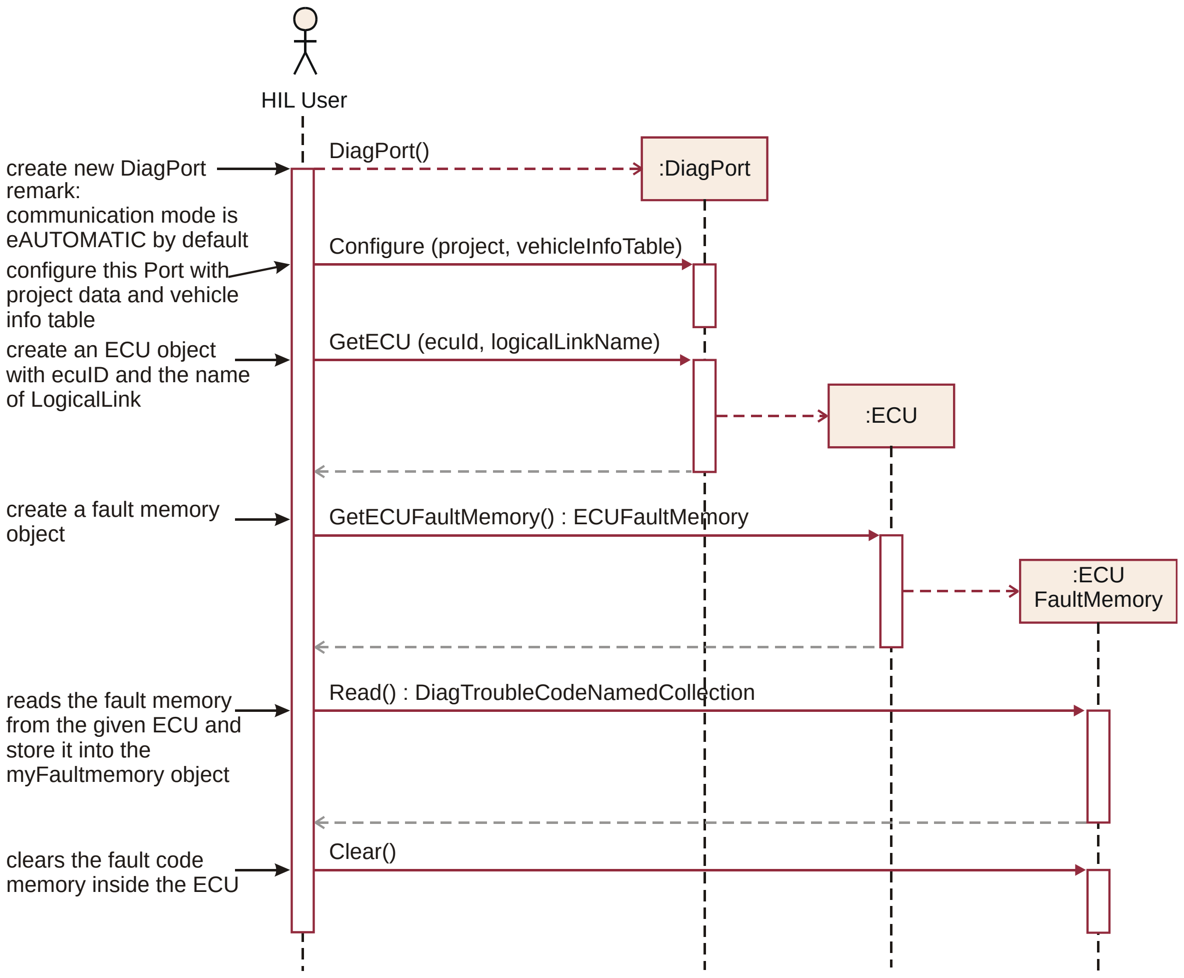

HIL User
:DiagPort
:ECU
:ECU FaultMemory
DiagPort()
Configure (project, vehicleInfoTable)
GetECU (ecuId, logicalLinkName)
GetECUFaultMemory() : ECUFaultMemory
Read() : DiagTroubleCodeNamedCollection
Clear()
create new DiagPort remark: communication mode is eAUTOMATIC by default
configure this Port with project data and vehicle info table
create an ECU object with ecuID and the name of LogicalLink
create a fault memory object
reads the fault memory from the given ECU and store it into the myFaultmemory object
clears the fault code memory inside the ECU

Figure 72 Reading and clearing the fault memory

When the client holds a reference to an `ECU` object (see chapter 6.2.1), reading the fault memory of the ECU is performed in several steps. The client invokes the `GetEcuFaultMemory()` method of the ECU object and gets an `ECUFaultMemory` object as return value. The `ECUFaultMemory` object is used to receive a dictionary of diagnostic trouble codes. In order to receive this dictionary the client of the diagnostic port has to invoke the `ECUFaultMemory` object’s `Read()` method. The `DiagTroubleCodeNamedCollection` object that is returned provides methods for getting diagnostic trouble codes by DTC value or listing all available trouble code entries. See the API documentation of the `DiagTroubleCodeNamedCollection` class for a full overview of available methods. Figure 72 depicts the steps needed to read a diagnostic trouble code by DTC value. The last step clears the fault memory by invoking the `Clear()` method of the `ECUFaultMemory` object. The `DiagTroubleCode` object contains getters for short name, long name, description and value of the DTC entry.

The sample code for this example will be found at

C#	C#\ASAM.HILAPI\SampleCode\DiagPort\DiagPortExample.cs
Python	Python\SampleCode\DiagPort\DiagPortExample.py
Java	JAVA\SampleCode\DiagPort\DiagPortExample.java

6.2.3 Reading the Variant Coding Data

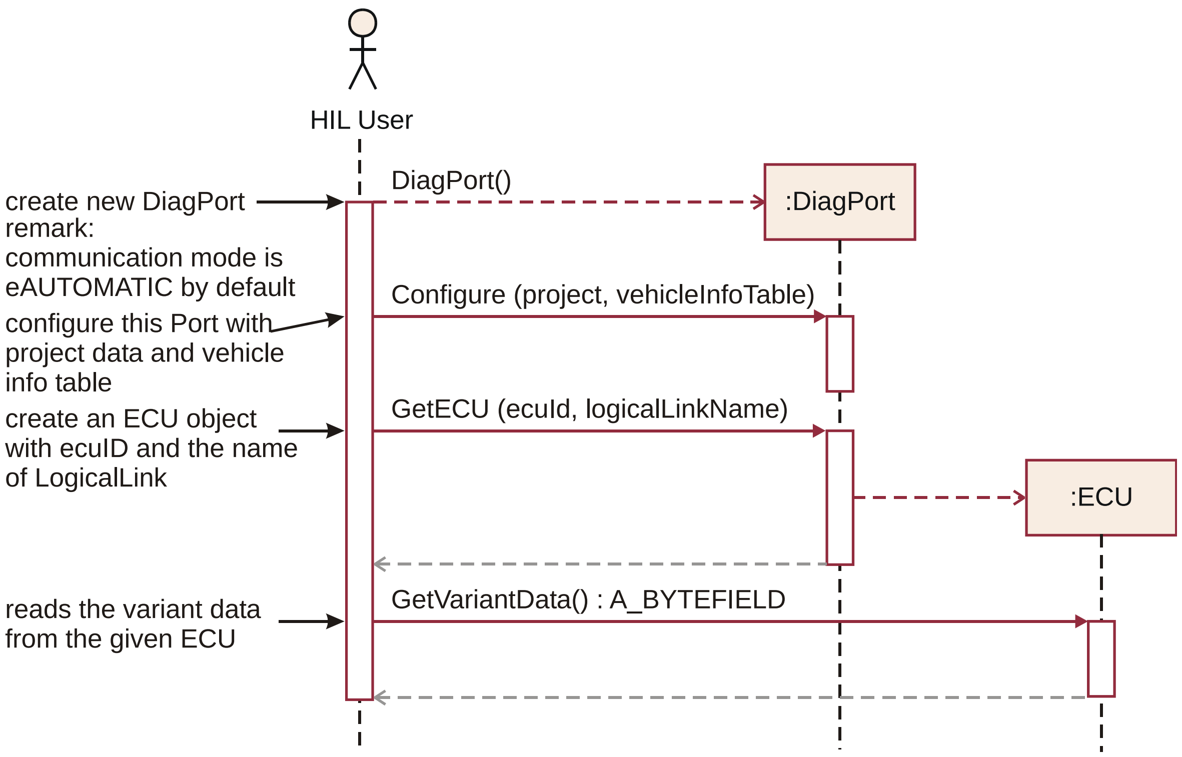

HIL User
DiagPort()
:DiagPort
create new DiagPort
remark:
communication mode is eAUTOMATIC by default
Configure (project, vehicleInfoTable)
configure this Port with project data and vehicle info table
GetECU (ecuId, logicalLinkName)
create an ECU object with ecuID and the name of LogicalLink
:ECU
GetVariantData() : A_BYTEFIELD
reads the variant data from the given ECU

Figure 73 Reading the variant coding data

Variant Coding is used to adapt the ECU's software to operating conditions. Typically, this is performed during the production of the vehicle. For example toggling between left-hand drive and right-hand drive according to the sales country is performed during variant coding. Figure 73 illustrates the steps needed to read the variant coding of an ECU. After having received the `ECU` object from the `DiagPort` object (see chapter 6.2.1) the `GetVariantData()` method is invoked to read the variant coding data.

The sample code for this example will be found at

C#	C#\ASAM.HILAPI\SampleCode\DiagPort\DiagPortExample.cs
Python	Python\SampleCode\DiagPort\DiagPortExample.py
Java	JAVA\SampleCode\DiagPort\DiagPortExample.java

6.2.4 READING IDENTIFICATION DATA

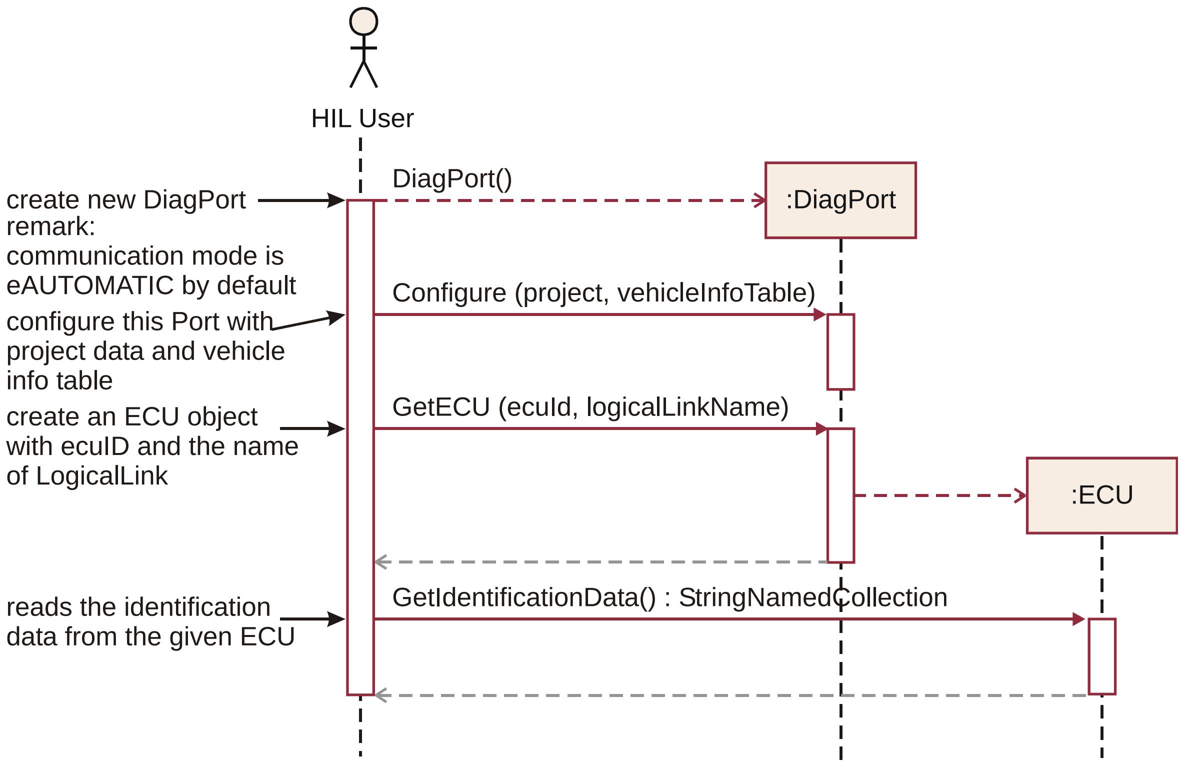

HIL User
DiagPort()
:DiagPort
create new DiagPort
remark:
communication mode is eAUTOMATIC by default
Configure (project, vehicleInfoTable)
configure this Port with project data and vehicle info table
GetECU (ecuId, logicalLinkName)
create an ECU object with ecuID and the name of LogicalLink
:ECU
GetIdentificationData() : StringNamedCollection
reads the identification data from the given ECU

Figure 74 Reading identification data

Identification data is used to identify a given ECU. Identification data may comprise the vehicle manufacturer's part number, hardware part number, hardware version, software version etc. The Diagnostic Port API client can read identification data by means of the `ECU` object's `GetIdentificationData()` method. See chapter 6.2.1 how to receive an `ECU` object. The result of an invocation of the `GetIdentificationData()` method is an `StringNamedCollection` object. This is an ordinary dictionary or map data type that holds key value pairs. These data types provide functions to access the contained data. For example `GetByName()` is an accessor method that takes the name of an (existing) key and returns the corresponding value.

The sample code for this example will be found at

C#	C#\ASAM.HILAPI\SampleCode\DiagPort\DiagPortExample.cs
Python	Python\SampleCode\DiagPort\DiagPortExample.py
Java	JAVA\SampleCode\DiagPort\DiagPortExample.java

6.2.5 READING MEASUREMENT DATA

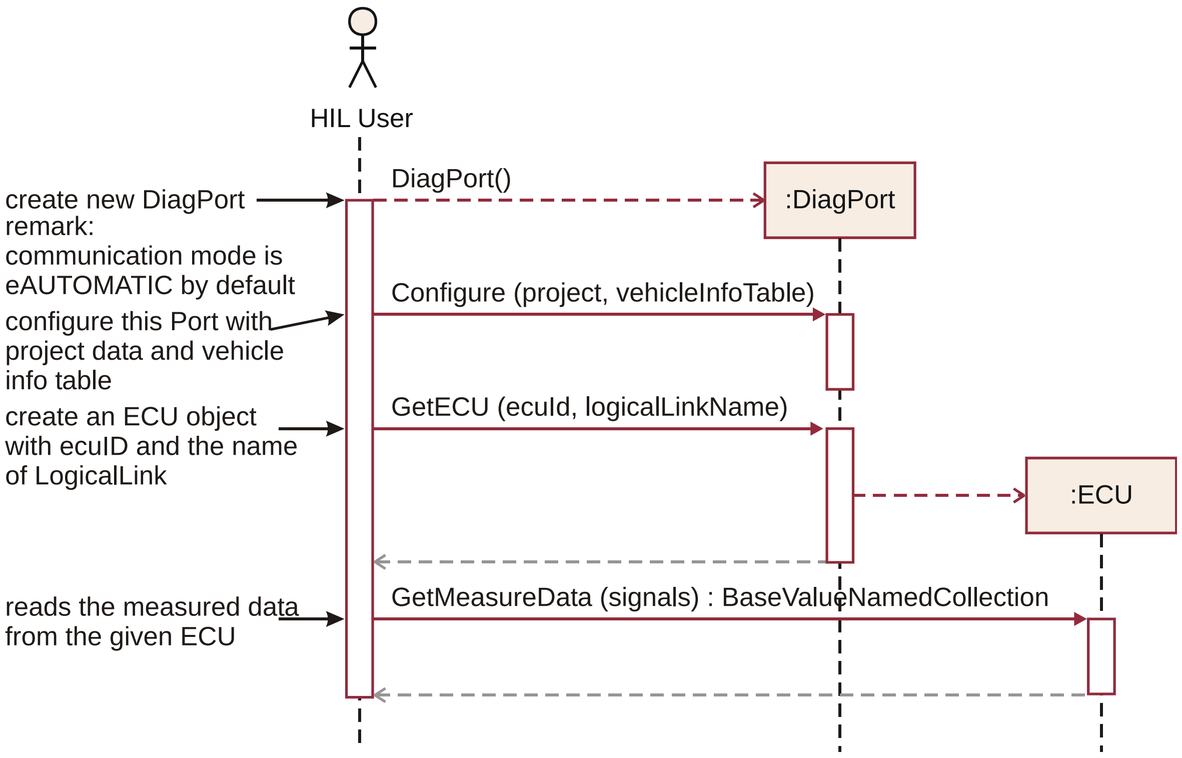

HIL User
DiagPort()
:DiagPort
Configure (project, vehicleInfoTable)
GetECU (ecuId, logicalLinkName)
:ECU
GetMeasureData (signals) : BaseValueNamedCollection
create new DiagPort remark: communication mode is eAUTOMATIC by default
configure this Port with project data and vehicle info table
create an ECU object with ecuID and the name of LogicalLink
reads the measured data from the given ECU

Figure 75 Reading measurement data

The Diagnostic Port API also allows reading measured values. The respective method is located in the `ECU` class. In order to read a bunch of measured values the client has to invoke the `ECU` class's `GetMeasureData()` method. This method takes an array of measure value names and returns a `BaseValueNamedCollection` object that contains the measured values.

The sample code for this example will be found at

C#	C#\ASAM.HILAPI\SampleCode\DiagPort\DiagPortExample.cs
Python	Python\SampleCode\DiagPort\DiagPortExample.py
Java	JAVA\SampleCode\DiagPort\DiagPortExample.java

6.2.6 EXECUTING MACROS

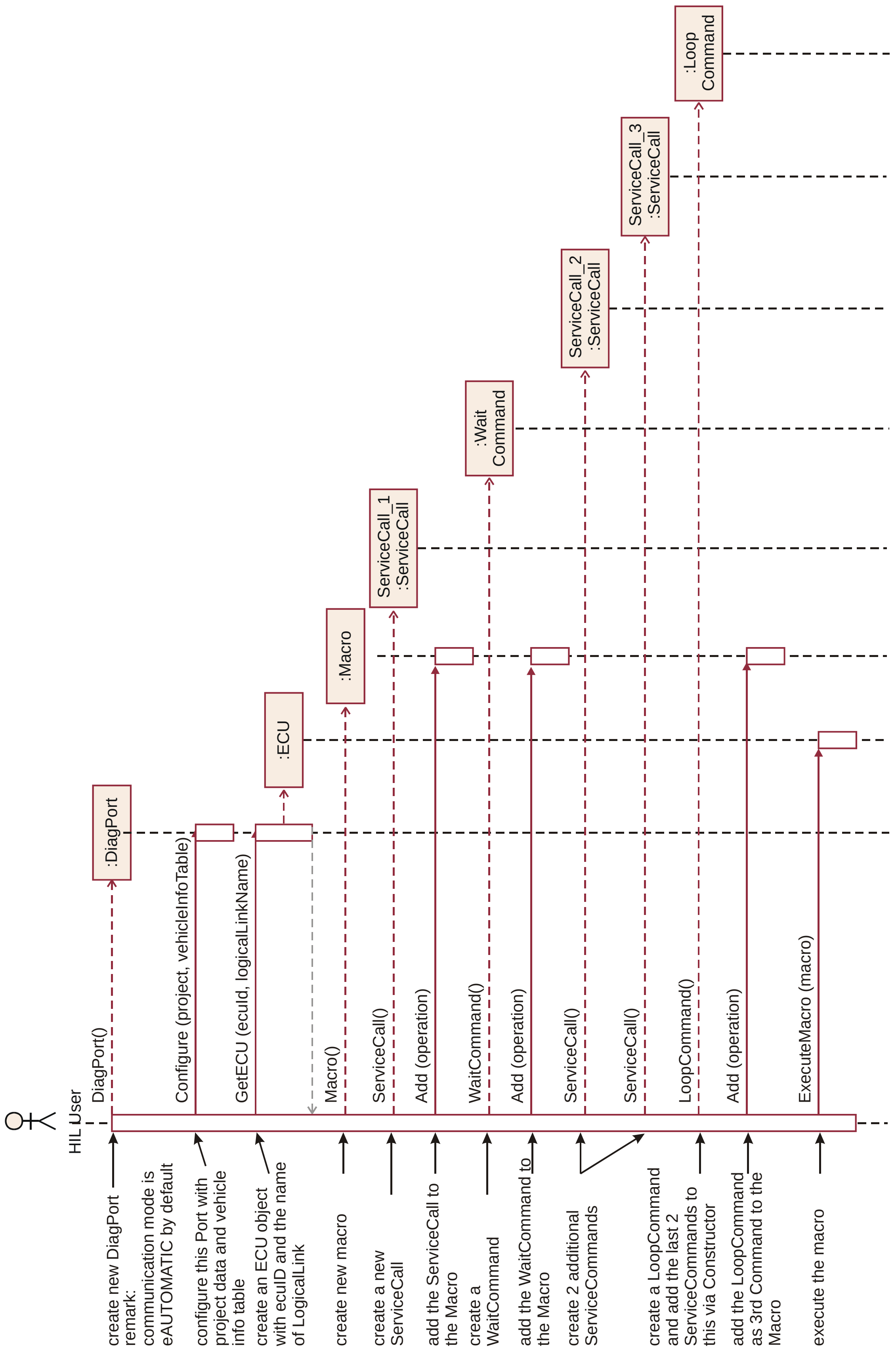

HIL User
:DiagPort
:ECU
:Macro
ServiceCall_1 :ServiceCall
:Wait Command
ServiceCall_2 :ServiceCall
ServiceCall_3 :ServiceCall
:Loop Command
DiagPort()
Configure (project, vehicleInfoTable)
GetECU (ecuId, logicalLinkName)
Macro()
ServiceCall()
Add (operation)
WaitCommand()
Add (operation)
ServiceCall()
ServiceCall()
LoopCommand()
Add (operation)
ExecuteMacro (macro)
create new DiagPort remark: communication mode is eAUTOMATIC by default
configure this Port with project data and vehicle info table
create an ECU object with ecuID and the name of LogicalLink
create new macro
create a new ServiceCall
add the ServiceCall to the Macro
create a WaitCommand
add the WaitCommand to the Macro
create 2 additional ServiceCommands
create a LoopCommand and add the last 2 ServiceCommands to this via Constructor
add the LoopCommand as 3rd Command to the Macro
execute the macro

Figure 76 Executing macros

The Diagnostic Port also allows for the execution of macro commands. Macro commands are compositions of service calls, wait commands and loop commands. Figure 76 outlines the execution of a macro that executes a service calls, a wait command and a loop of service calls. Before being able to call the `ExecuteMacro()` method of the `ECU` class the client has to receive an `ECU` object from the Diagnostic Port. See chapter 6.2.1 how to receive an `ECU` object. Also the client has to build the macro by creating the appropriate objects for service calls, wait commands and loop commands. Therefore the corresponding classes have to provide constructors to create instance objects. Then, the macro is constructed by adding these objects to macro or loop objects with the `Add()` method as depicted in Figure 76. The `ExecuteMacro()` method has to validate the given macro object structure. The `ExecuteMacro()` method will throw an exception if the given macro is not valid.

The sample code for this example will be found at

C#	C#\ASAM.HILAPI\SampleCode\DiagPort\DiagPortExample.cs
Python	Python\SampleCode\DiagPort\DiagPortExample.py
Java	JAVA\SampleCode\DiagPort\DiagPortExample.java

6.2.7 READING AND WRITING VALUES FROM AND TO THE EEPROM BY ALIAS NAMES

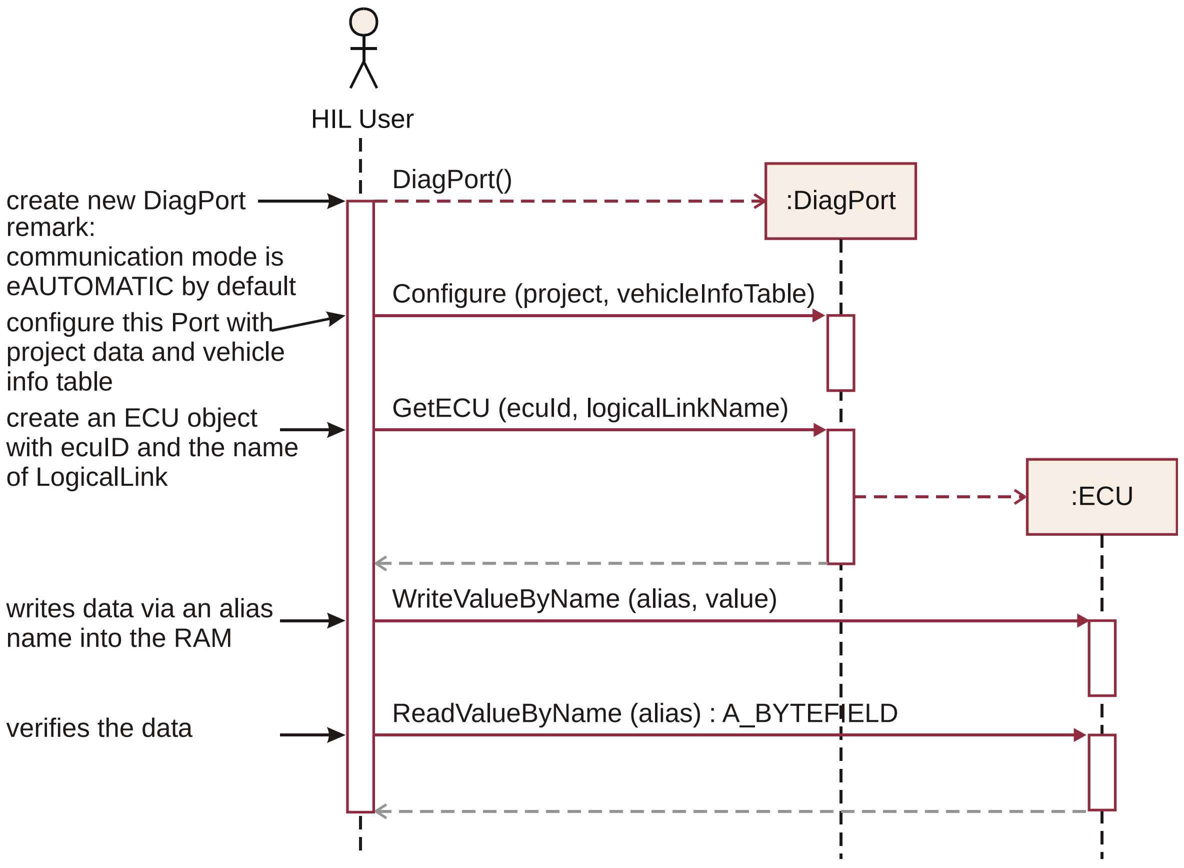

HIL User
:DiagPort
:ECU
DiagPort()
Configure (project, vehicleInfoTable)
GetECU (ecuId, logicalLinkName)
WriteValueByName (alias, value)
ReadValueByName (alias) : A_BYTEFIELD
create new DiagPort
remark:
communication mode is eAUTOMATIC by default
configure this Port with project data and vehicle info table
create an ECU object with ecuID and the name of LogicalLink
writes data via an alias name into the RAM
verifies the data

Figure 77 Reading and writing values from and to the EEPROM by an alias name

The client of the Diagnostic Port API can read and write data from and to the ECU's EEPROM by using symbolic (alias) names for the EEPROM address. The diagnostic tool is responsible for resolving the symbolic name to a specific address value. The corresponding methods are located in the `ECU` class of the Diagnostic Port API. The method for writing is named `WriteValueByName()` and the method for reading is named `ReadValueByName()`. Both methods take an alias name for the address as parameter. The diagnostic tool has to know how to map this alias name to a regular EEPROM address. Figure 77 depicts the steps needed to read and write from and to the ECU's EEPROM by an alias name.

The sample code for this example will be found at

C#	C#\ASAM.HILAPI\SampleCode\DiagPort\DiagPortExample.cs
Python	Python\SampleCode\DiagPort\DiagPortExample.py
Java	JAVA\SampleCode\DiagPort\DiagPortExample.java

6.2.8 READING FROM THE EEPROM

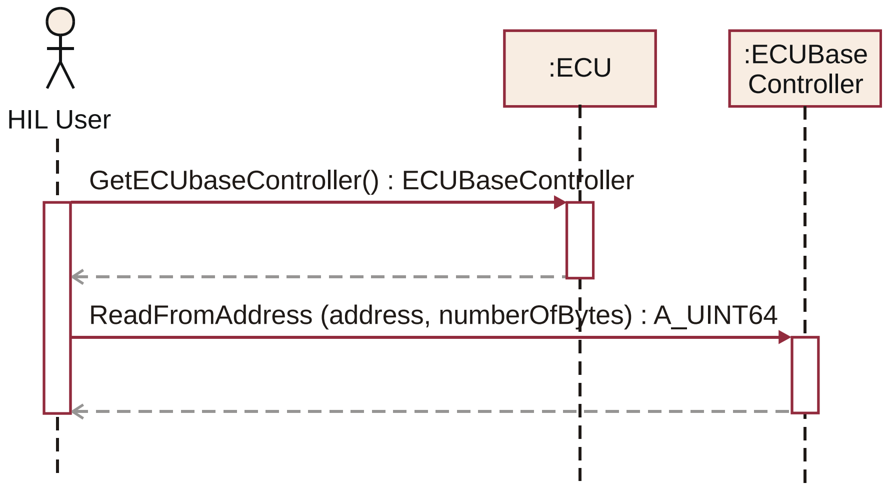

HIL User
:ECU
:ECUBase Controller
GetECUbaseController() : ECUBaseController
ReadFromAddress (address, numberOfBytes) : A_UINT64

Figure 78 Reading from the EEPROM

Reading data from an address in the EEPROM of the ECU is performed by using the ECUBaseController. A `ECUBaseController` object can be received from the `ECU` object by invoking its `GetECUBaseController()` method. See chapter 6.2.1 how to obtain an `ECU` object. With the ECUBaseController's `ReadFromAddress()` method the client can read a bunch of bytes from the ECU's EEPROM. The `ReadFromAddress()` takes two parameters: one for the address in the EEPROM and one for the number of bytes to read.

6.2.9 WRITING TO THE EEPROM

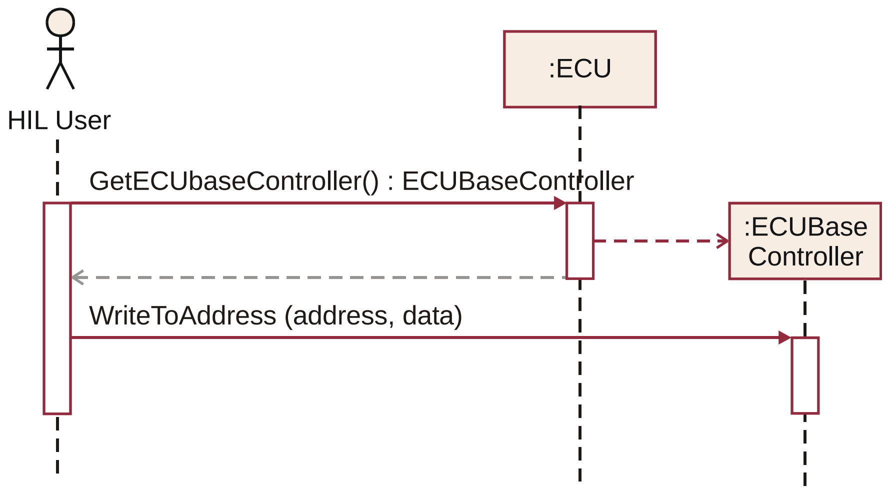

HIL User
:ECU
GetECUbaseController() : ECUBaseController
:ECUBase Controller
WriteToAddress (address, data)

Figure 79 Writing to the EEPROM

Writing data to an address in the EEPROM is performed by using the ECUBaseController. A client receives an `ECUBaseController` object by invoking the `ECU` object's `GetECUBaseController()` method. See chapter 6.2.1 how to obtain an ECU object. The ECUBaseController's `WriteToAddress()` method writes the data specified with the data parameter to the EEPROM at the address specified with the address parameter.

6.2.10 IMPLICIT AND EXPLICIT COMMUNICATION

The chapters above did all use the automatic (implicit) communication mode. In this case the client of the Diagnostic Port must not take care of creating and destroying communication channels in order to communicate with the ECU. In automatic communication mode it's up to the diagnostic tool–i.e. the HiL API Diagnostic Port server–to handle this task. The automatic communication mode is the *default* communication mode.
But there may also be use cases when a more sophisticated control over the communication is needed. That is why there is the explicit communication mode which is explained in the following chapter. In explicit communication mode the client is responsible for starting the communication with the `StartCommunication()` method before methods that need to communicate with the ECU can be invoked, e.g. `SendHexService()`. Also the client is responsible for stopping the communication with the `StopCommunication()` method after the mentioned method has been called.

6.2.11 SENDING HEX SERVICES WITH EXPLICIT COMMUNICATION

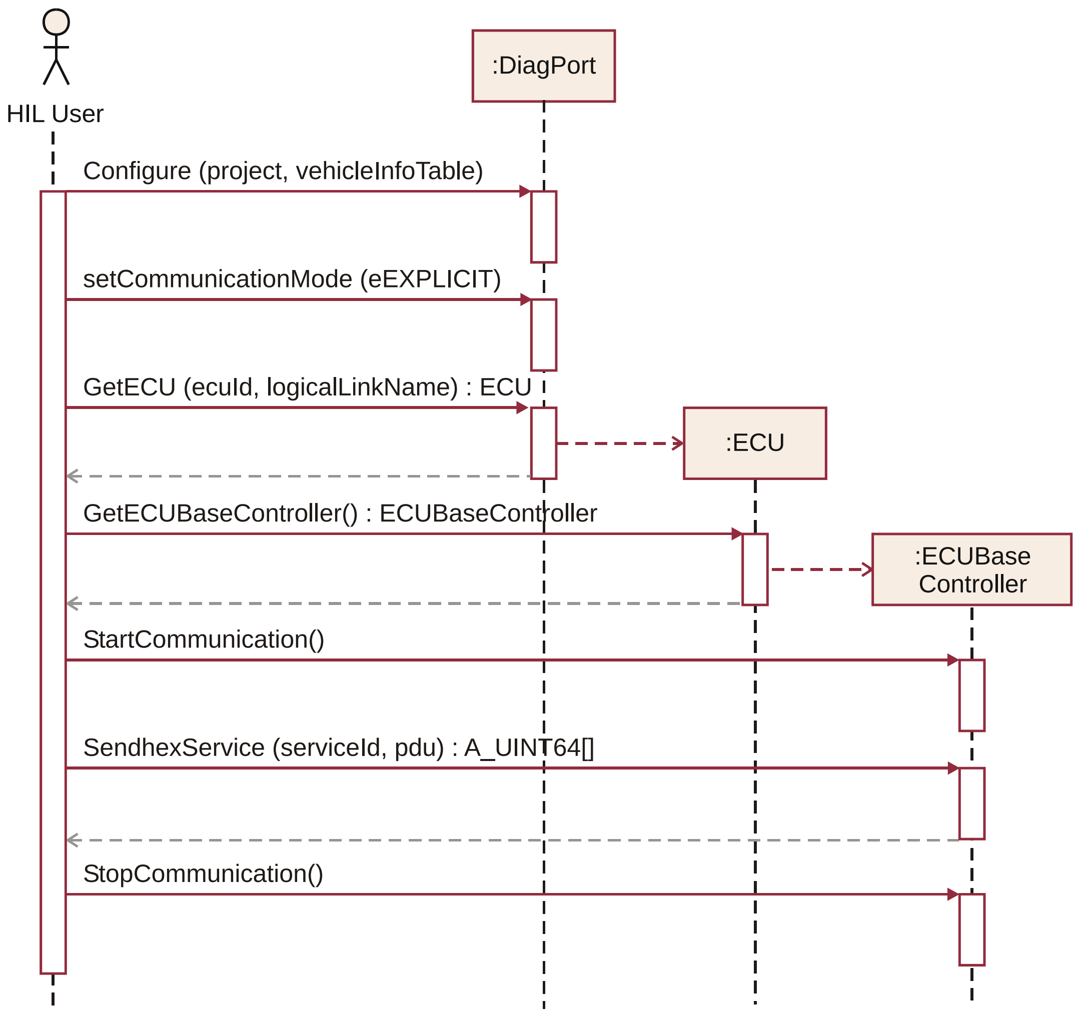

HIL User
:DiagPort
Configure (project, vehicleInfoTable)
setCommunicationMode (eEXPLICIT)
GetECU (ecuId, logicalLinkName) : ECU
:ECU
GetECUBaseController() : ECUBaseController
:ECUBase Controller
StartCommunication()
SendhexService (serviceId, pdu) : A_UINT64[]
StopCommunication()

Figure 80 Sending HEX service with explicit communication

This chapter illustrates the usage of the explicit communication mode. In explicit communication mode the client has to start and stop the communication with the ECU explicitly. After the project is configured the client receives the `ECU` object by invoking the

`GetECU()` method. Then the communication mode is set to explicit communication mode (`eEXPLICIT`) with the ECU class's `SetCommunicationMode()` method. Before calling an method that communicates with the ECU–e.g. the `SendHexService()` method–the `StartCommunication()` method of the `ECU` class has to be called. Finally the `StopCommunication()` method has to be called in order to stop the communication with the ECU. Figure 80 shows the steps needed to send an HEX service with explicit communication.
Since the Diagnostic Port only defines the two mentioned communication modes as state that can be changed via the `SetCommunicationMode()` method no state diagrams are provided here.

6.2.12 Executing Jobs

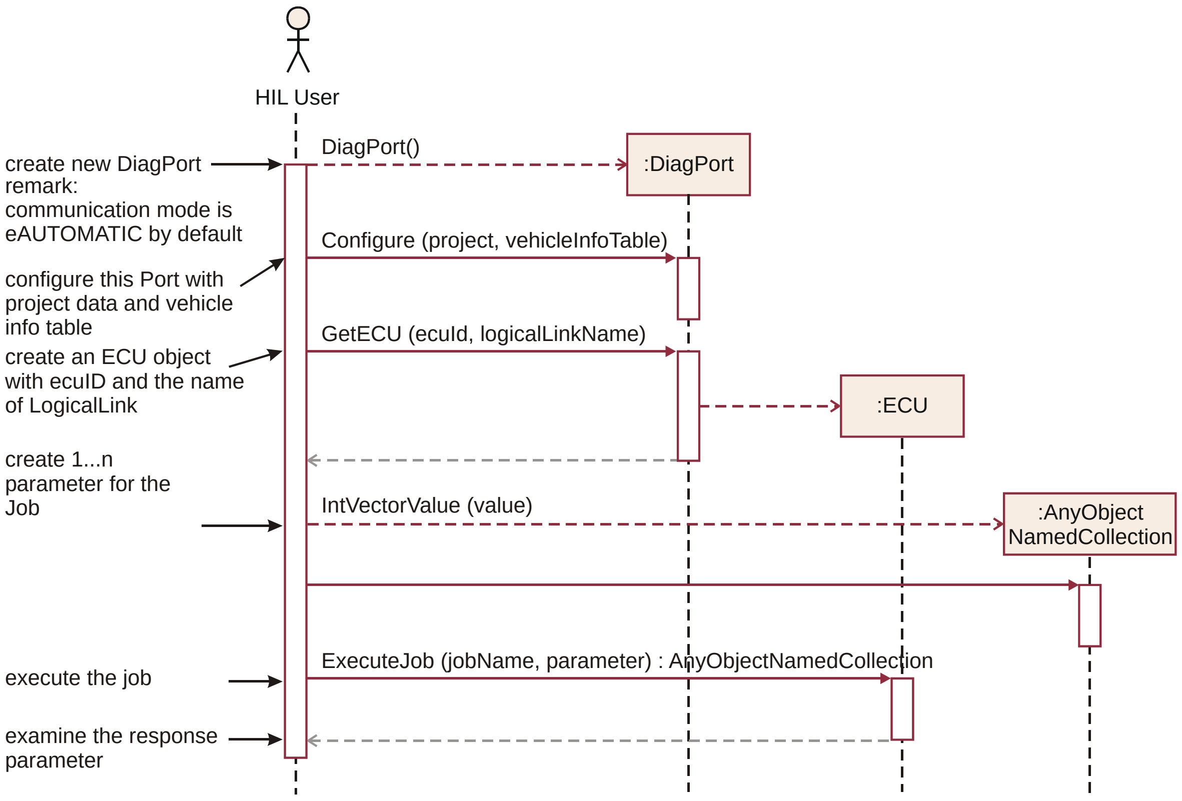


Figure 81 Executing a job

The Diagnostic Port API provides methods for the execution of diagnostic jobs. Figure 81 shows the steps needed to execute a diagnostic job. After configuration of the Diagnostic Port and receiving of the ECU object, the client creates the job parameters and invokes `the ExecuteJob()` method of the `ECU` class.

The sample code for this example will be found at

C#	C#\ASAM.HILAPI\SampleCode\DiagPort\DiagPortExample.cs
Python	Python\SampleCode\DiagPort\DiagPortExample.py
Java	JAVA\SampleCode\DiagPort\DiagPortExample.java

6.2.13 Reading Measurement Data from a Functional Group

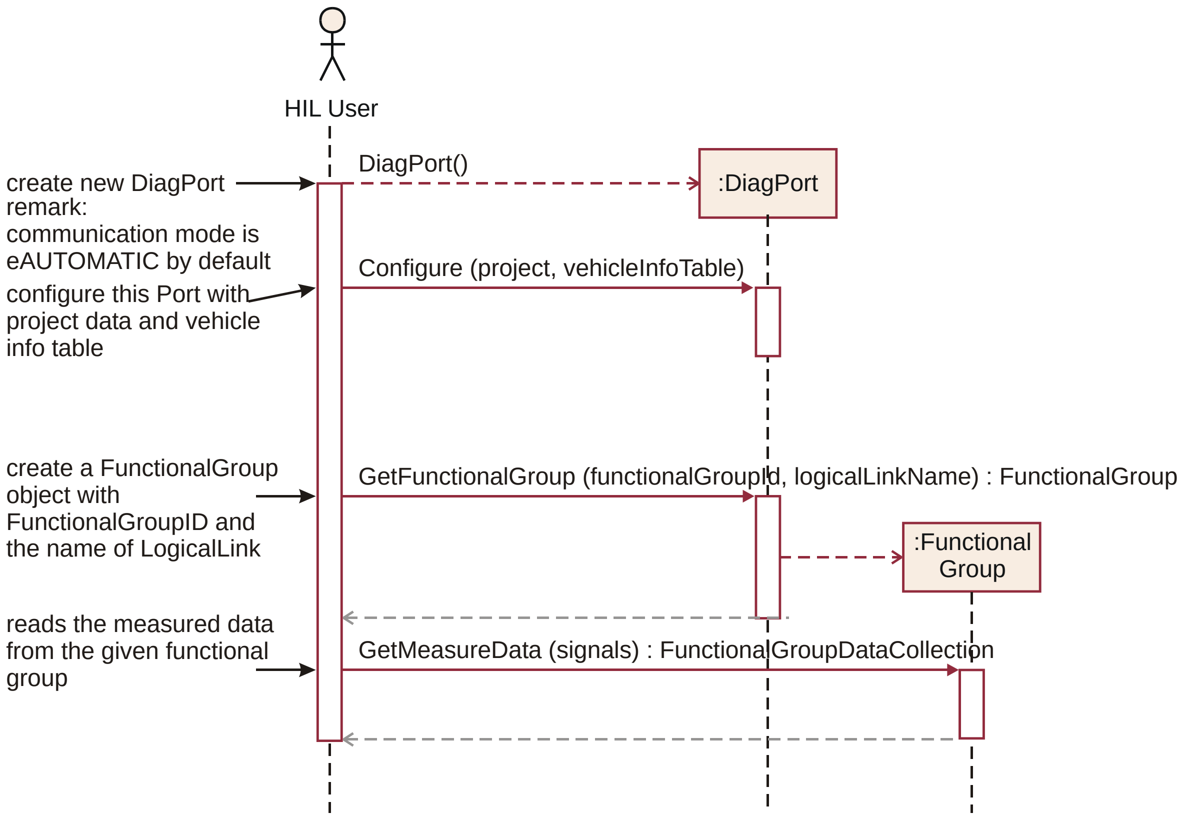

HIL User
create new DiagPort
remark:
communication mode is eAUTOMATIC by default
configure this Port with project data and vehicle info table
create a FunctionalGroup object with FunctionalGroupID and the name of LogicalLink
reads the measured data from the given functional group
DiagPort()
:DiagPort
Configure (project, vehicleInfoTable)
GetFunctionalGroup (functionalGroupId, logicalLinkName) : FunctionalGroup
:Functional Group
GetMeasureData (signals) : FunctionalGroupDataCollection

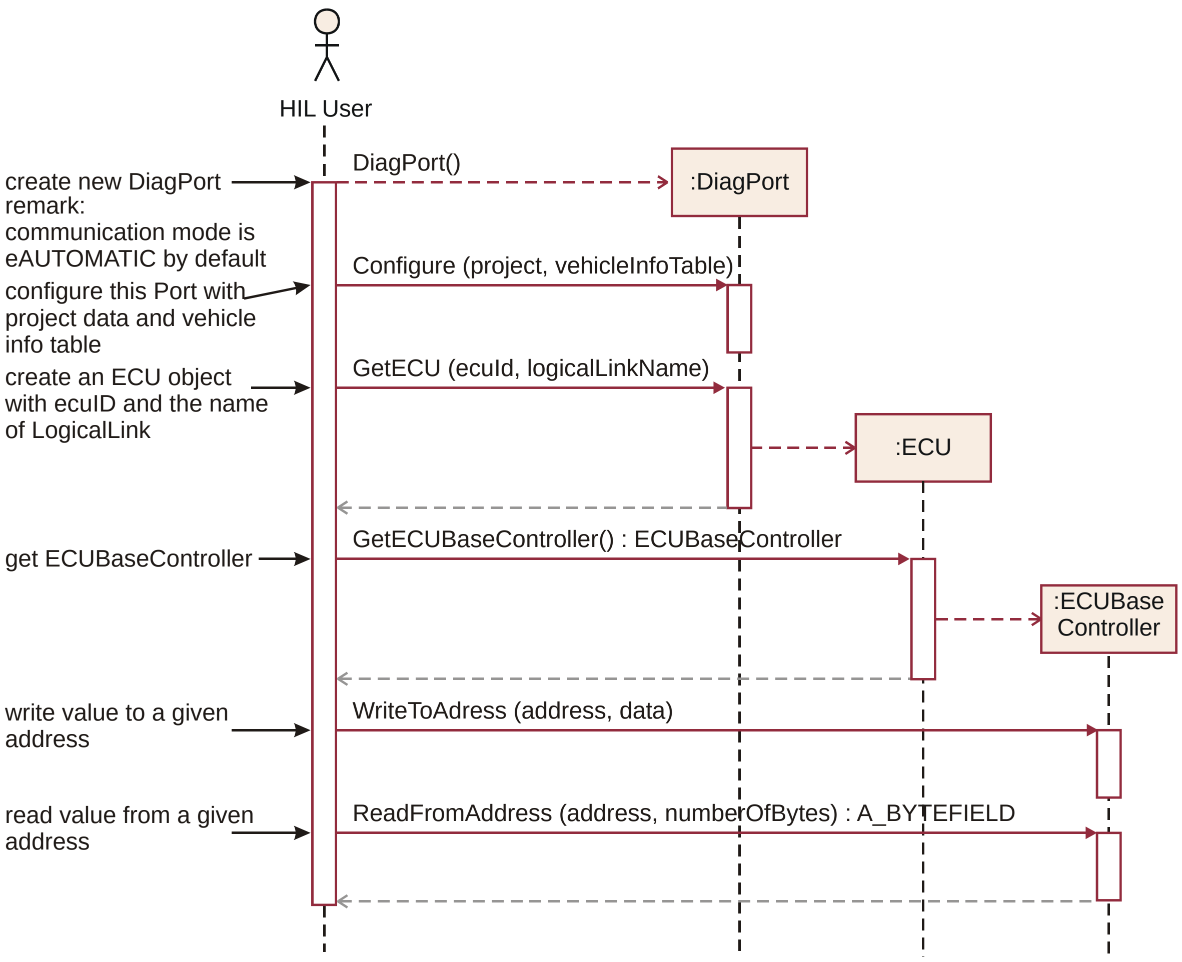
Figure 82 Reading measurement data from a functional group

Since the Diagnostic Port API supports functional groups, the client can read measurement data from a functional group, i.e. from a group of ECUs. Figure 82 depicts the steps needed to perform this task. After configuration of the `DiagPort` object and receiving of the `FunctionalGroup` object, the client invokes the Functional Group object's `GetMeasurementData()` method with the desired signal names as parameter.

The sample code for this example will be found at

C#	C#\ASAM.HILAPI\SampleCode\DiagPort\DiagPortExample.cs
Python	Python\SampleCode\DiagPort\DiagPortExample.py
Java	JAVA\SampleCode\DiagPort\DiagPortExample.java

6.2.14 USING THE BASECONTROLLER



HIL User
:DiagPort
:ECU
:ECUBase Controller
DiagPort()
Configure (project, vehicleInfoTable)
GetECU (ecuId, logicalLinkName)
GetECUBaseController() : ECUBaseController
WriteToAdress (address, data)
ReadFromAddress (address, numberOfBytes) : A_BYTEFIELD
create new DiagPort remark: communication mode is eAUTOMATIC by default
configure this Port with project data and vehicle info table
create an ECU object with ecuID and the name of LogicalLink
get ECUBaseController
write value to a given address
read value from a given address

Figure 83 Using the BaseController

The BaseController is used in cases a more sophistic access to the diagnostic tool is needed. With the BaseController's methods the client is able to read and write values directly to addresses of the ECU's memory. Figure 83 outlines the steps needed to read and write a number of bytes from and to the ECU's memory. The client receives the BaseController object directly from the `ECU` object as a return value of the `GetECUBaseController()` method.

The sample code for this example will be found at

C#	C#\ASAM.HILAPI\SampleCode\DiagPort\DiagPortExample.cs
Python	Python\SampleCode\DiagPort\DiagPortExample.py
Java	JAVA\SampleCode\DiagPort\DiagPortExample.java

6.3 SPECIAL HINTS

6.3.1 STRUCTURE OF RETURNED COLLECTIONS

The structures of the return collections of methods of the Diagnostic Port's classes depend on the ECU that is connected to the diagnostic tool. Therefore the Diagnostic Port API does not specify any return structures.

6.3.2 STATES IN THE DIAGNOSTIC TOOL

There are no restrictions on the invocation of methods that affect the communication mode or the communication status respectively.

7 EES PORT

7.1 USER CONCEPT

7.1.1 GENERAL

7.1.1.1 ELECTRICAL ERROR SIMULATION ON THE HIL SYSTEM

The pins of the system under test (SUT) are connected to the HIL system that provides power supply lines, communication busses like CAN, and simulated sensors and actuators. But test cases may not only comprise checking the behavior of the SUT in a fully functional environment. It is also important to check the SUT in case of electrical errors on the input and output pins. The question is: How does the SUT behave if the sensors and actuators are not working correctly or if they are not connected correctly?

Typical errors that have to be tested are electrical problems, mainly caused by wiring. To generate this class of errors the connections between the SUT and its environment (sensors, actuators, power supply, busses) have to be disturbed by an appropriate hardware system. This is the task of the so-called electrical error simulator (EES, see Figure 84).

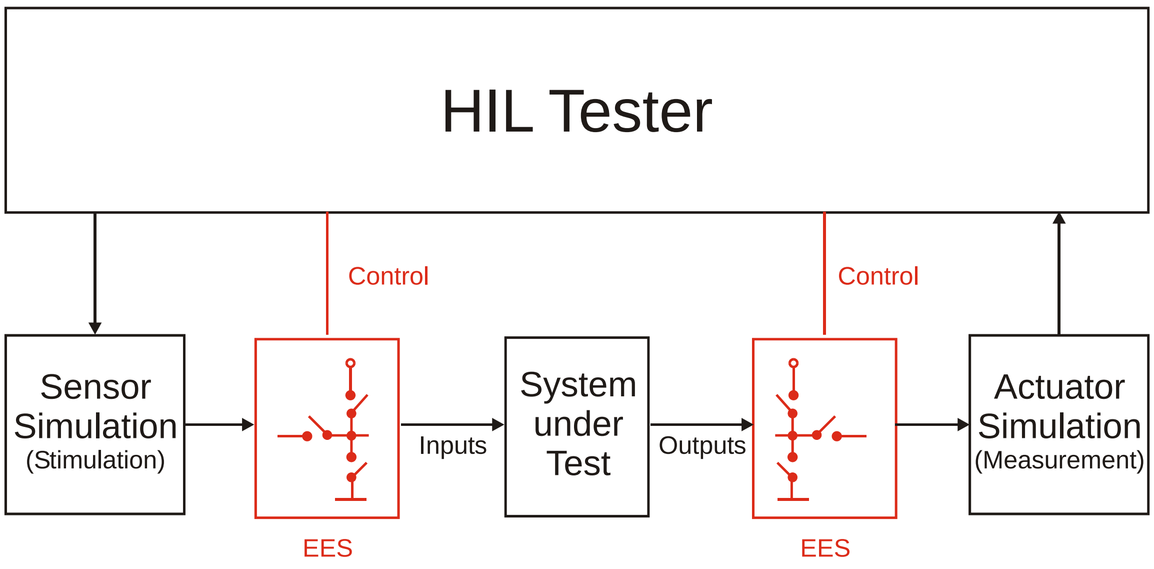

HIL Tester
Control
Control
Sensor Simulation (Stimulation)
Inputs
System under Test
Outputs
Actuator Simulation (Measurement)
EES
EES

Figure 84 Electrical Error Simulation is used to disturb the signals between the HIL system and the SUT

The electrical error simulator creates typical wiring errors like loose contacts, broken cables, short-circuits to neighboring pins, to ground (chassis) or to battery voltage. EES is provided by a special hardware in the HIL tester. But EES has not to be a separate component in general. It may be also integrated in a comprehensive HIL hardware.

The EES hardware is controlled by the test cases during the test, not by the real-time model. The HIL API EES port provides a general API for electrical error simulation hardware. The API hides the specific API of the used hardware, its driver software, and the communication between the machine the test is running on and the EES hardware. The EES port provides a defined set of functionality in an abstract manner. It is designed from the test case writer's point of view. Thus, the test case writer deals with some abstract error functionality. It is not necessary to know the technical details of the EES hardware.

7.1.1.2 Functional Principle of the EES Port

The general functional principle of the EES port is: A sequence of errors is defined using the HIL API by the test script. This is called the error configuration and may be stored in an XML file. The error configuration is downloaded to the specific EES hardware or software. The execution of the error sequence is completely transparent for the EES port user. It is done by the vendor-specific hardware, software, or driver.

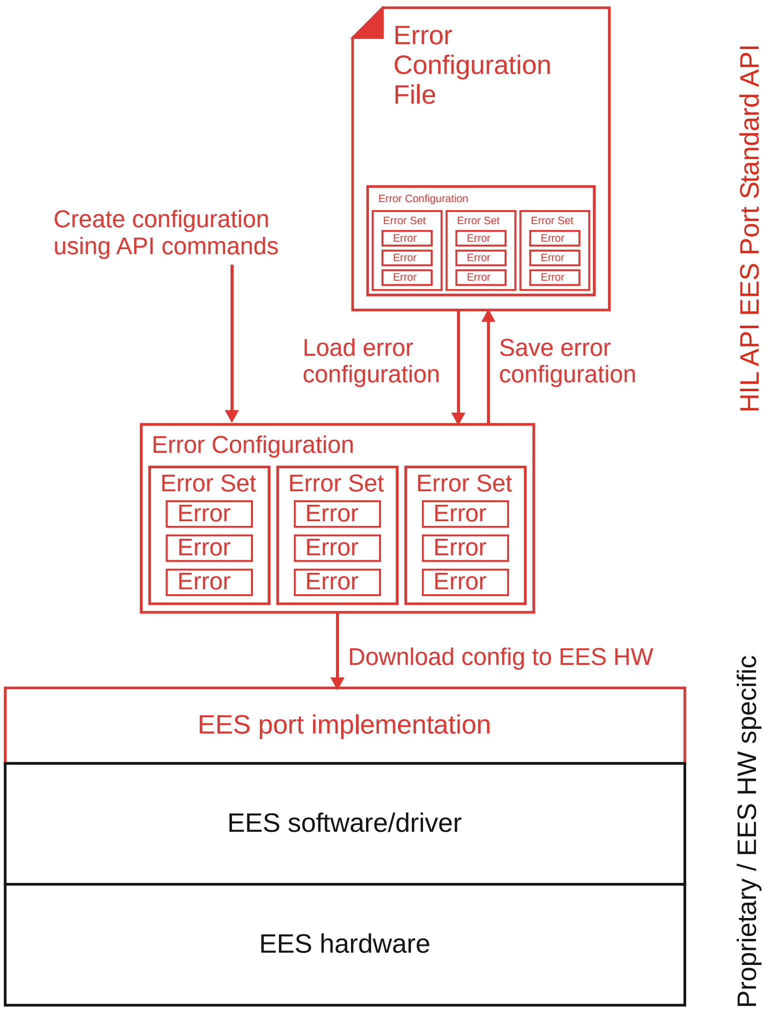


Figure 85 Error configurations are defined by EES ports and downloaded for execution to the vendor-specific EES hardware respective software

HIL API EES port mainly deals with the configuration of errors and starting a formerly configured error sequence by downloading (see Figure 85). Therefore the HIL API EES port is independent from a concrete EES implementation.

7.1.2 Configuration and Execution of Electrical Errors

7.1.2.1 Error Configuration

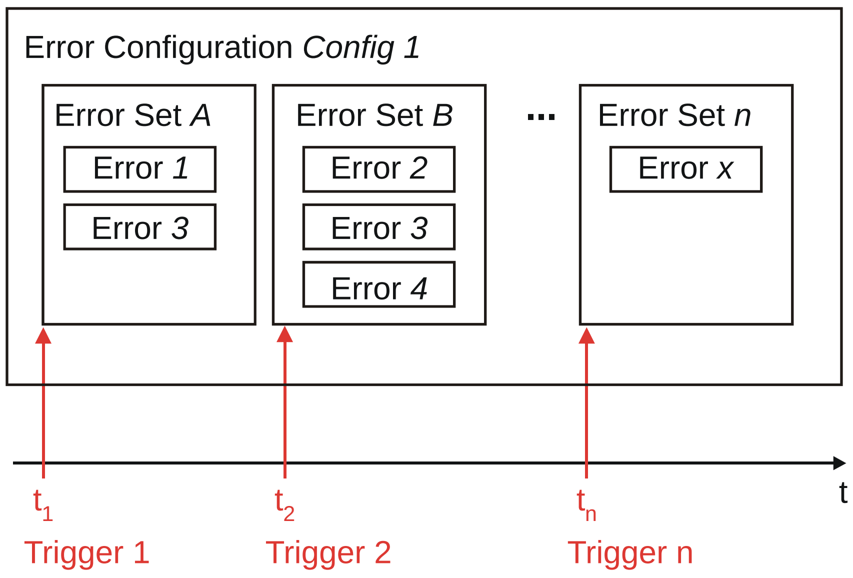


Figure 86 An error configuration comprises a sequence of error sets with several errors

An error is a defined disturbance of one or two electrical signals, typically pins of the SUT. E.g. an error may disturb a signal by interrupting the line and replacing it with a resistor. More than one error for different signals may be in effect at the same time. All errors that start at the same time are put together in an error set. An EES configuration comprises of a sequence of error sets (see Figure 86).

7.1.2.2 Execution of an Error Configuration

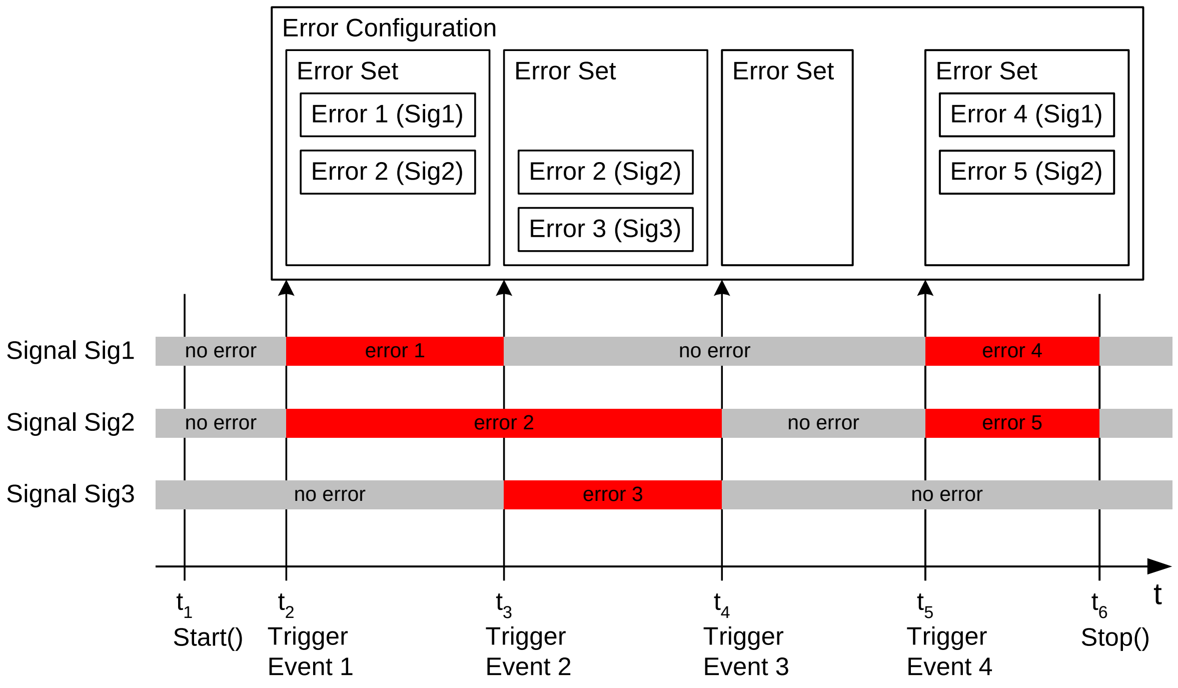


Figure 87 Example for the execution of an error configuration

To execute an error configuration, it has to be downloaded and started by the test case. When an error configuration is executed, the error sets are executed in the defined sequence (see Figure 87). That means the first error set is activated when the trigger for the first error set becomes true. All other triggers are not considered so far. The errors in the error set stay active as far as the trigger of the next error set becomes true.
The sequence of error sets is statically defined by the error configuration. It does not depend on the sequence the triggers of the error sets are fired. The triggers only determine when the next error set replaces the currently active error set.
The error configuration has to be stopped by the test case using the HIL API. The last error set remains active as long as the error configuration is not stopped. To get a defined end of signal disturbance, an empty error set can be used as the last error set in the configuration. Empty error sets can also be used to create error-free phases (refer to Figure 87 for an example).

If an error is defined in the same way in two consecutive error sets, the error will stay in action. There is no restart of the error or any other kind of influence when one error set is replaced by the consecutive error set containing the same error for the same signal.

7.1.2.3 Download of an Error Configuration

The EES error configuration has to be downloaded before it can be executed. Download means that the configuration has to be completely passed by the EES port to the specific EES driver, software, and hardware system. Typically the configuration will be physically downloaded to the hardware and executed there. But in general, this is not required by the HIL API definition. It is also possible that an error configuration is executed by the driver or another software system on the same PC.
Therefore, the conceptual sequence to create and use an error configuration is:

1. An error configuration is created by means of the HIL API EES port (construction by API calls or loading from a file).
2. Now the error configuration is stored on the computer the HIL API is running on and may be changed.
3. Then the error configuration is downloaded to the EES system, a specific software/hardware system. In any case, the error configuration cannot be changed any longer.
4. After starting the error configuration in the test case (using HIL API), the defined sequence of errors is executed. Execution is independent from the test case and the HIL API.
5. When the HIL API stops the execution, all kinds of disturbance immediately stop. Possibly the last error set of the error configuration remains active until the execution is stopped.

A downloaded error configuration can be used several times.

7.1.3 TRIGGERS IN EES

The EES system uses trigger events to switch from one error set to another error set. These triggers are handled by the EES hardware and software. They are not defined by the HIL API. And there are also no means to define trigger conditions for EES error sets in the EES port. In an EES error configuration only the type of the awaited trigger is defined. The type of a trigger can be thought as the trigger input connection of an appropriate hardware. But in fact, the trigger may be controlled by software also.
From EES port's point of view an EES system has three possible trigger inputs:

- Manual trigger: This trigger is fired by the controlling test script. The EES port offers a method to fire this trigger.
- Hardware trigger: The hardware trigger reacts on some kind of electrical trigger line of the EES hardware. Further details are not defined by the HIL API. It is just expected that the EES hardware has some kind of a hardware trigger input.
- Software trigger: The software trigger reacts on a trigger signal defined in the model or another software part of the HIL system. It is not defined by HIL API or the EES error configuration how the EES system is associated with the software system.

If an additional configuration of triggers is needed by the EES system, this has to be done using the EES specific software interface. There are no means in HIL API so far to define additional options for the triggers.

7.1.4 ELECTRICAL ERRORS

An error is defined by several independent aspects: the error category, the error type, and the option to disturb with or without load.

7.1.4.1 Error Category

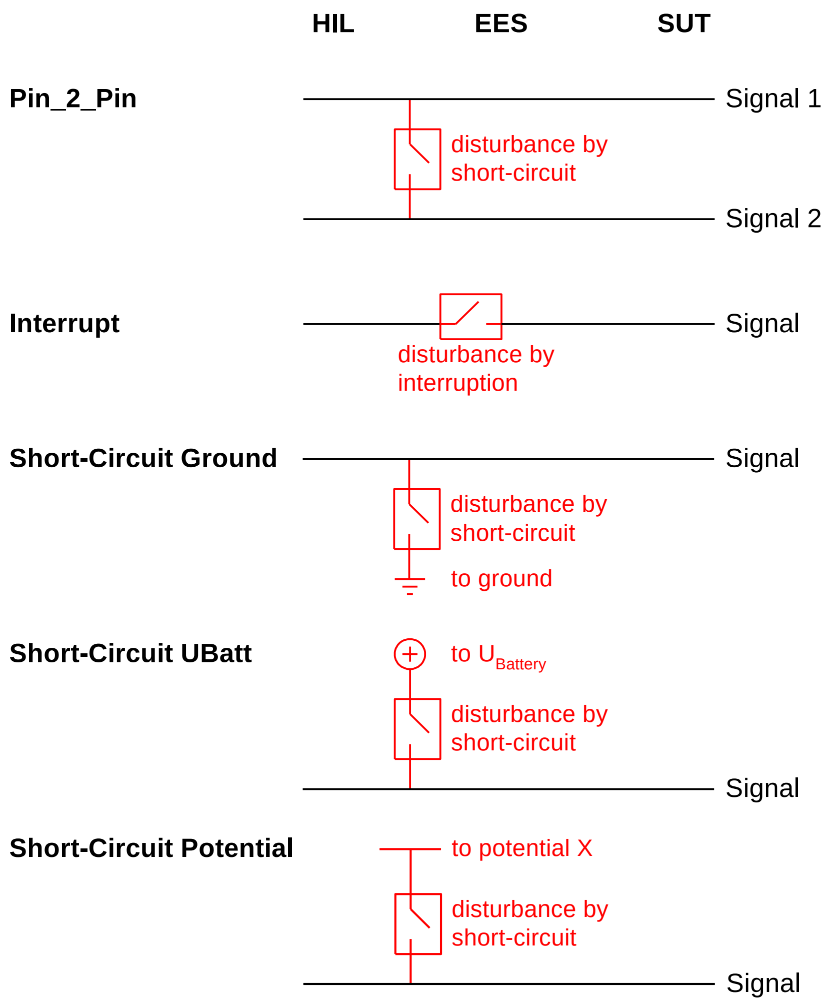


Figure 88 Illustration of the error categories defined by EES port

The error category defines how a signal should be disturbed. A signal is interrupted or connected to another signal or potential (see Figure 88). The way the interruption or short-circuit is provided is not defined by the error category (but by the error type, see chapter 7.1.4.2).
Typically the error category affects one signal. Only in case of a pin to pin error two signals are affected.

The error short-circuit to potential is the generalized form of a short-circuit error. For this category of errors the EES hardware has to provide additional potentials beside $U_{battery}$ and ground. Multiple potentials identified by numbers may be supported. $U_{battery}$ and ground are covered by separate categories because of their importance.

7.1.4.2 ERROR TYPE

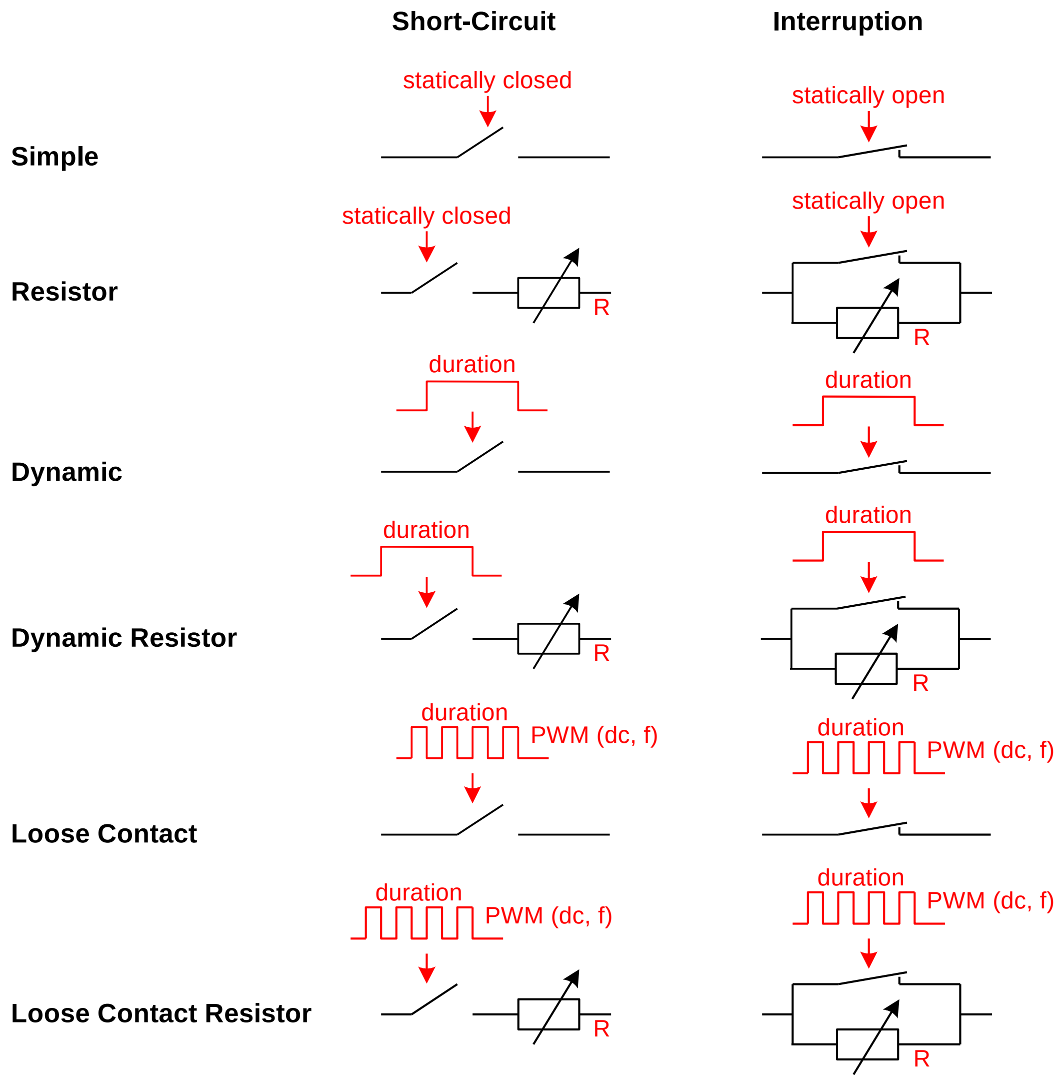


Figure 89 Illustration of the error types defined by EES port

The error type defines the disturbance itself. There are several possibilities that differ in the dynamic of the disturbance (static, for a defined duration, controlled by a PWM signal) and the resistance in case of the error (defined resistance or completely open/closed).
The concrete circuit also differs between short-circuit errors and interrupt errors, because in one case the error is caused by closing a connection, in the other by opening the connection. Nevertheless, the idea of an error of the same error type is the same in both cases.

Figure 89 shows the available error types and the principal circuits used for short-circuits and interrupts.

7.1.4.3 WITH OR WITHOUT LOAD

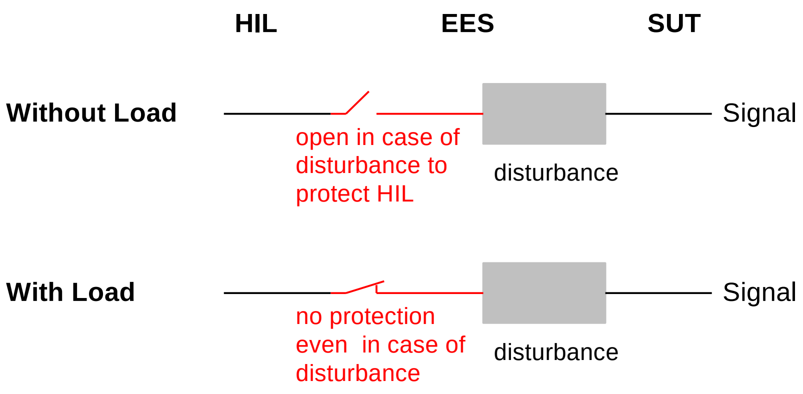

HIL
EES
SUT
Without Load
open in case of disturbance to protect HIL
disturbance
Signal
With Load
no protection even in case of disturbance
disturbance
Signal

Figure 90 Illustration of the option with load and without load

The option “with load” or “without load” is an additional aspect of an error. This aspect is orthogonal to error category and error type and can be freely chosen for almost every kind of error. Only interruptions do not provide this option because technically it does not make any sense in this case.
Background: If a signal between the HIL and the SUT is disturbed by the EES hardware, not only the SUT has to deal with the disturbance. A short-circuit for example effects the HIL hardware, too. To protect the HIL hardware the EES can open the connection between HIL and EES. Thus, the disturbance has an effect on the SUT but cannot damage the HIL.

Figure 90 shows the principal circuit of the with/without load protection in the EES hardware.

7.1.5 API

The EES port API consists of several classes to control the EES system and to create, store, load, and represent error configurations. In the following chapters the most important classes of the EES port and the relations between them are described.

7.1.5.1 EES PORT

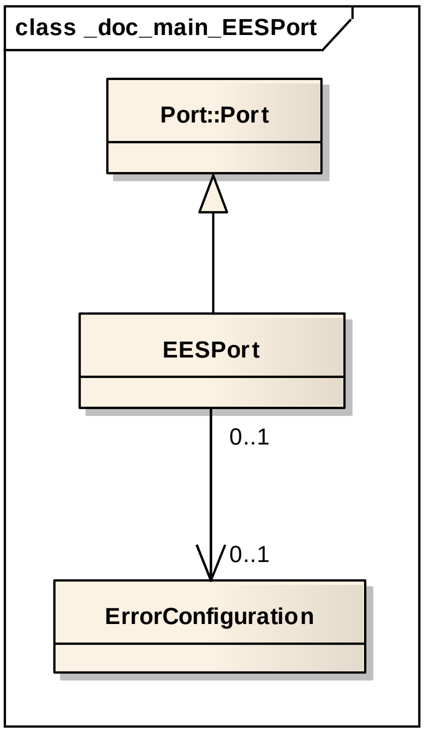


Figure 91 The main EES port classes

The EES port itself is represented by the class `EESPort`. This class is derived from the general HIL API Port class. `EESPort` provides methods to download an error configuration, to start and stop the execution of a downloaded error configuration on the EES system, and to trigger the EES system manually. This is the way a trigger of type "MANUAL" is fired.
To synchronize the test run of the test script with the execution of the error configuration, the synchronous method `WaitForTrigger` can be used. This method waits until the next trigger event defined in the error configuration. Alternatively the method returns with an error when the timeout time is reached.

An error configuration can be created using the constructor of the class `ErrorConfiguration`. It is possible to create several error configurations. But only one error configuration can be assigned to the `EESPort` instance. The assignment overrides an older assignment. Assigned error configurations can be changed. But after downloading the error configuration to the EES system no further changes are considered. Nevertheless, the error configuration may be downloaded again.

7.1.5.2 Error Configuration

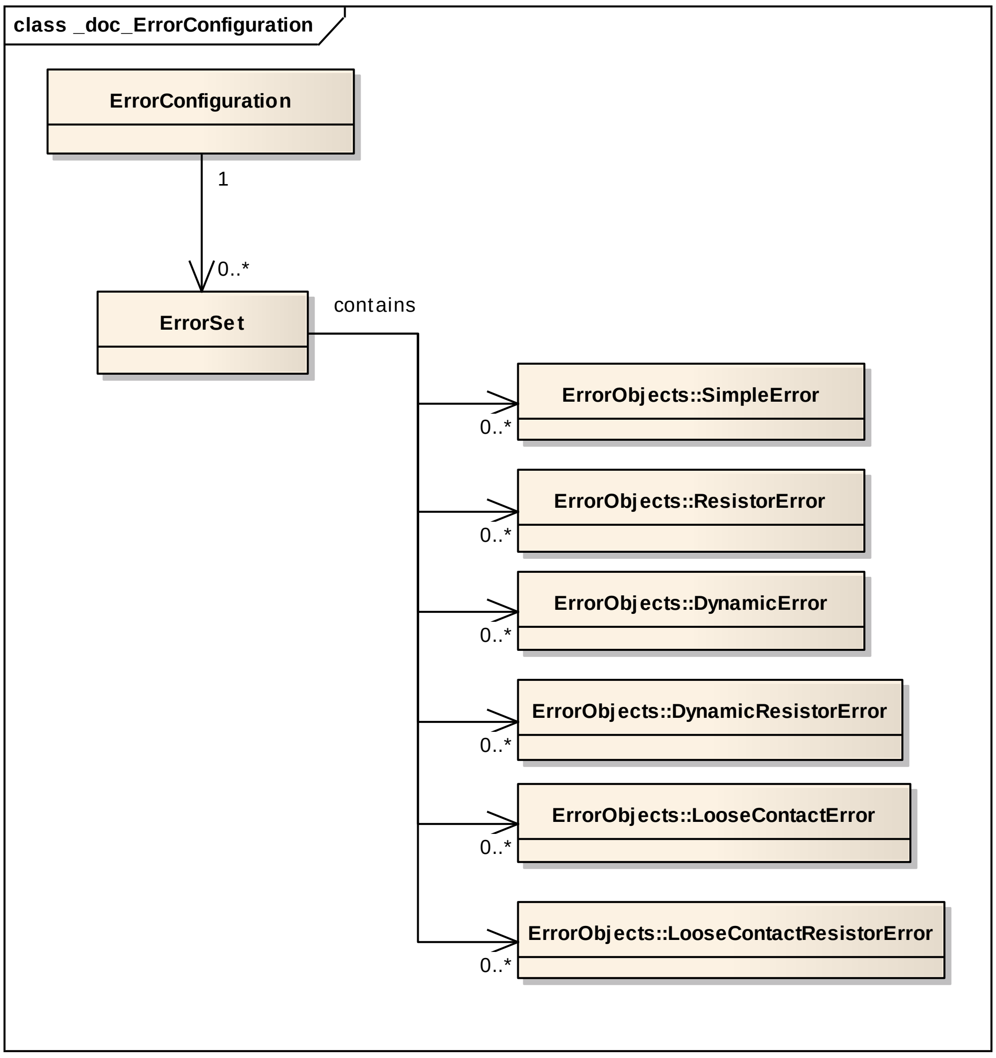

class _doc_ErrorConfiguration
ErrorConfiguration
1
0..*
ErrorSet
contains
ErrorObjects::SimpleError
0..*
ErrorObjects::ResistorError
0..*
ErrorObjects::DynamicError
0..*
ErrorObjects::DynamicResistorError
0..*
ErrorObjects::LooseContactError
0..*
ErrorObjects::LooseContactResistorError
0..*

Figure 92 Classes used to represent an error configuration

According to the structure of an error configuration there exists one class for the error configuration itself (`ErrorConfiguration`), one for the error sets (`ErrorSet`), and several classes for errors of the different error types (`SimpleError`, …).
Only the error configuration class can be instantiated by the constructor. All other classes are constructed using the factory method respective the error factory class of the error configuration object. Therefore error sets and error objects exist only in the context of an error configuration and will be destroyed automatically when the error configuration is destroyed by the user (using the destructor of `ErrorConfiguration`).
Errors (`SimpleError`, …) are created using the factory provided by the error configuration instance and then assigned to one or more error sets. It is not allowed to assign an error to another error configuration.

Properties of error objects and error sets (like name) cannot be changed after creation. These properties are set at creation by the factory class. In this sense, these objects are read-only.

7.1.5.3 Error Objects

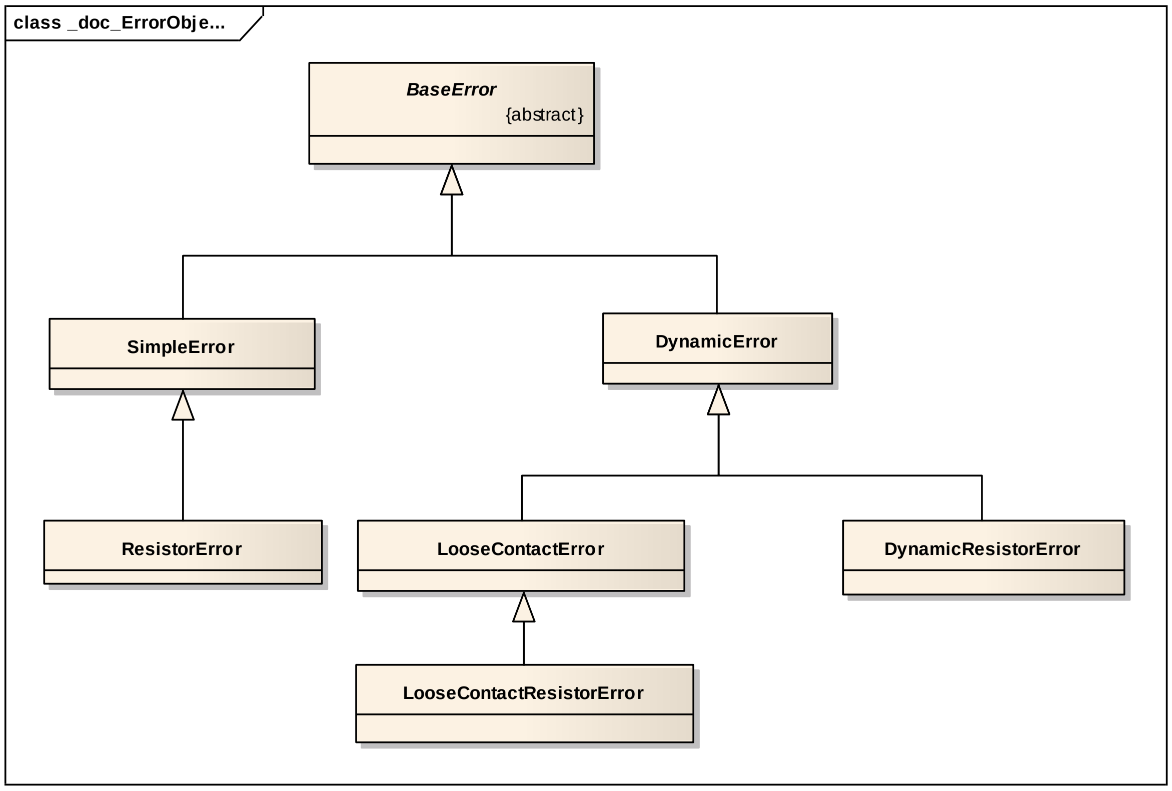


Figure 93 Class hierarchy of error objects

The six error classes represent the six error types. Other differentiation characteristics of errors like error category and option with/without load are stored as attributes in these classes.

The error classes are hierarchically organized with a common base class. The base class BaseError is abstract and cannot be instantiated. Attributes and variables of this type store an instance of an arbitrary error.

7.1.5.4 Creation of Error Objects

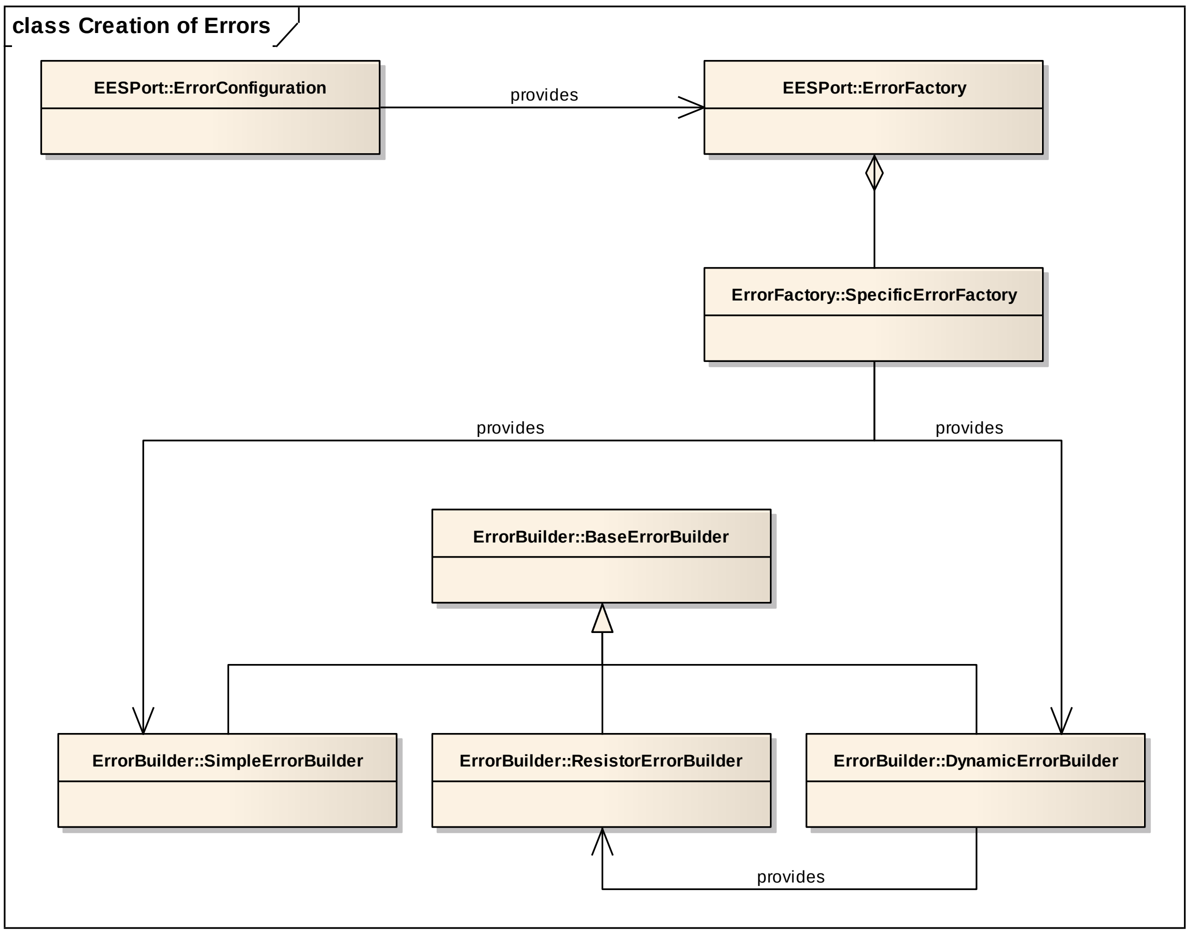

class Creation of Errors
EESPort::ErrorConfiguration
provides
EESPort::ErrorFactory
ErrorFactory::SpecificErrorFactory
provides
provides
ErrorBuilder::BaseErrorBuilder
ErrorBuilder::SimpleErrorBuilder
ErrorBuilder::ResistorErrorBuilder
ErrorBuilder::DynamicErrorBuilder
provides

Figure 94 Classes used to create error objects

Factory and builder classes are used to create a specific error object. The result of the builder is an instance of one error class (see chapter 7.1.5.3).
In principle, error objects are created by the following sequence:

1. Fetch factory `ErrorFactory` from error configuration. Use the error configuration the new error should belong to.
2. Choose the error category and define the affected signal or signals. For each signal the option with or without load is defined, too. The `ErrorFactory` will return a `SpecificErrorFactory`.
3. Choose whether you want to create a simple or a dynamic error. This is one aspect of the error type. The `SpecificErrorFactory` will return an object of class `SimpleErrorBuilder` or class `DynamicErrorBuilder`.
4. Configure now the error type using the error builder object. Possibly the error builder returns another error builder so that the building process may comprise several levels.
5. The method `ToBaseError()` returns the configured error object. `ToBaseError()` is available in all error builder classes.

Factories and builder objects cannot be created or destroyed by the user. They must be fetched again from the according father object each time a new error should be created.

This multi-level structure of factory/builder objects is designed to be used in a one line statement to create an error object with all its characteristics. The example code demonstrates how this works in the supported programming languages.

7.1.5.5 Reader and Writer for EES Configuration Files

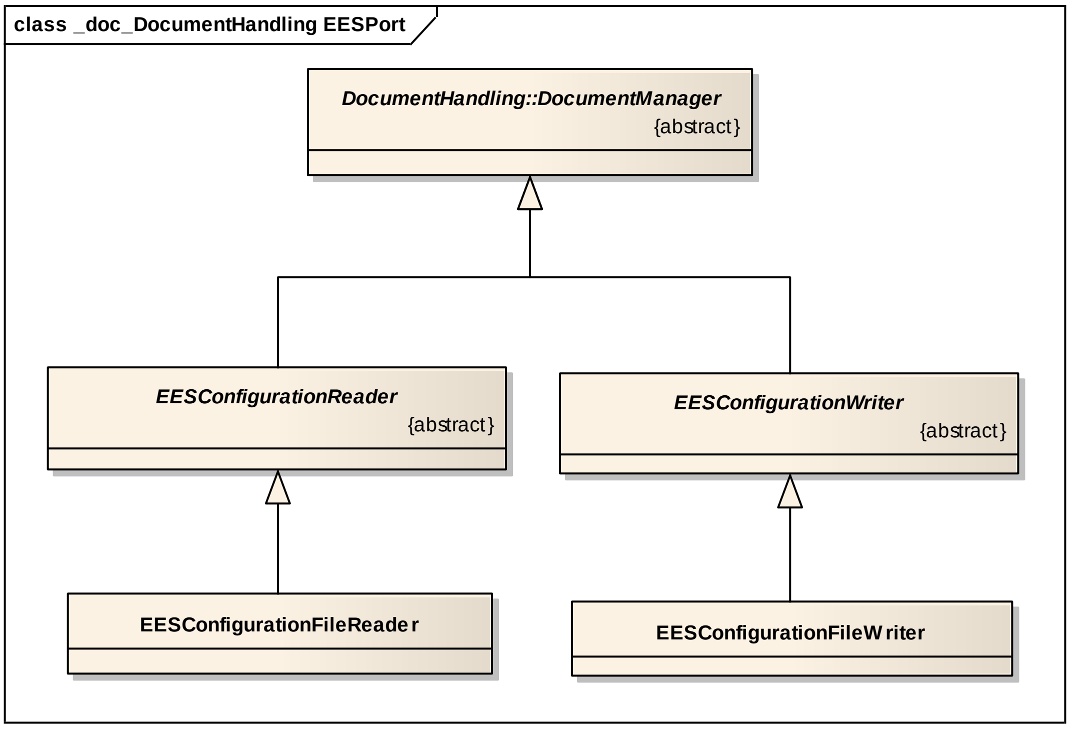

class _doc_DocumentHandling EESPort
DocumentHandling::DocumentManager
{abstract}
EESConfigurationReader
{abstract}
EESConfigurationWriter
{abstract}
EESConfigurationFileReader
EESConfigurationFileWriter

Figure 95 Reader and writer classes for error configuration files

Error configurations can be externally stored, normally in files. To allow different storage types and formats, reading and writing is handled by abstract reader and writer classes. For the EES error configuration these are the abstract classes `EESConfigurationReader` and `EESConfigurationWriter`. Both classes are derived from the common HIL API document handler class `DocumentManager`.

HIL API supports an XML format to store complete error configuration in a file. The schema definition of this XML file is part of the HIL API standard (EESConfiguration.xsd). The specific reader and writer classes are `EESConfigurationFileReader` and `EESConfigurationFileWriter`.

7.1.6 EES Port States

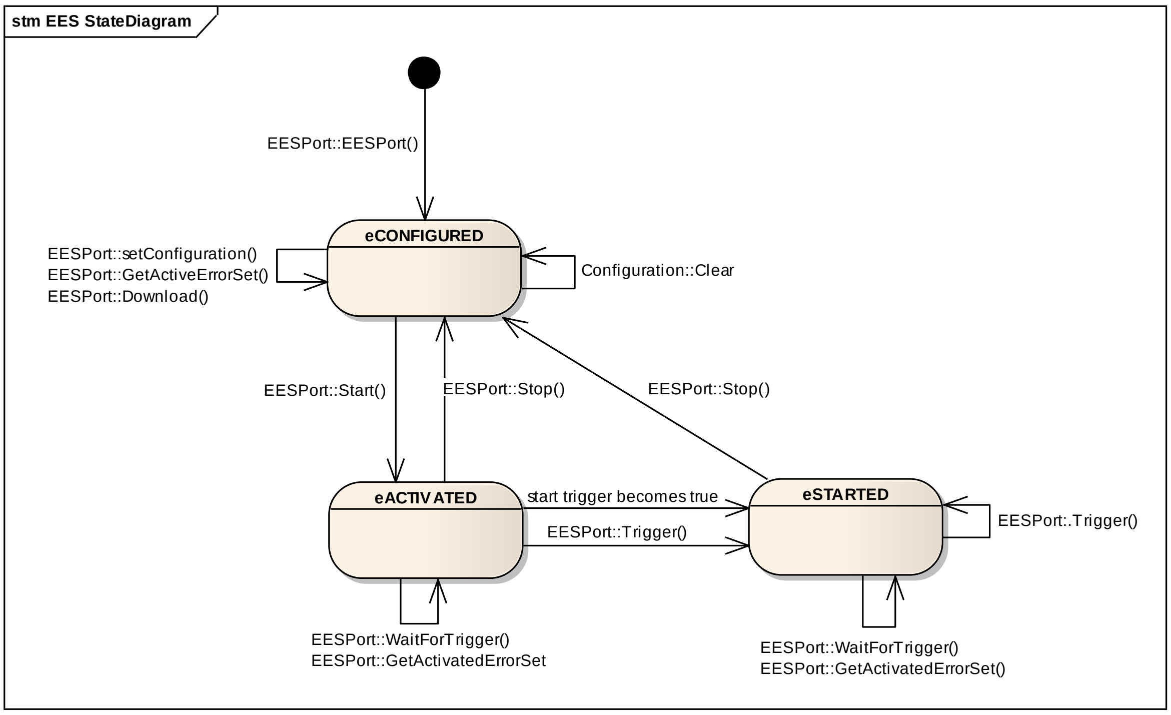

stm EES StateDiagram
EESPort::EESPort()
eCONFIGURED
EESPort::setConfiguration()
EESPort::GetActiveErrorSet()
EESPort::Download()
Configuration::Clear
EESPort::Start()
EESPort::Stop()
EESPort::Stop()
eACTIVATED
start trigger becomes true
EESPort::Trigger()
eSTARTED
EESPort::.Trigger()
EESPort::WaitForTrigger()
EESPort::GetActivatedErrorSet
EESPort::WaitForTrigger()
EESPort::GetActivatedErrorSet()

Figure 96 State diagram for EES port

The EES port has three states:

- `eCONFIGURED`:
  The base state of the EES port. In this state it is possible to configure the port and assign an error configuration.
- `eACTIVATED`:
  An error configuration is downloaded to the EES system and started. The EES system waits for the first defined trigger event to execute the first error set in the error configuration. This may be also a manual trigger, fired by using the method Trigger of the EES port object.
- `eSTARTED`:
  An error set is active. The EES system waits for the next defined trigger event to switch to the next error set.

The execution of an error configuration is divided into two states because the sequence of error sets does not start before the first trigger fires. From the user's point of view there is no difference between those two states. He can fire manual triggers, stop the execution of the error configuration, wait for the next trigger to synchronize the execution of the test script, or query the currently active error set.

Configuration of the EES port is not possible, not necessary, and not reasonable during execution of an error configuration. Therefore configuration is only possible in the configured state.

7.2 USAGE OF THIS PORT

7.2.1 CREATING ERROR CONFIGURATIONS BY API

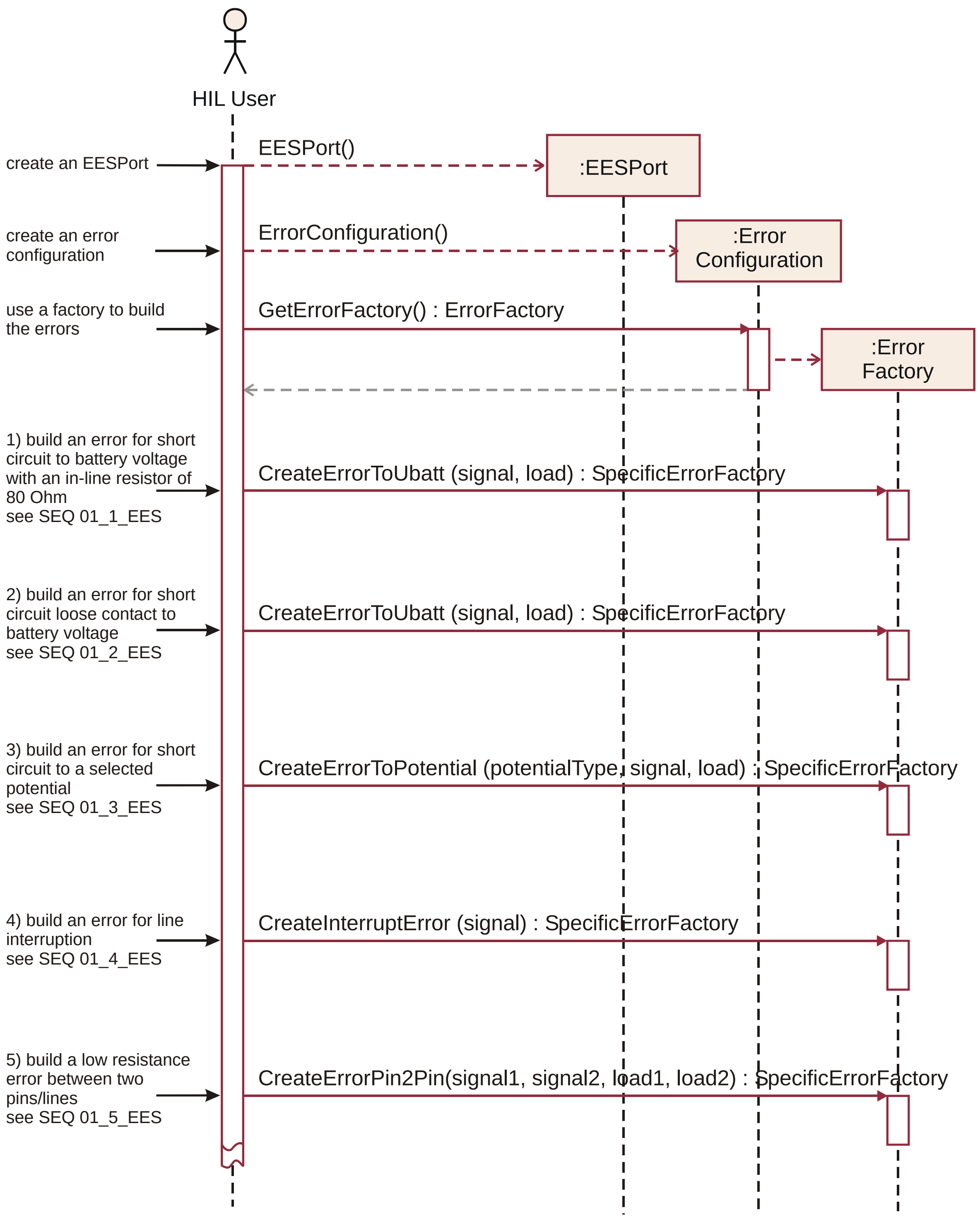

HIL User
create an EESPort
EESPort()
:EESPort
create an error configuration
ErrorConfiguration()
:Error Configuration
use a factory to build the errors
GetErrorFactory() : ErrorFactory
:Error Factory
1) build an error for short circuit to battery voltage with an in-line resistor of 80 Ohm see SEQ 01_1_EES
CreateErrorToUbatt (signal, load) : SpecificErrorFactory
2) build an error for short circuit loose contact to battery voltage see SEQ 01_2_EES
CreateErrorToUbatt (signal, load) : SpecificErrorFactory
3) build an error for short circuit to a selected potential see SEQ 01_3_EES
CreateErrorToPotential (potentialType, signal, load) : SpecificErrorFactory
4) build an error for line interruption see SEQ 01_4_EES
CreateInterruptError (signal) : SpecificErrorFactory
5) build a low resistance error between two pins/lines see SEQ 01_5_EES
CreateErrorPin2Pin(signal1, signal2, load1, load2) : SpecificErrorFactory

Figure 97 Sequence of example "create error configuration by API" (part1)

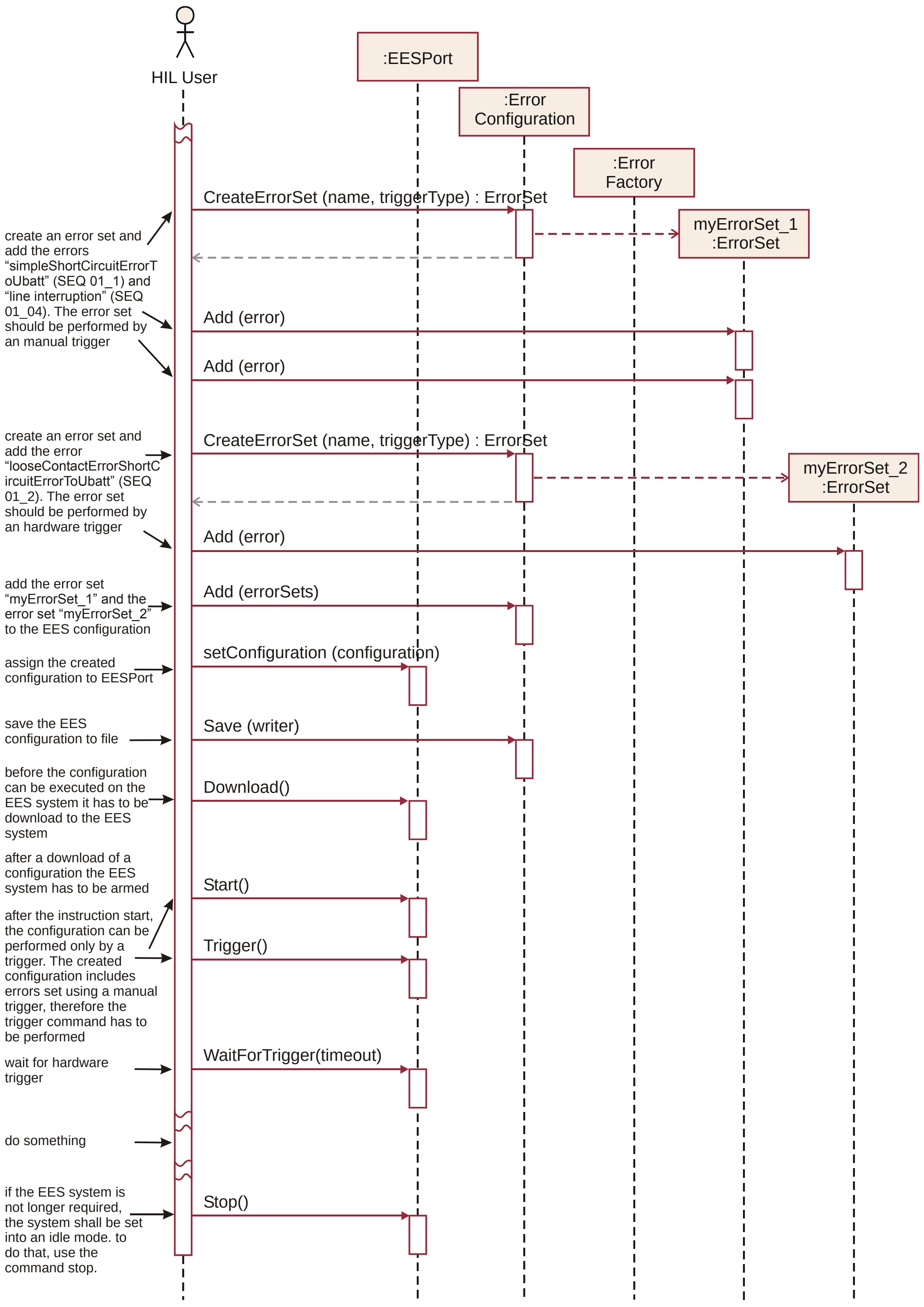

HIL User
:EESPort
:Error Configuration
:Error Factory
myErrorSet_1 :ErrorSet
myErrorSet_2 :ErrorSet
CreateErrorSet (name, triggerType) : ErrorSet
Add (error)
Add (error)
CreateErrorSet (name, triggerType) : ErrorSet
Add (error)
Add (errorSets)
setConfiguration (configuration)
Save (writer)
Download()
Start()
Trigger()
WaitForTrigger(timeout)
Stop()
create an error set and add the errors “simpleShortCircuitErrorToUbatt” (SEQ 01_1) and “line interruption” (SEQ 01_04). The error set should be performed by an manual trigger
create an error set and add the error “looseContactErrorShortCircuitErrorToUbatt” (SEQ 01_2). The error set should be performed by an hardware trigger
add the error set “myErrorSet_1” and the error set “myErrorSet_2” to the EES configuration
assign the created configuration to EESPort
save the EES configuration to file
before the configuration can be executed on the EES system it has to be download to the EES system
after a download of a configuration the EES system has to be armed
after the instruction start, the configuration can be performed only by a trigger. The created configuration includes errors set using a manual trigger, therefore the trigger command has to be performed
wait for hardware trigger
do something
if the EES system is not longer required, the system shall be set into an idle mode. to do that, use the command stop.

Figure 98 Sequence of example “create error configuration by API” (part2)

The example shows how an error configuration is created. First a new error configuration is created. This is independent from the EES port instance. In a second step, a specific error is created using the factory and builder objects of the configuration (see Figure 97 and also Figure 99 till Figure 103). In the third step, a new error set is requested from the error configuration and the error is assigned to this error set. The error configuration is assigned to the EES port instance and stored in a file.

The created error configuration is executed in the lower part of the sequence. This is straight forward: the error configuration is assigned to the EES port, it is downloaded, and started. Then a manual trigger is fired and the next trigger is awaited (a hardware trigger as defined in the second error set). At the end, the execution of the error set is stopped.

The sample code for this and the following examples will be found at

C#	C#/SampleCode/EESPort/EESPortExample.cs
Python	Python/SampleCode/EESPort/EESPortExample.py
Java	JAVA/SampleCode/EESPort/EESPortExample.java

7.2.2 Creating Error Objects

The error objects stored in the error configuration are created using the error factory and error builder classes. The following five sequence diagrams show how to create the different errors.

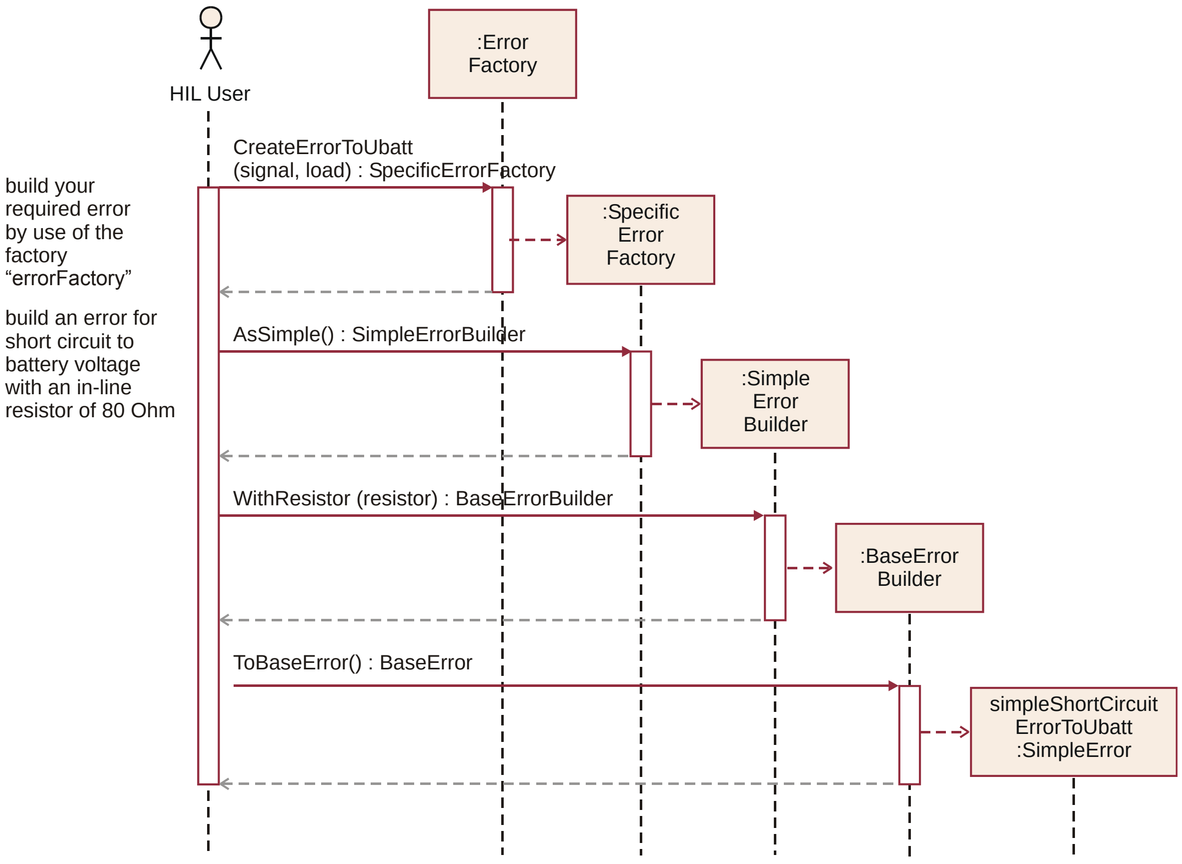

HIL User
:Error Factory
CreateErrorToUbatt (signal, load) : SpecificErrorFactory
build your required error by use of the factory “errorFactory”
:Specific Error Factory
build an error for short circuit to battery voltage with an in-line resistor of 80 Ohm
AsSimple() : SimpleErrorBuilder
:Simple Error Builder
WithResistor (resistor) : BaseErrorBuilder
:BaseError Builder
ToBaseError() : BaseError
simpleShortCircuit ErrorToUbatt :SimpleError

Figure 99 Sequence to create a short-circuit to $U_{battery}$ error object

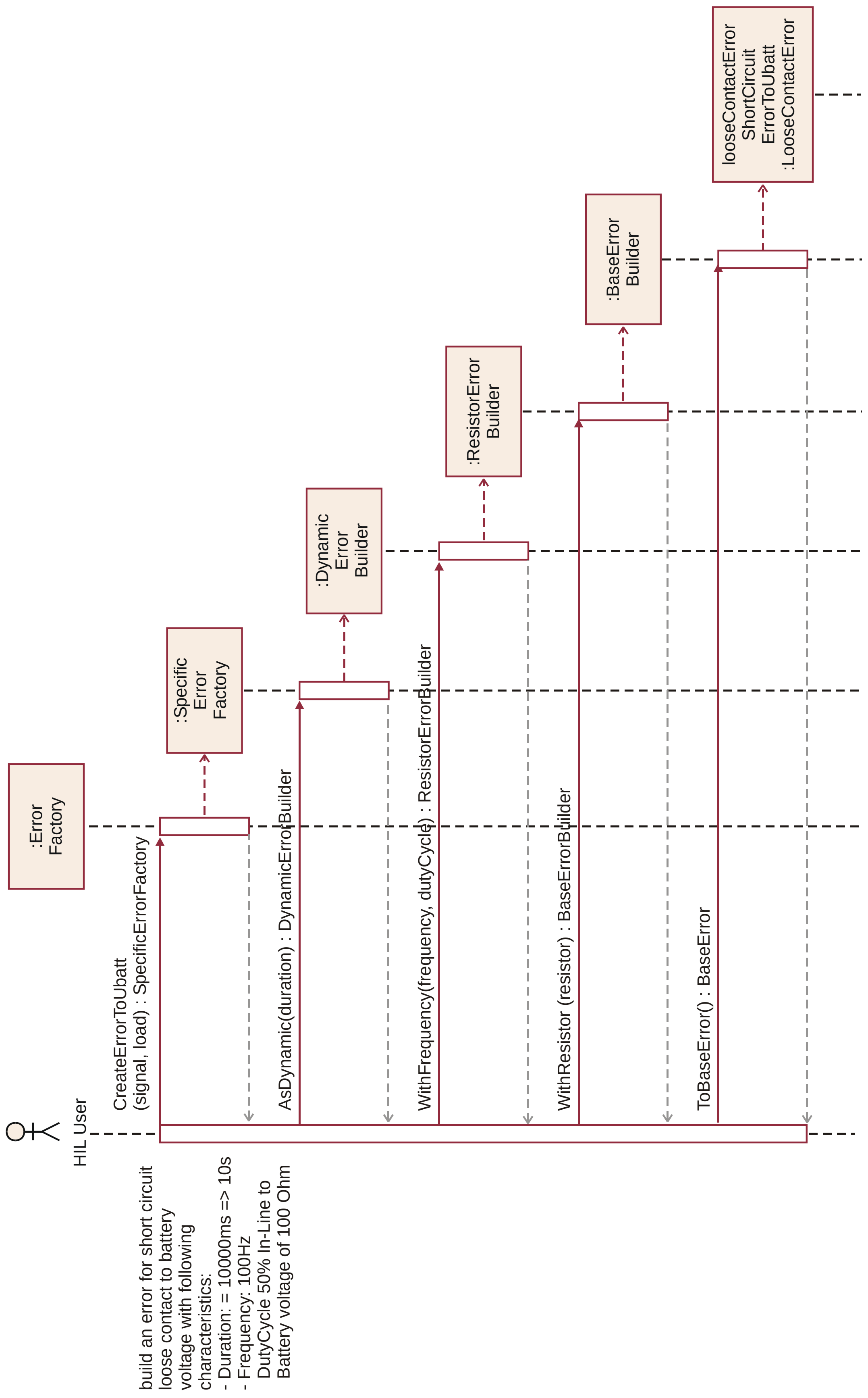

:Error Factory
:Specific Error Factory
:Dynamic Error Builder
:ResistorError Builder
:BaseError Builder
looseContactError ShortCircuit ErrorToUbatt :LooseContactError
HIL User
CreateErrorToUbatt (signal, load) : SpecificErrorFactory
AsDynamic(duration) : DynamicErrorBuilder
WithFrequency(frequency, dutyCycle) : ResistorErrorBuilder
WithResistor (resistor) : BaseErrorBuilder
ToBaseError() : BaseError
build an error for short circuit loose contact to battery voltage with following characteristics:
- Duration: = 10000ms => 10s
- Frequency: 100Hz
DutyCycle 50% In-Line to Battery voltage of 100 Ohm

Figure 100 Sequence to create a loose contact error object

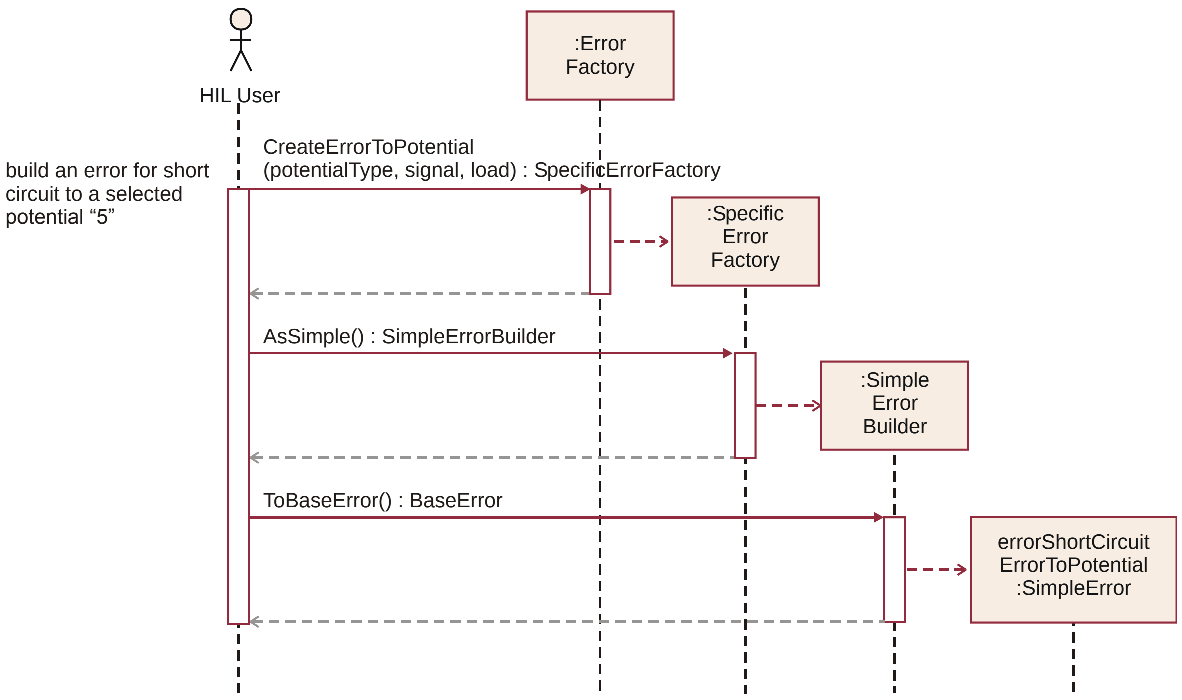

HIL User
:Error Factory
build an error for short circuit to a selected potential “5”
CreateErrorToPotential (potentialType, signal, load) : SpecificErrorFactory
:Specific Error Factory
AsSimple() : SimpleErrorBuilder
:Simple Error Builder
ToBaseError() : BaseError
errorShortCircuit ErrorToPotential :SimpleError

Figure 101 Sequence to create a short-circuit error object

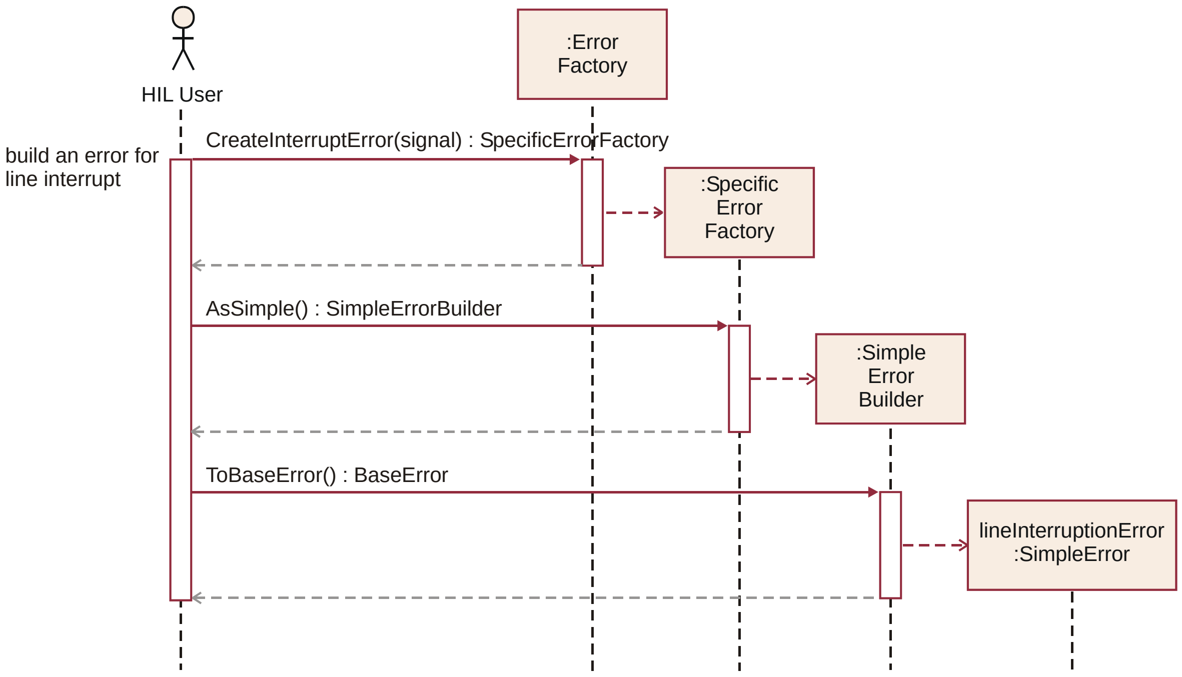

HIL User
:Error Factory
build an error for line interrupt
CreateInterruptError(signal) : SpecificErrorFactory
:Specific Error Factory
AsSimple() : SimpleErrorBuilder
:Simple Error Builder
ToBaseError() : BaseError
lineInterruptionError :SimpleError

Figure 102 Sequence to create a line interruption error object

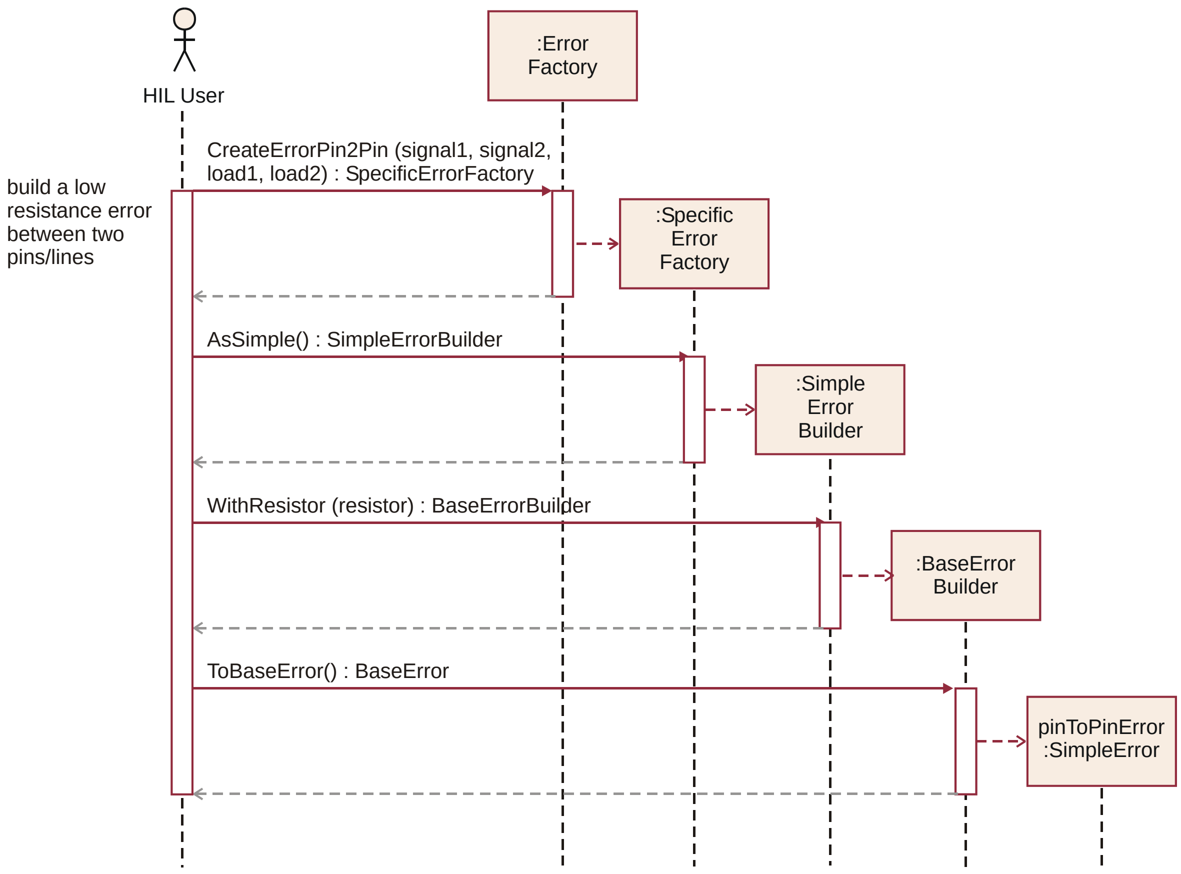

HIL User
:Error Factory
build a low resistance error between two pins/lines
CreateErrorPin2Pin (signal1, signal2, load1, load2) : SpecificErrorFactory
:Specific Error Factory
AsSimple() : SimpleErrorBuilder
:Simple Error Builder
WithResistor (resistor) : BaseErrorBuilder
:BaseError Builder
ToBaseError() : BaseError
pinToPinError :SimpleError

Figure 103 Sequence to create a pin-to-pin error object

These error creation sequences are part of the creation of an error configuration as shown in chapter 7.2.1.

The sample code for this example will be found at

C#	C#/SampleCode/EESPort/EESPortExample.cs
Python	Python/SampleCode/EESPort/EESPortExample.py
Java	JAVA/SampleCode/EESPort/EESPortExample.java

7.2.3 LOADING ERROR CONFIGURATIONS FROM FILE

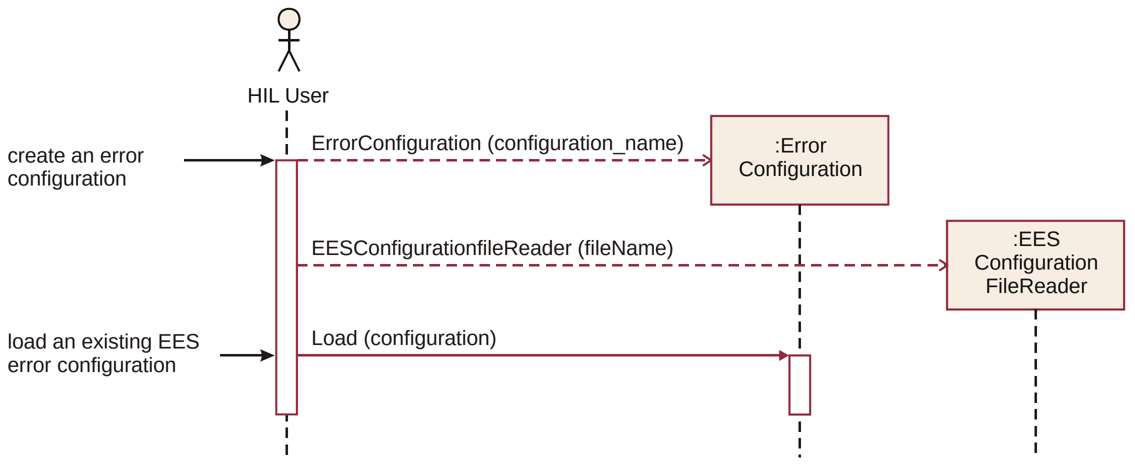

HIL User
create an error configuration
ErrorConfiguration (configuration_name)
:Error Configuration
EESConfigurationfileReader (fileName)
:EES Configuration FileReader
load an existing EES error configuration
Load (configuration)

Figure 104 Sequence of example "load error configuration from file"

In this sequence the error configuration is loaded from a file. The sequence shows how this is done using the error configuration file reader object. Other parts of usage, especially assignment of the error configuration to the EES port, downloading, starting, and stopping the configuration is independent from the creation. Therefore it is in principle the same as in the example above (see chapter 7.2.1).

The sample code for this example will be found at

C#	C#/SampleCode/EESPort/EESPortExample.cs
Python	Python/SampleCode/EESPort/EESPortExample.py
Java	JAVA/SampleCode/EESPort/EESPortExample.java

7.3 SPECIAL HINTS

7.3.1 EES HARDWARE LIMITATIONS AND EXTENSIONS

The EES port handles the EES hardware in an abstract manner. Therefore it is limited to a defined set of functionality. Possibly the hardware supports additional functions that are not supported by the HIL API EES port. These functions are not accessible with means of HIL API. Additional functionality may be used within a test case using additional APIs of the EES implementation. But strict compatibility to HIL API is lost for test cases that make use of such extended functionality, of course.
On the other hand, a specific EES hardware may lack some functionality that is defined by the HIL API. This is also a valid use case. The EES hardware respective port implementation is still HIL API compliant. In this case, the EES port returns an error when not implemented functions are used. HIL API compatible test cases can be executed but return an error due to lack of functionality of the underneath hardware ECU Port.

8 ECU ACCESS

8.1 USER CONCEPT

The ECU access part of the HIL API accesses an ECU via an MC system. It provides the following functionality:

- Measurement and capturing
- Calibration
- Management of ECU memory pages

The HIL API only communicates with the MC system, it does not communicate directly with the ECU. It is the task of the MC system to handle the communication with the ECU and to execute the methods defined by the HIL API. Therefore the HIL API does not need any knowledge about the interface being used for communication with the ECU (e.g. a CAN interface or a KWP2000 interface), and consequently this does not influence the code accessing the HIL API. Figure 105 illustrates this:

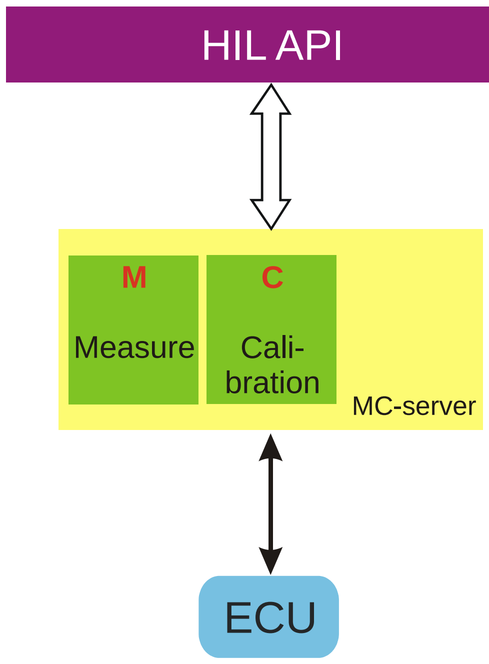


Figure 105 Accessing ECUs via the HIL API

The access to ECUs via the HIL API is done by two separate ECU access ports. Both ports are derived from the class Port (see Figure 106). These ports are:

ECUMPort	The `ECUMPort` class provides functionality for accessing measurement variables of an ECU. It also provides capturing functionality.
ECUCPort	The `ECUCPort` class provides functionality for reading and writing parameters of an ECU. It can also handle memory pages of an ECU.

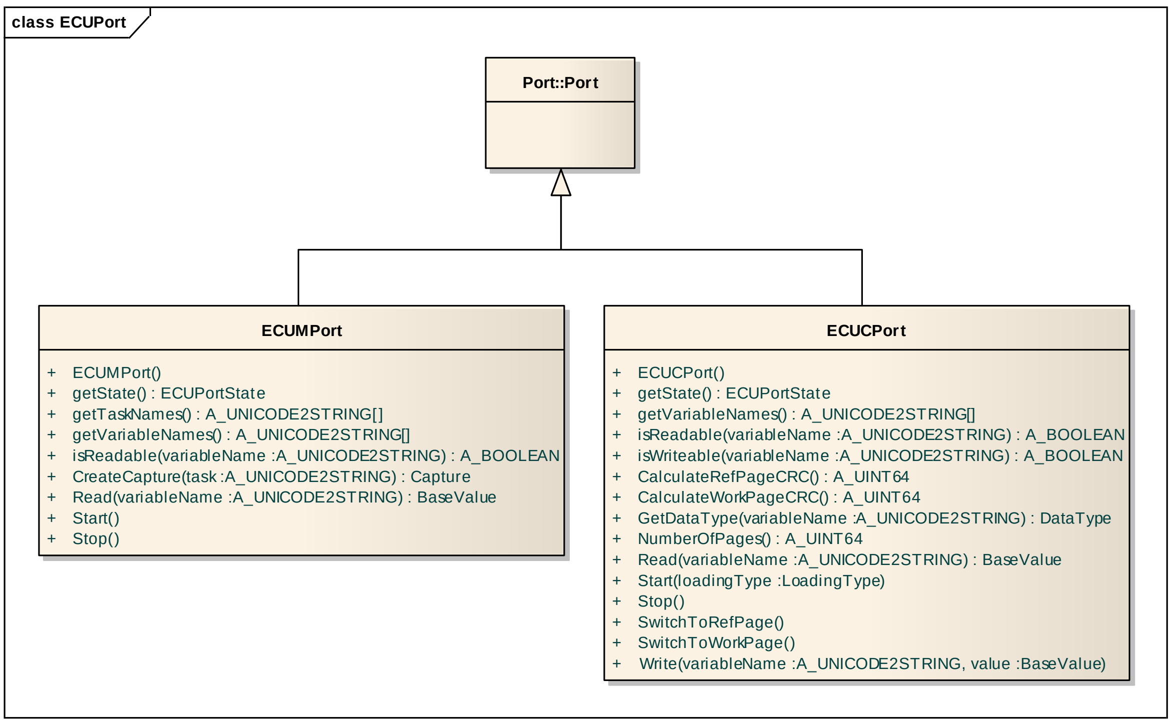

class ECUPort
Port::Port
ECUMPort
+ ECUMPort()
+ getState() : ECUPortState
+ getTaskNames() : A_UNICODE2STRING[]
+ getVariableNames() : A_UNICODE2STRING[]
+ isReadable(variableName :A_UNICODE2STRING) : A_BOOLEAN
+ CreateCapture(task :A_UNICODE2STRING) : Capture
+ Read(variableName :A_UNICODE2STRING) : BaseValue
+ Start()
+ Stop()
ECUCPort
+ ECUCPort()
+ getState() : ECUPortState
+ getVariableNames() : A_UNICODE2STRING[]
+ isReadable(variableName :A_UNICODE2STRING) : A_BOOLEAN
+ isWriteable(variableName :A_UNICODE2STRING) : A_BOOLEAN
+ CalculateRefPageCRC() : A_UINT64
+ CalculateWorkPageCRC() : A_UINT64
+ GetDataType(variableName :A_UNICODE2STRING) : DataType
+ NumberOfPages() : A_UINT64
+ Read(variableName :A_UNICODE2STRING) : BaseValue
+ Start(loadingType :LoadingType)
+ Stop()
+ SwitchToRefPage()
+ SwitchToWorkPage()
+ Write(variableName :A_UNICODE2STRING, value :BaseValue)

Figure 106 The ECU port classes

Both ports use the same state machine. The next chapters describe both ports and the state machines in more detail.

8.2 ECUCPort

The ECUCPort class provides functionality for reading and writing parameters of an ECU. It can also handle memory pages.

8.2.1 States of the ECUCPort

Figure 107 illustrates the state diagram of the ECUCPort.
There are two states, eOFFLINE and eONLINE. After creation, the ECUCPort instance is always in state eOFFLINE. The following table explains the states:

eOFFLINE	There is no connection to the ECU. Reading parameter values returns the values currently being stored in the server only, not coming from the ECU. Writing parameter values changes the values currently being stored on the server only and is not writing values to the ECU.
eONLINE	A connection to the ECU has been established. Reading parameter values returns the current values from the ECU. Writing parameter values changes the current values on the ECU.

The Start(eLoadingType) method switches from the eOFFLINE state to the eONLINE state (see also chapter 8.2.4 for details). The Stop() method switches from eONLINE state back to the eOFFLINE state.

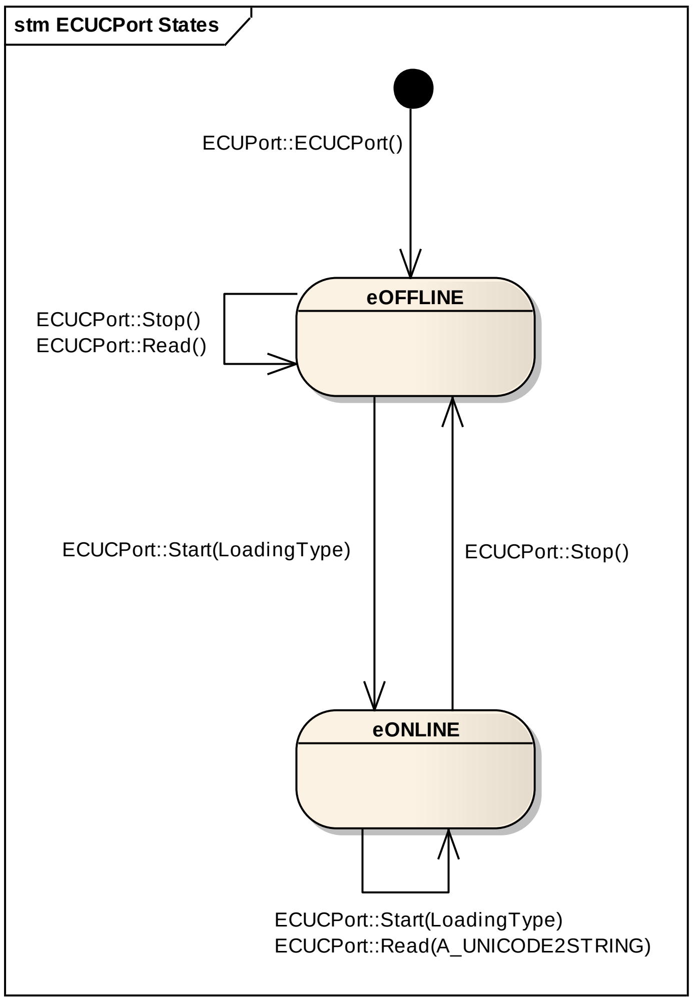


Figure 107 States of the ECU C port

8.2.2 ACCESSING ECU PARAMETERS

The `ECUCPort` allows reading and writing of parameter values of the ECU. If a parameter value is readable can be determined by using the `isReadable` function. If a parameter also can be modified can be determined using the isWritable function.

The following example shows how to read and write a scalar float ECU parameter value (see Figure 108).

First the connection to the ECU is set up using the `Start()` method. The value of the `LoadingType` parameter is not important for this example, any value can be used. Then the data type of the variable is fetched using the `GetDataType()` method. The following assumes that the chosen parameter value is readable and writable.

Then the new parameter value is written to the ECU. After writing, the current value of the parameter on the ECU is read back in order to check if it is the same value as the one which has been written. After that the connection to the ECU is stopped and the ECUCPort goes offline.

The sample code for this example will be found at

C#	C#\ASAM.HILAPI\SampleCode\ECUPort\ ECUCPortExample.cs
Python	Python\SampleCode\ECUPort\ ECUCPortExample.py
Java	JAVA\SampleCode\ECUPort\ ECUCPortExample.java

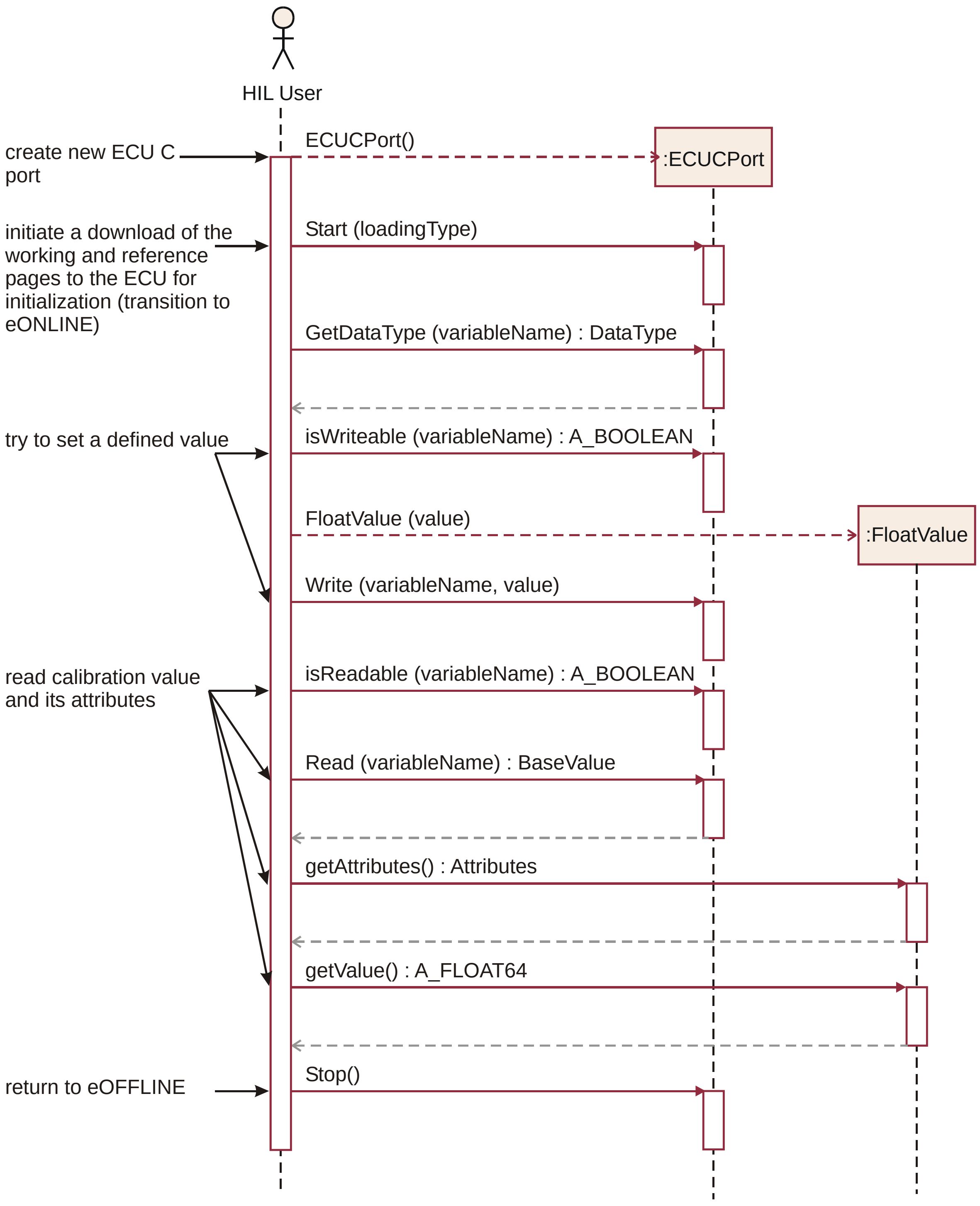

HIL User
create new ECU C port
initiate a download of the working and reference pages to the ECU for initialization (transition to eONLINE)
try to set a defined value
read calibration value and its attributes
return to eOFFLINE
ECUCPort()
:ECUCPort
Start (loadingType)
GetDataType (variableName) : DataType
isWriteable (variableName) : A_BOOLEAN
FloatValue (value)
:FloatValue
Write (variableName, value)
isReadable (variableName) : A_BOOLEAN
Read (variableName) : BaseValue
getAttributes() : Attributes
getValue() : A_FLOAT64
Stop()

Figure 108 Read and write a scalar float ECU parameter

8.2.3 GETTING THE LIST OF VARIABLES OF THE ECUCPORT

This example describes how a user can get the names of all parameters supported by this `ECUCPort` instance. A call of the `getVariableNames()` method returns a list of all parameter names.

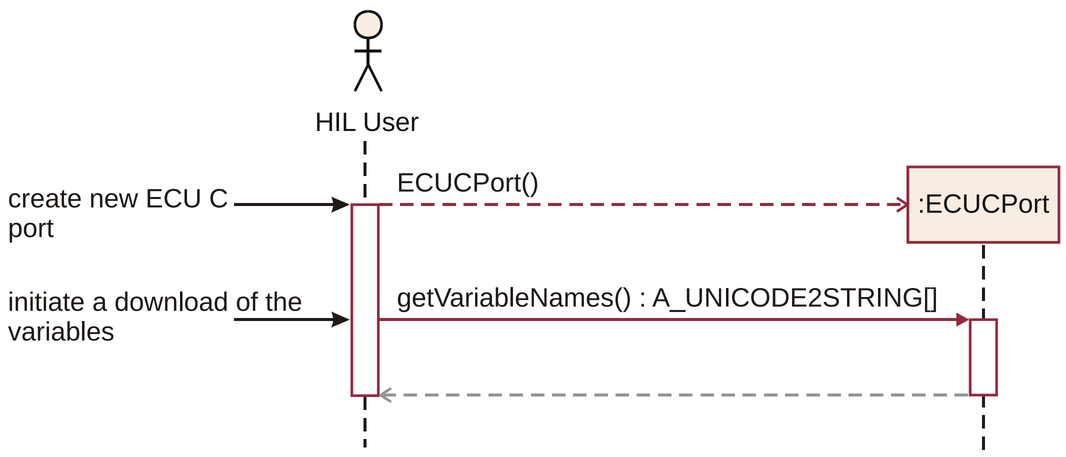

HIL User
create new ECU C port
ECUCPort()
:ECUCPort
initiate a download of the variables
getVariableNames() : A_UNICODE2STRING[]

Figure 109 Get the list of parameters of an ECUCPort instance

The sample code for this example will be found at

C#	C#\ASAM.HILAPI\SampleCode\ECUPort\ ECUCPortExample.cs
Python	Python\SampleCode\ECUPort\ ECUCPortExample.py
Java	JAVA\SampleCode\ECUPort\ ECUCPortExample.java

8.2.4 MANAGE ECU MEMORY PAGES

Another block of functionality of the `ECUCPort` is the management of ECU memory pages.
Most MC tools support the handling of several memory pages. Each of these memory pages is able hold all parameter values of the ECU. Two of the most popular memory pages are the working page and a read-only reference page. The following list gives a short overview of the memory page management functions of the `ECUCPort`:

- The `NumberOfPages()` method returns the number of pages which the current MC system supports. Most MC systems support 2 memory pages, a reference and a working page.
- If more than 1 memory page is supported by the MC system, the `SwitchToRefPage()` method allows to switch from the working page to the reference page, and the `SwitchToWorkPage()` method allows to switch from the reference page to the working page.
- To check if the contents of two memory pages are equal, the `CalculateRefPageCRC()` and `CalculateWorkPageCRC()` methods can be used.
- In order to set up a connection with the ECU, the `Start()` method must be used. The `LoadingType` parameter defines if the content of the current memory page is downloaded to the ECU (`eDOWNLOAD`) or if the content of the actual page is filled with the data coming from the ECU memory (`eUPLOAD`).

Setting a variable value with the `Write` method in state `eOFFLINE` sets the value on the actual page. This value can be written to ECU with the `Start(eDOWNLOAD)` method. Setting the value of the same variable in state `eONLINE` changes the value on the ECU directly.

The following example shows how to handle memory pages.

After creation of the `ECUCPort` the transition to `eONLINE` is performed. Then the number of pages is fetched. In this case a value of 2 is expected, to make sure that a working and a reference page exist. Then the reference page is made the current memory page and the checksum is calculated. Then a parameter value from the ECU is read. After a switch to the working page, the checksum of the working page is calculated. The value of the same parameter is read again – this time coming from the working page. The 2 values of the same variable can differ because they are coming from different memory pages. Then a new value is written to this variable and the checksum is calculated again. Now the value of the parameter on the ECU should be different to the checksum of the same page before. Finally, a call of the `Stop()` method executes a transition to the ECU port state to `eOFFLINE`.

The sample code for this example will be found at

C#	C#\ASAM.HILAPI\SampleCode\ECUPort\ ECUCPortExample.cs
Python	Python\SampleCode\ECUPort\ ECUCPortExample.py
Java	JAVA\SampleCode\ECUPort\ ECUCPortExample.java

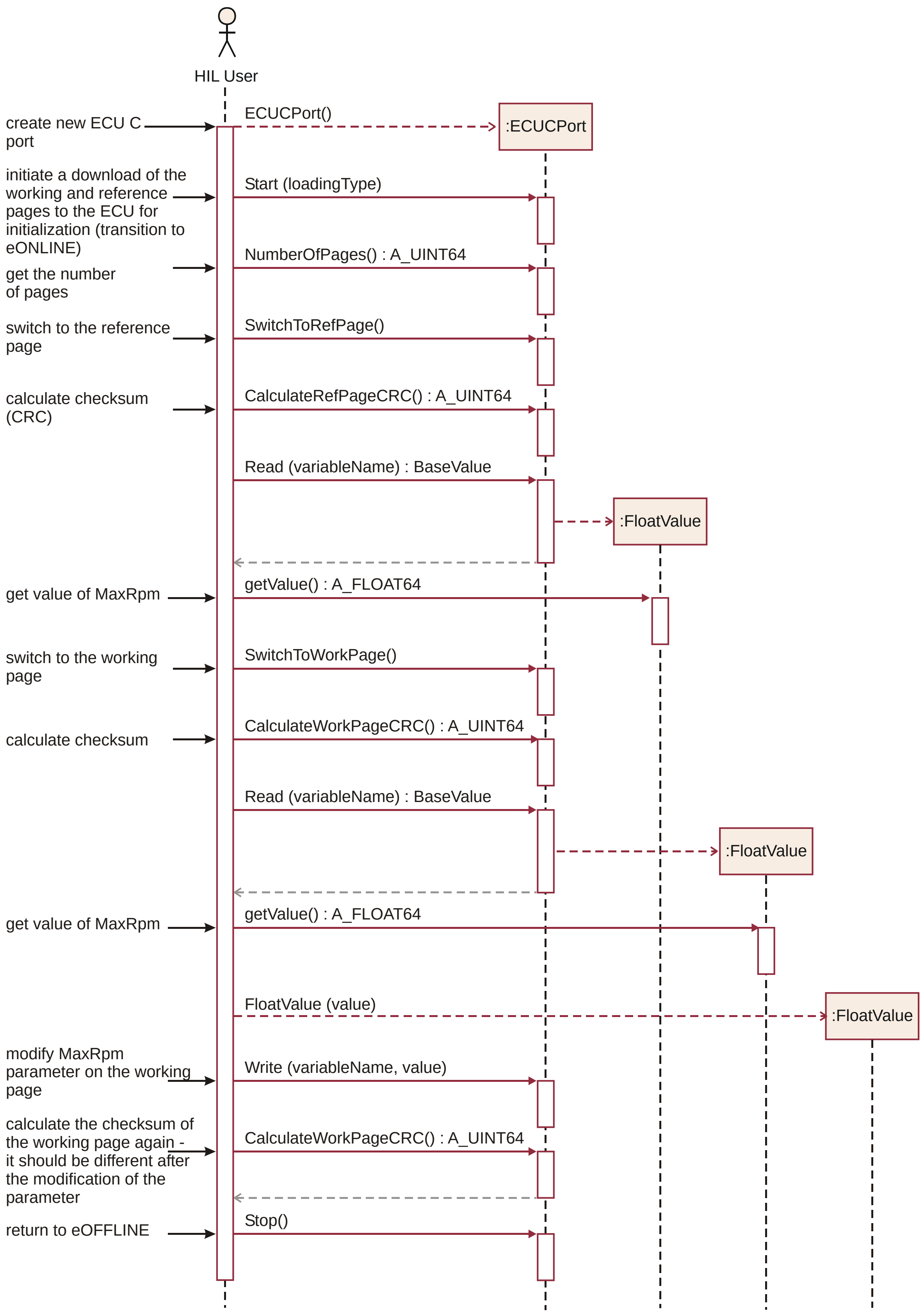

HIL User
create new ECU C port
ECUCPort()
:ECUCPort
initiate a download of the working and reference pages to the ECU for initialization (transition to eONLINE)
Start (loadingType)
get the number of pages
NumberOfPages() : A_UINT64
switch to the reference page
SwitchToRefPage()
calculate checksum (CRC)
CalculateRefPageCRC() : A_UINT64
Read (variableName) : BaseValue
:FloatValue
get value of MaxRpm
getValue() : A_FLOAT64
switch to the working page
SwitchToWorkPage()
calculate checksum
CalculateWorkPageCRC() : A_UINT64
Read (variableName) : BaseValue
:FloatValue
get value of MaxRpm
getValue() : A_FLOAT64
FloatValue (value)
:FloatValue
modify MaxRpm parameter on the working page
Write (variableName, value)
calculate the checksum of the working page again - it should be different after the modification of the parameter
CalculateWorkPageCRC() : A_UINT64
return to eOFFLINE
Stop()

Figure 110 Handling of memory pages

8.3 ECUMPort

The `ECUMPort` is used for measuring and capturing variables of an ECU. The measuring can be done

- As a snapshot using the `Read()` method,
- By measuring of different variables in a specified raster (using `Capture` objects, see chapter 4.7). Use the `CreateCapture()` method to create capture objects.

The `getVariableNames()` method returns a list of all available variable names of the `ECUMPort` instance.
The `getTaskNames()` method returns a list of all available raster names of the `ECUMPort` instance.

8.3.1 States of the ECUMPort

Figure 111 shows the state diagram of the `ECUMPort`. Like the `ECUCPort`, the state machine of the `ECUMPort` consists of two states:

`eOFFLINE`	There is no connection to the ECU. Capturing and reading ECU variable values are not possible. However, it is possible to create capture instances.
`eONLINE`	A connection to the ECU has been established. Reading ECU variable values is possible. Capturing can be started and stopped, without any influence on the ECU M port's state.

The `Start()` method switches from the `eOFFLINE` state to the `eONLINE` state.
The `Stop()` method switches from `eONLINE` state back to the `eOFFLINE` state.

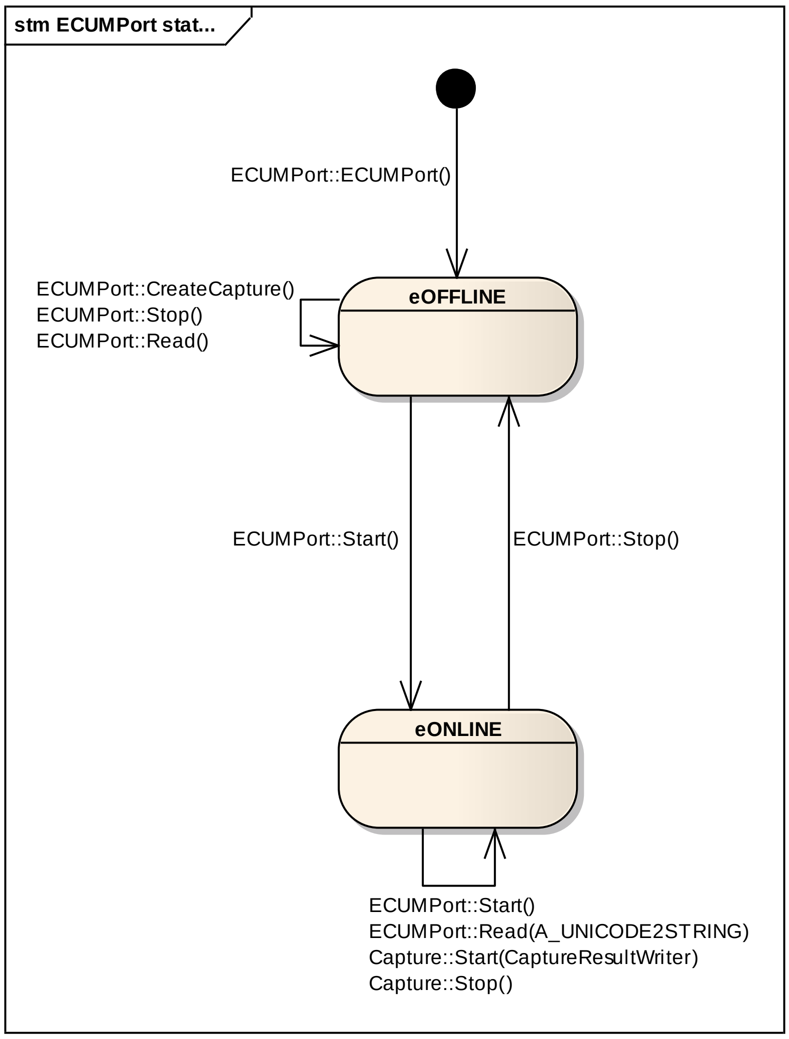


Figure 111 States of the ECU M port

8.3.2 GETTING LISTS OF VARIABLE AND TASK NAMES

The first example of the `ECUMPort` describes how a user can get the names of all variables and all tasks supported by this `ECUMPort` instance.
After the creation of the port the methods `getVariableNames()` and `getTaskNames()` are called.

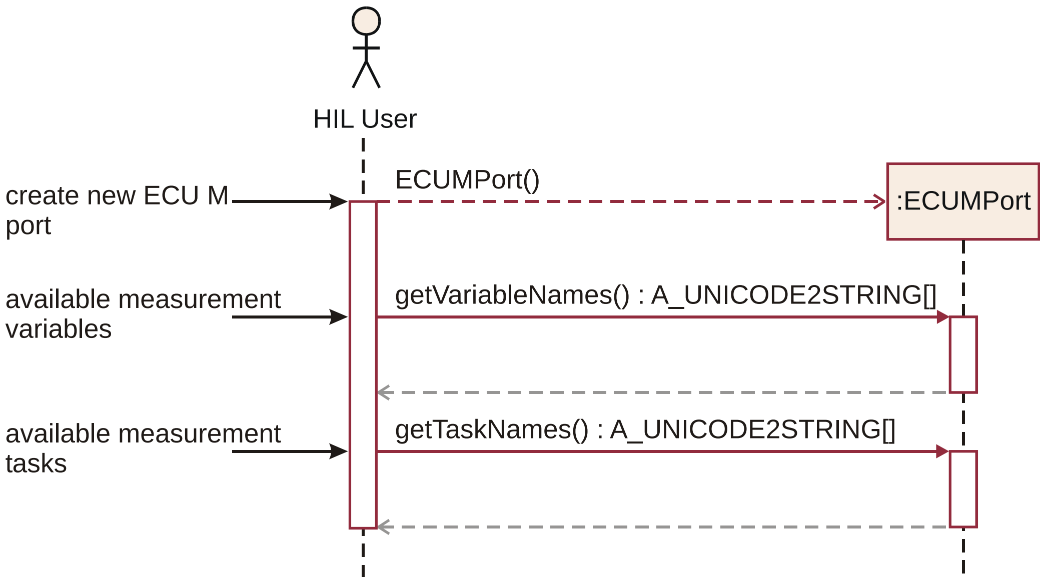

HIL User
create new ECU M port
ECUMPort()
:ECUMPort
available measurement variables
getVariableNames() : A_UNICODE2STRING[]
available measurement tasks
getTaskNames() : A_UNICODE2STRING[]

Figure 112 Get lists of variable and task names

The sample code for this example will be found at

C#	C#\ASAM.HILAPI\SampleCode\ECUPort\ECUMPortExample.cs
Python	Python\SampleCode\ECUPort\ ECUMPortExample.py
Java	JAVA\SampleCode\ECUPort\ ECUMPortExample.java

8.3.3 READ A SCALAR VARIABLE VALUE AND ITS PROPERTIES

This example shows how to read the value of a scalar ECU variable.
The `ECUMPort` instance is switched to the `eONLINE` mode after creation. Then the measure value is fetched using the `Read()` method. After switching back to the `eOFFLINE` mode, the properties (e.g. name, unit, etc.) of the variable value are examined.

The sample code for this example will be found at

C#	C#\ASAM.HILAPI\SampleCode\ECUPort\ECUMPortExample.cs
Python	Python\SampleCode\ECUPort\ ECUMPortExample.py
Java	JAVA\SampleCode\ECUPort\ ECUMPortExample.java

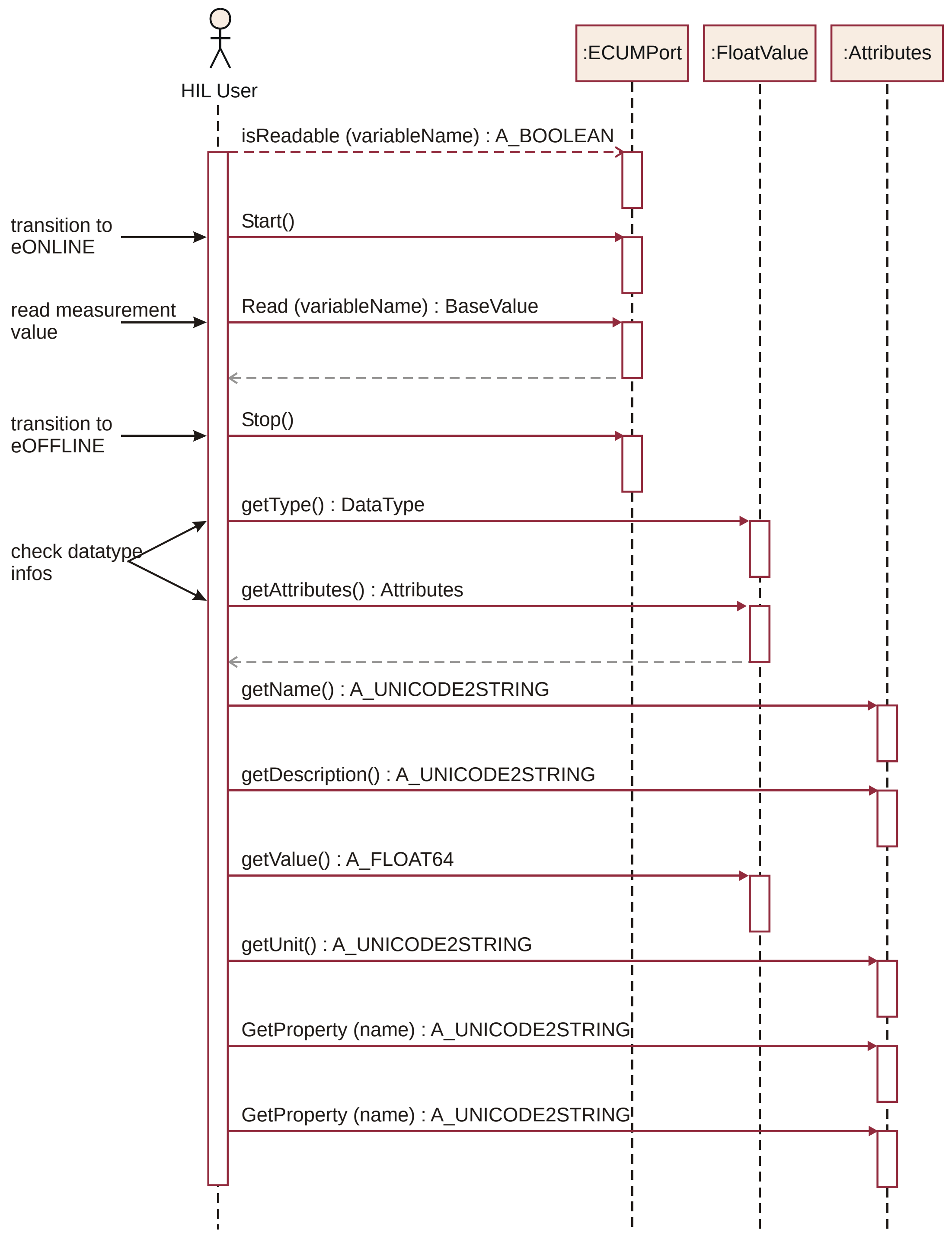

HIL User
:ECUMPort
:FloatValue
:Attributes
isReadable (variableName) : A_BOOLEAN
transition to eONLINE
Start()
read measurement value
Read (variableName) : BaseValue
transition to eOFFLINE
Stop()
check datatype infos
getType() : DataType
getAttributes() : Attributes
getName() : A_UNICODE2STRING
getDescription() : A_UNICODE2STRING
getValue() : A_FLOAT64
getUnit() : A_UNICODE2STRING
GetProperty (name) : A_UNICODE2STRING
GetProperty (name) : A_UNICODE2STRING

Figure 113 Read a scalar variable value and examine its properties

8.3.4 READ AN ARRAY VARIABLE VALUE AND ITS PROPERTIES

Reading an array value is nearly the same as reading a scalar value. The value returned by the `Read()` method now is a `VectorValue` which can be read out index by index.

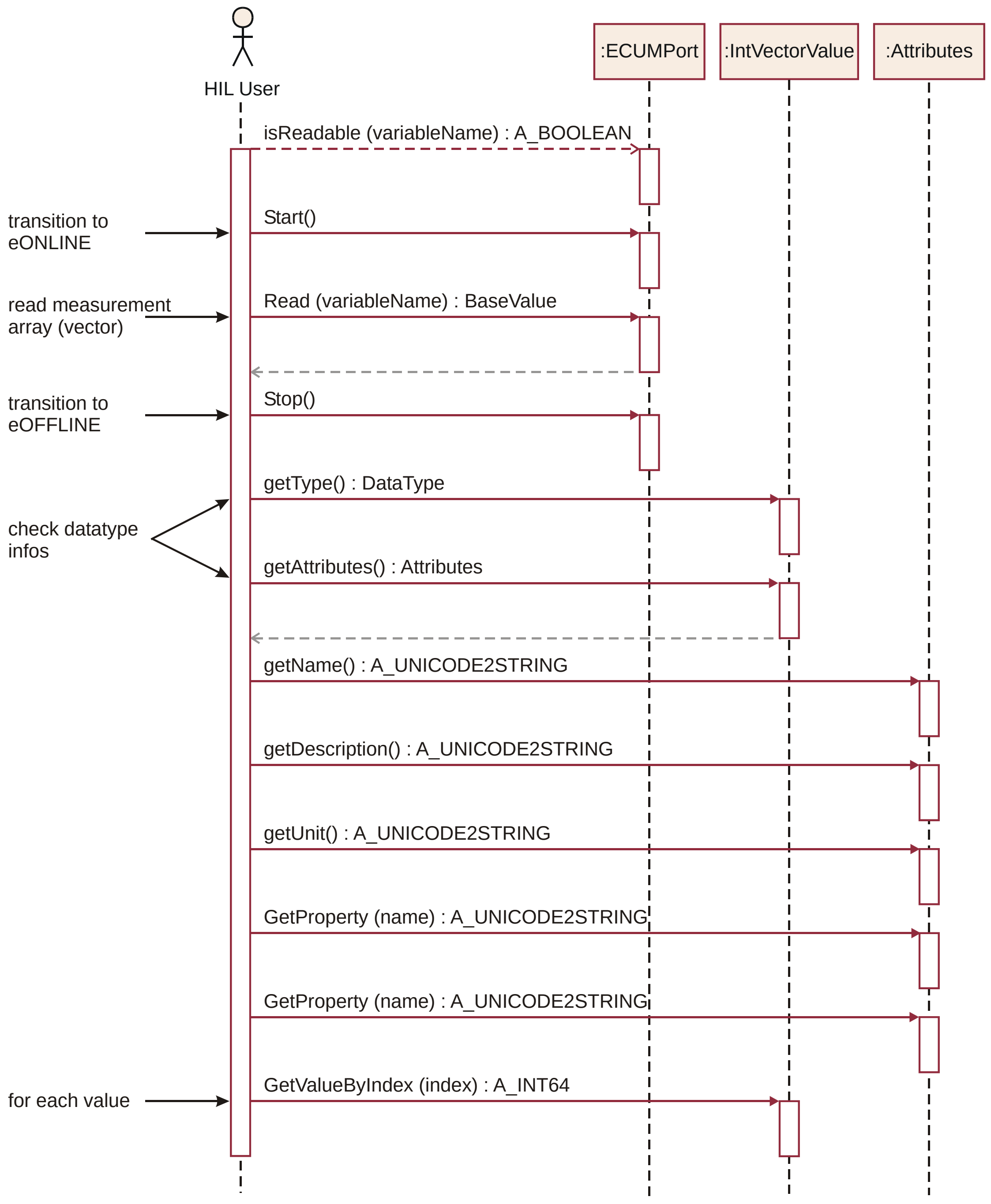

HIL User
:ECUMPort
:IntVectorValue
:Attributes
isReadable (variableName) : A_BOOLEAN
transition to eONLINE
Start()
read measurement array (vector)
Read (variableName) : BaseValue
transition to eOFFLINE
Stop()
check datatype infos
getType() : DataType
getAttributes() : Attributes
getName() : A_UNICODE2STRING
getDescription() : A_UNICODE2STRING
getUnit() : A_UNICODE2STRING
GetProperty (name) : A_UNICODE2STRING
GetProperty (name) : A_UNICODE2STRING
for each value
GetValueByIndex (index) : A_INT64

Figure 114 Read an array variable value and examine its properties

The sample code for this example will be found at

C#	C#\ASAM.HILAPI\SampleCode\ECUPort\ECUMPortExample.cs
Python	Python\SampleCode\ECUPort\ ECUMPortExample.py
Java	JAVA\SampleCode\ECUPort\ ECUMPortExample.java

8.3.5 READ A MATRIX VARIABLE VALUE AND ITS PROPERTIES

Reading a matrix value is nearly the same as reading an array value. The value returned by the `Read()` method now is a `MatrixValue` which can be read out using row and column indices.

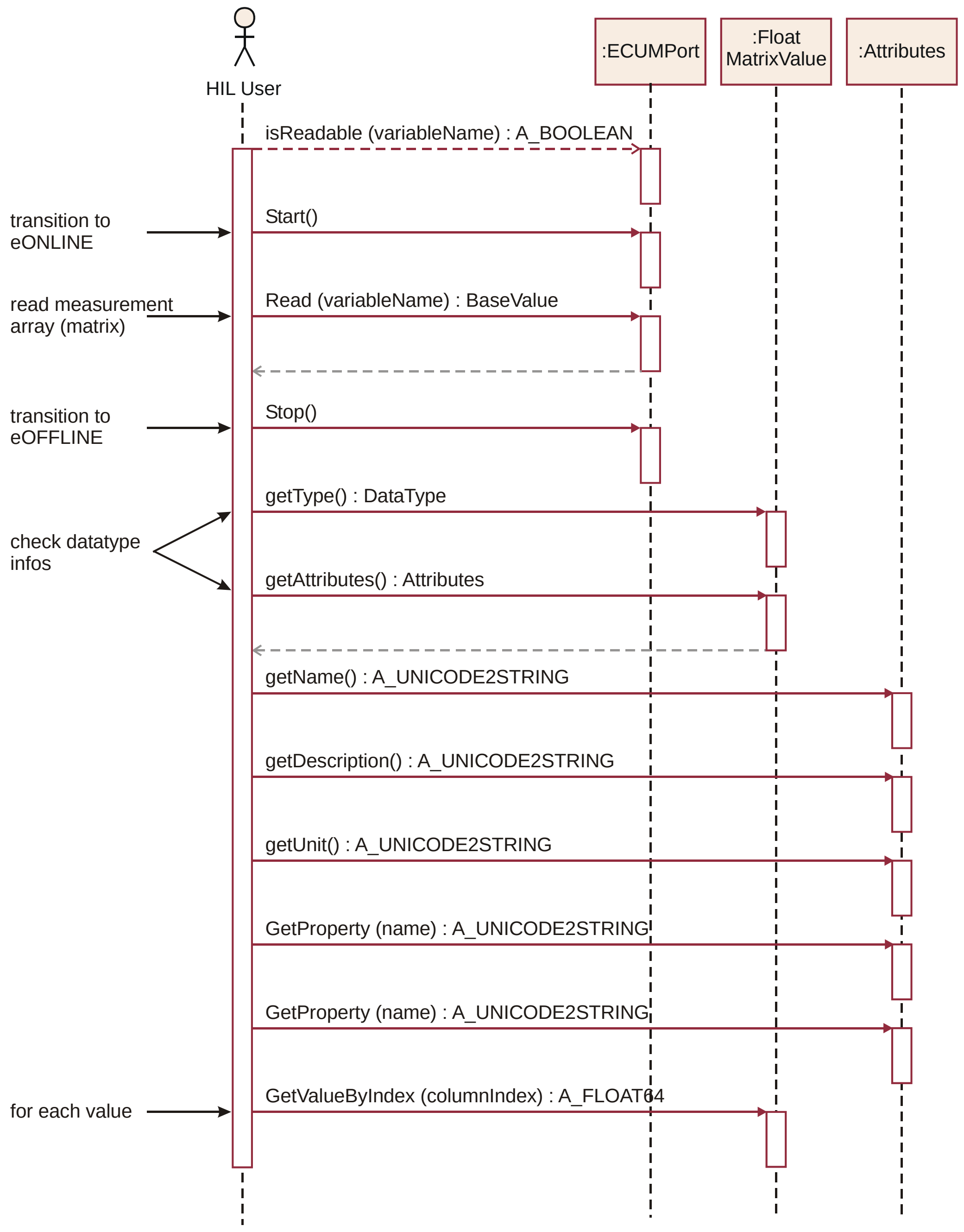

HIL User
:ECUMPort
:Float MatrixValue
:Attributes
isReadable (variableName) : A_BOOLEAN
transition to eONLINE
Start()
read measurement array (matrix)
Read (variableName) : BaseValue
transition to eOFFLINE
Stop()
check datatype infos
getType() : DataType
getAttributes() : Attributes
getName() : A_UNICODE2STRING
getDescription() : A_UNICODE2STRING
getUnit() : A_UNICODE2STRING
GetProperty (name) : A_UNICODE2STRING
GetProperty (name) : A_UNICODE2STRING
for each value
GetValueByIndex (columnIndex) : A_FLOAT64

Figure 115 Read a matrix variable value and examine its properties

The sample code for this example will be found at

C#	C#\ASAM.HILAPI\SampleCode\ECUPort\ECUMPortExample.cs
Python	Python\SampleCode\ECUPort\ ECUMPortExample.py
Java	JAVA\SampleCode\ECUPort\ ECUMPortExample.java

8.3.6 Capturing ECU variables

For reading more than one variable value, including its timestamps, a capture object must be used. The following example illustrates this.

A capture object always does the data acquisition using a specified raster. This raster is defined at creation time of the capture object using the `CreateCapture()` method. The subsequent call to `setVariables()` function defines which ECU variables shall be captured by this capture instance.

Capturing can only be done in the `eONLINE` state, which is initiated by a call of the `Start()` method. Now the capturing can be started and stopped by the `Capture.Start()` and `Capture.Stop()` methods. When capturing has been finished, the `ECUMPort` instance can be switched back to the `eOFFLINE` state. A call to the `Capture.getCaptureResult()` function returns all captured variable values.

See chapter 4.9 for more information about the `Capture` and `CaptureResult` classes.

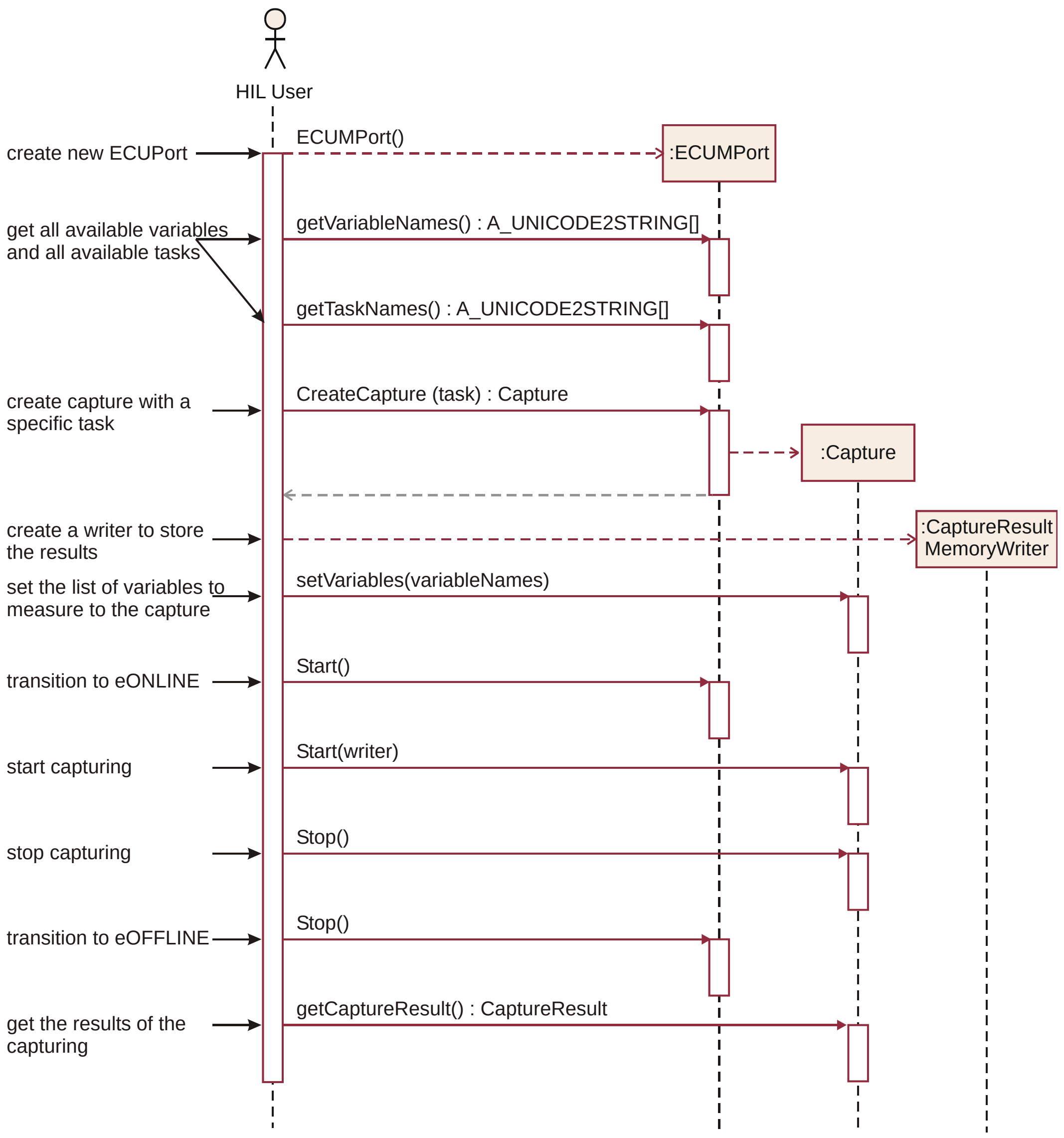

HIL User
create new ECUPort
ECUMPort()
:ECUMPort
get all available variables and all available tasks
getVariableNames() : A_UNICODE2STRING[]
getTaskNames() : A_UNICODE2STRING[]
create capture with a specific task
CreateCapture (task) : Capture
:Capture
create a writer to store the results
:CaptureResult MemoryWriter
set the list of variables to measure to the capture
setVariables(variableNames)
transition to eONLINE
Start()
start capturing
Start(writer)
stop capturing
Stop()
transition to eOFFLINE
Stop()
get the results of the capturing
getCaptureResult() : CaptureResult

Figure 116 Capturing ECU variables

The sample code for this example will be found at

C#	C#\ASAM.HILAPI\SampleCode\ECUPort\ECUMPortExample.cs
Python	Python\SampleCode\ECUPort\ ECUMPortExample.py
Java	JAVA\SampleCode\ECUPort\ ECUMPortExample.java

Directory of Figures

Directory of Tables

Books

[ASAM Expression]	ASAM AE Expression "General Expression Syntax", Version 1.0.0 (Draft of 23.07.2009)
[ASAM Data Types]	ASAM AIS "Abstract data type definition", Version 2.0.0
[ASAM MCD-2 MC]	ASAM MCD-2 MC "Measurement and Calibration Data Specification", Version 1.6.0
[ASAM MDF]	ASAM COMMON MDF "Measurement Data Format, Programmers Guide", Version 4.0.0
[ASAM MCD-2 NET]	ASAM MCD-2 NET "FIBEX - Field Bus Exchange Format", Version 3.0.0
[ASAM MCD-3]	ASAM MCD-3 "Application Programming Interface Specification", Version 2.2.0
[HIL C# Reference]	ASAM AE HIL "C# API Technology Reference", Version 1.0.0
[HIL Java Reference]	ASAM AE HIL "Java API Technology Reference", Version 1.0.0
[HIL Python Reference]	ASAM AE HIL "Python API Technology Reference", Version 1.0.0

ASAM e.V.

Arnikastrasse 2

D-85635 Höhenkirchen

Germany

Tel.:	(+49) 08102 / 8953 17
Fax.:	(+49) 08102 / 8953 10
E-mail:	info@asam.net
Internet:	www.asam.net