dSPACE HIL API MAPort Python– dSPACE XIL API MAPort

dSPACE HIL API MAPort Python Migration Guide

How to migrate dSPACE HIL API MAPort Python scripts to dSPACE XIL API MAPort .NET using Python for .NET

November 2016

Embedded Success dSPACE

How to Contact dSPACE

Mail:	dSPACE GmbH Rathenaustraße 26 33102 Paderborn Germany
Tel.:	+49 5251 1638-0
Fax:	+49 5251 16198-0
E-mail:	info@dspace.de
Web:	http://www.dspace.com

How to Contact dSPACE Support

To contact dSPACE if you have problems and questions, fill out the support request form provided on the website at http://www.dspace.com/go/supportrequest.
The request form helps the support team handle your difficulties quickly and efficiently.
In urgent cases contact dSPACE via phone: +49 5251 1638-941 (General Technical Support)

Software Updates and Patches

dSPACE strongly recommends that you download and install the most recent patches for your current dSPACE installation. Visit http://www.dspace.com/go/support for software updates and patches.

Important Notice

Contents

1 Introduction

Using the dSPACE XIL API .NET server in the Python scripting language is the method dSPACE recommends for migrating existing Python scripts that contain HIL API Python commands.

The dSPACE XIL API .NET implementation is based on the ASAM AE XIL 2.0.1 standard. It is the successor of the ASAM AE HIL 1.0.2 standard and comes with some enhancements and modifications. The main approach of this standard is decoupling test automation software and test hardware to let you reuse of test cases for different hardware systems.

To understand the concepts of the ASAM AE XIL standard, you are recommended to read the ASAM user documentation, which is included in the dSPACE XIL API .NET installation, before applying the dSPACE-specific API implementation.

The dSPACE .NET implementation of the ASAM XIL API Vs. 2.0.1 (dSPACE XIL API .NET server for short) currently covers large parts of the Model Access Port (MAPort) for access to real-time applications on dSPACE HIL, RCP and VEOS platforms; larger parts of the common namespace of the XIL API (to be found in `ASAM.XIL.Interfaces.Testbench.Common`), in which shared classes, data structures, and data types are defined for the XIL API Ports (especially the classes for data acquisition and real-time stimulation); and the Electrical Error Simulation Port (EESPort) for fault simulation on SCALEXIO systems.

This migration guide covers only the MAPort.

For a number of years, Python has supported the use of .NET assemblies directly in the Python scripting language via the Python package 'Python for .NET'. Python for .NET gives Python programmers nearly seamless integration with the .NET Common Language Runtime (CLR) and provides a powerful application scripting tool for .NET developers. When using this package, you can script .NET applications or build entire applications in Python by using .NET services and components written in any language that targets the CLR (Managed C++, C#, VB, JScript). Python for .NET is part of the dSPACE Python distribution.

2 Required Knowledge

You do not need previous knowledge of the Microsoft .NET framework or any of the .NET programming languages, such as C# or VB.NET.
However, you should have a basic knowledge of developing programs in the Python scripting language.

The following sections show concrete examples and code snippets that explain step by step how to migrate existing HIL API MAPort Python scripts to XIL API .NET.

First, the guide focuses on migrating of HIL API Python commands to classes/methods of the XIL API. The last sections of this guide provide additional information on exception handling, using namespaces, and how to use generic collections of the Microsoft .NET Framework.

3 Prerequisites/Preparatory Tasks

The following sections require that

1. You own a copy of the Platform API Package licenses
2. The dSPACE XIL API .NET server is installed (the easiest way to install all of your licensed software is to execute the dSPACE_MasterSetup.exe file, located in the root folder of your dSPACE Release DVD)
3. The dSPACE processor board was registered either with ControlDesk 4.x or later, or with the dSPACE Platform Management API
4. The real-time application that is used was generated with RTI from at least dSPACE Release 7.3. This is required so that the new measurement and calibration service is available in the real-time application.

4 Quick Overview

This introduction gives you a brief overview of the changes required for your script. For a more detailed description, refer to the following sections.

- Replace the HIL API Python import directive with calls to `clr.AddReference()` (a command of the Python for .NET package), by using the required ASAM XIL API .NET assemblies.
- Add references to the generic collections of the .NET framework.
- Replace the native Python collections, e. g., dictionaries and lists, with the corresponding generic collection from the .NET framework. You have to change only the collections directly used with XIL API .NET.
- Create a TestbenchFactory and a vendor-specific Testbench.
- Replace all HIL API Python constructor calls with the equivalent factory class call from the XIL API .NET.
- Replace the code for HIL API MAPort creation using a configuration dictionary with code that uses the new XIL API concept for MAPort creation using a port configuration file.

5 Migration Examples

This section uses specific examples to show you step by step how to convert HIL API Python commands into the classes and method calls of dSPACE XIL API .NET.
In each case, a short section of the HIL API Python code is compared to an equivalent section that uses XIL API .NET.

5.1 Importing the XIL API .NET Assemblies

For any .NET services and components, you have to import or reference the required assembly into the Python workspace. This requires the Python for .NET command `clr.AddReference()`.

```
import clr

# add references to the XIL API .NET assemblies
clr.AddReference("ASAM.XIL.Implementation.TestbenchFactory, Version=2.0.1.0, Cultu
re=neutral, PublicKeyToken=fc9d65855b27d387")
clr.AddReference("ASAM.XIL.Interfaces, Version=2.0.1.0, Culture=neutral, PublicKey
Token=bf471dff114ae984")
```

For further information, refer to section 6.1 Referencing the XIL API .NET assemblies.

5.2 Using the .NET Framework

XIL API .NET expects .NET data types. You therefore have to use them explicitly. The following example shows a .NET dictionary:

```
from System.Collections.Generic import Dictionary

netDict = Dictionary[str, str]()
netDict.Add("Key", "Value")
```

It is impossible to use Python data structures for XIL API .NET. For further information, refer to section 6.3 Generic Collections from the .NET Framework.

5.3 Creating dSPACE XIL API TestbenchFactory and Testbench

XIL API .NET uses the factory pattern for creating all kinds of Testbench-specific objects.
At first, you need to create a `TestbenchFactory` and a `VendorSpecificTestbench`:

```
testbenchFactory = TestbenchFactory()
testbench = testbenchFactory.CreateVendorSpecificTestbench("dSPACE GmbH", "XIL API
", "2015-B");
```

In the next step, you have to create an instance of the MAPort class to access the values of the real-time application. To configure an MAPort, the following information is required:

- The path and file name of the variable description belonging to the real-time application to be used
- The platform identifier as it appears in ControlDesk 4.x or later after successful registration

5.4 Creating and Configuring MAPorts

In the XIL API, the configuration dictionary of the HIL API MAPort was replaced by a port configuration file. The configuration contained in this file is loaded via the LoadConfiguration method of the MAPort and is applied to the MAPort via the Configure method.

HIL API Python code fragment:

```
configurationDict = {}
configurationDict["PlatformIdentifier"] = "ds1005"
configurationDict["ApplicationPath"] = r"C:\Path\To\MyApplication.sdf"
maPort = ASAM.HILAPI.dSPACE.MAPort.MAPort(configDictionary)
```

Corresponding XIL API .NET code fragment:

```
maPortConfigFilePath = r"..\MAPortConfiguration.xml"
maPort = testbench.MAPortFactory.CreateMAPort("DemoMAPort")
maPortConfig = maPort.LoadConfiguration(maPortConfigFilePath)
maPort.Configure(maPortConfig, False)
```

5.4.1 Example of a dSPACE XIL API MAPort Configuration File

With the following configuration file, you can access the real-time application that is specified in the variable description file (SDF file) running on a modular system based on DS1005.

In the `PlatformName` key, you have to enter the platform name that you specified when you registered the platform.

```
<?xml version="1.0" encoding="utf-8"?>
<PortConfigurations>
  <MAPortConfig>
    <PlatformName>ds1005</PlatformName>
    <SystemDescriptionFile>C:\Path\To\MyApplication.sdf</SystemDescriptionFile>
  </MAPortConfig>
</PortConfigurations>
```

For details, refer to dSPACE XIL API Implementation Guide > Implementing an MAPort Application > Basics on the MAPort > Configuring the MAPort.

5.5 Reading and Writing Values

Reading and writing values using an MAPort instance is similar in HIL API and XIL API. The only difference is the construction of a value container object used to specify the value to be written: while in HIL API a constructor was used, in XIL API a factory function of the Testbench's ValueFactory must be called instead.

HIL API Python code fragment:

```
turnSignalLever = "Model Root/TurnSignalLever[-1..1]/Value";
readValue = maPort.Read(turnSignalLever);
maPort.Write(turnSignalLever, FloatValue(1.0));
```

Corresponding XIL API .NET code fragment:

```
turnSignalLever = "Model Root/TurnSignalLever[-1..1]/Value"
readValue = maPort.Read(turnSignalLever);
maPort.Write(turnSignalLever, testbench.ValueFactory.CreateFloatValue(1.0));
```

5.6 Measuring Variables

When you measure variables, the `Start` method of the Capture instance requires a capture result writer object to be passed as an argument. With the XIL API you have to replace the constructor of the capture result writer with a factory method of the capturing factory.

The following example shows the creation of a `CaptureResultMemoryWriter` object using HIL API Python:

```
taskName = "HostService"
batteryVoltage = "Model Root/BatteryVoltage[V]/Value"

capture = maPort.CreateCapture(taskName)
capture.Variables = [batteryVoltage]
```

```
captureResultWriter = CaptureResultMemoryWriter()
capture.Start(captureResultWriter)
time.sleep(1)
capture.Stop()
```

Corresponding XIL API .NET code fragment:

```
taskName = "HostService"
batteryVoltage = "Model Root/BatteryVoltage[V]/Value"

capture = maPort.CreateCapture(taskName)
capture.Variables = Array[str]([batteryVoltage])
captureResultWriter = testbench.CapturingFactory.CreateCaptureResultMemoryWriter()
capture.Start(captureResultWriter)
time.sleep(1)
capture.Stop()
```

5.7 Triggered Measurements

You have to create the Watcher objects used for triggering a measurement by using a WatcherFactory instead of constructors for the ConditionWatcher and the DurationWatcher objects.

HIL API Python code fragment:

```
defines = StringNamedCollection()
defines.Add("Lever", turnSignalLever)

conditionWatcher = ConditionWatcher()
conditionWatcher.Defines = defines
conditionWatcher.Condition = "posedge(Lever, 0.5)"

durationWatcher = DurationWatcher(8.0)

capture.SetStartTriggerCondition(conditionWatcher, 0.0)
capture.SetStopTriggerCondition(durationWatcher, 0.0)
```

Corresponding XIL API .NET code fragment:

```
defines = Dictionary[str,str]()
defines.Add("Lever", turnSignalLever)

conditionWatcher = testbench.WatcherFactory.CreateConditionWatcher("posedge(Lever,
 0.5)", defines)
durationWatcher = testbench.WatcherFactory.CreateDurationWatcher(8.0)

capture.SetStartTriggerCondition(conditionWatcher, 0.0)
capture.SetStopTriggerCondition(durationWatcher, 0.0)
```

5.8 Creating Stimulus Signals

To create the signals and symbols of the stimulus domain of the XIL API, use the `SignalFactory` and `SymbolFactory` classes instead of constructors.

HIL API Python code fragment:

```
leverSegment1 = ConstSegment()
leverSegment1.Duration = ConstSymbol(4)
leverSegment1.Value = ConstSymbol(0)

batterySegment = RampSegment()
batterySegment.Duration = ConstSymbol(18)
batterySegment.Start = ConstSymbol(4.0)
batterySegment.Stop = ConstSymbol(13.5)

signalLever = SegmentSignalDescription()
signalLever.Name = "Signal_For_Lever"
signalLever.Add(leverSegment1)

signalBattery = SegmentSignalDescription()
signalBattery.Name = "Signal_For_Battery"
signalBattery.Add(batterySegment)

signalDescriptionSet = SignalDescriptionSet()
signalDescriptionSet.Add(signalLever)
signalDescriptionSet.Add(signalBattery)
```

Corresponding XIL API .NET code fragment:

```
leverSegment1 = testbench.SignalFactory.CreateConstSegment()
leverSegment1.Duration = testbench.SymbolFactory.CreateConstSymbolByValue(4)
leverSegment1.Value = testbench.SymbolFactory.CreateConstSymbolByValue(0)

batterySegment = testbench.SignalFactory.CreateRampSegment()
batterySegment.Duration = testbench.SymbolFactory.CreateConstSymbolByValue(18)
batterySegment.Start = testbench.SymbolFactory.CreateConstSymbolByValue(4.0)
batterySegment.Stop = testbench.SymbolFactory.CreateConstSymbolByValue(13.5)

signalLever = testbench.SignalFactory.CreateSegmentSignalDescriptionByName("Signal
_For_Lever")
signalLever.Add(leverSegment1)

signalBattery = testbench.SignalFactory.CreateSegmentSignalDescriptionByName("Sign
al_For_Battery")
signalBattery.Add(batterySegment)

signalDescriptionSet = testbench.SignalFactory.CreateSignalDescriptionSet();
signalDescriptionSet.Add(signalLever)
signalDescriptionSet.Add(signalBattery)
```

5.9 Generating Stimulus Signals

All classes from the `Common.SignalGenerator` namespace are constructed via methods of the `SignalGeneratorFactory` class in the XIL API.

HIL API Python code fragment:

```
signalGenerator = ASAM.HILAPI.dSPACE.MAPort.Stimulus.SignalGenerator()
stzReader = SignalGeneratorSTZReader(r"..\DemoStimulus.stz")
```

```
stzReader.Load(signalGenerator)
```

Corresponding XIL API .NET code fragment:

```
signalGenerator = None
stzReader = testbench.SignalGeneratorFactory.CreateSignalGeneratorSTZReaderByFileN
ame(r"..\DemoStimulus.stz")
signalGenerator = stzReader.Load(signalGenerator)
```

6 Additional Information on Using dSPACE XIL API .NET in Python

In the previous section, source code fragments were used to compare how sections of a typical HIL API Python script can be ported to XIL API .NET. Some details were omitted to keep a clear view of the essential points.

The following sections describe a few points in greater detail; points you must consider when using the dSPACE XIL API .NET server.

6.1 Referencing the XIL API .NET assemblies

.NET assemblies must be loaded to Python before you can use them. This requires using the Python for .NET command `clr.AddReference()`.

To reference the assemblies, you can use the available namespaces. These contain the classes and methods of the .NET assemblies for use in the Python source code.

The following .NET assemblies are needed for using the dSPACE XIL API .NET server:

- `ASAM.XIL.Interfaces.dll`
- `ASAM.XIL.Implementation.TestbenchFactory.dll`

For the above assemblies, the relevant Python code is:

```
# add references to the XIL API .NET assemblies
clr.AddReference("ASAM.XIL.Implementation.TestbenchFactory, Version=2.0.1.0, Cultu
re=neutral, PublicKeyToken=fc9d65855b27d387")
clr.AddReference("ASAM.XIL.Interfaces, Version=2.0.1.0, Culture=neutral, PublicKey
Token=bf471dff114ae984")
```

Because the XIL API .NET assemblies are in the Global Assembly Cache (GAC), you do not have to specify path information.
To determine the fully qualified assembly name, you can use Windows PowerShell with the following command:

```
[System.Reflection.AssemblyName]::GetAssemblyName("C:\Path\To\MyAssembly.dll").FullName
```

6.2 Namespaces

In .NET, using namespaces is common. All classes of .NET frameworks are organized like this. The XIL API .NET is also divided into namespaces.

Example:
In XIL API .NET, the `TestbenchFactory` class is located in the namespace `ASAM.XIL.Implementation.TestbenchFactory.Testbench`.

In Python, you can create a new instance of the `TestbenchFactory` class as follows:

```
# create TestbenchFactory
testbenchFactory = ASAM.XIL.Implementation.TestbenchFactory.Testbench.TestbenchFac
tory()
```

You can use the import directive in Python so you do not have to write the complete namespace each time you create a new instance:

```
from ASAM.XIL.Implementation.TestbenchFactory.Testbench import TestbenchFactory
```

This makes the code much more compact:

```
# create TestbenchFactory
testbenchFactory = TestbenchFactory()
```

6.3 Generic Collections from the .NET Framework

Using generic collections from the .NET framework requires a special approach in Python. This section shows you how to create such a dictionary, which is required for creating a XIL API `ConditionWatcher`.

The dictionary must have keys of the data type string; the values that are assigned to each key must also have the data type string.

For any class that belongs to the .NET Framework, the corresponding namespace needs to be referenced first before it can be used:

```
from System.Collections.Generic import Dictionary

netDict = Dictionary[str, str]()
netDict.Add("Key", "Value")
```

A similar procedure applies to .NET lists which are, for example, required for the `Capture.Variables` property:

```
from System import Array

netList = Array[str](["String1", "String2"])
```

6.4 Exception Handling

Any errors during the execution of a XIL API .NET method are shown by .NET exceptions. Therefore, you are strongly recommended to use comprehensive exception handling. The exceptions defined by ASAM are located in the namespace `ASAM.XIL.Interfaces.Testbench.Common.Error`.

HIL API Python code fragment:

```
try:
    maPort = ASAM.HILAPI.dSPACE.MAPort.MAPort(configDictionary)
    # ...
except HILAPIException as ex:
    print "HILAPIException occurred:"
    print ex.message
    raise
except Exception as ex:
    print "Exception occurred:"
    print ex.message
    raise
```

Corresponding XIL API .NET code fragment:

```
maPort = None

try:
    maPort = testbench.MAPortFactory.CreateMAPort("DemoMAPort")
    # ...
except TestbenchPortException as ex:
    print "TestbenchPortException occurred:"
    print "Code: %d" % (ex.Code)
    print "CodeDescription: %s" % (ex.CodeDescription)
    print "VendorCode: %d" % (ex.VendorCode)
    print "VendorCodeDescription: %s" % (ex.VendorCodeDescription)
    raise
except Exception as ex:
    print "Exception occurred:"
    print ex.message
    raise
finally:
    # free the allocated memory and any system resources of the MAPort instance
    if maPort != None:
        maPort.Dispose()
        maPort = None
```

As you can see in the example, not only ASAM exceptions, but also .NET exceptions must be caught. In addition, some objects require a call to the `Dispose()` method to ensure that in any case (not only when an exception occurs) the memory and, possibly, the system resources associated with the object are released. This applies to objects of the `MAPort` and `Capture` classes. For instances of SignalGenerator class, call the `DestroyOnTarget()` method.

In ASAM exceptions, detailed information about the cause of the error is shown by `Code`, `CodeDescription`, `VendorCode` and `VendorCodeDecription`. We recommend to show the `VendorCodeDescription` in error messages or log files of your application; the `VendorCodeDescriptions` in most cases gives detailed information on the cause of an error.

6.5 Scaling Information in the Variable Description File

The variable description file defines whether a source value on a dSPACE platform can be represented as a converted value on the host PC and how it is to be converted. As an example, a specific parameter of some fixed point data type in the real-time application might be represented as a real world physical value in the test scenario.

The dSPACE HIL API uses source values by default when reading, writing or capturing variables. For further information on scaling, refer to Software > Test Automation > dSPACE HIL API Implementations > dSPACE HIL API Python Implementation Document > Using dSPACE HIL API Python Implementation > Specific Enhancements to the HIL API Standard > Value scaling enable/disable via TRC file.

With the dSPACE XIL API, the behavior changed to converted values. The scaling is enabled by default. This means that the value conversion is executed as specified by the scaling attribute in the variable description file. If this behavior is not desired, it can be disabled within the MAPort configuration file. For further information, refer to Software > Test Automation > dSPACE XIL API Implementations > dSPACE XIL API Reference > Model Access Port Implementation > MAPort Configuration > EnableScaling.

7 Complete Sample Scripts

This document is accompanied by a ZIP archive containing a complete sample solution, and a Simulink model.

To run the sample scripts, you have to execute some preparatory tasks as described in section 3 Prerequisites/Preparatory Tasks of this document.

The file names of the samples indicate if the dSPACE HIL API Python server or the dSPACE XIL API .NET server is used. The header of each sample gives a short explanation of the use case of the script. The first few lines of code must be adjusted to the dSPACE processor platform and location of the real-time application you use.

Further Reading

For details on the dSPACE XIL API .NET server, refer to Software > dSPACE XIL API Implementations in dSPACE HelpDesk.

For details on the ASAM XIL API standard, visit http://www.asam.net
(ASAM Automotive Electronics (AE) / AE X-in-the-Loop API V2.0.1).