

Real-Time Testing

Guide

For Real-Time Testing 5.0

Release 2021-A – May 2021

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

If you encounter a problem when using dSPACE products, contact your local dSPACE representative:

- Local dSPACE companies and distributors: <http://www.dspace.com/go/locations>
- For countries not listed, contact dSPACE GmbH in Paderborn, Germany.
Tel.: +49 5251 1638-941 or e-mail: support@dspace.de

You can also use the support request form: <http://www.dspace.com/go/supportrequest>. If you are logged on to mydSPACE, you are automatically identified and do not need to add your contact details manually.

If possible, always provide the relevant dSPACE License ID or the serial number of the CmContainer in your support request.

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/patches> for software updates and patches.

Important Notice

This publication contains proprietary information that is protected by copyright. All rights are reserved. The publication may be printed for personal or internal use provided all the proprietary markings are retained on all printed copies. In all other cases, the publication must not be copied, photocopied, reproduced, translated, or reduced to any electronic medium or machine-readable form, in whole or in part, without the prior written consent of dSPACE GmbH.

© 2006 - 2021 by:
dSPACE GmbH
Rathenaustraße 26
33102 Paderborn
Germany

This publication and the contents hereof are subject to change without notice.

AUTERA, ConfigurationDesk, ControlDesk, MicroAutoBox, MicroLabBox, SCALEXIO, SIMPHERA, SYNECT, SystemDesk, TargetLink and VEOS are registered trademarks of dSPACE GmbH in the United States or other countries, or both. Other brand names or product names are trademarks or registered trademarks of their respective companies or organizations.

Contents

About This Document	9
Safety Precautions	11
General Warning.....	11
Introduction to Real-Time Testing	13
Features of Real-Time Testing.....	14
Real-Time Testing in a dSPACE System.....	16
Hardware and Software Requirements.....	18
Application Areas.....	19
Example of Using Real-Time Testing.....	21
Demo Examples of Using Real-Time Testing.....	22
Basics of Real-Time Testing	25
Software Components of Real-Time Testing.....	25
Basics on RTT Sequences.....	27
Basics on Running RTT Sequences.....	29
Workflow for Real-Time Testing.....	31
Enabling Real-Time Testing for dSPACE Platforms	33
Basics on Enabling Real-Time Testing.....	34
How to Enable Real-Time Testing for a DS1006 or MicroAutoBox II.....	34
How to Enable Real-Time Testing for a DS1006 Multiprocessor System.....	36
How to Enable Real-Time Testing for a SCALEXIO System.....	37
How to Enable Real-Time Testing for MicroAutoBox III.....	38
Real-Time Testing and VEOS.....	41
Implementing RTT Sequences	43
Implementing Generator Functions.....	45
Basics on Generator Functions.....	46
Using Generator Functions.....	47
Implementing an RTT Sequence Using User-Defined Generator Functions.....	49

Terminating a Generator Function.....	51
Using the Parallel() Generator Function.....	53
Using the ParallelRace() Generator Function.....	54
General Information on Implementing RTT Sequences.....	57
Encoding the Scripts of RTT Sequences.....	58
Simple RTT Sequence.....	59
Getting Properties of an RTT Sequence.....	60
Using Local and Global Variables in RTT Sequences.....	61
Using Variables Accessible by Several RTT Sequences.....	62
Read/Write Access to Variables of the Simulation Application.....	62
Checking Conditions According to the ASAM GES Standard.....	65
Printing Variable Values of an RTT Sequence on the Host PC.....	68
Printing Messages in the dSPACE Log from an RTT Sequence.....	69
Avoiding a Timeout During Initialization of an RTT Sequence.....	70
Exception Handling and Using Modules.....	73
Implementing an Exception Handling.....	73
Using User Python Modules.....	74
Using Modules from the Standard Python Library.....	75
Exchanging Data Between RTT Sequence and Host Python Script.....	77
Basics on Dynamic Variables.....	77
Example of Using Dynamic Variables.....	80
Exchanging Data Between RTT Sequences and Python Scripts	
Running on the Host PC.....	83
Example of Exchanging User-Defined Data.....	85
Data Replay in RTT Sequences.....	87
Basics of Data Replay Using MAT Files.....	87
Basics of Data Replay Using ASAM MDF (MF4) Files.....	90
Replay Mode.....	93
Example of Data Replay Using MAT Files.....	98
Example of Data Replay Using ASAM MDF (MF4) Files.....	101
Handling CAN Messages.....	104
CAN Messages in RTT Sequences.....	104
Implementing CAN Communication in RTT Sequences.....	104
Handling CAN Messages Using the rttlib.canlib Module.....	105
Basics on the rttlib.canlib Module.....	106
Basics on Handling CAN FD Messages with the rttlib.canlib Module.....	108
How to Prepare the Simulink Model for CAN Message Handling.....	109
How to Prepare the Simulink Model for CAN FD Message Handling.....	111
How to Use a Prepared RTT CAN Model with a SCALEXIO System.....	115

Accessing the CAN Bus with the rttlib.canlib Module.....	117
Sending CAN Messages with the rttlib.canlib Module.....	118
Receiving CAN Messages with the rttlib.canlib Module.....	121
Handling CAN Messages Using the rttlib.dscanapilib Module.....	123
Basics of the rttlib.dscanapilib Module.....	124
How to Prepare a CAN Channel for Using it with the rttlib.dscanapilib Module.....	125
Accessing the CAN Bus with the rttlib.dscanapilib Module.....	127
Sending CAN Messages with the rttlib.dscanapilib Module.....	129
Receiving CAN Messages with the rttlib.dscanapilib Module.....	132
Unregistering CAN Channels with the rttlib.dscanapilib Module.....	136
Implementing Communication via Ethernet.....	137
Basics on the dsethernetapilib Module.....	137
How to Use Ethernet with Real-Time Testing on a SCALEXIO Platform.....	138
Using Ethernet with Real-Time Testing on a VEOS Platform.....	139
Example of Sending and Receiving Frames via Ethernet.....	140
Implementing a Communication via a Serial Interface.....	144
Basics of Working with the rs232lib Module.....	144
Example of Sending Data via a Serial Interface.....	145
Example of Receiving Data via a Serial Interface.....	147
Accessing Variables of a Simulation Application on a Remote Node.....	150
Basics on Accessing Variables of a Simulation Application on a Remote Node.....	150
Notes on Accessing Variables of a Simulation Application on a Remote Node.....	153
Example of Accessing Variables of a Remote Node.....	156
Tips and Tricks.....	162
Writing Effective RTT Sequences.....	162

Managing RTT Sequences 165

Basics on Managing RTT Sequences.....	166
States of RTT Sequences.....	166
Basics on Executing RTT Sequences.....	167
Managing RTT Sequences Using the Real-Time Test Manager.....	170
Basics on the Real-Time Test Manager.....	171
How to Customize the Screen Arrangement.....	172
How to Specify and Use a Filter.....	174
How to Start the Real-Time Test Manager and Access a Platform.....	176

How to Create a New RTT Sequence on the Platform.....	178
How to Manage RTT Sequences on the Real-Time Platform.....	180
Managing RTT Sequences in Python Scripts.....	181
Basics on the Real-Time Test Manager Server Interface.....	182
Creating and Starting RTT Sequences in Python Scripts.....	182
Starting RTT Sequences with Arguments in Python Scripts.....	185
Controlling one RTT Sequence in Python Scripts.....	187
Controlling All the Created RTT Sequences in Python Scripts.....	189
Handling Events of RTT Sequences in Python Scripts.....	191
Handling OnHostCall Events of an RTT Sequence in Python Scripts.....	193
Getting the Run Time of an RTT Sequence.....	195
Error Management.....	198
Debugging RTT Sequences.....	198
Example of Debugging an RTT Sequence.....	198
Introduction to the Message Reader API.....	201
Reading dSPACE Log Messages via the Message Reader API.....	201
Supported dSPACE Products and Components.....	203
Example of Reading Messages with Python.....	203
Example of Reading Messages with C#.....	205

Test Scenario Demos 209

Example of Testing ECUs for Turn Signals and Warning Signals Control.....	209
--	-----

Troubleshooting 213

Troubleshooting for RTT Sequences.....	214
Peaks in Turnaround Time.....	215
A TestOverrun Exception Occurs.....	216
The Output of Print Is Lost.....	217
No Output in Running State.....	217
Timeout During Initialization.....	217
Exceptions Occur During Initialization.....	218
Exceptions Occur During Data Streaming.....	218
Problem when Using Too Many Data Streams.....	219
The RTT Sequence Does Not Run in Real-Time.....	220
Timing Problem When Removing Objects in a Loop.....	221
Variables Cannot Be Accessed After Migrating to MATLAB R2011a or Later.....	221
Lost Variable Values.....	222
Waiting for Barrier Timed Out.....	223





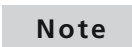


Exception Occurs When Creating a Variable Object.....	224
Memory Leaks Caused by Circular References.....	224
Using Real-Time Testing in Large SCALEXIO Systems.....	224
Troubleshooting for Host PC Python Scripts.....	226
Using Sleep() Function.....	226
Using Different Versions of Real-Time Testing.....	226
Error Message: RPC Server is Unavailable.....	228
Limitations	229
General Limitations for Real-Time Testing.....	229
Limitations When Using Real-Time Testing.....	230
Limitations for Platforms.....	234
Index	237


About This Document

Content	This guide describes the features of Real-Time Testing. It gives information on the basics of Real-Time Testing and shows how you can implement it on your dSPACE platform.
Required knowledge	Knowledge in handling the host PC and the Microsoft Windows operating system is assumed. This document is primarily targeted at engineers who have experience with the Python programming language.
Documented product versions	This documentation is part of several product versions of Real-Time Testing. As long as it is not stated, the descriptions are valid for all product versions. If there are differences, the product versions are stated.

Symbols

dSPACE user documentation uses the following symbols:

Symbol	Description
	Indicates a hazardous situation that, if not avoided, will result in death or serious injury.
	Indicates a hazardous situation that, if not avoided, could result in death or serious injury.
	Indicates a hazardous situation that, if not avoided, could result in minor or moderate injury.
	Indicates a hazard that, if not avoided, could result in property damage.
	Indicates important information that you should take into account to avoid malfunctions.
	Indicates tips that can make your work easier.
	Indicates a link that refers to a definition in the glossary, which you can find at the end of the document unless stated otherwise.

Symbol	Description
	Precedes the document title in a link that refers to another document.

Naming conventions

dSPACE user documentation uses the following naming conventions:

%name% Names enclosed in percent signs refer to environment variables for file and path names.

< > Angle brackets contain wildcard characters or placeholders for variable file and path names, etc.

Special folders

Common Program Data folder A standard folder for application-specific configuration data that is used by all users.

%PROGRAMDATA%\dSPACE\<InstallationGUID>\<ProductName>

or

%PROGRAMDATA%\dSPACE\<ProductName>\<VersionNumber>

Documents folder A standard folder for user-specific documents.

%USERPROFILE%\Documents\dSPACE\<ProductName>\<VersionNumber>

Local Program Data folder A standard folder for application-specific configuration data that is used by the current, non-roaming user.

%USERPROFILE%\AppData\Local\dSPACE\<InstallationGUID>\<ProductName>

Accessing dSPACE Help and PDF Files


After you install and decrypt dSPACE software, the documentation for the installed products is available in dSPACE Help and as PDF files.

dSPACE Help (local) You can open your local installation of dSPACE Help:

- On its home page via Windows Start Menu
- On specific content using context-sensitive help via **F1**

dSPACE Help (Web) You can access the Web version of dSPACE Help at www.dspace.com/go/help.

To access the Web version, you must have a *mydSPACE* account.

PDF files You can access PDF files via the  icon in dSPACE Help. The PDF opens on the first page.

Safety Precautions

Introduction

To avoid risk of injury and/or property damage, read and ensure compliance with the safety precautions given.

General Warning

Danger potential

Using dSPACE software can be dangerous. You must observe the following safety instructions and the relevant instructions in the user documentation.

WARNING

Improper or negligent use can result in serious personal injury and/or property damage.

Using Real-Time Testing can have a direct effect on technical systems (electrical, hydraulic, mechanical) connected to it. You can download real-time tests described in Python by the Real-Time Test Manager or scripts running in a Python interpreter directly to the dSPACE real-time processor. When these tests are started they are running in parallel to the real-time application as long as they are stopped explicitly (closing the Real-Time Test Manager application or the Python interpreter is not sufficient to stop running real-time tests). Real-time tests are able to change potentially any model parameter at run-time without any further user interaction and user notification. By this, the output behavior of the dSPACE hardware is directly affected.

- **Only persons who are qualified to use dSPACE software, and who have been informed of the above dangers and possible consequences, are permitted to use this software.**
- All applications where malfunctions or operating errors involve the danger of injury or death must be examined for potential hazards by the user, who must if necessary take additional measures for protection (for example, an emergency off switch).

Liability

It is your responsibility to adhere to instructions and warnings. Any unskilled operation or other improper use of this product in violation of the respective safety instructions, warnings, or other instructions contained in the user documentation constitutes contributory negligence, which may lead to a limitation of liability by dSPACE GmbH, its representatives, agents and regional dSPACE companies, to the point of total exclusion, as the case may be. Any exclusion or limitation of liability according to other applicable regulations, individual agreements, and applicable general terms and conditions remain unaffected.

Data loss during operating system shutdown

The shutdown procedure of Microsoft Windows operating systems causes some required processes to be aborted although they are still being used by dSPACE software. To avoid data loss, the dSPACE software must be terminated manually before a PC shutdown is performed.

Introduction to Real-Time Testing

Introduction

This is a brief introduction to Real-Time Testing, describing the features of Real-Time Testing, and the software and hardware requirements. Application areas are described, and an example is given.

Where to go from here

Information in this section

[Features of Real-Time Testing..... 14](#)

To give you a brief introduction to Real-Time Testing.

[Real-Time Testing in a dSPACE System..... 16](#)

Giving an overview of how real-time testing is embedded in a dSPACE system.

[Hardware and Software Requirements..... 18](#)

There are some system requirements for working with Real-Time Testing. Check whether your system fulfills these requirements before you start using Real-Time Testing.

[Application Areas..... 19](#)

Describing some application areas where real-time testing is useful.

[Example of Using Real-Time Testing..... 21](#)

An example demonstrates where real-time testing is useful.

[Demo Examples of Using Real-Time Testing..... 22](#)

Some demo examples are installed with Real-Time Testing. Examine these examples to learn about the features of Real-Time Testing.

Features of Real-Time Testing

Introduction

To give you a brief introduction to Real-Time Testing.

Features

Real-Time Testing has the following key features:

- Basic features
 - Tests are programmed in Python, the object-oriented scripting language. Tests are called RTT sequences.
 - RTT sequences can be performed synchronously with the simulation application.
 - RTT sequences are executed on the processor board in real time.
 - RTT sequences can use standard Python modules and modules provided by dSPACE for real-time testing.
 - A single RTT sequence can contain concurrent elements, that is an RTT sequence can stimulate and monitor signals in parallel.
 - Several independent RTT sequences can be performed at the same time.
 - Time measurements in RTT sequences are performed at the resolution of the simulation step size.
 - RTT sequences can be parameterized.
- Variables
 - Python data objects can be exchanged between RTT sequences running on the simulation platform and Python scripts running on the host PC by using host calls.
 - RTT sequences have read/write access to the signals and parameters of the simulation application.
 - Dynamic variables can be created in an RTT sequence during simulation run time. You have read/write access to them from other RTT sequences running on the same simulation platform and from the host PC as well.
 - Data replay in RTT sequences can be performed by streaming data piecewise from a MAT file or an ASAM MDF file to a certain RTT variable object. Data replay can be restarted within a single RTT sequence.
- Managing RTT sequences
 - RTT sequences are managed via a graphical user interface (Real-Time Test Manager) or a Python script running in a Python interpreter on the host PC.
 - RTT sequences can be created or started with arguments on the simulation platform.
 - Paused, stopped, and terminated RTT sequences can be restarted with the same namespace.
- CAN messages
 - RTT sequences can send and receive CAN messages in the raw format using dSPACE CAN hardware and the RTI CAN MultiMessage Blockset. The canlib module contains all the necessary classes and methods.
 - For SCALEXIO, DS6001, MicroAutoBox III, and VEOS, RTT sequences can send and receive CAN messages in the raw format using the `dscanapilib`

module. It is not necessary that the simulation model contains blocks of the RTI CAN MultiMessage Blockset.

- Multiprocessor systems
 - In multiprocessor systems, RTT sequences have transparent read and write access to the signals and parameters on a remote CPU.
- RS232 library
 - An RS232 communication can be implemented fully with RTT sequences. No RTI block in the simulation model is necessary.
 - Sending and receiving data via the onboard RS232 interfaces of the DS1006 processor board.

Python version on the platform

Real-Time Testing supports Python with a specific version on the platform, i.e. the tests can use standard modules of this version.

Note that older versions of Real-Time Testing used older Python versions for RTT sequences. Old real-time models and BCG files have to be recompiled with the current version of Real-Time Testing to upgrade them to the required Python.

Real-Time Testing Version	Python Version
Real-Time Testing 1.2 and earlier	Python 2.4.3
Real-Time Testing 1.3 ... Real-Time Testing 2.6	Python 2.5.1
Real-Time Testing 3.0	Python 2.7.10
Real-Time Testing 3.1 and later	Python 2.7.11 ¹⁾
Real-Time Testing 4.2	Python 3.6.4 ²⁾ for VEOS and MicroAutoBox III Python 2.7.11 for all other platforms
Real-Time Testing 4.4 and later	Python 3.6.4 ²⁾ for SCALEXIO, VEOS, and MicroAutoBox III Python 2.7.11 for all other platforms

¹⁾ Recompiling the BCG files are not necessary when they are compiled with Python 2.7.10.

²⁾ Note that all strings in Python 3.6 are unicode objects. You must use the correct encoding. For more information on migrating to this Python version, refer to www.dspace.com/go/Python36Migration.

Python version on the host PC

Real-Time Testing supports Python 3.9 on the host PC.

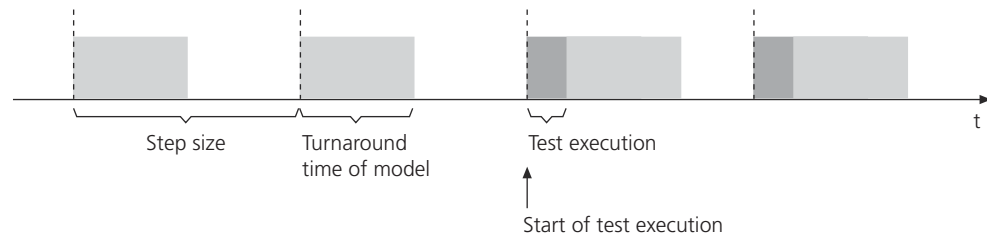
Host software

As of Real-Time Testing 3.1, the host software of Real-Time Testing is available only as 64-bit variant.

In Real-Time Testing 3.1 and lower, the host software of Real-Time Testing is available only as 32-bit variant.

Functionality

The basis of real-time testing is a Python interpreter running on the simulation platform. The Python interpreter contains special modifications which make it suitable for real-time environments. The simulation platform calls the Python interpreter and executes the simulation application in each sampling step. Executing the Python interpreter requires some processing time, see the following illustration.



The real-time Python interpreter processes the RTT sequences. The RTT sequences have read/write access to the variables of the real-time application. The test and real-time application are executed in one simulation step. You can stimulate the simulation application in one simulation step and monitor the results in the next simulation step. There are no latencies between starting a stimulation or measurement and executing a stimulation or measurement, because both are done in the same simulation step on the simulation platform.

The RTT sequences are executed without latency in relation to the execution of the simulation application. There are no latencies caused by the host PC. This may happen when the tests are performed on the host PC, for example, if you work with AutomationDesk. Executing the RTT sequences requires some computing time on the simulation platform. The calculation of the model (turnaround time) must therefore not take up the whole simulation step size.

Related topics**Basics**

[Application Areas.....](#) 19

Real-Time Testing in a dSPACE System

Introduction

The following overview shows you how real-time testing is embedded in a dSPACE system.

System overview

Real-time testing requires no extension of the simulation system. The RTT sequences can be implemented and managed from the host PC. The real-time model does not have to be changed but the simulation application must include

the service call to the Python interpreter. The service call is included in the simulation application during its build process. Thus, simulation applications where the service call is not included must only be recompiled.

Host PC

The host PC is a PC on which Real-Time Testing is installed to control the RTT sequences on the platforms. Real-Time Testing can run on Windows and Ubuntu Linux operating systems. Not all tasks can be performed on a Linux PC.

Windows operating system If you have a Windows PC, you can control the RTT sequences using a graphical user interface (RTT Manager) or Python scripts.

Linux operating system On a Linux PC, you can control the RTT sequences using Python scripts only. A graphical user interface is not available. As the dSPACE tools required for the build process do not run on a Linux PC, you cannot build simulation applications on Linux PCs. This must be done on a Windows PC on which the required dSPACE tools are installed. To control simulation applications (load, start, stop) from the Linux PC, the CmdLoader tool is available. Furthermore, Real-Time Testing on Linux supports only VEOS and SCALEXIO (SCALEXIO Processing Unit and DS6001 Processor Board) platforms.

Real-time system

The real-time system calculates the real-time application and executes the RTT sequences. The real-time system must be connected to the host PC where Real-Time Testing is installed. The kind of connection depends on the real-time system. For example, a SCALEXIO system is connected via Ethernet.

Real-Time Testing and the other dSPACE tools are not required to run on the same host PC. You can build and download the real-time application on a host PC and use another host PC for Real-Time Testing.

VEOS

The VEOS platform calculates the offline simulation application and executes the RTT sequences. VEOS can run on Windows PCs or Linux PCs. Real-Time Testing and VEOS are not required to run on the same host PC. Real-Time Testing can control the RTT sequences on a VEOS remotely if the remote PC is in the same network.

Related topics

Basics

[Enabling Real-Time Testing for dSPACE Platforms.....](#) 33

Hardware and Software Requirements

Introduction

There are some system requirements for working with Real-Time Testing. Check whether your system fulfills these requirements before you start using Real-Time Testing.

Simulation platform

Real-Time Testing supports the following dSPACE platforms:

- Modular system based on a DS1006 Processor Board
Handling CAN messages requires an I/O board with CAN interface: DS2202, DS2210, DS2211, or DS4302 boards.
Handling CAN FD messages requires an I/O board with CAN FD interface: DS4505 with DS4342 modules.
- Modular system based on a DS1007 PPC Processor Board
Handling CAN messages requires an I/O board with CAN interface: DS2202, DS2210, DS2211, or DS4302 boards.
Handling CAN FD messages requires an I/O board with CAN FD interface: DS4505 with DS4342 modules.
- MicroAutoBox with at least 800 MHz
- MicroLabBox
- SCALEXIO Processing Unit
Handling CAN messages requires an I/O board or module with CAN interface: DS2671 Bus Board, DS2672 Bus Module, or DS6301 CAN/LIN Board.
Handling CAN FD messages requires an I/O board with CAN FD interface: DS2671 Bus Board or DS6301 CAN/LIN Board.
- DS6001 Processor Board
Handling CAN or CAN FD messages requires a DS6301 CAN/LIN Board.
- VEOS
 - As single platform
 - With V-ECU on VEOS
 - With multiple V-ECUs on VEOS
 - Running 64-bit application
- MicroAutoBox III

Preconditions for the real-time system

Computing time RTT sequences require computing time. As a rule of thumb, there should be at least 10% of the sample time or at least the following free computing time.

Real-Time System	Free Computing Time
DS1006 2.6 GHz	50 μ s
DS1007	100 μ s
MicroAutoBox 800 MHz	200 μ s
MicroAutoBox III	150 μ s

Real-Time System	Free Computing Time
MicroLabBox	100 μ s
SCALEXIO system	100 μ s 200 μ s if data replay is used

You should read the turnaround time of the real-time application and check whether enough computing time remains.

The actual computing time which is required to perform Real-Time Testing depends on the calculation which are executed in the RTT sequence.

Memory Running an RTT sequence requires memory for the Python interpreter (with preloaded Python libraries) and for the RTT sequence. Additionally, the TRC file is downloaded to the real-time platform, where it requires about 20% to 25% of its size.

Software requirements

The following software is required for the host PC.

Operating system Real-Time Testing supports Windows and Linux operating systems. For details on the supported operating systems, refer to [Operating System \(New Features and Migration\)](#).

Note that Linux operating systems are not supported by all dSPACE tools. For example, it is not possible to build simulation applications or control them on such operating systems. Furthermore, the RTT Manager (graphical user interface) is not available. You can control the RTT sequence only via Python scripts.

ControlDesk ControlDesk is required for managing the real-time platform and real-time application.

MATLAB/Simulink Real-Time Testing does not require MATLAB/Simulink during the tests. However, the real-time application must be compiled for real-time testing (see [Enabling Real-Time Testing for dSPACE Platforms](#) on page 33).

Text editor To write the tests, a text editor is required.

RTI CAN MultiMessage Blockset To handle CAN messages, your Simulink model must contain blocks of the RTI CAN MultiMessage Blockset.

Application Areas

Introduction

Some application areas where real-time testing is useful are described below.

Stimulus without latency

You have read/write access to the variables of the simulation application via the RTT sequence without latency. You can access the variables of the real-time application in the same sampling step before or after the simulation application is calculated. All values can be accessed synchronously to the simulation application. Signals can be stimulated at a precisely specified point of time.

Time measurement	Time measurement is more precise. There is no latency between time measurement in RTT sequences and the simulation application. The resolution of the measurement depends only on the step size of the simulation. You can access the precise current time of the current sampling step from the RTT sequence.
Signal monitoring without data capture and postprocessing	<p>Since the RTT sequence can read model variables and evaluate conditions containing these variables, there is no need for data capturing and postprocessing on the host PC.</p> <p>Test results can therefore be evaluated <i>online</i> instead of <i>offline</i>.</p>
Sending and receiving CAN or CAN FD messages in raw data format	With the canlib or dscanapilib module, you can handle CAN or CAN FD messages in the RTT sequences. By this, the RTT sequences are able to receive or send CAN or CAN FD messages in raw data format. The canlib module requires that your Simulink model contains blocks of the RTI CAN MultiMessage Blockset. For SCALEXIO, DS6001, MicroAutoBox III, and VEOS, you can use the dscanapilib module.
Using parameters for RTT sequences	You can specify parameters for RTT sequences. The parameters can be specified on the host PC. This makes it possible to create an RTT sequence once and run it several times with different data.
Dynamic variables	Dynamic variables are created in the RTT sequences during simulation run time. By using dynamic variables, you can perform some calculation in the RTT sequence without changing the simulation model. Using dynamic variables creates a user-defined interface between the host PC and the RTT sequences without having to modify existing model variables.
Replaying data	If you have saved data in a MAT or MDF4 file, you can use it in your RTT sequences, for example, to stimulate model variables. The data file saved on the host PC is copied in parts to the simulation platform. This makes it possible to replay even large data files.
Sending and receiving data via RS232 interface	You can use the serial interface of a DS1006 Processor Board to send and receive data in the RS232 standard. The RS232 communication can be fully implemented in RTT sequences. The real-time model must not be modified.
Sending and receiving frames via Ethernet in raw data format	Only for SCALEXIO, DS6001, MicroAutoBox III, VEOS, and VEOS V-ECU platforms: With the dsethernetapilib module, you can handle frame messages of the Ethernet communication in the RTT sequences. By this, the RTT sequences

are able to receive or send frames in raw data format. This feature requires that the Simulink model contains the Ethernet Setup block.

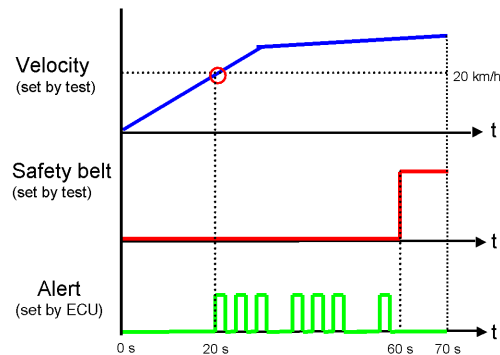
Example of Using Real-Time Testing

Introduction

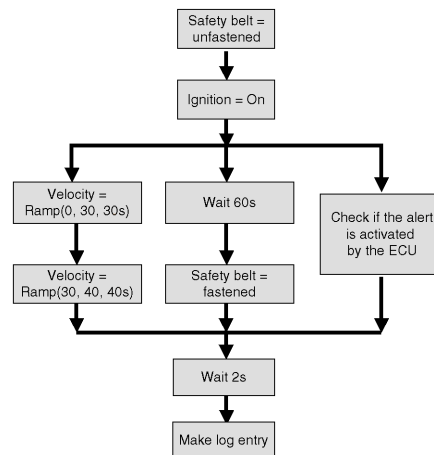
The following example demonstrates where real-time testing is useful.

Belt minder

The example shows a belt minder in a car. The belt minder's behavior is implemented by an ECU. It warns the driver when he or she drives faster than a specified velocity (20 km/h) and the safety belt is not fastened. The following illustration shows an example of the expected behavior. The velocity increases. When the car exceeds 20 km/h, an alert is output. When the driver fastens the belt, the alert must be stopped immediately.



This is a schematic of the belt minder test:



The following listing shows the corresponding RTT sequence for testing the belt minder. The RTT sequence is not ready to use. It shows only the principle structure.

```

from rttlib import variable
from rttlib import scheduler
from rttlib import utilities
Ignition = variable.Variable(r'Model Root/Ignition')
Velocity = variable.Variable(r'Model Root/Velocity/Value')
BeltLock = variable.Variable(r'Model Root/BeltLock/Value')
BeltMinder = variable.Variable(r'Model Root/BeltMinder/Value')
TestPass = 0
def SetVelocity(VariableObject):
    yield GenerateRamp(VariableObject, 0.0, 30.0, 30)
    yield GenerateRamp(VariableObject, 30.0, 40.0, 40)
def SetBeltLock(Delay):
    yield utilities.Wait(Delay)
    BeltLock.Value = 1
def CheckAlert():
    global TestPass
    while(1):
        if(((Velocity.Value >= 20) and (BeltLock.Value == 0) and \
            (BeltMinder.Value == 0)) or
            ((Velocity.Value >= 20) and (BeltLock.Value == 1) and \
            (BeltMinder.Value == 1)) or
            ((Velocity.Value < 20) and (BeltMinder.Value == 1))):
            TestPass = 0
        yield None
def MainGenerator(*args):
    # set driver unfastened
    BeltLock.Value = 0
    # Ignition on
    Ignition.Value = 1
    # Generate Signals and check.
    yield scheduler.ParallelRace(SetVelocity(Velocity),\
        SetBeltLock(60.0),\
        CheckAlert())
    # wait for 2 sec.
    yield utilities.Wait(2.0)
    if(TestPass==1):
        print("Test passed")
    else:
        print ("Test failed")
    # continue script execution
    ...

```

Demo Examples of Using Real-Time Testing

Introduction

Some demo examples are installed with Real-Time Testing. Examine these examples to learn about the features of Real-Time Testing.

Accessing the demos

The demos are zipped in archives which are installed under the installation folder of Real-Time Testing:

C:\Program Files\Common Files\dSPACE\RealTimeTesting\5.0\Demos

To work with the demos, unzip them to a work folder. You must have write access to the work folder. You can use the Real-Time Test Manager for unzipping the demos, refer to [Unzip Demos \(Real-Time Testing Library Reference !\[\]\(35e4f762fc1cfea5610d92e2d225d5b4_img.jpg\)](#)).

Contents of a demo

Real-Time Testing provides several ready-to-use demos. Each demo contains the following files and folders.

StartDemo.bat file The **StartDemo.bat** file can be used to start the demo. This batch file guides you through the process of loading the demo application and the execution of an RTT sequence. Press **Ctrl+C** or **Ctrl+Break** to abort the batch file.

ReadMe.txt file The **ReadMe.txt** file describes the demo. Start an ASCII editor to read it.

Python folder The **Python** folder contains the Python loader script. It demonstrates how to load and execute an RTT sequence with Python.

RTTSequences folder The **RTTSequences** folder contains the RTT sequence to be loaded and other supporting files.

<DemoName>.zip The **<DemoName>.zip** file is an archive that contains all the files of the demo.

ControlDesk projects

In addition to the Python scripts (RTT sequences and loader scripts), the demos folder contains ControlDesk projects for all the supported platforms in the **SampleExperiments** folder. The projects are stored as backup files. You can open the backup files in ControlDesk. Refer to [Open Project + Experiment from Backup \(ControlDesk Project and Experiment Management !\[\]\(d0262bbe9d2356661a2e89321dfcc781_img.jpg\)](#)).

Starting the Python demos

Batch files are a convenient way to start the tests. The **StartDemo.bat** batch files are in the root folder of the demo and perform the following tasks:

- Let you select the platform type.
- Display a short instruction how to start the experiment.
- Start Python.exe with the Python script which starts the selected test (RTT sequence).

Related topics

References

[Explore Demos Folder \(Real-Time Testing Library Reference !\[\]\(06a315363e7801bba8c7489a6694af19_img.jpg\)](#))
[Open Project + Experiment from Backup \(ControlDesk Project and Experiment Management !\[\]\(e6891337bae9c3e53d940590bb38555a_img.jpg\)](#))
[Unzip Demos \(Real-Time Testing Library Reference !\[\]\(3b3087d58101bd9c870ed8cf31cc67b6_img.jpg\)](#))

Basics of Real-Time Testing

Introduction	The basics of Real-Time Testing (RTT) are described below, including the software components, RTT sequences, and how Real-Time Testing is integrated into a dSPACE system.
---------------------	--

Where to go from here	Information in this section
	Software Components of Real-Time Testing..... 25 Real-Time Testing consists of several software components.
	Basics on RTT Sequences..... 27 RTT sequences are the tests which are programmed in Python for real-time testing.
	Basics on Running RTT Sequences..... 29 The application and the RTT sequence run on the same platform. The platform calculates the application and scheduled parts of the RTT sequence in one sampling step.
	Workflow for Real-Time Testing..... 31 You must perform several steps for real-time testing.
	Information in other sections
	Implementing RTT Sequences..... 43 Describes how you can implement RTT sequences for real-time testing.

Software Components of Real-Time Testing

Introduction	The software components of Real-Time Testing are described below.
---------------------	---

Python interpreter

The test engine is a Python interpreter running on the simulation platform in parallel to the simulation model. The Python interpreter processes the RTT sequences. The version of the Python interpreter and the kind of integration depend on the platform type, refer to the following table.

Platform	Python Version	Integration	
		Linked to Real-Time Application ¹⁾	Part of Firmware ²⁾
DS1006 Processor Board	2.7.11	✓	–
DS1007 PPC Processor Board	2.7.11	–	✓
MicroLabBox	2.7.11	–	✓
MicroAutoBox II	2.7.11	✓	–
MicroAutoBox III	3.6.4	–	✓
SCALEXIO Processing Unit	3.6.4	–	✓
DS6001 Processing Board	3.6.4	–	✓
VEOS	3.6.4	–	✓
V-ECU on VEOS	3.6.4	–	✓

¹⁾ The Python interpreter is linked to the real-time application during the build process.

²⁾ The Python interpreter is part of the firmware.

dSPACE Python libraries

dSPACE provides some Python libraries for real-time testing. These Python libraries are used in the RTT sequences, for example, to access variables of the real-time application. For an overview, refer to [dSPACE Python Modules for Implementing RTT Sequences \(Real-Time Testing Library Reference !\[\]\(de95854c7ee024cfadc48187bbb781b2_img.jpg\)](#).

Real-Time Test Manager Server

The Real-Time Test Manager Server is a server with a Python interface. The Python interface is used in host scripts which manage the RTT sequences. It has all the required functions to download, start, pause, continue, or stop RTT sequences. Additionally, it can be used to implement an event handling on the host.

For information on how to work with the Real-Time Test Manager Server, refer to [Managing RTT Sequences in Python Scripts](#) on page 181.

For information on the module, refer to [rttmanagerlib Module \(Real-Time Testing Library Reference !\[\]\(6a9b39b98eb945faa14c645ec99e4eaa_img.jpg\)](#).

Real-Time Test Manager

The Real-Time Test Manager is a graphical user interface for managing RTT sequences on the real-time platform. You can use the Real-Time Test Manager to download the RTT sequences (as PY file or BCG files) and control their execution. It has the Log Viewer for the outputs of the RTT sequences.

For information on how to work with the Real-Time Test Manager, refer to [Managing RTT Sequences Using the Real-Time Test Manager](#) on page 170.

For information on the commands, refer to [Real-Time Test Manager Commands \(Real-Time Testing Library Reference !\[\]\(c507f772dba2b921f86777f01218e570_img.jpg\)\)](#).

Related topics

Basics

[Managing RTT Sequences in Python Scripts..... 181](#)

References

[dSPACE Python Modules for Implementing RTT Sequences \(Real-Time Testing Library Reference !\[\]\(e474458956c9a37fbf9586ddb60a7fa1_img.jpg\)\)](#)
[rttmanagerlib Module \(Real-Time Testing Library Reference !\[\]\(4d1d3f2547aeece54bb6babd23f4121b_img.jpg\)\)](#)

Basics on RTT Sequences

Introduction

RTT sequences are the tests which are implemented in Python for real-time testing. The RTT sequences are downloaded to the simulation platform. The RTT sequences and the real-time application are executed synchronously with the simulation.

File names of RTT sequences

The file names of the RTT sequences must fulfill the rules for Python modules. This means that no special characters are allowed, for example, spaces, dots. The following characters are allowed:

- Lower case: "a" ... "z"
- Upper case: "A" ... "Z"
- Digits: "0" ... "9"
- Underscore: "_"

Characteristics

An RTT sequence is implemented in the Python programming language. It is executed together with the simulation application on the simulation platform. Both are executed in parallel. Usually an RTT sequence is not executed completely in one sampling step. The execution normally spans several sampling steps. To distribute the execution of an RTT sequence over several sampling steps, the **yield** statement is used. A **yield None** suspends the execution of an RTT sequence in one sampling step and lets it resume in the next sampling step. All the values of the variables are retained, so the script continues execution directly after the **yield** statement. For more information on **yield**, refer to the Python documentation or [Implementing Generator Functions](#) on page 45.

Generator function

Generator functions allow stepwise execution of a test procedure in which its local variables are retained. Each `yield None` statement marks the boundary between the execution steps.

This is the structure of a generator function:

```
def generator_function():
    ...
    myVar = 1
    ...
    # The execution is paused here
    yield None # sampling step n: the execution of the function is suspended
    ...
    myVar = myVar + 1 # sampling step n+1: The function is resumed here
    ...
```

Every function call to a generator function must also be prefixed with the `yield` statement:

```
yield generator_function()
```

MainGenerator(*args)

The `MainGenerator(*args)` generator is the topmost entry point of the RTT sequence. It is called by the test engine after the initialization phase, when periodic execution begins. Each RTT sequence must have one `MainGenerator(*args)` function that organizes the functionality of the entire RTT sequence at a high level. The function is called by the Python interpreter when an RTT sequence is started.

A `MainGenerator(*args)` function must have at least one `yield` statement:

- A `yield` statement which returns another generator object
- A `yield` statement that simply returns "None"

```
def MainGenerator(*args):
    yield None
```

With the `(*args)` parameter, the host PC Python script passes an argument list containing all the parameters for the RTT sequence. This parameter is optional. Refer to [Starting RTT Sequences with Arguments in Python Scripts](#) on page 185.

Related topics**Basics**

Basics on Running RTT Sequences.....	29
Implementing Generator Functions.....	45
Implementing RTT Sequences.....	43

Basics on Running RTT Sequences

Introduction

The application and the RTT sequences run on the same platform. The platform must calculate the application and scheduled parts of the RTT sequences in one sampling step.

Preconditions

As the platform calculates the application and parts of the RTT sequence, the calculation of the application must therefore leave enough computation time to execute RTT sequences.

Managing RTT sequences

You can manage the RTT sequences from the host PC. The following management functions are possible:

- Creating RTT sequences at the platform
To generate a BCG file, a developer version is required. To create a new RTT sequence for the simulation platform (i.e. downloading the BCG file to it), you must have the developer or operator version.
- Specifying the sequence priorities
The sequence priorities specify the order in which the RTT sequences are executed.
- Starting and stopping the periodic execution of RTT sequences (each RTT sequence individually or all the created RTT sequences)
- Pausing and resuming RTT sequences (each RTT sequence individually or all the created RTT sequences)
- Restarting stopped, terminated, or paused RTT sequences (each RTT sequence individually). An RTT sequence that stopped with an error message cannot be restarted.
When you restart an RTT sequence, its namespace is maintained. The sequence is not initialized but starts directly with executing the **MainGenerator** function.
- Deleting RTT sequences from the platform
- Getting the scripts state (RUNNING, PAUSED, STOPPED, ERROR, etc.)

You can manage the RTT sequences in two ways:

- Using the Real-Time Test Manager
The Real-Time Test Manager provides the basic management functions in a graphical user interface.
- Using `rttmanagerlib` in Python scripts
The `rttmanagerlib` library contains the management function. You can use the library in your Python scripts.

Executing RTT sequences

The running RTT sequences and application are executed consecutively in a simulation step. You can specify whether an RTT sequence is executed before or after the application. The priorities specified when the RTT sequences are created

specify the order of their execution. If RTT sequences have the same priority, they are executed in the reverse order in which they are downloaded to the platform. The RTT sequence created last is then executed first in a sampling step.

The platform computes RTT sequences and application in one simulation step in the following order:

1. Start a simulation step
2. The (PreComputation) RTT sequences are executed.
3. The simulation model is calculated.
4. The (PostComputation) RTT sequences are executed.
5. End of simulation step

If the execution time of RTT sequences within a PreComputation or a PostComputation channel is longer than 1 s, a TestOverrun exception occurs. For details to the exception, refer to [A TestOverrun Exception Occurs](#) on page 216.

RTT sequence behavior on task overrun

If Ignore overrun is selected in the real-time model, the RTT sequences are executed synchronously with the model but real-time is not guaranteed anymore. If the Current time variable is used and an overrun is ignored, Current time is delayed and does not correspond with the actual time anymore.

Example If a real-time model is executed with a step size of 0.1 seconds and the model runs in real-time, 10 steps of the model and the RTT sequence are executed per second. If Ignore overrun is selected and an overrun occurs, for example, only 5 steps of the model and the RTT sequence are executed per second and the model and the RTT sequence are not running in real-time. If the RTT sequence is set to poll 10 seconds on the Current time variable, an ignored overrun causes the RTT sequence to poll 20 seconds but output a polling time of 10 seconds.

Refer to [How to Specify Overrun Strategies for Tasks \(RTI and RTI-MP Implementation Guide\)](#), and [RTI Task Configuration Dialog \(RTI and RTI-MP Implementation Reference\)](#) or [RTI Task Configuration Dialog \(Multiprocessor Setup Dialog\) \(RTI and RTI-MP Implementation Reference\)](#).

RTT sequence behavior on simulation stop

When the simulation is stopped, all running RTT sequences are stopped, too. Restarting or pausing the simulation does not restart or continue the RTT sequences. This prevents the simulation and the RTT sequences running unsynchronously. Otherwise, the RTT sequences would resume their execution at the point where they stopped.

TestOverrun exception

If the execution time of RTT sequences within one PreComputation or PostComputation channel is longer than 1 s, an TestOverrun exception occurs. The error message of the exception gives you information where it has occurred (name of the RTT sequence, code line and function name).

Related topics**Basics**

[Managing RTT Sequences in Python Scripts](#)..... 181

Workflow for Real-Time Testing

Introduction

The following overview shows you how real-time tests are performed.

Workflow

Real-time testing is performed in several steps:

1. Prepare the simulation application
A simulation model does not need to be changed for real-time testing. However, it must contain the service call to the Python interpreter. The simulation application usually contains the service call by default. If not, you can enable Real-Time Testing manually. For instructions, refer to [Enabling Real-Time Testing for dSPACE Platforms](#) on page 33.
2. Implementing an RTT sequence
An RTT sequence is a Python script which is executed on the platform. For information on how to implement an RTT sequence, refer to [Implementing RTT Sequences](#) on page 43.
3. Generate the BCG file
A BCG file contains the RTT sequence and all the modules which are imported by the RTT sequence. Only signed BCG files can be downloaded to the platform.
4. Executing the real-time test
When the implementation of the RTT sequence is finished, you can download it to the simulation platform for executing the test. You can control the execution using a Python script on the host or the Real-Time Test Manager. Refer to [Managing RTT Sequences in Python Scripts](#) on page 181.
5. Error management and troubleshooting
If your real-time test does not work in the desired way, you can stop the execution and change the implementation of the RTT sequence.
 - For tips and tricks for implementing, refer to [Writing Effective RTT Sequences](#) on page 162.
 - For information on how you can debug an RTT sequence, refer to [Debugging RTT Sequences](#) on page 198.
 - For information on general errors, refer to [Using Sleep\(\) Function](#) on page 226.

Enabling Real-Time Testing for dSPACE Platforms

Introduction

If Real-Time Testing is not enabled by default, you can enable it manually.

Where to go from here

Information in this section

[Basics on Enabling Real-Time Testing..... 34](#)

Provides some basic information on how to enable Real-Time Testing for dSPACE platforms.

[How to Enable Real-Time Testing for a DS1006 or MicroAutoBox II..... 34](#)

Provides instructions on how to enable Real-Time Testing on a DS1006 single-processor or MicroAutoBox.

[How to Enable Real-Time Testing for a DS1006 Multiprocessor System..... 36](#)

Provides instructions on how to enable Real-Time Testing on multiprocessor systems based on a DS1006.

[How to Enable Real-Time Testing for a SCALEXIO System..... 37](#)

Provides instructions on how to enable Real-Time Testing on a SCALEXIO Processing Unit or a DS6001 Processor Board (single-processor, multicore and multiprocessor systems).

[How to Enable Real-Time Testing for MicroAutoBox III..... 38](#)

Provides instructions on how to enable Real-Time Testing on a MicroAutoBox III.

[Real-Time Testing and VEOS..... 41](#)

You can use VEOS for real-time testing. Real-time testing is always enabled in offline simulation application.

Basics on Enabling Real-Time Testing

Basics	A simulation model does not have to be modified if you want to use it for real-time testing, but its simulation application (real-time application for real-time platforms or offline simulation application for VEOS) must contain the service call to the Python interpreter.
Preconditions for the real-time system	For information on the hardware requirements, refer to Hardware and Software Requirements on page 18.
Default settings	Real-Time Testing is enabled for most dSPACE platforms by default. For MicroAutoBox III, Real-Time Testing is disabled by default.
Enabling Real-Time Testing	<p>To support Real-Time Testing, you need to insert a service call in the simulation application. This is done during the build process.</p> <p>The kind of application you have to use depends on the platform type, see:</p> <ul style="list-style-type: none"> ▪ How to Enable Real-Time Testing for a DS1006 or MicroAutoBox II on page 34 ▪ How to Enable Real-Time Testing for a DS1006 Multiprocessor System on page 36 ▪ How to Enable Real-Time Testing for a SCALEXIO System on page 37 ▪ How to Enable Real-Time Testing for MicroAutoBox III on page 38 <p>It is not necessary to enable Real-Time Testing for VEOS. Real-Time Testing is always enabled on this platform.</p>
Related topics	<p>Basics</p> <p>Real-Time Testing and VEOS..... 41</p>

How to Enable Real-Time Testing for a DS1006 or MicroAutoBox II

Objective	If you want to use Real-Time Testing on a DS1006 single-processor system or MicroAutoBox II, the support must be enabled before code generation.
------------------	--

Enabling Real-Time Testing support

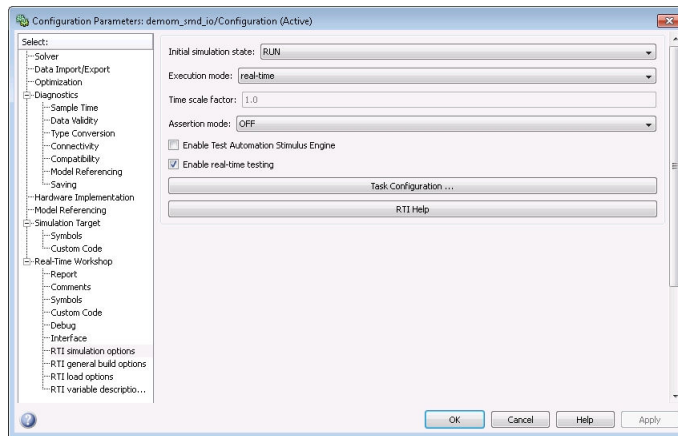
For DS1006 single-processor system or MicroAutoBox II, you enable support on the RTI simulation options page of the Code Generation dialog in MATLAB/Simulink.

Method

To enable Real-Time Testing for a DS1006 or MicroAutoBox II

- 1 Open the real-time model in MATLAB/Simulink.
- 2 Open the configuration page of Code Generation.
- 3 On the RTI simulation options page, select Enable real-time testing.

The following illustration shows the settings in MATLAB.



- 4 Build and download the real-time application.

Tip

To check the turnaround time of the real-time application (Timer Task 1), the variable description file contains the `turnaroundTime` variable which you can read in ControlDesk. For details, refer to [turnaroundTime \(RTI and RTI-MP Implementation Reference\)](#).

Result

You can use the real-time application for real-time testing.

Related topics

References

[Code Generation Dialog \(Model Configuration Parameters Dialogs\) \(RTI and RTI-MP Implementation Reference\)](#)

How to Enable Real-Time Testing for a DS1006 Multiprocessor System

Objective

If you want to use Real-Time Testing on a DS1006 multiprocessor system, the support must be enabled before code generation.

Enabling Real-Time Testing support

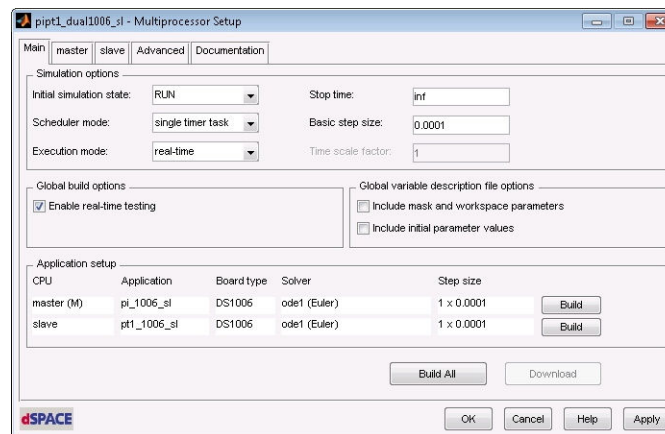
In a DS1006 multiprocessor system, you enable support on the Main page of the Multiprocessor Setup block in MATLAB/Simulink.

Method

To enable Real-Time Testing for a DS1006 multiprocessor system

- 1 Open the real-time model in MATLAB/Simulink.
- 2 Double-click the Multiprocessor Setup block.
- 3 On the Main page, select Enable real-time testing.

The following illustration shows the settings in MATLAB.



This enables real-time testing for each CPU of your system.

- 4 Build and download the real-time application.

Tip

To check the turnaround time of the real-time application (Timer Task 1), the variable description file contains the `turnaroundTime` variable, which you can read in ControlDesk. For details, refer to [turnaroundTime \(RTI and RTI-MP Implementation Reference\)](#).

Result

You can use the real-time application for real-time testing.

Related topics

References

[Multiprocessor Setup Dialog \(RTI and RTI-MP Implementation Reference !\[\]\(bd1a142de767a21e5362c595f844a4ff_img.jpg\)](#))

How to Enable Real-Time Testing for a SCALEXIO System

Objective

If you want to use Real-Time Testing on a SCALEXIO system, the support must be enabled before code generation. The processing unit of a SCALEXIO system can be a SCALEXIO Processing Unit or a DS6001 Processor Board. It can be used as single-processor, multicore, or multiprocessor system.

Enabling Real-Time Testing support

For a SCALEXIO system, Real-Time Testing support must be enabled in ConfigurationDesk. When a model is added to the application, the preconfigured application process is used by default. This process enables Real-Time Testing for the fastest periodic task.

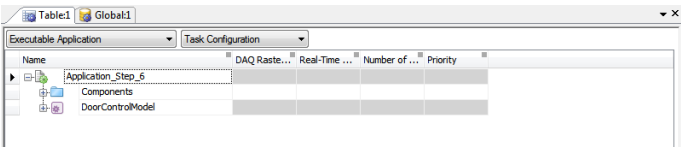
Real-Time Testing must be enabled for exactly one task. If it is not enabled or enabled for multiple tasks, a conflict occurs and the real-time application cannot be built.

Method

To enable Real-Time Testing for a SCALEXIO system

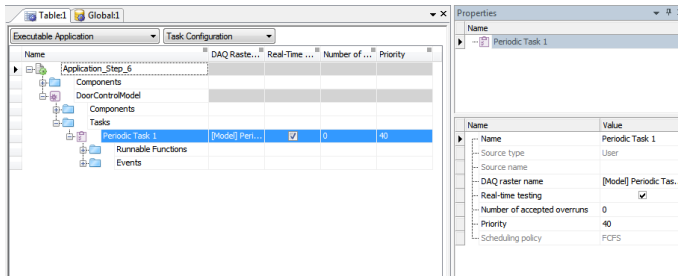
- 1 In ConfigurationDesk, open the ConfigurationDesk application which corresponds to the real-time application.
- 2 Choose Build – Open Table Executable Application.

A table window opens with the Executable Application view. The view displays the settings of the real-time application, for example, the tasks.



- 3 In the Executable Application view, select the task to which you want to synchronize Real-Time Testing.
The Properties Browser displays the properties of the selected task.

4 In the Properties Browser, select Real-time testing.



Real-Time Testing must be enabled for only one task; you can check this in the Conflicts Viewer.

- 5 Open the Conflicts Viewer and search for the Application process: Multiple tasks with Real-Time Testing conflict. If you have found this conflict, resolve the conflict in the Conflicts Viewer or the Executable Application view. For details of resolving conflicts, refer to [Resolving Conflicts \(ConfigurationDesk Real-Time Implementation Guide !\[\]\(065aacad479feea1b3f501fa02b79a7a_img.jpg\)](#)).
- 6 Build and download the real-time application.

Tip

To check the turnaround time of the task selected for Real-Time Testing, the variable description file contains the **Task Turnaround Time** variable, which you can read in ControlDesk. For details, refer to [Simulation and RTOS Group \(ConfigurationDesk Real-Time Implementation Guide !\[\]\(5a132f13505a6571904d622757b7a8f0_img.jpg\)](#)).

Result

You can use the real-time application for real-time testing.

Related topics

Basics

[Modeling Executable Applications and Tasks \(ConfigurationDesk Real-Time Implementation Guide !\[\]\(35dc653d59570f8f891c312eeece91a2_img.jpg\)](#))

References

[Executable Application Table \(ConfigurationDesk User Interface Reference !\[\]\(104fbf564e2e5a8fbd84f31656d114c7_img.jpg\)](#))

How to Enable Real-Time Testing for MicroAutoBox III

Objective

If you want to use Real-Time Testing on MicroAutoBox III, the support must be enabled.

Enabling Real-Time Testing support

Real-Time Testing must be enabled for the real-time application and on the MicroAutoBox III.

- Real-Time Testing is enabled for the fastest periodic task by default. You can enable RTT for exactly one task. RTT is automatically disabled for the other tasks. You can disable or enable support in ConfigurationDesk. Refer to [Part 1](#) on page 39.
- You can disable or enable the use of Real-Time Testing via the web interface of MicroAutoBox III. Real-Time Testing is disabled for MicroAutoBox III by default. Refer to [Part 2](#) on page 40.

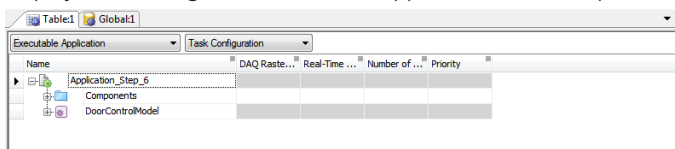
Part 1

To enable Real-Time Testing in the real-time application implemented for MicroAutoBox III

- In ConfigurationDesk, open the ConfigurationDesk application which corresponds to the real-time application.

- Choose Build – Open Table Executable Application.

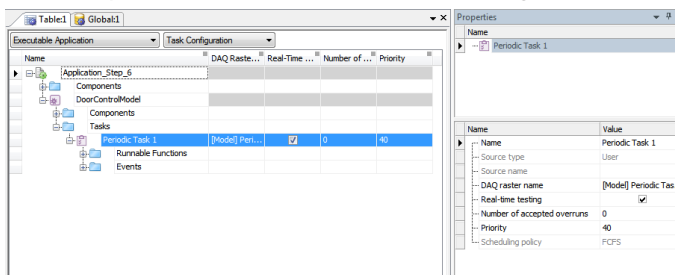
A table window opens with the Executable Application view. The view displays the settings of the real-time application, for example, the tasks.



- In the Executable Application view, select the task to which you want to synchronize Real-Time Testing.

The Properties Browser displays the properties of the selected task.

- In the Properties Browser, select Real-time testing.



Real-Time Testing must be enabled for only one task; you can check this in the Conflicts Viewer.

- Open the Conflicts Viewer and search for the Application process: Multiple tasks with Real-Time Testing conflict. If you have found this conflict, resolve the conflict in the Conflicts Viewer or the Executable Application view. For details of resolving conflicts, refer to [Resolving Conflicts \(ConfigurationDesk Real-Time Implementation Guide\)](#).

6 Build and download the real-time application.

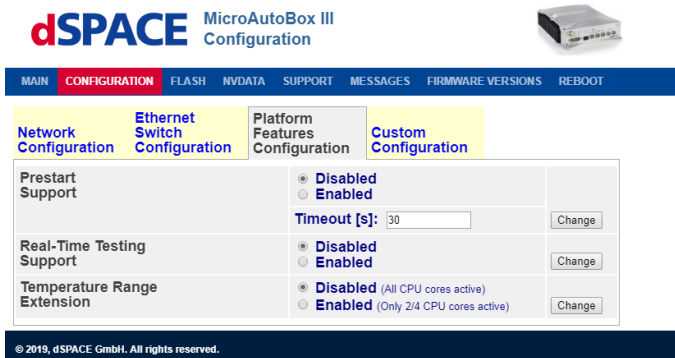
Tip

To check the turnaround time of the task selected for Real-Time Testing, the variable description file contains the **Task Turnaround Time** variable, which you can read in ControlDesk. For details, refer to [Simulation and RTOS Group \(ConfigurationDesk Real-Time Implementation Guide !\[\]\(3d8c13c92b853674f749aac6fa869926_img.jpg\)](#)).

Part 2

To enable Real-Time Testing on MicroAutoBox III platform

- 1 Open the web interface. Refer to [How to Open the Web Interface \(MicroAutoBox III Hardware Installation and Configuration !\[\]\(750841ae7100dc832cb0a4b3af4492f3_img.jpg\)](#)).
- 2 Open the CONFIGURATION – Platform Features Configuration page.



- 3 At Real-Time Testing Support, select Enabled.
- 4 Click Change.
- 5 Open the MicroAutoBox III System Reboot page and click Restart.
The new configuration becomes active after you restart the MicroAutoBox III.

Result

You enabled Real-Time Testing.

Related topics

References

[CONFIGURATION – Platform Features Configuration Page \(MicroAutoBox III Hardware Installation and Configuration !\[\]\(e9474ce1d70442456f8fe9c393ea149c_img.jpg\)](#))

Real-Time Testing and VEOS

Introduction

You can use VEOS for real-time testing. Real-time testing is always enabled in offline simulation application.

VEOS and Real-Time Testing versions

The following table displays the release number and versions of Real-Time Testing (RTT) and VEOS contained in the Release.

dSPACE Release	VEOS Version	RTT Version	Remarks
RLS2021-A	5.2	5.0	–
RLS2020-B	5.1	4.4	–
RLS2020-A	5.0	4.3	–
RLS2019-B	4.5	4.2	Real-Time Testing supports VEOS 4.5 with the Python 3.6.4 interpreter.
RLS2019-A	4.4	4.1	Real-Time Testing supports VEOS 64-bit applications.
RLS2018-B	4.3	4.0	Real-Time Testing supports multiple V-ECUs on VEOS
RLS2018-A	4.2	3.4	–
RLS2017-B	4.1	3.3	–
RLS2017-A	4.0	3.2	VEOS running on a remote PC is supported.
RLS2016-B	3.7	3.1	–
RLS2016-A	3.6	3.0	–
RLS2015-B	3.5	2.6	–
RLS2015-A	3.4	2.5	Real-Time Testing supports V-ECUs on VEOS. You must consider some limitations. Refer to General Limitations for Real-Time Testing on page 229.
RLS2014-B	3.3	2.4	Real-Time Testing can be used in all environment models. Real-Time Testing support is always enabled.
RLS2014-A	3.2	2.3	VEOS supports several environment models, but Real-Time Testing can only be used in one environment model. Real-Time Testing support must be enabled in the offline simulation application.
RLS2013-B	3.1	2.2	Real-Time Testing support is always enabled for offline simulation applications.

Related topics

Basics

[Basics on Enabling Real-Time Testing](#)..... 34

Implementing RTT Sequences

Introduction

This section describes how to implement RTT sequences for real-time testing, with simple examples. It is recommended to read the topics in the specified order.

Where to go from here

Information in this section

[Implementing Generator Functions.....45](#)

Generator functions are the basis of real-time testing. These are the functions which are executed in steps.

[General Information on Implementing RTT Sequences.....57](#)

Describing how you can program simple RTT sequences.

[Exception Handling and Using Modules.....73](#)

You can implement your own modules and use standard Python modules.

[Exchanging Data Between RTT Sequence and Host Python Script.....77](#)

Host calls can transfer Python data objects from the simulation platform to the host PC and vice versa.

[Data Replay in RTT Sequences.....87](#)

You can stimulate variable objects in an RTT sequence by data replay of MAT file variables or ASAM MDF file channels.

[Handling CAN Messages.....104](#)

You can send and receive CAN messages in the raw format.

[Implementing Communication via Ethernet.....137](#)

You can receive and transmit messages via Ethernet.

[Implementing a Communication via a Serial Interface.....144](#)

You can send and receive data via a serial interface.

Accessing Variables of a Simulation Application on a Remote Node.....	150
An RTT sequence can access variables of a simulation application running on another node (processor board in a multiprocessor system, CPU on a multicore board, or a remote VPU in an offline simulation application).	
Tips and Tricks.....	162
Giving some tips and tricks for implementing RTT sequences.	

Information in other sections

Basics of Real-Time Testing.....	25
Describes the software components, RTT sequences, and how Real-Time Testing is integrated into a dSPACE system. Also explains the workflow for implementing real-time tests and how a real-time application must be prepared.	
Managing RTT Sequences.....	165
Describes how you can execute RTT sequences on dSPACE platforms. It explains how to start and monitor one or more RTT sequences and provides information on how you can debug an RTT sequence.	
Demo Examples of Using Real-Time Testing.....	22
Some demo examples are installed with Real-Time Testing. Examine these examples to learn about the features of Real-Time Testing.	

Implementing Generator Functions

Introduction

Generator functions are the basis of real-time testing. These are the functions which are executed step-by-step. This section introduces the use of generator functions and explains how nested execution can be implemented.

Where to go from here

Information in this section

[Basics on Generator Functions.....46](#)

Generator functions are more powerful and flexible than normal Python functions, because they can suspend and resume their execution later.

[Using Generator Functions.....47](#)

Generator functions can be executed concurrently in an RTT sequence.

[Implementing an RTT Sequence Using User-Defined Generator Functions.....49](#)

Usually, RTT sequences contain generator functions. Generator functions can suspend their execution and resume it in the next simulation step.

[Terminating a Generator Function.....51](#)

You can terminate a generator function by different methods.

[Using the Parallel\(\) Generator Function.....53](#)

You can execute several generator functions in parallel. If you use the Parallel() generator function, the RTT sequence continues only when all the embedded generator functions have finished.

[Using the ParallelRace\(\) Generator Function.....54](#)

You can execute several generator functions in parallel. If you use the ParallelRace() generator function, the RTT sequence continues when the first of the embedded generator functions has finished.

Information in other sections

[General Information on Implementing RTT Sequences.....57](#)

Describing how you can program simple RTT sequences.

[Demo Examples of Using Real-Time Testing.....22](#)

Some demo examples are installed with Real-Time Testing. Examine these examples to learn about the features of Real-Time Testing.

Basics on Generator Functions

Basics

Generator functions are more powerful and flexible than normal Python functions, because they can suspend and resume their execution later. Normal Python functions either "return" or throw an exception. In both cases the function's state (local variable and its instruction pointer) is lost:

```
def function_interval(a,b):
    i = a
    while i < b:
        i += 1          # increment local variable
        print(" i = ", i)
    return
>>> function_interval(4,8)
i = 5
>>>
```

A normal Python function becomes a generator function if it contains a **yield** statement. Generator functions are different because they preserve their state across calls. This enables them to suspend and resume their execution in the middle of while loops or other control structures. To resume a generator function after suspension, its state is saved in a dedicated "generator object". The generator object can be regarded as an instance of a generator function, for example:

```
def generate_interval(a, b):
    j = a
    while j < b:
        j += 1          # increment local variable
        print(" j = ", j)
        yield None      # suspend execution
>>> gen_obj = generate_interval(4, 8) # create generator object
>>> print(gen_obj)
<generator object at 0x01D52E68>
```

Every generator object automatically provides a **next()** function. Calling the **next()** function resumes execution of the generator object where it was suspended during the last call to **next()**.

```
>>> gen_obj.next()
j = 5
>>> gen_obj.next()
j = 6
>>> gen_obj.next()
j = 7
>>> gen_obj.next()
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
StopIteration
```

The **StopIteration** exception signals to the caller that the generator object is ended and it does not make sense to call it again. Another advantage of generator functions is the fact that multiple independent instances of a generator function can be created:

```
>>> gen_obj1 = generate_interval(4, 7) # create generator object
>>> print(gen_obj1)
<generator object at 0x01D52E68>
```

```

>>> gen_obj2 = generate_interval(10, 13) # create 2nd generator object
>>> print(gen_obj2)
<generator object at 0x01D52F88>
>>> gen_obj1.next() #advance 1st generator object
j = 5
>>> gen_obj2.next() #advance 2nd generator object
j = 11
>>> gen_obj1.next()
j = 6
>>> gen_obj2.next()
j = 12
>>> gen_obj1.next()
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
StopIteration
>>> gen_obj2.next()
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
StopIteration

```

Generator functions are ideally suited for RTT sequences, because their stepwise execution allows fine grained synchronization with the periodic calculation of the real-time model. Preserving local variables across calls is another important factor that simplifies the RTT sequences and speeds up their execution.

Related topics

Basics

Implementing an RTT Sequence Using User-Defined Generator Functions.....	49
Terminating a Generator Function.....	51
Using Generator Functions.....	47

Using Generator Functions

Nesting generator functions

The step-wise execution of generator objects can be seen as a form of cooperative multitasking. This allows the concurrent execution of multiple test "threads" within an RTT sequence. The tedious task of calling `next()` for each generator function is performed by the dSPACE script scheduler, so you can concentrate on specifying the test logic. The dSPACE script scheduler advances each active generator function exactly once per simulation step. You can therefore design the generator on this assumption. Another feature of the dSPACE script scheduler is support for nested generator functions. A nested generator function is a generator function which calls another (sub) generator function, that is, the execution of the outer generator function is suspended as long as the inner generator is active. The dSPACE scheduler supports infinite nesting levels.

```

from rttlib import variable
from rttlib import utilities
CurrentTime = utilities.currentTime
SpringConstant = variable.Variable(r'Model Root/Model Parameters/C/Value')

```

```

# wait for 'duration' seconds
def inner_generator(duration):
    start = CurrentTime.Value
    while (CurrentTime.Value < (start + duration)):
        yield None
# outer generator function
def outer_generator(signal):
    # do something
    signal.Value = 1.0
    # wait for 10 ms
    yield inner_generator(0.01)
    # do something
    signal.Value = 2.0
    # wait for 10 ms
    yield inner_generator(0.01)
    # do something
    signal.Value = 3.0
# Root of the generator tree
# and topmost entry point for the scheduler.
# The name MainGenerator is fixed
def MainGenerator(*args):
    # call outer generator
    yield outer_generator(SpringConstant)

```

As the example shows, calling a (sub) generator function requires a **yield** statement before the instantiation of the generator object. This is a requirement because it is the only way the caller (the outer generator function) can be suspended until the inner generator function finishes. Omitting the **yield** statement would create the inner generator object, but as it is not assigned to a variable (or "bound to name" as this is often called in Python), the newly created object would be discarded immediately. Even worse, the outer generator function would continue execution instead of waiting for the inner generator to complete.

Nesting generator objects as described above creates a hierarchical tree structure with exactly one generator function at its root (the outermost generator function). This outermost generator function must have a specific name, because it is the first generator function that is called by the dSPACE script scheduler. The name is **MainGenerator(*args)**, and it has to be present in every RTT sequence.

The dSPACE script scheduler starts the RTT sequence's real-time part at the beginning of **MainGenerator(*args)** and descends into its (sub) generator functions until the end of **MainGenerator(*args)** is reached. The end of the **MainGenerator(*args)** function is also the end of the whole RTT sequence.

You do not have to call the dSPACE script scheduler or the **MainGenerator(*args)** function explicitly. This is done automatically by loading an RTT sequence to the embedded Python interpreter and by starting it with the Real-Time Test Manager.

Concurrency within an RTT sequence

The mechanisms (including nested generator) described above only allow the execution of a single generator step in each simulation step. Nested generator functions, however, permit concurrent execution of multiple generator objects within a single simulation. The **Parallel()** and **ParallelRace()** generator

functions provide the means to execute multiple generator functions within a simulation step. They take a list of (sub) generator functions as arguments and execute them in parallel, that is, advance each (sub) generator exactly once per sampling step. `Parallel()` and `ParallelRace()` have different termination conditions. For further details, refer to [Using the Parallel\(\) Generator Function](#) on page 53 and [Using the ParallelRace\(\) Generator Function](#) on page 54.

Finally statement

It is not allowed to use `yield` in a `finally` statement. If you do so, a warning ("Runtime error: generator ignored GeneratorExit") is given in the Log Viewer and some objects may not be cleared.

Limitations for accuracy when executing RTT sequences

When executing an RTT sequence, the latencies can be erroneous because of the inaccurate representation of floating point numbers. For details on the floating point representation and examples, refer to *B. Floating Point Arithmetic: Issues and Limitations* in the *Python reference*.

Related topics

Basics

[Basics on Generator Functions.....](#) 46

Implementing an RTT Sequence Using User-Defined Generator Functions

Introduction

Usually, RTT sequences contain generator functions. Generator functions can suspend their execution and resume it in the next simulation step.

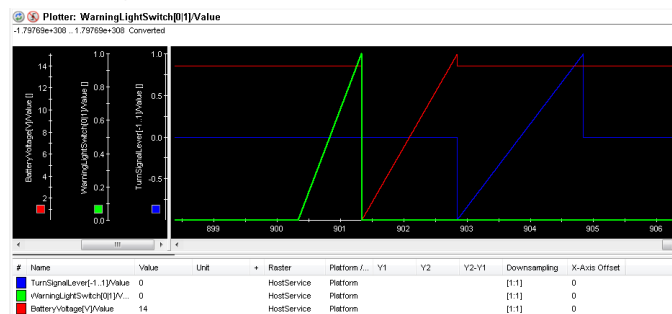
Example

The following example is an RTT sequence for the TurnSignal demo application. The demo application is located in `<MyDocuments>\dSPACE\Real-Time Testing\5.0\SampleExperiments\TurnSignal_<platform>`.

```
from rttlib import variable
WarningLightSwitch = variable.Variable(r'Model Root/WarningLightSwitch[0][1]/Value')
BatteryVoltage      = variable.Variable(r'Model Root/BatteryVoltage[V]/Value')
TurnSignalLever     = variable.Variable(r'Model Root/TurnSignalLever[-1..1]/Value')
def GenerateRampGen(Variable, MinVal, MaxVal, ValueToAdd):
    Variable.Value = MinVal
    while(Variable.Value <= MaxVal):
        # in each step the value is increased.
        Variable.Value += ValueToAdd
        # The execution is suspended here.
        yield None
        # The execution is continued here.
```

```
def MainGenerator(*args):
    # stimulate WarningLightSwitch
    yield GenerateRampGen(WarningLightSwitch, 0.0, 1.0, 0.001)
    WarningLightSwitch.Value = 0.0
    # stimulate BatteryVoltage
    yield GenerateRampGen(BatteryVoltage, 0.0, 15.0, 0.01)
    BatteryVoltage.Value = 14.0
    # stimulate TurnSignalLever
    yield GenerateRampGen(TurnSignalLever, -1.0, 1.0, 0.001)
    TurnSignalLever.Value = 0.0
```

The following illustration shows the capture result of the variables stimulated by the RTT sequence in ControlDesk.



Description

There are two functions in this example. The `GenerateRampGen()` is the user-defined generator function. It raises the value of a variable (`Variable`) from a minimum value (`MinVal`) to a maximum value (`MaxVal`) by a specified increment (`ValueToAdd`). After changing the value, a `yield None` statement is executed, which suspends the generator function for the current simulation step until the next sampling step.

The `MainGenerator(*args)` function uses the `GenerateRampGen()` generator function to stimulate the value of `WarningLightSwitch`, `BatteryVoltage`, and `TurnSignalLever`. These are parameters of the real-time application that are defined in the variable description file (TRC file). They can also be found in ControlDesk's Variables controlbar. The parameters are changed one after another (in a sequence). Note that `yield` must be inserted before the function call of `GenerateRampGen()`. This is required for calling a generator function.

Tip

You can label generator functions for easier identification, for example, by using function names which end with `Gen`. This makes it easy to identify which functions have to be called with a `yield` statement as prefix.

Related topics**Basics**

[Basics on RTT Sequences..... 27](#)

Examples

[Demo Examples of Using Real-Time Testing..... 22](#)

Terminating a Generator Function

Introduction

You can terminate a generator function by several methods.

Termination

A single generator function terminates in one of the following cases:

- It terminates with **return**.
- It reaches the end of the generator function (if it is a function with no explicit return statement).

The whole RTT sequence terminates in one of the following cases:

- The topmost generator (**MainGenerator(*args)**) terminates.
- A (fatal) exception occurs at any level.
- The RTT sequence is STOPPED or REMOVED via the Real-Time Test Manager.

The running RTT sequences are not automatically stopped when the simulation is stopped.

Example

The following example shows how generator functions are terminated.

```
def TerminateMethod_1_Gen():
    Loops    = 1000
    Counter  = 0
    while(Counter < Loops):
        Counter += 1
        yield None
    print("First while loop is ended.")
    # The function is terminated here
```

```

def TerminateMethod_2_Gen():
    Loops = 1000
    Counter = 0
    while(1):
        if(Counter > Loops):
            break
        Counter += 1
        yield None
    # The following code is executed after the loop
    print("Second while loop is ended.")
    yield None
    # The function is terminated here
def TerminateMethod_3_Gen():
    Loops = 1000
    Counter = 0
    while(1):
        if(Counter > Loops):
            # The function is terminated here
            return
        Counter += 1
        yield None
    # The following code is never executed
    print("Third while loop is ended.")
def MainGenerator(*args):
    yield TerminateMethod_1_Gen()
    yield TerminateMethod_2_Gen()
    yield TerminateMethod_3_Gen()
    yield None

```

Description

Generator functions are terminated when the end of the function definition is reached or `return` is called explicitly. The `TerminateMethod_2_Gen()` and `TerminateMethod_3_Gen()` functions have an endless while loop. The while loop is only terminated when `break` or `return` is called. `break` terminates only the while loop. `return` terminates the function.

The `MainGenerator(*args)` function calls the generator functions. Note that `yield` must be inserted before the function call.

Related topics

Basics

Basics on Generator Functions.....	46
Implementing an RTT Sequence Using User-Defined Generator Functions.....	49
Using Generator Functions.....	47

Using the Parallel() Generator Function

Introduction

You can execute several generator functions in parallel. All generator functions that are part of a `Parallel()` construct are calculated in each simulation step. If you use the `Parallel()` generator function, the part of the RTT sequence containing the `Parallel()` function continues only when all the embedded generator functions have finished.

Parallel()

A `Parallel()` generator function calls two or more embedded generator functions. Each embedded generator function is executed once in a simulation step. They are executed in consecutively in the order specified in `Parallel()`'s argument list, starting from the last entry of the list. From the perspective of the real-time application, however, they are executed in parallel, because the actions take place in a single simulation step.

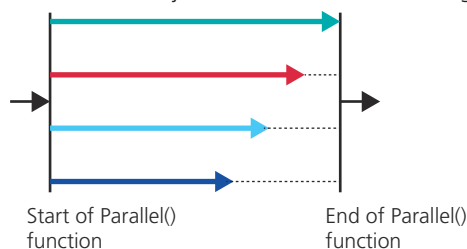
If the RTT sequence must continue when the first embedded generator function has finished, use the `ParallelRace()` generator function, see [Using the ParallelRace\(\) Generator Function](#) on page 54.

Example

The following RTT sequence shows the use of a `Parallel()` generator function.

```
from rttlib import scheduler
from rttlib import variable
WarnLightSwitch = variable.Variable(r'Model Root/WarnLightSwitch[0|1]/Value')
BatteryVoltage = variable.Variable(r'Model Root/BatteryVoltage[V]/Value')
TurnSignalLever = variable.Variable(r'Model Root/TurnSignalLever[-1..1]/Value')
def GenerateRampGen(Variable, MinVal, MaxVal, ValueToAdd):
    Variable.Value = MinVal
    while (Variable.Value <= MaxVal):
        # In each step the value is increased.
        Variable.Value += ValueToAdd
        # The execution is suspended here.
        yield None
        # The execution is continued here.
def MainGenerator(*args):
    # test the parallel() generator function
    yield scheduler.Parallel(GenerateRampGen(WarnLightSwitch, 0.0, 1.0, 0.001),\
                             GenerateRampGen(BatteryVoltage, 0.0, 15.0, 0.01),\
                             GenerateRampGen(TurnSignalLever, -1.0, 1.0, 0.001))
```

The following illustration shows the times when the `Parallel()` function starts and ends in conjunction with its embedded generator functions.



Description

The `TestParallelGen()` function contains a `Parallel()` generator function containing three generator functions (`GenerateRampGen()` for different variables). The embedded generator functions increase the specified variables from a minimum to maximum value in a specified step size. They require different numbers of simulation steps to do this. The following table shows the parameters passed to `GenerateRampGen` and the resulting number of simulation steps.

Variable	Min. Value	Max. Value	Step	Number of Steps
WarnLightSwitch	0.0	1.0	0.001	1,000
BatteryVoltage	0.0	15.0	0.01	1,500
TurnSignalLever	-1.0	1.0	0.001	2,000

Because a `Parallel()` generator function is used, it is ended after 2,000 simulation steps. The last simulation step is the one in which all the embedded function generators have finished.

Related topics**Basics**

Implementing an RTT Sequence Using User-Defined Generator Functions.....	49
Read/Write Access to Variables of the Simulation Application.....	62

Using the `ParallelRace()` Generator Function

Introduction

You can execute several generator functions in parallel. All generator functions that are part of a `ParallelRace()` construct are calculated in each simulation step. If you use the `ParallelRace()` generator function, the part of the RTT sequence containing `ParallelRace()` continues when the first of the embedded generator functions has finished.

`ParallelRace()`

A `ParallelRace()` generator function calls two or more embedded generator functions. Each embedded generator function is executed once in a simulation step. They are executed in consecutively in the order specified in `ParallelRace()`'s argument list, starting from the last entry of the list. From the perspective of the real-time application, however, they are executed in parallel. When the first embedded generator function is finished, all the other embedded generator functions are terminated.

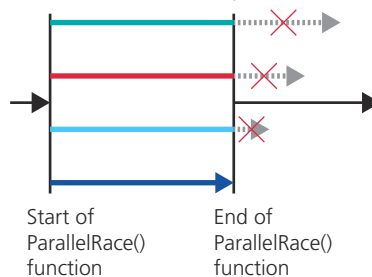
If the RTT sequence must continue when all the embedded generator functions have finished, use the `Parallel()` generator function, see [Using the `Parallel\(\)` Generator Function](#) on page 53.

Example

The following RTT sequence shows the use of a `ParallelRace()` generator function.

```
from rttlib import scheduler
from rttlib import variable
WarnLightSwitch = variable.Variable(r'Model Root/WarningLightSwitch[0|1]/Value')
BatteryVoltage = variable.Variable(r'Model Root/BatteryVoltage[V]/Value')
TurnSignalLever = variable.Variable(r'Model Root/TurnSignalLever[-1..1]/Value')
def GenerateRampGen(Variable, MinVal, MaxVal, ValueToAdd):
    Variable.Value = MinVal
    while (Variable.Value <= MaxVal):
        # In each step the value is increased.
        Variable.Value += ValueToAdd
        # The execution is suspended here.
        yield None
        # The execution is continued here.
def MainGenerator(*args):
    # test the parallel() generator function
    yield scheduler.ParallelRace(GenerateRampGen(WarnLightSwitch, 0.0, 1.0, 0.001),\
                                GenerateRampGen(BatteryVoltage, 0.0, 15.0, 0.01),\
                                GenerateRampGen(TurnSignalLever, -1.0, 1.0, 0.001))
```

The following illustration shows the times when the `ParallelRace()` function starts and ends in conjunction with its embedded generator functions.

**Description**

The `TestParallelRaceGen()` function contains the function calls to three generator functions (`GenerateRampGen()` for different variables). The embedded generator functions increase the specified variables from a minimum to maximum value in a specified step size. They require different numbers of simulation steps to do this, see the following table.

Variable	Min. Value	Max. Value	Step	Number of Steps
WarnLightSwitch	0.0	1.0	0.001	1,000
BatteryVoltage	0.0	15.0	0.01	1,500
TurnSignalLever	-1.0	1.0	0.001	2,000

Because the generator functions are embedded in `ParallelRace()`, they are ended after 1,000 simulation steps. This is the simulation step in which the first embedded generator function finishes.

Related topics

Basics

Implementing an RTT Sequence Using User-Defined Generator Functions.....	49
Read/Write Access to Variables of the Simulation Application.....	62

General Information on Implementing RTT Sequences

Introduction

This section describes how you can program RTT sequences. It gives several short examples of an RTT sequence.

Where to go from here

Information in this section

[Encoding the Scripts of RTT Sequences..... 58](#)

Real-Time Testing uses the UTF-8 coding for the Python interpreter on the simulation platform. You must consider this for the encoding of the files of the RTT sequences.

[Simple RTT Sequence..... 59](#)

The template has the minimum structure of an RTT sequence. You can use it as a basis for your own RTT sequence.

[Getting Properties of an RTT Sequence..... 60](#)

The SequenceProperties class of the rttlib.utilities module provides some properties of the RTT sequence.

[Using Local and Global Variables in RTT Sequences..... 61](#)

If the function definitions use local instead of global variables, the function runs faster.

[Using Variables Accessible by Several RTT Sequences..... 62](#)

You can use global variables which are accessible by several RTT sequences, so that you can exchange data between RTT sequences.

[Read/Write Access to Variables of the Simulation Application..... 62](#)

Real-Time Testing provides access to the variables of a simulation application. You have read/write access at any point during execution of the RTT sequence. This enables perfect monitoring and stimulation of signals, because of the stepwise execution of the RTT sequence.

[Checking Conditions According to the ASAM GES Standard..... 65](#)

RTT sequences can be paused and resumed depending on conditions that can be specified according to the ASAM General Expression Syntax (GES) standard.

[Printing Variable Values of an RTT Sequence on the Host PC..... 68](#)

The values of variables can be read in the Log Viewer of the Real-Time Test Manager.

[Printing Messages in the dSPACE Log from an RTT Sequence..... 69](#)

You can print messages of different severity levels (info, warning, or error) to the standard output or the dSPACE Log file.

[Avoiding a Timeout During Initialization of an RTT Sequence..... 70](#)

The time available for initializing an RTT sequence is limited. If your RTT sequence requires more time for the initialization, for example, when long lists are created, a timeout occurs. In this case you can increase the available initialization time.

Information in other sections

Implementing Generator Functions.....	45
Generator functions are the basis of real-time testing. These are the functions which are executed in steps.	
Demo Examples of Using Real-Time Testing.....	22
Some demo examples are installed with Real-Time Testing. Examine these examples to learn about the features of Real-Time Testing.	

Encoding the Scripts of RTT Sequences

Introduction

Real-Time Testing uses the UTF-8 encoding for the Python interpreter on the simulation platform. You must consider this for the encoding of the files of the RTT sequences.

Encoding and Python version

The behavior depends on the Python version used.

Python version 2.7 Python version 2.7 is used on DS1006, DS1007, MicroAutoBox, and MicroLabBox platforms.

If Python version 2.7 is installed on the platform and the RTT sequence contains non-ascii characters, you must save the files in the UTF-8 encoding. You can also use the string prefix `u'...'` for unicode strings.

Python version 3.6 Python version 3.6 is used on VEOS, MicroAutoBox III, SCALEXIO, and DS6001 platforms.

If Python version 3.6 is installed on the platform, it is sufficient to specify the encoding in the first line of the RTT sequence, for example:

```
# -*- coding: Latin -*-
```

However, it is recommended to use the UTF-8 encoding.

Setting the encoding

The encoding is set in the text editor. How you specify the encoding, depends on the text editor you are using.

For example, PythonWin automatically sets the encoding according to the first line of the RTT sequence. The UTF-8 encoding is specified using the following line.

```
# -*- coding: utf-8 -*-
```

Note that other text editors ignore the coding settings in the RTT sequence. In this case, they may have other possibilities to specify the encoding.

Simple RTT Sequence

Introduction

The following example has the minimum structure of an RTT sequence. You can use it as a basis for your own RTT sequences.

Example

The following RTT sequence shows a simple example. It prints the simulation time to the Log Viewer of the Real-Time Test Manager twice. The difference between the two time values is the simulation step size.

```
# module import
from rttlib import variable
from rttlib import utilities
# Create a reference to a variable object containing the current time
CurrentTime = utilities.currentTime
# Generator function called by the scheduler
# The name 'MainGenerator' is mandatory
def MainGenerator(*args):
    # Begin of synchronized execution
    print(" First step's CurrentTime Value: ", CurrentTime.Value)
    yield None
    print(" Second step's CurrentTime Value: ", CurrentTime.Value)
    yield None
```

Description

An RTT sequence has two different sections: an initialization section and the function definition of `MainGenerator(*args)`.

Initialization section All code lines at the top level belong to the initialization of the RTT sequence. The code lines are processed when the RTT sequence is created. This is the same portion of code which would be executed if this source file were imported by another script. In this part, the required Python modules are imported, and time-consuming operations, for example, initializing variables and arrays, should be performed. The execution of the initialization section is completely unsynchronized with model execution. Executing the initialization section can take several sample steps.

MainGenerator(*args) All code lines in the `MainGenerator(*args)` function definition are executed synchronously to the real-time model. It is executed stepwise from the dSPACE script scheduler. Every RTT sequence must have at least a simple `MainGenerator ("yield None")`. `yield` is used to control the processing of the RTT sequence. The RTT sequence is processed until a `yield` keyword is found. In the next simulation step, the RTT sequence resumes after the `yield` keyword. The example above demonstrates the use of `yield`.

Related topics

References

[rttlib.utilities Module \(Real-Time Testing Library Reference !\[\]\(b792654f2cef9719eabeb6c5be00811e_img.jpg\)](#))

Getting Properties of an RTT Sequence

Introduction

The `SequenceProperties` class of the `rttlib.utilities` module provides some properties of the RTT sequence.

Example

The following RTT sequence shows the simple example which is introduced in [Simple RTT Sequence](#) on page 59. The example is extended by the `SequenceProperties` object to provide some properties of the RTT sequence.

```
# module import
from rttlib import variable
from rttlib import utilities
# Create a reference to a variable object containing the current time
CurrentTime = utilities.currentTime
# Get a SequenceProperties object
SequenceProperties = utilities.SequenceProperties()
# To get the name of the RTT sequence.
SequenceName = SequenceProperties.Name
# To get the description
SequenceDescription = SequenceProperties.Description
# To get the file name
SequenceFileName = SequenceProperties.FileName
# To get the priority of the RTT sequence
SequencePriority = SequenceProperties.Priority
# To get the sequence channel
SequenceChannel = SequenceProperties.SequenceChannel
# Generator function called by the scheduler
# The name 'MainGenerator' is mandatory
def MainGenerator(*args):
    # Begin of synchronized execution
    print(" First step's CurrentTime Value: ", CurrentTime.Value)
    yield None
    print(" Second step's CurrentTime Value: ", CurrentTime.Value)
    yield None
```

Description

You get or set some properties of the RTT sequence when you use the `SequenceProperties` class of the `rttlib.utilities` module.

The following properties are read-only: `Name`, `Description`, `FileName`, and `Priority`.

The `SequenceChannel` property can be read and written. This property specifies whether the RTT sequence is executed before or after the simulation model. For details of the execution order, refer to [Basics on Running RTT Sequences](#) on page 29.

Related topics**Basics**

Simple RTT Sequence..... 59

References

SequenceProperties (Real-Time Testing Library Reference )

Using Local and Global Variables in RTT Sequences

Introduction

You can use local and global variables in RTT sequences. To optimize your RTT sequence, you should use local variables in function definitions. Local variables are much faster to access than global variables because they do not access the dictionary.

Example

The example shows the using of local and global variables.

```
threshold = 2.0
def check_threshold(signal):
    local_threshold = threshold
    while (1):
        if (signal.Value < local_threshold):
            yield None
```

Description

`threshold` is a global variable. `local_threshold` is a local variable in the `check_threshold()` function. Both variables reference the same object. In the function, only the `local_threshold` variable is used. Python accesses local variables much more efficiently than global variables.

Arguments of a function definition are local variables. It is not necessary to copy their values to a local variable explicitly.


Tip

For more performance tips, refer to <http://wiki.python.org/moin/PythonSpeed/PerformanceTips>.

Related topics**Basics**

Using Variables Accessible by Several RTT Sequences..... 62

Using Variables Accessible by Several RTT Sequences

Introduction	You can use global variables which are accessible by several RTT sequences, so that you can exchange data between RTT sequences.
Limitation	Global variables cannot be used by RTT sequences running on different CPUs in a multiprocessor system.
Example	<p>The following example shows how to create a global variable in an RTT sequence.</p> <pre>from rttlib import globalvariables # create "myvariable1" globalvariables.myvariable1 = 42.0</pre> <p>The created global variable (myvariable1) can be accessed by other RTT sequences like a normal module attribute.</p> <pre>from rttlib import globalvariables print(globalvariables.myvariable1)</pre>
Description	When you want to use global variables for several RTT sequences, you only have to import the globalvariables module from the rttlib package. The globalvariables module is a dSPACE-provided module that is shared between all RTT sequences. It persists as long as the Python interpreter is running. Global variables can be created by simply inserting a new attribute into the namespace of the globalvariables module.
Related topics	<p>Basics</p> <p>Basics on Dynamic Variables..... 77</p> <p>Using Local and Global Variables in RTT Sequences..... 61</p> <p>References</p> <p>rttlib.globalvariables Module (Real-Time Testing Library Reference )</p>

Read/Write Access to Variables of the Simulation Application

Introduction	Real-Time Testing provides access to the variables of a simulation application. You have read/write access at any point during execution of the RTT sequence.
---------------------	---

This enables perfect monitoring and stimulation of signals, because of the stepwise execution of the RTT sequence.

Enhanced TRC file

Real-Time Testing 2.6 and higher supports the enhanced TRC file generation that have been release with dSPACE Release 2015-B. When the enhanced TRC file generation is used instead of the previous TRC file generation, it can lead to a modified variable path, so that they become invalid in your test scripts.

Multiprocessor system

The method described below can also be used if you want to access a variable from an RTT sequence running on the *same* CPU in a multiprocessor system. If you want to access a variable of another CPU (remote CPU) in a multiprocessor system, refer to [Accessing Variables of a Simulation Application on a Remote Node](#) on page 150.

Fixed parameters

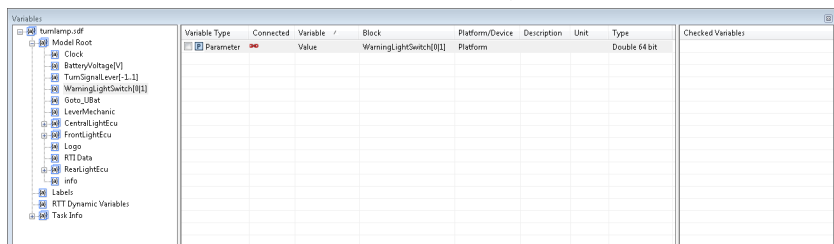
Fixed parameters cannot be modified. The INITONLY flag is set for fixed parameters, i.e., they have a fixed value during a running simulation and can be modified only during the initialization phase of a simulation application.

Scaling

In a TRC variable description file, a scaling can be specified for a variable. Real-Time Testing 3.3 and higher can consider the scaling of variables. Real-Time Testing supports only the linear scaling of parameters ($\hat{x} = m \cdot x + b$), signals, and single elements of vectors and matrices. Other kinds of scaling, for example, formula scaling, nonlinear scaling, or string-formatted scalings (for example, used for bitmasks), are not supported. You can enable or disable the scaling support for each variable separately. The consideration of the scaling is enabled by default.

Example

The following example is an RTT sequence for the TurnSignal demo. This demo is installed in <MyDocuments>\dSPACE\Real-Time Testing\5.0\SampleExperiments\TurnSignal_<Platform>. The following illustration shows the Variables controlbar with the demo opened in ControlDesk.



```
# module import
from rttlib import variable
# Create a variable object
WarningLightSwitch = variable.Variable('r'Model Root/WarningLightSwitch[0][1]/Value')
```

```
def MainGenerator(*args):
    print("Variable Name: ", WarningLightSwitch.Name())
    WarningLightSwitch.Value = 0.0
    yield None
    print("Value of WarningLightSwitch: ", WarningLightSwitch.Value)
    WarningLightSwitch.Value = 1.0
    yield None
    print("Value of WarningLightSwitch: ", WarningLightSwitch.Value)
    WarningLightSwitch.Value = 0.0
    yield None
    print(WarningLightSwitch.Value)
    yield None
```

Description

To access a variable of the real-time application, you must import the **variable** module from the **rttlib** package. This module lets you create a variable object which you can use in your RTT sequence. To create a variable object, you must specify the variable with its full path as specified in the real-time model and TRC file. The full path of the variable can be read in ControlDesk's Variables controlbar (see example above). You can read a value of a variable (`a = WarningLightSwitch.Value`) and write to a variable (`WarningLightSwitch.Value = 42.0`). For more information on the **variable** module, refer to [rttlib.variable Module \(Real-Time Testing Library Reference\)](#).

Tip

An easy way to get a variable's name is to drag & drop the variable from ControlDesk's Variables controlbar to the source code editor. There are some limitations applying to the path names. Refer to [Limitations for variables](#) on page 230. In multiprocessor systems, ControlDesk displays an extended path to the real-time variable. To distinguish the variables of different CPUs, the name of the submodel is added to the path. For RTT sequences you can ignore the name of the submodel. Real-Time Testing requires only the path as specified in the TRC or A2L file. If the RTT sequence and the Simulink variable are located on the same CPU, the name of the subsystem in the constructor of the variable object can be ignored.

Example

The following example shows how you can handle exception errors for variables using **VariableManagerError**.

```
from rttlib import variable
try:
    VariableList.append(variable.Variable(path))
except variable.VariableManagerError:
    # do something
except:
    raise Exception("Unexpected exception type. Expected:
%s" % (variable.VariableManagerError))
```


Related topics**Basics**

[Checking Conditions According to the ASAM GES Standard..... 65](#)
[Using Local and Global Variables in RTT Sequences..... 61](#)

References

[Variable Class \(Real-Time Testing Library Reference !\[\]\(d66ff64371a51729ac8c1cdaa685ba6f_img.jpg\)\)](#)

Checking Conditions According to the ASAM GES Standard

Introduction

RTT variables can be monitored depending on conditions that can be specified according to the ASAM General Expression Syntax (GES) standard. The watcherlib module provides the necessary class for this.

Basics on the watcherlib

The watcherlib module has the watcher class. You can use this class to specify a watcher that checks a condition in each model step. The watcher checks the variable value until the condition specified for the watcher is true or a timeout is reached. When the condition is true, the watcher generator function returns. When the timeout is reached before the condition is fulfilled, a timeout exception occurs.

Specifying the condition

Conditions are specified according to the ASAM GES standard. Real-Time Testing supports more operators and functions than the ASAM GES standard. For a list of all the supported operators and functions, refer to [Operators and Functions Supported by the watcherlib \(Real-Time Testing Library Reference !\[\]\(4b7a79268f6ba26c1471d4232fffa85a_img.jpg\)\)](#).

You can use variable objects in the expression. Therefore, it is possible to observe the values of variables of the simulation application.

Examples The following table describes some condition examples.

Condition	Purpose
"var1 > 10.0"	To check whether var1 is greater than a specified value.
"var1 > var2"	To check whether var1 is greater than var2 .
"var1 == var2"	To check whether var1 and var2 are equal. Do not use the condition for float variables. Refer to the next row.

Condition	Purpose
"abs(var1 - var2) < 1e-14"	To check whether the var1 and var2 float variables are equal. Note If you check two float variables for equality, using the == operator might lead to unexpected results. To take into account precision effects of the binary representation of float values, it is useful to allow for tolerances.
"var1 > sqrt(2)"	To check whether the value of var1 is greater than the square root of 2.
"var1 != var2"	To check whether var1 and var2 are non-equal.

Example

The following example shows how to define and use a watcher.

```
#-----
# Import the variable and watcherlib class from the rttlib module.
# This Python Library provides all Real-Time Testing modules.
# ALL imports must be defined in the global part of the script.
#-----
from rttlib import variable
from rttlib import watcherlib
from rttlib import utilities
#-----
# Module-global variables
#-----
# Create a variable object for the watcherlib.
if variable.IsA2L():
    # the variable description for this application is an A2L file.
    TurnSignalLeftFront = variable.Variable(r'FrontLightEcuTurnSignalLeft')
    TurnSignalLeftRear = variable.Variable(r'RearLightEcuTurnSignalLeft')
    BatteryVoltage = variable.Variable(r'BatteryVoltageValue')
    TurnSignalLever = variable.Variable(r'TurnSignalLeverValue')
else:
    # the variable description for this application is a TRC file.
    TurnSignalLeftFront = variable.Variable(r'Model Root/FrontLightEcu/TurnSignalLeft')
    TurnSignalLeftRear = variable.Variable(r'Model Root/RearLightEcu/TurnSignalLeft')
    BatteryVoltage = variable.Variable(r'Model Root/BatteryVoltage[V]/Value')
    TurnSignalLever = variable.Variable(r'Model Root/TurnSignalLever[-1..1]/Value')
def MainGenerator():
    # Set the TurnSignalLever to the left(-1); use (+1) for to set it to the right
    TurnSignalLever.Value = -1.0
    BatteryVoltage.Value = 12.0
```

```

# Define the Watcher_01 Object.
# Set a condition.
condition_01 = "var1 > sqrt(100)"
# Set a timeout.
timeout = 15
Watcher_01 = watcherLib.Watcher(condition_01, {'var1':BatteryVoltage}, timeout)
# Define a Watcher from the Watcher_01 Object.
WatcherGenerator_01 = Watcher_01.Watch()
# Wait until the condition is true or the timeout is reached.
yield WatcherGenerator_01
yield None
# Toggle BatterieVoltage to force errors in the Turnsignal system.
BatteryVoltage.Value = 7.5
# Define Watcher 2.
# Set a condition.
condition_02 = "var1 != var2"
# Set a timeout.
timeout = 30
Watcher_02 = watcherLib.Watcher(condition_02, {'var1':TurnSignalLeftFront, 'var2':TurnSignalLeftRear}, timeout)
# Define Watcher from the Watcher_02 Object.
WatcherGenerator_02 = Watcher_02.Watch()
# Wait until the condition is true or the timeout is reached.
yield WatcherGenerator_02

```

Description

When you work with the `watcherlib`, you first have to define a watcher object. Then you can use the watcher object to create a watcher iterator.

Defining a watcher object To define a watcher object, you have to specify three parameters:

- **Condition:** A string containing an expression specifying the condition. Refer to [Specifying the condition](#) on page 65.
- **VariablesDictionary:** A dictionary with keys and values. The keys must be names that are used in the condition string. The values must be the names of variable objects that correspond to model variables. Refer to [Read/Write Access to Variables of the Simulation Application](#) on page 62.
- **Timeout:** A time value. If the condition is not fulfilled within the specified time, a timeout exception occurs.

In the example, the `Watcher_01` and `Watcher_02` objects are defined:

```

Watcher_01 = watcherlib.Watcher(condition_01,
{'var1':BatteryVoltage}, timeout)
and
Watcher_02 = watcherlib.Watcher(condition_02,
{'var1':TurnSignalLeftFront, 'var2':TurnSignalLeftRear},
timeout)

```

Defining a watcher iterator When you execute the `Watch` method of the watcher object, you get the watcher iterators:

```

WatcherGenerator_01 = Watcher_01.Watch()

```

and

```
WatcherGenerator_02 = Watcher_02.Watch()
```

Using the watcher generator The watcher generators are the points in the RTT sequence where the RTT sequence is paused and the condition is checked. Watcher generators are generator objects, so they must be prefixed with the `yield` statement:

```
yield WatcherGenerator_01
```

and

```
yield WatcherGenerator_02
```

The watcher generators can also be used in `Parallel` and `ParallelRace` functions.

Related topics

References

[rttlib.watcherlib Module \(Real-Time Testing Library Reference\)](#)

Printing Variable Values of an RTT Sequence on the Host PC

Introduction

The values of variables can be read in the Log Viewer of the Real-Time Test Manager.

Example

The following example shows how a value of a variable can be printed. The output is downsampled. Only every 100th value is printed.

```
from rttlib import variable
from rttlib import utilities
CurrentTime = utilities.currentTime
ExecTime = variable.Variable(r'Platform()::currentTime')
def MainGenerator(*args):
    counter = 0
    start_time = CurrentTime.Value
    end_time = start_time + 5.0
    while CurrentTime.Value < end_time:
        counter += 1
        if (counter % 100) == 0:
            print(ExecTime.Value)
        yield None
```

Description

You can use the `print` command as usual. The output is printed in the Log Viewer of the Real-Time Test Manager. You should use only few print statements, because it can happen that the real-time system writes the output faster than the host can read. Moreover, using a `print` command requires a lot of

computation time. You should use it only for transferring the final test results or debugging purposes.

Note

Print output can be lost

Before the output is transferred to the host, it is copied to a ring buffer. If the buffer is full, older data is overwritten and not transmitted. This can happen if you print too much data, which the host cannot read. The size of the buffer is currently 4 kbit.

Monitoring variables in ControlDesk

It is not possible to monitor local variables of the RTT sequence in ControlDesk. ControlDesk can only monitor variables which are contained in the TRC file.

Tip

If you want to monitor variables of the RTT sequence in ControlDesk, you must copy their values to a TRC variable. To get TRC variables for this purpose, you can add Constant blocks to the real-time model.

Related topics

Basics

Read/Write Access to Variables of the Simulation Application.....	62
Using Local and Global Variables in RTT Sequences.....	61

Printing Messages in the dSPACE Log from an RTT Sequence

Introduction

You can print messages of different severity levels (info, warning, or error) to the standard output or the dSPACE Log file.

Example

The following example shows how messages of different severity levels can be printed.

```
from rttlib import utilities
# Messages to be printed
Info_Text = 'My information message'
Warning_Text = 'My warning message'
Error_Text = 'My error message'
# Enable message printing
utilities.Logging.Enable = True
# Specify the target for printing
utilities.Logging.Direction = utilities.Logging.TO_ON_WRITE
```

```
# Print messages
utilities.Logging.info(Info_Text, sep=' ', end='\n')
utilities.Logging.warning(Warning_Text, sep=' ', end='\n')
utilities.Logging.error(Error_Text, sep=' ', end='\n')
```

Description

To print messages, Real-Time Testing provides the **Logging** class. This class has the **Enable** attribute that enables or disables the printing. The **Direction** attribute specifies the target of the message. You can print the message to the standard output (**utilities.Logging.TO_ON_WRITE**) and/or the dSPACE log file (**utilities.Logging.TO_LOGFILE**). The example demonstrates how you can send messages to the standard output. It is also possible to write the messages to the dSPACE Log file.

The messages are printed under different severity levels using the **info()**, **warning()**, or **error()** method.

Related topics**References**

[Logging \(Real-Time Testing Library Reference !\[\]\(10f8862fc183b400327470ea85afe9ae_img.jpg\)\)](#)

Avoiding a Timeout During Initialization of an RTT Sequence

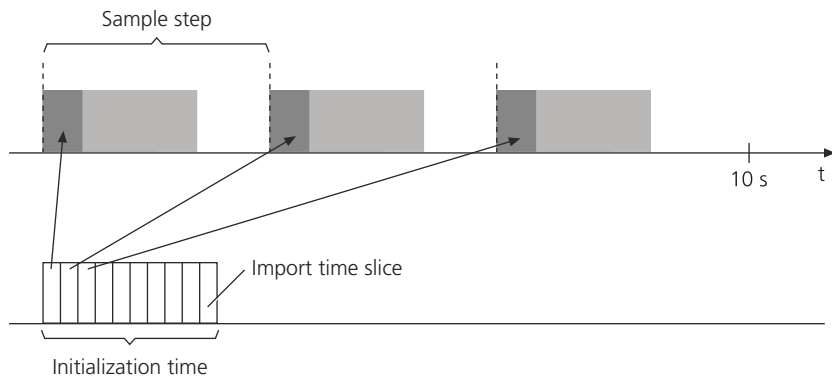
Introduction

The time available for initializing an RTT sequence is limited. If your RTT sequence requires more time for the initialization, for example, when long lists are created, a timeout occurs. In this case you can increase the available initialization time.

Initialization time

When an RTT sequence is created, the sequence is initialized first. During initialization all code lines at the top level are processed, the required Python modules are imported, and time-consuming operations, for example, initializing variables and arrays, are performed.

Usually, the RTT sequence is initialized in several sampling steps. Each step executes part of the initialization section in an initialization time slice. By default, the initialization time slice is 20 μ s.



If the initialization of an RTT sequence is not finished after 8 s, a timeout occurs. An error message similar to the following is generated:

```
Real-Time Testing Real-Time Test Manager Server: Error 14:08:48 72,0:
  An exception was raised for sequence 'RTTSequence_2'.
  Exception: '
RTT Traceback:
: '. (0x36,1)
dsxyz: WARNING 14:08:51 [#22] dsxyz - RTPYTHON:
      Initialization phase timed out
      Check the model's sim state. (-8211)(0xCA)
```

The following formula shows the dependencies between initialization time, initialization time slice, sampling step, and timeout:

$$\text{Timeout [s]} / \text{Sampling step [s]} = \text{Number of steps}$$

$$\text{Number of steps} \cdot \text{Initialization time slice [s]} = \text{Initialization time [s]}$$

Example The example calculates the initialization time of an RTT sequence:

$$8 \text{ s} / 0.001 \text{ s} = 8,000$$

$$8,000 \cdot 0.00002 \text{ s} = 0.16 \text{ s}$$

In this example the RTT sequence must be initialized within 8,000 steps. Because an RTT sequence gets 20 μs initialization time slice per sampling step, the total computation time for initializing the RTT sequence is 0.16 s.

Adapting the initialization time

If a timeout occurs during initialization, you must adapt the initialization time. The timeout of 8 s is fixed and cannot be changed.

The initialization time is adapted by changing the initialization time slices by calling the `SetImportTimeslice` function. Larger initialization time slices yield a larger initialization time. For example, with sampling steps of 1 ms and a timeout of 8 s, initialization time slices of 20 μs yield an initialization time of 0.16 s, and time slices of 40 μs yield an initialization time of 0.32 s.

Example The following example shows how to set the initialization time slices to 40 μs :

```
from rttlib import utilities
utilities.SetImportTimeslice(0.00004)
```

Tip

If you cannot extend the initialization time slices because there is not enough time in the sampling steps, you can split an RTT sequence into several RTT sequences. Each RTT sequence has its own initialization time slice. You can then work with global variables to make the variables accessible for every RTT sequence. Refer to [Using Variables Accessible by Several RTT Sequences](#) on page 62.

Related topics

References

[SetImportTimeslice Function \(Real-Time Testing Library Reference !\[\]\(96cc62f861fdd6e50510c0224a756dff_img.jpg\)\)](#)

Exception Handling and Using Modules

Introduction

You can implement your own modules and use standard Python modules.

Where to go from here

Information in this section

[Implementing an Exception Handling..... 73](#)

You can handle exceptions which occur in your RTT sequence.

[Using User Python Modules..... 74](#)

Describes how you can use your own Python modules in your RTT sequences.

[Using Modules from the Standard Python Library..... 75](#)

You can use standard Python modules for Real-Time Testing. The modules must only be imported in your RTT sequence.

Information in other sections

[General Information on Implementing RTT Sequences..... 57](#)

Describing how you can program simple RTT sequences.

[Implementing Generator Functions..... 45](#)

Generator functions are the basis of real-time testing. These are the functions which are executed in steps.

Implementing an Exception Handling

Introduction

Exceptions can occur in an RTT sequence, for example, a division by zero exception. You can handle exceptions in your RTT sequence.

Note

Exceptions can only be used inside of generator functions. They are not propagated through a `yield` statement. Propagating exceptions across `yield` statements is planned for future versions.

Example

The following example demonstrates how an exception can be handled.

```
from rttlib import errors
```

```
def TestException():
    try:
        10/0
        yield None
    except ZeroDivisionError:
        raise errors.RTException("integer division by zero")
def MainGenerator(*args):
    yield TestException()
```

Description

The `TestException` function contains a `try` and `except` statement. An exception (division by zero) is caused in the `try` statement. The `except` statement is therefore executed and raises the exception.

Related topics**References**

[RTException Class Description \(Real-Time Testing Library Reference !\[\]\(a870788d6ed9b8fd294b7654a8c8526b_img.jpg\)](#))

Using User Python Modules

Introduction

This section describes how you can use your own Python modules in your RTT sequences. You can build your own Python module for use in Real-Time Testing. When you group common code into separate modules, the RTT sequences become clearer and shorter, and you can reuse the code in several RTT sequences.

Compiled modules in the PYC format can also be imported if they were compiled with the same Python version. However, compiling makes the contents of the imported library non-human-readable and difficult to alter.

Functions included in your Python module must be suitable for Real-Time Testing. For example, they must not use functions of unsupported Python modules. For a list of the supported modules, refer to [Supported Python Modules \(Real-Time Testing Library Reference !\[\]\(6a9b39b98eb945faa14c645ec99e4eaa_img.jpg\)](#)).

Example

The following script shows a user Python module example, for example, `myrtttools.py`.

```
from rttlib import variable
from rttlib import utilities
CurrentTime = utilities.currentTime
def WaitGen(Duration):
    print(" WaitGen() for %s sec called." %Duration)
    StopTime = CurrentTime.Value + Duration
    while (CurrentTime.Value < StopTime):
        yield None
```

The following script shows how the function of `myrtttools.py` can be called from an RTT sequence.

```
from rttlib import variable
import myrtttools
WarningLightSwitch = variable.Variable(r'Model Root/WarningLightSwitch[0]
1]/Value')
def MainGenerator(*args):
    # skip the first two simulation steps
    yield None
    yield None
    # Set the value of WarningLightSwitch
    WarningLightSwitch.Value = 0.0
    yield None
    # Call a function from user-written module
    yield myrtttools.WaitGen(10)
    yield None
    # Set the value of WarningLightSwitch
    WarningLightSwitch.Value = 1.0
    yield None
```

Description

`myrtttools.py` is a user Python module. It contains only one function definition. The `WaitGen()` function suspends the RTT sequence and resumes it after the specified duration.

The RTT sequence sets the value of `WarningLightSwitch` to 0.0. The `WaitGen()` generator function is used in the `MainGenerator(*args)`.

The RTT sequence does the following:

- Sets the value of `WarningLightSwitch` to 0.0.
- Waits 10 seconds using the `WaitGen()` function.
- Sets the value of `WarningLightSwitch` to 1.0.

To be able to call the `WaitGen()` generator function, the `myrtttools` module must be imported.

Related topics

References

[rttlib.utilities Module \(Real-Time Testing Library Reference !\[\]\(4fe57c3593bf1b21d272ae7ac8dfaf77_img.jpg\)\)](#)

Using Modules from the Standard Python Library

Introduction

Only a subset of the whole standard library is suitable for Real-Time Testing. One reason for this limitation is the lack of file system and networking support. Another reason is the fact that most modules lack a deterministic behavior required for real-time systems. During byte-code generation, it is checked whether Real-Time Testing supports the imported modules. For information on

which Python modules are supported, refer to [Supported Python Modules \(Real-Time Testing Library Reference !\[\]\(21199eb166cc97331a0c54c649195dcc_img.jpg\)](#)).

Example

The following example demonstrates how to import standard Python modules and use their function definitions.

```
from rttlib import variable
from rttlib import utilities
import math
ModelStepSize = utilities.modelStepSize
CurrentTime = utilities.currentTime
BatteryVoltage = variable.Variable(r'Model Root/BatteryVoltage[V]/Value')
def SinusGen(var, duration, amplitude, offset):
    period = 1.0 # in Hz
    start = CurrentTime.Value
    while CurrentTime.Value < start + duration:
        # calculate relative time
        t = CurrentTime.Value - start
        # calculate new sinus : value = off + amp * sin(w * t)
        var.Value = offset + amplitude * math.sin(2 * math.pi * (1/period) * t)
        # suspend until next simulation step
        yield None
def MainGenerator(*args):
    yield SinusGen(BatteryVoltage, 20.0, 7.5, 7.5)
```

Description

The example uses a function from Python's math module. The `SinusGen` function stimulates a variable using a sinus signal. The `sin` function is contained in the `math` module, which is therefore imported.

Note

Not all standard Python function are suitable for real-time use. Functions from `math` are generally real-time-safe, but if you use functions like `sort()` you can get problems because their run time depends on the size of their argument.

Related topics

References

[BCGServiceProvider \(Real-Time Testing Library Reference !\[\]\(758ebdf4629c903da74c2e079717ae32_img.jpg\)](#))

Exchanging Data Between RTT Sequence and Host Python Script

Introduction

Host calls can transfer Python data objects from the simulation platform to the host PC and vice versa.

Where to go from here

Information in this section

[Basics on Dynamic Variables..... 77](#)

You can create dynamic variables in an RTT sequence. You can access them from other RTT sequences running on the same processor board and from the host PC as well.

[Example of Using Dynamic Variables..... 80](#)

The examples show how to use dynamic variables in RTT sequences and a host PC Python script.

[Exchanging Data Between RTT Sequences and Python Scripts Running on the Host PC..... 83](#)

You can transfer Python data objects from the simulation platform to the host PC and vice versa by using host calls.

[Example of Exchanging User-Defined Data..... 85](#)

You can transfer user-defined Python data objects from the simulation platform to the host PC's Python interpreter and vice versa by using host calls.

Basics on Dynamic Variables

Introduction

You can create dynamic variables in an RTT sequence during simulation run time. You can access them from other RTT sequences running on the same processor board and from the host PC as well.

Dynamic variables cannot be used by RTT sequences running on different CPUs in a multiprocessor system.

Description

You can create dynamic variables in an RTT sequence while the real-time application is running. By using dynamic variables, you can perform some calculation in the RTT sequence without changing the real-time model. You can use dynamic variables to create a user-defined interface between the host PC and the RTT sequences. The name of a dynamic variable therefore has to be unique in all RTT sequences on the same processor board. The dynamic variable

remains on the hardware even if the RTT sequence where the variable was created is removed.

Dynamic variables cannot be read or written from the host to the RTT sequence in real time. The values are read or written as fast as possible.

Possible use cases are:

- The `RTT_TestResult` dynamic variable, which you can use to store the result of the latest executed RTT sequence
- The `RTT_Maneuver` dynamic variable, which is written by the host PC script to trigger a specific action in one or more of the RTT sequences

After you created a dynamic variable in an RTT sequence, you can read and write this variable:

- In the RTT sequence where you created it
- In all other RTT sequences running on the same simulation platform
- In the Python script which manages the RTT sequences on the host PC

Creating dynamic variables

You can create a dynamic variable in an RTT sequence. The variable's name is unique in the namespace of the Python interpreter on the board where all RTT sequences are created. If the variable name does not yet exist, a new variable object is created. If the variable name already exists on a board, the variable object references the existing variable object.

Tip

Create dynamic variables during the initialization phase of the RTT sequence to save computation time. Dynamic variables which are not created during the initialization phase cannot be accessed by the host PC and are not visible in the variable collection of the board until the next RTT sequence is created.

Example The example creates two dynamic variables in an RTT sequence.

```
from rttlib import dynamicvariable
# Create dynamic variable objects
DynamicVariable1 = dynamicvariable.DynamicVariable("TestVariable1")
DynamicVariable2 = dynamicvariable.DynamicVariable("TestVariable2")
```

The object names in the RTT sequence are `DynamicVariable1` and `DynamicVariable2`. The parameters used during creation specify the name attributes for the dynamic variables. Each name is unique on a board. The objects are global, and each one can be accessed from different RTT sequences by using the same name string to create a dynamic variable object. Only one dynamic variable object with any specific name is available on the host for the board where it was created. Other boards can have dynamic variable objects with the same name.

Name attribute If dynamic variable objects are created with the same name string (even in different RTT sequences), they access the same dynamic variable object. For example, if `DynamicVariable1` in the first RTT sequence and `DynamicVariable2` in the second RTT sequence use the name string

"TestVariable" and the value of `DynamicVariable2` changes, this automatically changes the value of `DynamicVariable1`.

Value and DynamicValue attributes The `DynamicVariable` class has two attributes (`Value` and `DynamicValue`). The attributes are independent.

The `Value` attribute supports only the float data type.

The `DynamicValue` attribute supports several data types (Boolean, integer, float, string, tuple) or a combination of them. If a list data type is used, it is converted to a tuple for the host-platform communication. The dictionary data type is not supported. Before the `DynamicValue` is read for the first time, it must be set initially. Otherwise an exception is triggered.

Accessing dynamic variables in an RTT sequence

You can access dynamic variables from the RTT sequence where you created them or from any other RTT sequence on the same simulation platform.

Example The example shows the values of the dynamic variables, changes their values, and shows them again.

```
def MainGenerator(*args):
    print("Start of real-time test execution on target platform")
    yield None
    print("Name of dynamic variable is '%s', \
        Value = %f " %(DynamicVariable1.Name(), DynamicVariable1.Value))
    print("Name of dynamic variable is '%s', \
        Value = %f " %(DynamicVariable2.Name(), DynamicVariable2.Value))
    yield None
    DynamicVariable1.Value = 44
    DynamicVariable2.Value = 45
    yield None
    print("Name of dynamic variable is '%s', \
        Value = %f " %(DynamicVariable1.Name(), DynamicVariable1.Value))
    print("Name of dynamic variable is '%s', \
        Value = %f " %(DynamicVariable2.Name(), DynamicVariable2.Value))
    yield None
```

Accessing dynamic variables from the host PC Python interpreter

You can access dynamic variables from the Python interpreter via the Real-Time Test Manager Server on the host PC.

Note

You cannot read and write data from the host PC to the simulation platform in real time.

Example The example shows how you can read and write dynamic variables from a host PC Python script.

```
# Create new sequence
RTTManager = rttmanagerlib.RealTimeTestManagerServer()
Board = RTTManager.AccessBoard(BoardName)
for Variable in Board.Variables:
    print("Dynamic Variable '%s' = %f" %(Variable.Name, Variable.Value))
# write variable
for Variable in Board.Variables:
    Variable.Value = 0
```

Event handling

You can implement event handling for events triggered by the dynamic variables. The events can be evaluated in the Python script running on the host.

You can implement event handling for the following events:

Event	Description
OnAdd	A dynamic variable was created on the real-time platform.

If you create event handlers, they must use a specified method. For details on the method, refer to [VariablesEvents Class Description \(Real-Time Testing Library Reference !\[\]\(96cc62f861fdd6e50510c0224a756dff_img.jpg\)](#)).

Restrictions

- The value of dynamic variables is always in the float64 data type.
- Dynamic variables can only be removed by resetting the Python interpreter or reloading the simulation application
- Dynamic variables can only be created and accessed on the local CPU in a multiprocessor system.

Related topics**References**

[rttlib.dynamicvariable Module \(Real-Time Testing Library Reference !\[\]\(e1c624d4757f08486e89482c18364c17_img.jpg\)](#))

[Variable \(Real-Time Testing Library Reference !\[\]\(fd44bd93e945cfa8875a8962f08e5b64_img.jpg\)](#))

[Variables \(Collection\) \(Real-Time Testing Library Reference !\[\]\(4a7bd0d19449e9ae6d04f317c9f2938f_img.jpg\)](#))

Example of Using Dynamic Variables

Introduction

The following scripts show how an RTT sequence creates dynamic variables and how the host PC Python script reads these variables. The event handling for events triggered by the dynamic variables is also shown.

RTT sequence

The following RTT sequence creates two dynamic variables and sets their values. For the complete example, refer to the demo archive <MyDocuments>\dSPACE\Real-Time Testing\5.0\06_DynamicVariables and see the RTTSequence.py file.

```
from rttlib import dynamicvariable
# Create dynamic variable objects
DynamicVariable1 = dynamicvariable.DynamicVariable("TestVariable1")
DynamicVariable2 = dynamicvariable.DynamicVariable("TestVariable2")
# Identifier for real-time testing print messages
RTT_PREFIX = "RTT:* "
def MainGenerator(*args):
    print(RTT_PREFIX + "Start of real-time test execution on target platform")
    yield None
```



```

print(RTT_PREFIX + "Name of dynamic variable is '%s', Value = %f "\
      %(DynamicVariable1.Name(), DynamicVariable1.Value))
print(RTT_PREFIX + "Name of dynamic variable is '%s', Value = %f "\
      %(DynamicVariable2.Name(), DynamicVariable2.Value))
yield None
print(RTT_PREFIX + "Set new values of variables. ")
NewValue = 42
DynamicVariable1.Value = NewValue
DynamicVariable1.DynamicValue = NewValue
DynamicVariable2.Value = NewValue + 1
DynamicVariable2.DynamicValue = NewValue + 1
yield None
print(RTT_PREFIX + " '%s' = %f " %(DynamicVariable1.Name(),\
      DynamicVariable1.Value))
print(RTT_PREFIX + " '%s' = %f " %(DynamicVariable2.Name(),\
      DynamicVariable2.Value))
yield None
print(RTT_PREFIX + "Suspend RTT sequence execution for host interaction.")
while DynamicVariable2.Value < 100:
    yield None
    print(RTT_PREFIX + "Get new values of variables, changed by host. ")
    print(RTT_PREFIX + " '%s' Value = %f, DynamicValue = %s " \
          % (dynamicVariable1.Name(), dynamicVariable1.Value, dynamicVariable1.DynamicValue))
    print(RTT_PREFIX + " '%s' Value = %f, DynamicValue = %s " \
          % (dynamicVariable2.Name(), dynamicVariable2.Value, dynamicVariable2.DynamicValue))
    yield None
    print(rttPrefix + "Suspend RTT sequence execution for host interaction.")
    # Wait until the host sets a dynamic value
    while dynamicVariable2.DynamicValue == 43:
        yield None
        print(rttPrefix + "Get new dynamic values of variables, changed by host. ")
        print(rttPrefix + " '%s' Value = %f, DynamicValue = %s " \
              % (dynamicVariable1.Name(), dynamicVariable1.Value, dynamicVariable1.DynamicValue))
        print(rttPrefix + " '%s' Value = %f, DynamicValue = %s " \
              % (dynamicVariable2.Name(), dynamicVariable2.Value, dynamicVariable2.DynamicValue))
        yield None
    print(RTT_PREFIX + "End of real-time test execution.")

```

Host PC Python script

The following Python script runs the RTT sequences and reads the names and values of the dynamic variables. The script also sets a new value for the dynamic variable so that the script on the real-time hardware continues. For the complete example, refer to the `RTTLoader.py` file of the demo.

```

import rttmanagerlib
import rttutilities
import pythoncom
import os
import sys
workingDir = os.path.dirname(os.path.realpath(__file__))
if os.path.isdir(workingDir) == 0:
    workingDir = os.getcwd()
workingDir = os.path.abspath(workingDir)
workingDir = workingDir[:workingDir.rfind("\\")]
testScriptName = workingDir + r'\RTTSequences\RTTSequence.py'
import rttutilities

```

```

def executeDemo(boardName):
    rttManager = rttmanagerlib.RealTimeTestManagerServer()
    bcgFileName = rttManager.BCGServiceProvider.Generate(testScriptName,[])
    print("BCG file created: ", bcgFileName)
    board = rttManager.AccessBoard(boardName)
    print("Board '%s' successfully connected." % board.Name)
    sequencesEvents = rttdemoutilities.RTTMSequencesEvents(board.Sequences)
    variablesEvents = rttdemoutilities.RTTMVariablesEvents(board.Variables)
    try:
        sequence = board.Sequences.Create(bcgFileName)
        print("Sequence '%s' on real-time platform created. " % bcgFileName)
        sequence.Run()
        print("Sequence on real-time platform started.\n")
        # Suspend host script for the execution time of the real-time testing
        # script
        numberOfSeconds = 2
        rttutilities.RTTSleep(numberOfSeconds)
        print("\nCheck the dynamic variable collection on the host.")
        print("The dynamic variables were created during the RTT sequence " \
              "import phase.")
        for variable in board.Variables:
            print("Dynamic Variable '%s' = %f" % (variable.Name,
                                                  variable.Value))
        rttutilities.RTTSleep(numberOfSeconds)
        print("\nSet new values for dynamic variables on the real-time " \
              "hardware from host.")
        # Set new values for dynamic variables
        valueForDynamicVariables = 123.456
        for variable in board.Variables:
            print("Dynamic Variable '%s' = %f" % (variable.Name,
                                                  valueForDynamicVariables))
            variable.Value = valueForDynamicVariables
            valueForDynamicVariables += 1
            print("\nSuspend host script execution for %d seconds." \
                  % numberOfSeconds)
        rttutilities.RTTSleep(numberOfSeconds)
    ...

```

Event handling

The following example shows how an event handler can be implemented for events triggered by the dynamic variables.

```

class RTTMVariablesEvents(rttmanagerlib._IRTVariablesEvents):
    def __init__(self, EventSource = None, Parent = None):
        # Call base class constructor to connect to event source
        rttmanagerlib._IRTVariablesEvents.__init__(self, EventSource)
        self.Parent = Parent
    def OnAdd(self, Variable):
        """Method OnAdd"""
        Variable = rttmanagerlib.IRTVariable(Variable)
        print(VARIABLE_PREFIX)
        print(VARIABLE_PREFIX + r"OnAdd: New RTT variable '%s' created." \
              %(Variable.Name))
        print(VARIABLE_PREFIX + r"   Name:           '%s'" % (Variable.Name))
        print(VARIABLE_PREFIX + r"   SequenceName: '%s'" \
              %(Variable.SequenceName))

```

Related topics**Examples**

Demo Examples of Using Real-Time Testing..... 22

Exchanging Data Between RTT Sequences and Python Scripts Running on the Host PC

Introduction

You can transfer Python data objects from the simulation platform to the host PC's Python interpreter and vice versa by using host calls.

The following examples show an RTT sequence running on the simulation platform and a Python script running in the host PC's Python interpreter.

RTT sequence

The following RTT sequence is an example of sending data to a Python script on the host PC and receiving the return result. For a more complex example, refer to the demo <MyDocuments>\dSPACE\Real-Time Testing\5.0\11_HostCalls and see the RTTSequence.py file. For an example of a user-defined RTT sequence, refer to [Example of Exchanging User-Defined Data](#) on page 85.

Example The following Python script is a short example of an RTT sequence using a host call:

```
from rttlib import hostcall
def MainGenerator(*args):
    print("Start of real-time test execution on target platform")
    # List to store the results returned from host
    HostCallResults = []
    print("Send to Host: 42, 2")
    # Send a host call to the host PC and wait for the result
    # Syntax of host calls: yield hostcall.Hostcall([ReturnResult], \
    # Execution argument 1, Execution argument 2, ..., Execution argument n)
    yield hostcall.Hostcall(HostCallResults, 42, 2)
    print("Received from host: ", HostCallResults[0])
    # Insert 'yield None' to avoid a task overrun.
    # The script proceeds in the next sample step.
    yield None
```

Description `MainGenerator(*args)` is the main Real-Time Testing generator function. The name of the function is mandatory because it is the defined entry point for the script scheduler.

The `HostCallResults` variable is initialized as a list. It is used to store the return value received from the host PC.

`yield hostcall.Hostcall(HostCallResults, 42, 2)` is the function which sends the Python data object to the Python interpreter running on the host PC. This call suspends the script execution until the host PC responds and sends the return arguments. The Python data objects are evaluated on the host

PC by an event handler (see below). The event handler writes the result to the `HostCallResults` variable.

Host PC Python script

The following Python script shows the event handling for receiving a Python data object from the simulation platform and returning the result. For a more complex example, refer to the `RTTLoader.py` file of the demo. For an example of a user-defined host PC Python script, refer to [Example of Exchanging User-Defined Data](#) on page 85.

Note

You can use host calls only with the `rttmanagerlib` module. You cannot use the Real-Time Test Manager for host calls.

The Python interpreter on the host PC must be running and the event object, for example, `RTTMHostCallEvents`, must be valid.

Only one host PC Python script can handle host events of an RTT sequence.

If a host PC Python script tries to connect to a host event that is already connected to another host PC Python script, an exception occurs.

You can exchange data with a maximum size of 10 kBits.

Example The following Python script shows a part of a host Python script handling a host call:

```
...
class RTTMHostCallEvents(rttmanagerlib._IRTSequenceEvents):
    def __init__(self, Sequence, BoardName):
        # Call base class constructor to connect to event source
        rttmanagerlib._IRTSequenceEvents.__init__(self, Sequence)
        self.CurrentBoardName = BoardName
    def OnHostCall(self, *Data):
        """This method is called if an OnHostCall event is received
        in the current sequence.
        """
        # Calculate the return value
        ReturnResultToRT = Data[0]/Data[1]
        # The return arguments are sent back to the calling sequence
        # running on the platform (and now waiting for the answer
        # from the host).
        return ReturnResultToRT
...
# Initialize events
HostCallEvent = None
# Create new sequence
RTTManager = rttmanagerlib.RealTimeTestManagerServer()
# Generate byte code from the RTT sequence
BcgFileName = RTTManager.BCGServiceProvider.Generate(TEST_SCRIPT_NAME, [])
Board = RTTManager.AccessBoard(BoardName)
# Load a sequence to the platform.
Sequence = Board.Sequences.Create(BcgFileName)
# Connect to sequence event handle.
HostCallEvent = RTTMHostCallEvents(Sequence, BoardName)
```

```
# Start the sequence on the platform.
Sequence.Run()
```

Description The `RTTHostCallEvents` class is an event class to attach to Real-Time Testing host call events.

In the main function, the RTT sequence is created and registered for this event class. `OnHostCall(self, *Data)` is therefore called every time an `OnHostCall` event is received in the current sequence.

In this example two values are expected. The first value is divided by the second value. The result is returned to the RTT sequence.

In the last part, the simulation platform is accessed and the RTT sequence is created on the platform.

`HostCallEvent = RTTHostCallEvents(Sequence, BoardName)` connects the event handler to the RTT sequence. Now the Python script is able to receive host call events from the RTT sequence and evaluate them in the `OnHostCall` function definition.

For more details to handling host calls, refer to [Handling OnHostCall Events of an RTT Sequence in Python Scripts](#) on page 193.

Related topics

Basics

[Handling Events of RTT Sequences in Python Scripts](#)..... 191

Example of Exchanging User-Defined Data

Introduction

You can transfer user-defined Python data objects from the simulation platform to the host PC's Python interpreter and vice versa by using host calls. The following examples show a user-defined class, an RTT sequence running on the simulation platform, and a Python script running in the host PC.

User-defined class

The following script is an example of a user-defined class which can be used in the RTT sequence:

```
import math
class Circle:
    def __init__(self, radius):
        self.radius = radius
    def circumference(self):
        return 2 * math.pi * self.radius
```

RTT sequence

The following RTT sequence is an example of sending user-defined data to a Python script on the host PC and receiving the return result:

```
from Circle import *
from rttlib import hostcall
MyCircle=Circle(10.0)
return_args = []
def MainGenerator(*args):
    yield hostcall.Hostcall(return_args, MyCircle)
    print("<Host> Received Circle (c=", return_args[0].circumference(), ")")
```

Host PC Python script

The following Python script is an example of the event-handling for receiving a user-defined Python data object from the simulation platform and returning the result:

```
class RTTHostCallEvents(rttmanagerlib._IRTSequenceEvents):
    ...
    def OnHostCall(self, *Data):
        CircleData = Data[0]
        print("<Host> Received Circle (c=", CircleData.circumference(), ")")
        CircleData.radius = CircleData.radius / 2.0
        return CircleData
    def main():
    ...
```

Result

When this example is executed, the following message is output in the host PC's Python interpreter:

```
<Host> Received Circle ( c = 62.8318530718 )
<RTT> Received Circle ( c = 31.4159265 )
```

The RTT sequence creates the first circle object on the simulation platform with a radius of 10. It is sent to the host PC Python script via host call. On the host PC, the radius is divided by two and returned to the RTT sequence on the simulation platform.

Related topics**Basics**

Exchanging Data Between RTT Sequences and Python Scripts Running on the Host	
PC.....	83
Using User Python Modules.....	74

Data Replay in RTT Sequences

Introduction

You can stimulate variable objects in an RTT sequence by data replay of MAT file variables or ASAM MDF file channels.

Where to go from here

Information in this section

[Basics of Data Replay Using MAT Files..... 87](#)

You can stimulate variable objects in an RTT sequence by data replay of MAT file variables.

[Basics of Data Replay Using ASAM MDF \(MF4\) Files..... 90](#)

You can stimulate variable objects in an RTT sequence by data replaying ASAM MDF file variables.

[Replay Mode..... 93](#)

Real-Time Testing provides different modes for data replay that specifies the model step when data values are streamed.

[Example of Data Replay Using MAT Files..... 98](#)

The example shows how you can use data streaming in an RTT sequence to stream data of a MAT file.

[Example of Data Replay Using ASAM MDF \(MF4\) Files..... 101](#)

The example shows how you can use data streaming in an RTT sequence to stream data of an ASAM MDF (MF4) file.

Basics of Data Replay Using MAT Files

Introduction

You can use MAT file data to replay them on variable and dynamic variable objects in your RTT sequence. Data is streamed from the host PC to the real-time system during the run time of the RTT sequence. By using this mechanism, even large MAT files can be used for stimulation purposes.

MAT files

MAT files for data replay must contain at least two arrays. One array must contain the values for the time. All other arrays can be used to stimulate variables of the RTT sequence and finally the real-time application. Data must be of 'double' type. MAT files can only be used if they do not contain a substructure.

The data of the MAT file vectors is replayed without any modifications in the default replay mode. This includes the following:

- The variable value is set at the time given by the variable map time vector with a value given by the mapped data vector.
- There is no normalization of the time vector to zero. This means if the first entry in the MAT file time vector is '5.5', for example, the first value is stimulated 5.5 seconds after the start of the replay generator.
- A MAT file time vector resolution higher than the model step size causes an exception.

Tip

- The step size used in the MAT file can be higher than the model step size. If it is an integer multiple of the model step size, the amount of data is reduced which must be transferred for one variable and you can stream more variables in parallel.
- The number of data streams is limited and depends on the platform type. To reduce the number of data streams, you can combine signals from the same time base in one data stream. Refer to [Problem when Using Too Many Data Streams](#) on page 219.

Note

The MAT file you use must fulfill the following preconditions:

- The MAT file must contain at least two one-dimensional arrays. One array must contain monotonically increasing values for the time axis (x-axis).
- The data must be of 'double' type.
- The MAT file can be used only if it does not contain a substructure.

Variable Map

A variable map object is the mapping of MAT file data to variable objects and needed as input for data streaming. Each variable map refers to a unique time vector, whose name is passed to the constructor of the variable map object. Then a variable object and its associated MAT file vector name can be added to the map object.

The mapping is designed to be independent of any specific MAT file. The MAT file used for data streaming must include the vector names used in the mapping.

Before you can add variables to a variable map object, it must be created using the `CreateVariableMap` method. A MAT file vector/variable object pair can be added to an existing variable map. The objects in the MAT file must be of "double" data type. The variable objects are stimulated with the data content of the associated MAT file vector at the time stamps of the variable map time vector. Only mapped variables are stimulated.

The variable mapping must be created in the initialization phase of the RTT sequence and completed before the variable map object is used for creating a `MatFile` object.

Replay mode	<p>When signals are replayed using data streaming, only one value of a signal can be used in a model step. As time stamps of signals are normally not recorded in the same time period as the model is calculated (model step), algorithms are used to select the signal value to be used. Real-Time Testing provides four different modes for data replay, each mode presenting an implemented algorithm. The modes are selected by specifying the optional ReplayMode parameter of the MatFile class. For details, refer to Replay Mode on page 93.</p>
Starting data replay	<p>Data replay is started by the Replay method in the MainGenerator function. The replay starts in the execution step in which the generator function is called for the first time and ends after the generator finished.</p> <p>When the MAT file was replayed completely, you can call the Replay method again to restart the replay, so you can replay the same data over and over again. It is not necessary to reload the RTT sequence. However, if a datastream object is deleted (for example, by using del() or assigning None), it is not possible to restart the replay.</p>
Finishing data replay	<p>There are several ways for the generator to finish:</p> <ul style="list-style-type: none"> ▪ The MAT file was replayed completely. ▪ The generator is terminated by scheduler.ParallelRace(). ▪ Shutting down the Real-Time Test Manager Server stops data replay. ▪ Stopping or removing the RTT sequence. <p>You should remove RTT sequences using data replay in the following cases:</p> <ul style="list-style-type: none"> ▪ If they are not used anymore. ▪ Before the real-time application is reloaded. <p>When RTT sequences are removed, the datastream objects are deleted.</p>
Limitations for data replay	<ul style="list-style-type: none"> ▪ For DS1006-based modular systems connected to the host PC via Ethernet, data replay is not recommended, because it provides limited performance. The recommended connection type is a bus. Use a DS817 Link Board (PC) for the host PC. ▪ The performance of a DS817 Link Board depends on the architecture of the host PC. A DS817 has a PCI host interface. In newer PC models, the performance of the PCI interface has decreased. This leads to a reduced bandwidth for host communication of a connected DS817. ▪ For SCALEXIO real-time PCs of the first generation (based on the i7-860 processor) that have the Linux-based operating system, the performance is slightly reduced at the Linux-based operating system.

- To replay data, a DataStreaming Server is created at the host PC. The DataStreaming Server handles the data streaming for the corresponding RTT sequence. If an exception occurs during the initialization of the DataStreaming Server, it cannot be created and there is no exception or other error message transmitted to the RTT sequence. If the RTT sequence is started anyway, another exception is displayed, because the DataStreaming server is missing and data cannot be streamed.
- Exceptions detected during data replay cannot be handled using a try-except statement in the RTT sequence if they occur on the host PC. For example, if the difference between consecutive values of the 'Time' variable is lower than the step size of the model, an exception occurs. The DataStreaming Server passes this exception to the RTT sequence, but it cannot be handled because it occurred on the host PC. You can recognize exceptions occurred on the host PC by means of the missing "RTT Traceback" keyword and missing line number in the message text.
- It is not recommended to set up more than 8 data streams in parallel, even if they contain only one signal and one time vector.
- The path for MAT files used in RTT sequences, for example, for data replay, must not include non-ascii characters, i.e., characters that are not part of the local code page of the PC which is used for the test.
- Writing to target variables on the simulation platform is limited. Refer to [Variable Class \(Real-Time Testing Library Reference\)](#).
- The supported data type for MAT file vectors is 'double'.
- An instance of a `datastream.MatFile` object cannot be started by the `Replay` method twice in parallel.

Related topics

References

[rttlib.datastream Module \(Real-Time Testing Library Reference\)](#)

Basics of Data Replay Using ASAM MDF (MF4) Files


Introduction

You replay data of ASAM MDF version 4.1.x files (extension: **MF4**) on variable and dynamic variable objects in your RTT sequence. Data is streamed from the host PC to the real-time system during the run time of the RTT sequence. By using this mechanism, even large MDF files can be used for stimulation purposes.

Supported MDF files

Real-Time Testing supports ASAM MDF files in version 4.1.x. The files have the extension MF4.

MDF files must fulfill the following requirements so that they can be used for real-time testing:

- MDF files must include at least one data channel that a master channel is assigned to.
- The synchronization type of the channels to be streamed must be time.
- The MDF files must have 6 strings to identify the channels: group name, group source, group path, channel name, channel source, and channel path.
- All channels to be streamed in the one data stream must be in the same group of the MDF file.
- The MDF file must contain only 1:1 conversion, linear conversion, or rational conversion (if it can be traced back into a linear conversion).
- Some additional limitations apply to the MDF file. Refer to [Limitations for ASAM MDF Files \(Version 4.x\)](#) ([ControlDesk Measurement and Recording](#) ).

To stream only a part of the MDF file, you can specify a start value and duration for data streaming. The start value specifies the first value of the channels that are used for streaming. The duration value specifies the time range that is used.

Tip

- The step size used in the MDF file group's master channel can be higher than the model step size. If it is an integer multiple of the model step size, the amount of data that must be transferred for one variable is reduced and you can stream more variables in parallel.
- The number of data streams is limited and depends on the platform type. To reduce the number of data streams, you can combine signals from the same time base in one data stream. Refer to [Problem when Using Too Many Data Streams](#) on page 219.

Variable map

A variable map object is the mapping of MDF file channels to variable objects and is needed as input for data streaming. Each variable map refers to a unique time vector, which is specified by the MDF group's master channel. This channel is specified by the group name, group source, and group path and passed to the constructor of the variable map object. Then a variable object and its associated MDF file channels can be added to the map object.

Before you can add variables to a variable map object, it must be created using the `CreateVariableMapMDF` method. A MDF file channel/variable object pair can be added to an existing variable map. The variable objects are stimulated with the data content of the associated MDF channels at the time stamps of the MDF group's master channel. Only mapped channels are stimulated.

The variable mapping must be created in the initialization phase of the RTT sequence and completed before the variable map object is used for creating a `MDFFile` object.

Replay mode

When signals are replayed using data streaming, only one value of a signal can be used in a model step. Because time stamps of signals are normally not

recorded in the same time period in which the model is calculated (model step), algorithms are used to select the signal value to be used. Real-Time Testing provides four different modes for data replay, each mode representing an implemented algorithm. The modes are selected by specifying the optional **ReplayMode** parameter of the **MDFFile** class. Refer to [Replay Mode](#) on page 93.

Starting data replay

Data replay is started by the **Replay** method in the **MainGenerator** function. The replay starts in the execution step in which the generator function is called for the first time and ends after the generator finished.

When the MDF file was replayed completely or a specified duration expired, you can call the **Replay** method again to restart the replay, so you can replay the same data over and over again. It is not necessary to reload the RTT sequence. However, if a datastream object is deleted (for example, by using **del()** or assigning **None**), it is not possible to restart the replay.

Finishing data replay

There are several ways for the generator to finish:

- The specified duration expires.
- The MDF file was replayed completely.
- The generator is terminated by **scheduler.ParallelRace()**.
- Shutting down the Real-Time Test Manager Server stops data replay.
- Stopping or removing the RTT sequence.

You should remove RTT sequences using data replay in the following cases:

- If they are not used anymore.
- Before the real-time application is reloaded.

When RTT sequences are removed, the datastream objects are deleted.

Limitations for data replay

- For DS1006-based modular systems connected to the host PC via Ethernet, data replay is not recommended, because it provides limited performance. The recommended connection type is a bus. Use a DS817 Link Board (PC) for the host PC.
- The performance of a DS817 Link Board depends on the architecture of the host PC. A DS817 has a PCI host interface. In newer PC models, the performance of the PCI interface has decreased. This leads to a reduced bandwidth for host communication of a connected DS817.
- For SCALEXIO real-time PCs of the first generation (based on the i7-860 processor) that have the Linux-based operating system, the performance is slightly reduced at the Linux-based operating system.

- To replay data, a `DataStreaming Server` is created at the host PC. The `DataStreaming Server` handles the data streaming for the corresponding RTT sequence. If an exception occurs during the initialization of the `DataStreaming Server`, it cannot be created and there is no exception or other error message transmitted to the RTT sequence. If the RTT sequence is started anyway, another exception is displayed, because the `DataStreaming server` is missing and data cannot be streamed.
- Exceptions detected during data replay cannot be handled using a try-except statement in the RTT sequence if they occur on the host PC. For example, if the difference between consecutive values of the 'Time' variable is lower than the step size of the model, an exception occurs. The `DataStreaming Server` passes this exception to the RTT sequence, but it cannot be handled because it occurred on the host PC. You can recognize exceptions occurred on the host PC by means of the missing "RTT Traceback" keyword and missing line number in the message text.
- It is not recommended to set up more than 8 data streams in parallel, even if they contain only one signal and one time vector.
- The path for MAT files used in RTT sequences, for example, for data replay, must not include non-ascii characters, i.e., characters that are not part of the local code page of the PC which is used for the test.
- An instance of a `datastream.MDFFile` object cannot be started by the `Replay` method twice in parallel.

Related topics

References

[rttlib.datastream Module \(Real-Time Testing Library Reference\)](#) 

Replay Mode

Introduction

When data values are replayed, only one data value can be used per model step. As time stamps of the data values are normally not recorded in the same time period as the model is calculated (model step), algorithms are implemented to select the model step when the data values should be used.

Replay modes

Real-Time Testing provides four different modes for data replay. The modes are selected by specifying the `ReplayMode` attribute of the `MatFile` class.

RM_STRICT Data values are always used as soon as possible. If the time stamp meets exactly the model step, the data value is replayed in the model step. If the time stamp of the data value is between two model steps, the data value is replayed in the next (second) model step.

If more than one data value are available for a model step or the time between two data values is less than a model step, an exception is thrown.

This mode is used by default.

RM_SAMPLED Data values are used in the model step which is next to the time stamp of the value (the following or previous model step).

If more than one data value are available for a model step, the data value with the time stamp next to the model step is used. The other data values are discarded.

RM_LINEAR The replayed values are interpolated from the data values directly before and after the model step.

If the time stamp of a data value matches the model step, this data value is used.

If more than one value exists during a model step, the other data values are discarded.

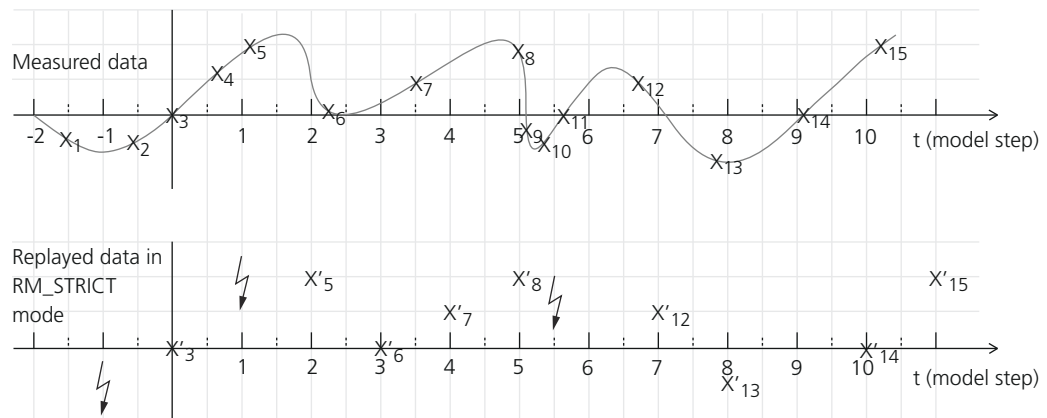
RM_BACKWARD Data values are always used as soon as possible. If the time stamp meets exactly the model step, the data value is replayed in the model step. If the time stamp of the data value is between two model steps, the data value is replayed in the next (second) model step.

If more than one data value are available for a model step, the data value with the time stamp next to the model step is used. The other data values are discarded.

Example

The following examples show an analog signal and its samples that are written to a data file. Below the measured signal the replayed data is shown. The replayed data is different in time and value depending on the replay mode used.

RM_STRICT mode The following graph shows the measured data and replayed data in the RM_STRICT mode.



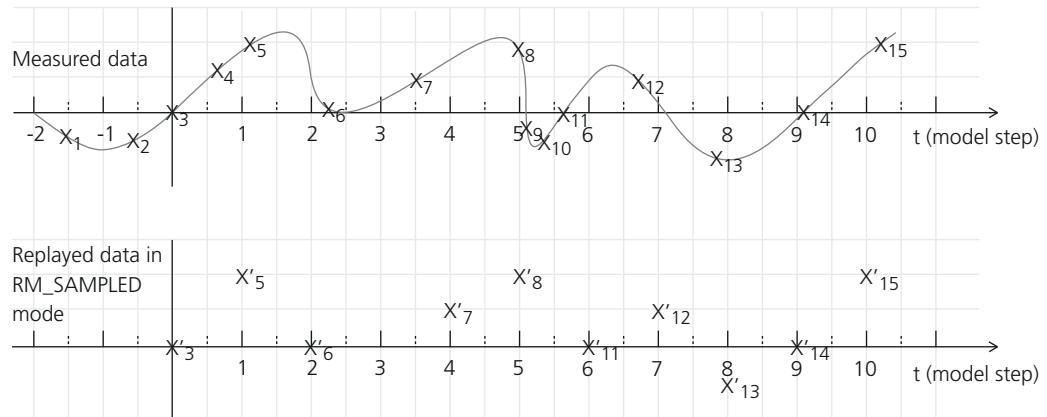
In this mode, the data values are used as soon as possible. If the time stamp meet exactly the model step, the data value is replayed in the model step (for example, X_3 , X_8). If the time stamp of the data value is between two model steps, the data value is replayed in the next model step (for example, X_5 , X_6).

If several data values are within one model step, an exception is thrown (for example, X_{10} , X_{11} , see the lightning symbol in the illustration above).

If the time between a data value and the previous data value is less than the model step size, an exception is thrown (for example, X_4 , X_9).

If a time stamp is negative, an exception is thrown (for example, X_1 , X_2).

RM_SAMPLED mode The following graph shows the measured data and replayed data in the RM_SAMPLED mode.

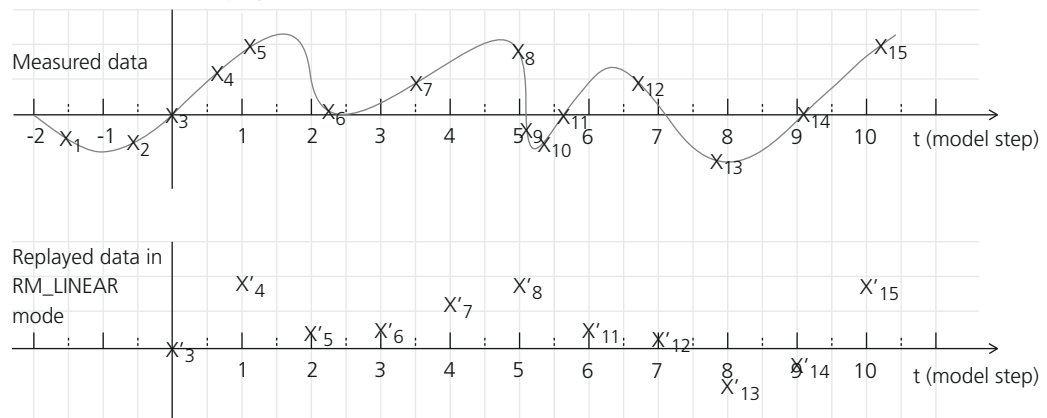


In this mode, the data values are used earlier (for example, X_5 , X_6) or later (for example, X_7 , X_{11}) than measured. If several data values are within one model step, only the nearest value to the model step is used, the other data values are discarded (for example, X_4 , X_9 , X_{10}). Data values with time stamps which fits exactly on a model step are replayed on the model step (X_3 , X_8).

If the last time stamp is not a multiple of a model step, the replay ends in the model step before the time stamp.

If time stamps are negative, their data values are discarded (for example, X_1 , X_2).

RM_LINEAR mode The following graph shows the measured data and replayed data in the RM_LINEAR mode.



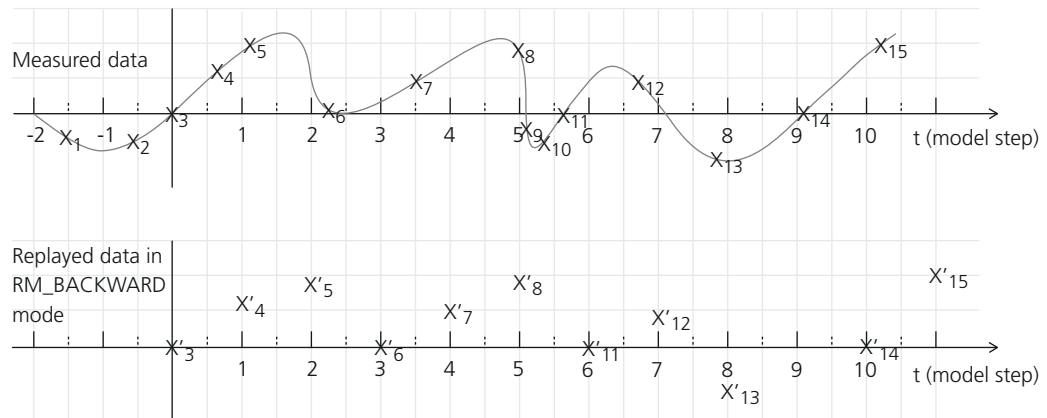
In this mode, the replayed value is interpolated using the data values that are directly before and after the model step. This means that the replayed value and time stamps are adapted. Only if data values with time stamps which fits exactly on a model step are replayed on the model step (for example, X_3 , X_8).

When time stamps are negative, the replay starts in the first model step ($t=0$) using the replayed value that is interpolated by the data values of the last

negative time stamp and first positive time stamp. All other data values which have negative time stamps are discarded (for example, X_1 , X_2).

When the last time stamp is not a multiple of a model step, the replay ends before the last time stamp. Otherwise the replay ends in the model step which matches the last time stamp.

RM_BACKWARD mode The following graph shows the measured data and replayed data in the RM_BACKWARD mode.

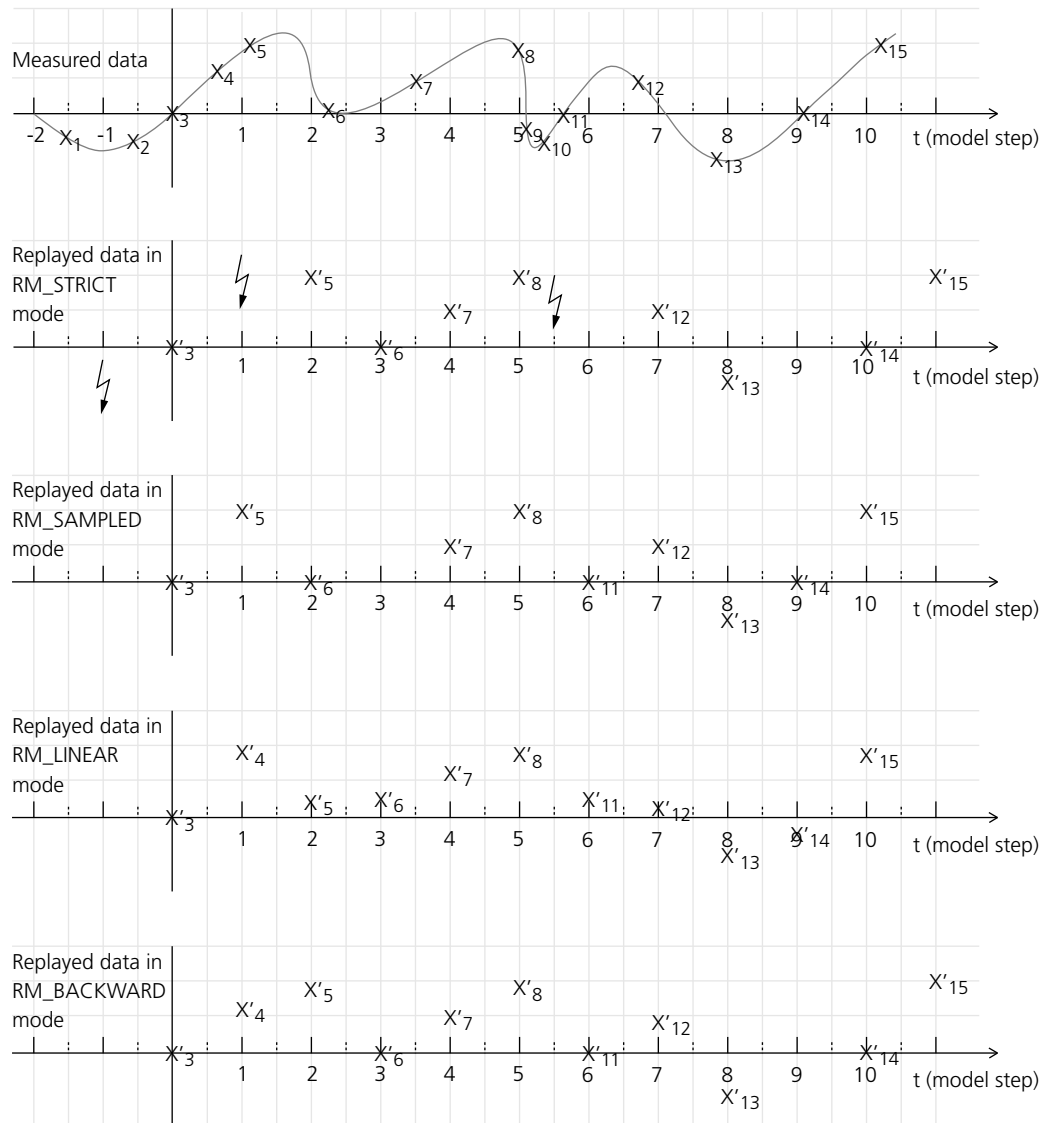


In this mode, the data values are used as soon as possible. If the time stamp meet exactly the model step, the data value is replayed in the model step (for example, X_3 , X_8). If the time stamp of the data value is between two model steps, the data value is replayed in the next model step (for example, X_5 , X_6). If several data values are within one model step, only the nearest data value to the model step is used, the other data values are discarded (for example, X_9 , X_{10}).

If time stamps are negative, the replay starts in the first model step at $t = 0$ using the data value of the previous negative time stamp. All other data values which have negative time stamps are discarded.

The replay ends in the model step after the last time stamp.

Overview of all the replay modes The following illustration shows an overview of the replay modes.



Related topics

Basics

[Data Replay in RTT Sequences.....87](#)

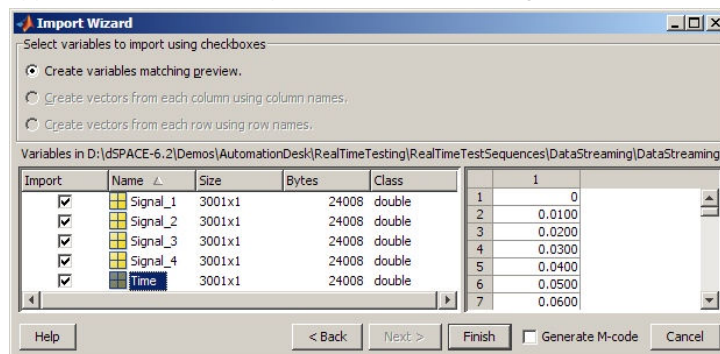
References

[rttlib.datastream Module \(Real-Time Testing Library Reference !\[\]\(339a16584d5da0f0a3ca4e9ec17bf6a1_img.jpg\)](#))

Example of Data Replay Using MAT Files

Introduction

The example shows how data replay in an RTT sequence works. The MAT file used for data replay is named 'DataStreaming.mat'. In this demo, the full path is passed to the RTT sequence as a sequence argument. It contains five arrays with the same dimension named **Time** and **Signal_x** (x = 1 ... 4). The signal values ("Signal_x") are independent of each other but have a common time vector ("Time"). The values in these arrays are used to stimulate variables in a real-time application. MATLAB's Import Wizard shows the signal values:



Example

The example below is a part of the RTT sequence of a demo in `<MyDocuments>\dSPACE\Real-Time Testing\5.0\10_DataStreaming`. The experiment with the real-time application which is stimulated is installed in `SampleExperiments\TurnSignal_<platform>`.

```
#-----
# Import all classes from the rttlib module.
# This Python library provides all real-time testing modules.
# All imports must be defined in the global part of the script.
#-----
from rttlib import variable
from rttlib import utilities
from rttlib import datastream
```

```

#-----
# Module global variables
#-----
# Create variable objects for accessing Simulink signals
currentTime      = utilities.currentTime
if variable.IsA2L():
    # the variable description for this application is an A2L file.
    WarningLightSwitch = variable.Variable(r'WarningLightSwitchValue')
    TurnSignalLeft = variable.Variable(r'RearLightEcuTurnSignalLeft')
    BatteryVoltage = variable.Variable(r'BatteryVoltageValue')
    TurnSignalLever = variable.Variable(r'TurnSignalLeverValue')
else:
    # the variable description for this application is a TRC file.
    WarningLightSwitch = variable.Variable(r'Model Root/WarningLightSwitch[0|1]/Value')
    TurnSignalLeft = variable.Variable(r'Model Root/RearLightEcu/TurnSignalLeft')
    BatteryVoltage = variable.Variable(r'Model Root/BatteryVoltage[V]/Value')
    TurnSignalLever = variable.Variable(r'Model Root/TurnSignalLever[-1..1]/Value')
fileToBeStreamed = ''
#-----
# Read sequence argument passed to sequences.Create()
#-----
try:
    sequenceArgument = utilities.GetSequenceArgument()
    # Check for optional arguments
    if sequenceArgument:
        if isinstance(sequenceArgument, str) \
            or isinstance(sequenceArgument, unicode):
            # Get the name of the MAT file
            fileToBeStreamed = sequenceArgument
        else:
            raise Exception("Sequence argument is not a string.")
    except:
        raise Exception("Sequence argument is missing. Start the sequence with\"
            " the script 'RTTLoader.py'.")
# Map variable objects to the MAT file variables
# The constructor uses the MAT file time variable name
variablesToStimulate = datastream.CreateVariableMap("Time")
# ALL variables objects must be mapped to a MAT file variable name.
variablesToStimulate.AddVariable("Signal_1", WarningLightSwitch)
variablesToStimulate.AddVariable("Signal_2", TurnSignalLeft)
variablesToStimulate.AddVariable("Signal_3", BatteryVoltage)
variablesToStimulate.AddVariable("Signal_4", TurnSignalLever)
# Create a data stream
myStream = datastream.MatFile(fileToBeStreamed, variablesToStimulate)
# Identifier for real-time testing print messages
rttPrefix = r" *RTT:* "
def MainGenerator():
    """
    Function: MainGenerator
    This function is the main real-time testing generator function.
    The name of the function is mandatory because it is the defined
    entry point for the script scheduler.
    """
    print(rttPrefix + "Start of real-time test execution on target platform")
    yield None
    print(rttPrefix + "Start data streaming at '%0.6f' " % currentTime.Value)
    # Start the data replay
    yield myStream.Replay()
    yield None
    print(rttPrefix + "Start data streaming again at '%0.6f' " \
        % currentTime.Value)

```

```
# Start the data replay again
yield myStream.Replay()
print(rttPrefix + "Finished data streaming at '%0.6f' " % currentTime.Value)
yield None
print(rttPrefix + "End of real-time test execution.")
```

Description

This section describes only the parts of the RTT sequence which relate to the data streaming feature.

```
from rttlib import datastream
```

To access the array of a MAT file, you must include the `datastream` module. It contains all necessary methods.

```
VariablesToStimulate = datastream.CreateVariableMap("Time")
```

The `CreateVariableMap` method creates an empty map for mapping MAT file variables to RTT variables. It is called with the name of the time variable included in the MAT file ("Time") and acts as a common time base for all stimulation variables. The method returns the map object, which is used to store information on mapping the variables of the MAT file to the variables of the RTT sequence.

```
VariablesToStimulate.AddVariable("Signal_1", WarningLightSwitch)
```

The `AddVariable` method must be called to map a variable of the MAT file to an RTT variable object. The variable object (dynamic or Simulink variable) must be created before being inserted in the map object. You must map each variable which you want to use for stimulation.

```
MyStream = datastream.MatFile(MatFileName, VariablesToStimulate)
```

The `MatFile` method is called with the name of the MAT file and the mapping object. It returns a stream object which coordinates data streaming. All these preparatory operations (creating variable objects, map objects, `datastream` objects) must be executed in the RTT sequence init phase.

```
yield MyStream.Replay()
```

The `Replay()` method executes data streaming. The host PC reads the values of the variables in the MAT file continuously from the MAT file and writes them to the mapped variables of the RTT sequence. Because it is a generator function, it must be prefixed with the `yield` statement. The `Replay` method must only be used in the `MainGenerator` function.

Related topics

Basics

[Peaks in Turnaround Time..... 215](#)

References

[rttlib.datastream Module \(Real-Time Testing Library Reference !\[\]\(cbd8541a32dfc32f356f5c6c994b0a21_img.jpg\)](#))

Example of Data Replay Using ASAM MDF (MF4) Files

Introduction

The example shows how data replay in an RTT sequence works. The ASAM MDF file used for data replay is named 'DataStreaming.mf4'. In this demo, the full path is passed to the RTT sequence as a sequence argument. It contains four channels with the same dimension named **Sine Wave**, **Stairs**, **Add Const Sine Wave**, and **Ramp**. The values in these channels are used to stimulate variables in a real-time application.

Example

The following example below is a part of the RTT sequence of a demo in `<MyDocuments>\dSPACE\Real-Time Testing\5.0\10_DataStreaming`. The experiment with the real-time application that is stimulated is installed in `SampleExperiments\TurnSignal_<platform>`.

```
#-----
# Import all classes from the rttlib module.
# This Python library provides all Real-Time Testing modules.
# All imports must be defined in the global part of the script.
#-----
from rttlib import variable
from rttlib import utilities
from rttlib import datastream
#-----
# Module-global variables
#-----
# Create variable objects for accessing Simulink signals.
currentTime = utilities.currentTime
if variable.IsA2L():
    # the variable description for this application is an A2L file.
    WarningLightSwitch = variable.Variable(r'WarningLightSwitchValue')
    TurnSignalLeft = variable.Variable(r'RearLightEcuTurnSignalLeft')
    BatteryVoltage = variable.Variable(r'BatteryVoltageValue')
    TurnSignalLever = variable.Variable(r'TurnSignalLeverValue')
else:
    # the variable description for this application is a TRC file.
    WarningLightSwitch = variable.Variable(r'Model Root/WarningLightSwitch[0|1]/Value')
    TurnSignalLeft = variable.Variable(r'Model Root/RearLightEcu/TurnSignalLeft')
    BatteryVoltage = variable.Variable(r'Model Root/BatteryVoltage[V]/Value')
    TurnSignalLever = variable.Variable(r'Model Root/TurnSignalLever[-1..1]/Value')
fileToBeStreamed = ''
#-----
# Read sequence argument passed to sequences.Create().
#-----
try:
    sequenceArgument = utilities.GetSequenceArgument()
```

```

# Check for optional arguments.
if sequenceArgument:
    if isinstance(sequenceArgument, str) \
        or isinstance(sequenceArgument, unicode):
        # Get the name of the MF4 file.
        fileToBeStreamed = sequenceArgument
    else:
        raise Exception("Sequence argument is not a string.")
except:
    raise Exception("Sequence argument is missing. Start the sequence with" \
        " the script 'RTTLoader.py'.")

# Map variable objects to the channels of the MDF file.
# The constructor uses the group name, source, and path of a group in an MDF file.
variablesToStimulate = datastream.CreateVariableMapMDF("Base Task", "Turnlamp", "Demo Signals")
# ALL variables objects must be mapped to a channel within the group used for the variable map constructor.
# The method uses the name, source and path of a channel in an MDF File.
variablesToStimulate.AddVariable("Sine Wave", WarningLightSwitch, "Turnlamp", "Demo Signals")
variablesToStimulate.AddVariable("Stairs", TurnSignalLeft, "Turnlamp", "Demo Signals")
variablesToStimulate.AddVariable("Add Const Sine Wave", BatteryVoltage, "Turnlamp", "Demo Signals")
variablesToStimulate.AddVariable("Ramp", TurnSignalLever, "Turnlamp", "Demo Signals")
# Create a data stream.
myStream = datastream.MDFFile(mf4FileName, variablesToStimulate, ReplayMode = datastream.RM_BACKWARD)
# Identifier for Real-Time Testing print messages.
rttPrefix = r" *RTT:* "
def MainGenerator():
    """
    Function: MainGenerator
    This function is the main Real-Time Testing generator function.
    The name of the function is mandatory because it is the defined
    entry point for the script scheduler.
    """
    print(rttPrefix + "Start of real-time test execution on target platform")
    yield None
    print(rttPrefix + "Start data streaming at '%0.6f' " % currentTime.Value)
    # Start the data replay.
    yield myStream.Replay()
    yield None
    print(rttPrefix + "Start data streaming again at '%0.6f' " \
        % currentTime.Value)
    # Start the data replay again.
    yield myStream.Replay()
    print(rttPrefix + "Finished data streaming at '%0.6f' " % currentTime.Value)
    yield None
    print(rttPrefix + "End of real-time test execution.")

```

Description

This section describes only the parts of the RTT sequence that relate to the data streaming feature.

```
from rttlib import datastream
```

To access the channels of an MDF file, you must include the `datastream` module. It contains all necessary methods.

```
VariablesToStimulate = datastream.CreateVariableMapMDF("Base
Task", "Turnlamp", "Demo Signals")
```

The `CreateVariableMapMDF` method creates an empty map for mapping MDF file channels to RTT variables. It is called with the group name, group source, and group path. The MDF group's master channel provides the common time base

for all stimulation variables. The method returns the map object, which is used to store information on mapping the channels of the MDF file to the variables of the RTT sequence.

```
VariablesToStimulate.AddVariable("Sine Wave",
WarningLightSwitch, "Turnlamp", "Demo Signals")
```

The **AddVariable** method must be called to map a channel of the MDF file to an RTT variable object. The variable object (dynamic or Simulink variable) must be created before being inserted in the map object. You must map each channel you want to use for stimulation.

```
MyStream = datastream.MDFFile(mf4FileName, VariablesToStimulate, ReplayMode =
datastream.RM_BACKWARD)
```

The **MDFFile** method is called with the name of the MDF file and the mapping object. It returns a stream object that coordinates data streaming. All these preparatory operations (creating variable objects, map objects, and datastream objects) must be executed in the RTT sequence init phase.

```
yield MyStream.Replay()
```

The **Replay()** method executes data streaming. The host PC reads the values of the variables continuously from the MDF file and writes them to the mapped variables of the RTT sequence. Because it is a generator function, it must be prefixed with the **yield** statement. The **Replay** method must be used only in the **MainGenerator** function.

Related topics

Basics

[Peaks in Turnaround Time..... 215](#)

References

[rttlib.datastream Module \(Real-Time Testing Library Reference !\[\]\(c444627dab9fee9a1550c053ffaaaae2_img.jpg\)\)](#)

Handling CAN Messages

Introduction You can send and receive CAN messages in the raw format in an RTT sequence.

Where to go from here

Information in this section

[CAN Messages in RTT Sequences](#)..... 104

You can use Real-Time Testing to send and receive CAN messages.

[Handling CAN Messages Using the `rttlib.canlib` Module](#)..... 105

You can use the `rttlib.canlib` module when the Simulink model uses the RTI CAN MultiMessage Blockset.

[Handling CAN Messages Using the `rttlib.dscanapilib` Module](#)..... 123

You can use the `rttlib.dscanapilib` module when the Simulink model does not use the RTI CAN MultiMessage Blockset.

CAN Messages in RTT Sequences

Implementing CAN Communication in RTT Sequences

Introduction You can transmit and receive CAN and CAN FD messages in RTT sequences. Real-Time Testing has two different modules for implementing CAN communication.

`rttlib.canlib` module

The `rttlib.canlib` module can be used if the CAN communication of the simulation model is implemented via the RTI CAN MultiMessage Blockset. Simulink models that are implemented using this blockset, provide experimental CAN messages. The experimental messages are used by the `rttlib.canlib` module to send and receive CAN messages in raw format.

Refer to [Handling CAN Messages Using the `rttlib.canlib` Module](#) on page 105.

`rttlib.dscanapilib` module

The `rttlib.dscanapilib` module provides an interface to CAN controller of the simulation system where the RTT sequence is executed. Currently SCALEXIO systems, VEOS, and MicroAutoBox III are supported. The `rttlib.dscanapilib` module can access the CAN controller to send and receive CAN messages in raw format. In contrast to the `rttlib.canlib` module, no predefined experimental

messages are necessary and it is not necessary that the Simulink model is implemented using the RTI CAN MultiMessage Blockset.

Refer to [Handling CAN Messages Using the rttlib.dscanapilib Module](#) on page 123.

Related topics

Basics

Handling CAN Messages Using the rttlib.canlib Module.....	105
Handling CAN Messages Using the rttlib.dscanapilib Module.....	123

Handling CAN Messages Using the rttlib.canlib Module

Introduction

You can handle CAN messages using the `rttlib.canlib` module when the Simulink model uses the RTI CAN MultiMessage Blockset.

Where to go from here

Information in this section

Basics on the rttlib.canlib Module.....	106
Real-Time Testing provides the <code>rttlib.canlib</code> module to receive and send CAN or CAN FD messages in raw data format. This feature requires that your Simulink model contains blocks of the RTI CAN MultiMessage Blockset.	
Basics on Handling CAN FD Messages with the rttlib.canlib Module.....	108
You can receive or send CAN FD messages in raw data format in an RTT sequence using the <code>rttlib.canlib</code> module.	
How to Prepare the Simulink Model for CAN Message Handling.....	109
To handle CAN messages, you must configure blocks of the RTI CAN MultiMessage Blockset for real-time testing.	
How to Prepare the Simulink Model for CAN FD Message Handling.....	111
To handle CAN FD messages, you must configure blocks of the RTI CAN MultiMessage Blockset for real-time testing.	
How to Use a Prepared RTT CAN Model with a SCALEXIO System.....	115
To make communication via the CAN bus for a SCALEXIO system available, you must build the signal chain for CAN communication in ConfigurationDesk using CAN function blocks.	
Accessing the CAN Bus with the rttlib.canlib Module.....	117
Before you send or receive CAN messages, you must access the CAN bus.	

Sending CAN Messages with the `rttlib.canlib` Module..... 118

You can send raw data of a CAN message.

Receiving CAN Messages with the `rttlib.canlib` Module..... 121

You can receive raw data of a CAN message.

Basics on the `rttlib.canlib` Module

Introduction

Real-Time Testing provides the `rttlib.canlib` module to receive and send CAN or CAN FD messages in raw data format. This feature requires that your Simulink model contains blocks of the RTI CAN MultiMessage Blockset.

Supported real-time platforms

You can use the `rttlib.canlib` module on

- A modular system based on a DS1006 or DS1007 that contains an I/O board with CAN interface, for example, a DS2211
- MicroAutoBox
- SCALEXIO systems that contain I/O boards with a CAN controller (DS2671 Bus Board, DS6301 CAN/LIN Board) or a DS2680 I/O Unit with the DS2672 Bus Module

Experimental message

Experimental messages are the CAN messages which are specified in an experiment software, for example, Real-Time Testing. Experimental messages are specified in RTT sequences and need not be implemented in the Simulink model. This means you can specify them even when the real-time application is already running.

CAN message format

This version of Real-Time testing supports raw data CAN messages in the standard or extended format.

RTI CAN MultiMessage Blockset

Handling CAN messages with the `rttlib.canlib` module requires the RTI CAN MultiMessage Blockset, which prepares the experimental messages. You must therefore specify the maximum experimental messages in an RTI block. This maximum is valid for all RTT sequences running on the real-time hardware. Refer to [How to Prepare the Simulink Model for CAN Message Handling](#) on page 109 or [How to Use a Prepared RTT CAN Model with a SCALEXIO System](#) on page 115.

rttlib.canlib module

The `rttlib.canlib` module contains all classes and methods which are necessary for handling CAN messages. For reference information on the `rttlib.canlib` module, refer to [rttlib.canlib Module \(Real-Time Testing Library Reference !\[\]\(c507f772dba2b921f86777f01218e570_img.jpg\)](#)).

Implementing CAN communication

Before you send or receive CAN messages, you must access the CAN bus. Refer to [Accessing the CAN Bus with the rttlib.canlib Module](#) on page 117.

You can send CAN messages to the CAN bus. The CAN messages are specified as raw data. They can have the standard or extended frame format. Refer to [Sending CAN Messages with the rttlib.canlib Module](#) on page 118.

You can read CAN messages which are transmitted on the CAN bus. The messages are read in raw format. They can have the standard or extended frame format. Refer to [Receiving CAN Messages with the rttlib.canlib Module](#) on page 121.

Bus Navigator

To monitor the CAN messages or to send CAN messages, you can use the Bus Navigator of ControlDesk. For information on its features, refer to [Overview of the Bus Navigator \(ControlDesk Bus Navigator !\[\]\(3e2231b1ad3ca8da8658228c00dd08e0_img.jpg\)](#)).

Demos

Three demos for using the canlib module are available in the `RTTSequences\canlib` folder of the demo archive `<MyDocuments>\dSPACE\Real-Time Testing\5.0\12_CAN`.

Name	Description
<code>RTTSequence_Basic.py</code>	Demonstrates how you can send a CAN message in standard frame format. A cyclic message is transferred every 50 ms (ten times).
<code>RTTSequence_Advanced.py</code>	Demonstrates how you can send CAN messages in standard and extended frame format. A standard CAN message is sent in the standard frame format and after a delay of one second in the extended frame format.
<code>RTTSequence_Professional.py</code>	Demonstrates how you can receive CAN messages in an RTT sequence. The functions for transmitting and receiving CAN messages are executed synchronously.

A model, which has to be prepared for use with the demo, is installed with the RTI CAN MultiMessage Blockset. To work with the tutorial model, you must adapt it to the platform which you use for real-time testing. For details, refer to [Working with the Prepared Demo Results \(RTI CAN MultiMessage Blockset Tutorial !\[\]\(0d5ec72f61334709c3fc9450209b754f_img.jpg\)](#)).

Limitation

You can access only the CAN or CAN FD controllers which are on the CPU where the RTT sequence is running. Accessing a CAN or CAN FD controller on a remote CPU in a multiprocessor system is not supported.

Related topics**HowTos**

[How to Prepare the Simulink Model for CAN Message Handling.....](#) 109
[How to Use a Prepared RTT CAN Model with a SCALEXIO System.....](#) 115

Examples

[Demo Examples of Using Real-Time Testing.....](#) 22

References

[rttlib.canlib Module \(Real-Time Testing Library Reference !\[\]\(aa53ad6fea213b8b2226d3077e30533a_img.jpg\)](#))

Basics on Handling CAN FD Messages with the rttlib.canlib Module

Introduction

You can receive or send CAN FD messages in raw data format in an RTT sequence using the `rttlib.canlib` module.

Real-Time Testing version

This feature is supported as of Real-Time Testing 2.5.

Support of CAN FD protocol

Real-Time Testing supports the classic CAN protocol and the CAN FD protocol. The CAN FD protocol allows data rates higher than 1 Mbit/s and payloads of up to 64 bytes per message. CAN FD messages are handled in the same way as CAN messages.

Keep in mind that the CAN FD protocol is supported only by dSPACE platforms equipped with a CAN FD-capable CAN controller.

For further information, refer to [Basics on Working with CAN FD \(RTI CAN MultiMessage Blockset Reference !\[\]\(cbd8541a32dfc32f356f5c6c994b0a21_img.jpg\)](#)).

Data code length

For CAN FD messages, the maximum data length is 64 byte. Only the following DLC values are valid: 0, 1, 2, 3, 4, 5, 6, 7, 8, 12, 16, 20, 24, 32, 48, 64.

Checking CAN FD support

You can check whether the CAN FD support is enabled for the controller. To check this, the controller class has the `IsCANFDSupportEnabled` attribute which you can evaluate. See the following example.

```
# Access the bus
MyCANBus = r"BusSystems/CAN/Chassis"
MyController = canmlib.GetController(MyCANBus)
MyChannel = MyController.GetChannel()
# Get experimental message
Msg1 = MyChannel.GetRawMessage()
# Initialize message1
If MyController.IsCANFDSupportEnabled == 1:
    # CAN FD is supported by this controller
    Msg1.Format = canmlib.canmmbase.lib.ftFDSTD
```

Related topics**References**

[rttlib.canlib Module \(Real-Time Testing Library Reference !\[\]\(0aff635c4179ba9e710b00f4b01d3b20_img.jpg\)](#))

How to Prepare the Simulink Model for CAN Message Handling

Objective

To handle CAN messages, you must configure blocks of the RTI CAN MultiMessage Blockset for real-time testing. This document describes only the steps required for real-time testing. For information on how to work with the RTI CAN MultiMessage Blockset, refer to [RTI CAN MultiMessage Blockset Tutorial !\[\]\(8bba887393ca45b761e5cb49e755e762_img.jpg\)](#).

Demo model

A demo model which can be modified for real-time testing is installed with the RTI CAN MultiMessage Blockset. For details, refer to [Working with the Prepared Demo Results \(RTI CAN MultiMessage Blockset Tutorial !\[\]\(47734e4656765d20df4fdbd5b7aff048_img.jpg\)](#)).

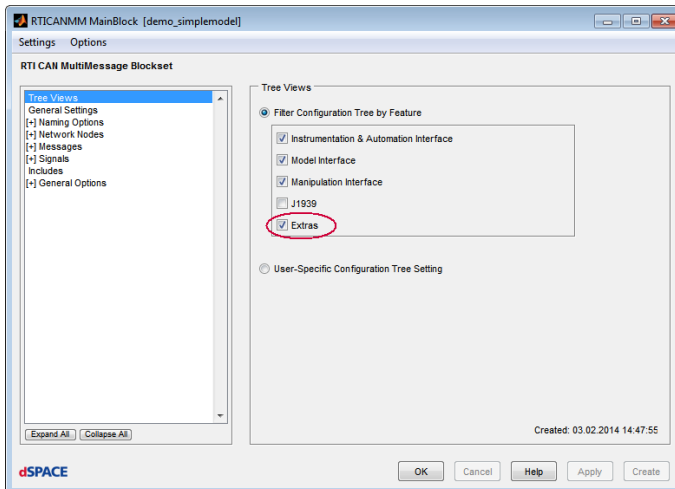
Preconditions

- Your Simulink model must contain the RTI blocks of the RTI CAN MultiMessage Blockset:
 - RTICANMM ControllerSetup
 - RTICANMM GeneralSetup
 - RTICANMM MainBlock
- The RTICANMM ControllerSetup block must be configured for the dSPACE hardware used. For details, refer to [Working with the Prepared Demo Results \(RTI CAN MultiMessage Blockset Tutorial !\[\]\(11a0966cbb90b5c1d6ebfc666ec75f78_img.jpg\)](#)).

Method

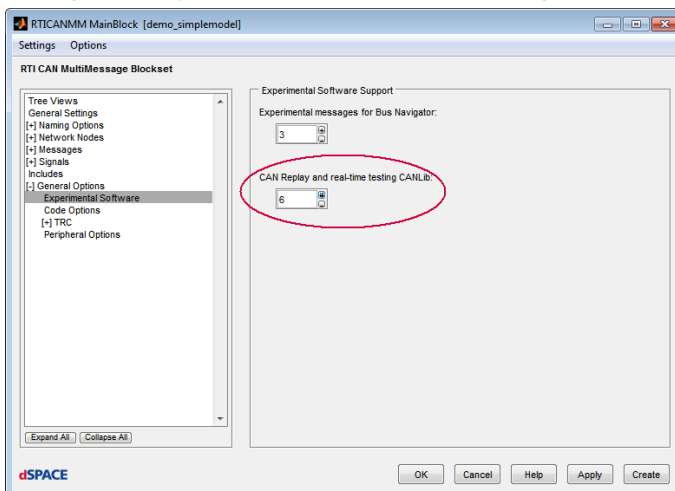
To prepare the Simulink model for CAN message handling

- 1 Open your Simulink model.
- 2 Open the RTICANMM MainBlock block.
- 3 On the Tree Views page, select Extras.






The dialog tree gets some additional pages, in particular, the Experimental Software page.




- 4 In the dialog tree, click General Options – Experimental Software to open the Experimental Software page.
- 5 On the Experimental Software page in the CAN Replay and real-time testing CANLib field, specify the maximum number of experimental messages which you want to use for real-time testing.



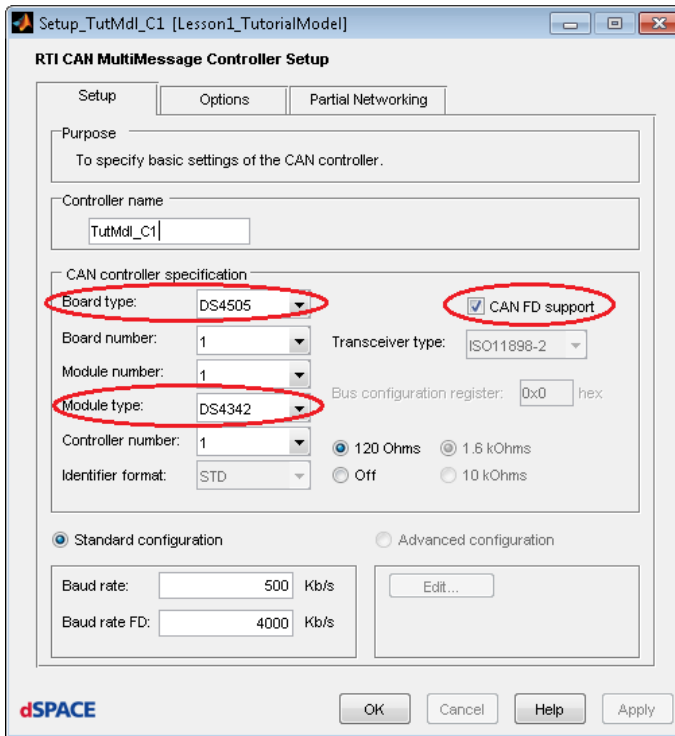
- 6 Click Apply to confirm your settings.
- 7 Click Create to create all the necessary files and close the dialog.
- 8 Click OK to confirm your settings and close the block dialog.

Result	The Simulink model is prepared for CAN message handling by RTT sequences.
Next steps	<p>If you have a SCALEXIO system, the signal chain must be built in ConfigurationDesk, see How to Use a Prepared RTT CAN Model with a SCALEXIO System on page 115.</p> <p>Build the real-time application, refer to Enabling Real-Time Testing for dSPACE Platforms on page 33.</p>
Related topics	<p>References</p> <div> RTICANMM ControllerSetup (RTI CAN MultiMessage Blockset Reference ) RTICANMM GeneralSetup (RTI CAN MultiMessage Blockset Reference ) RTICANMM MainBlock (RTI CAN MultiMessage Blockset Reference ) </div>

How to Prepare the Simulink Model for CAN FD Message Handling

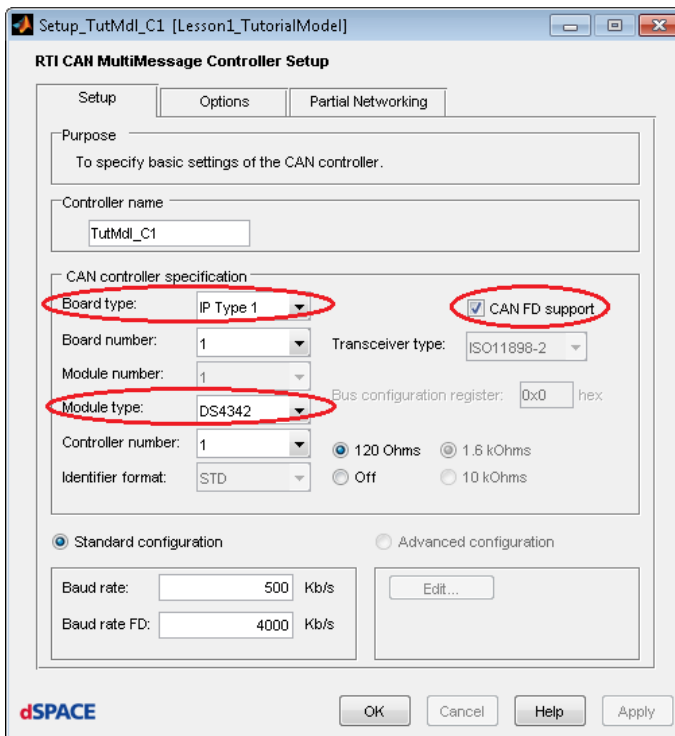
Objective	To handle CAN FD messages, you must configure blocks of the RTI CAN MultiMessage Blockset for real-time testing. This document describes only the steps required for real-time testing. For information on how to work with the RTI CAN MultiMessage Blockset, refer to RTI CAN MultiMessage Blockset Tutorial  .
Demo model	A demo model which can be modified for Real-Time Testing is installed with the RTI CAN MultiMessage Blockset. For details, refer to Working with the Prepared Demo Results (RTI CAN MultiMessage Blockset Tutorial ) .
Preconditions	<ul style="list-style-type: none"> ▪ Your Simulink model must contain the RTI blocks of the RTI CAN MultiMessage Blockset: <ul style="list-style-type: none"> ▪ RTICANMM ControllerSetup ▪ RTICANMM GeneralSetup ▪ RTICANMM MainBlock ▪ The RTICANMM ControllerSetup block must be configured for the dSPACE hardware used. For details, refer to Working with the Prepared Demo Results (RTI CAN MultiMessage Blockset Tutorial ).
Method	<p>To prepare the Simulink model for CAN FD message handling</p> <ol style="list-style-type: none"> 1 Open your Simulink model. 2 Open the RTICANMM Controller Setup block.

- 3 For DS1006 or DS1007 platforms:
 1. In Board type list, select the I/O board type used, for example, DS4505.
 2. In Module type list, select the CAN FD controller used, for example, DS4342.
 3. Select CAN FD Support.

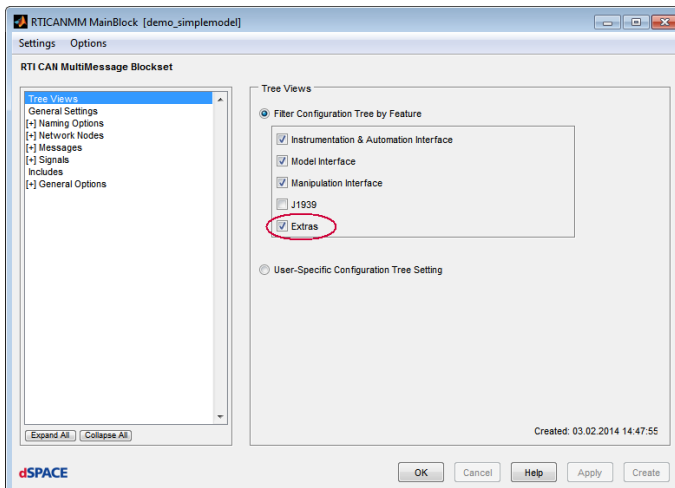


For DS1401:

1. In Board type list, select the I/O board type used, for example, IP Type 1.
2. In Module type list, select the CAN FD controller used, for example, DS4342.
3. Select CAN FD Support.



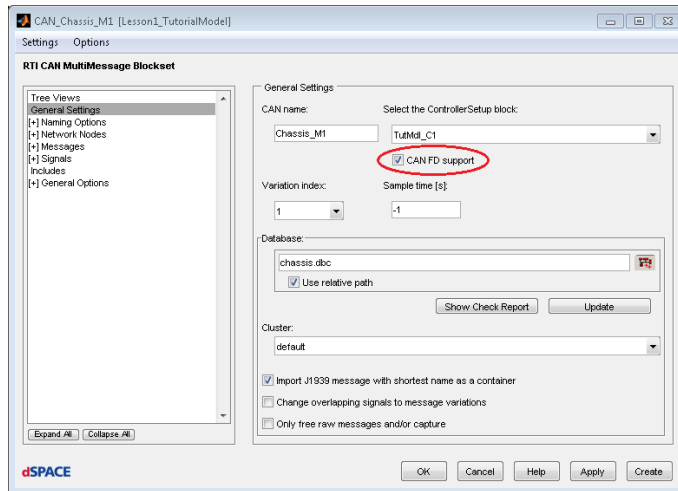
- 4 Open the RTICANMM MainBlock block.
- 5 On the Tree Views page, select Extras.



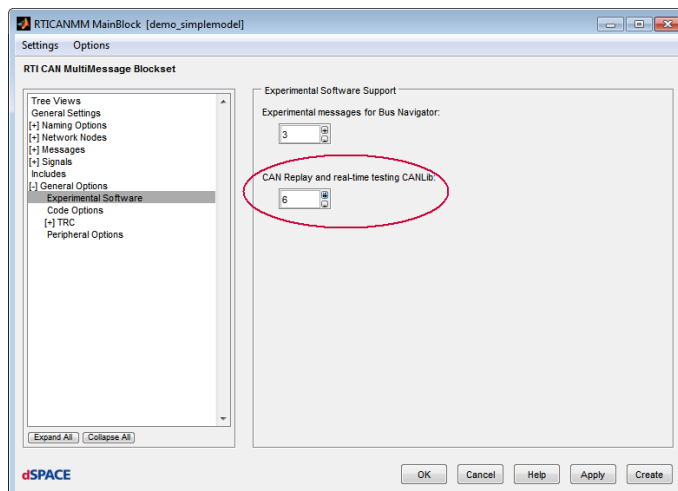
The dialog tree gets some additional pages, in particular, the Experimental Software page.

- 6 In the dialog tree, click General Settings to open the General Settings page.

- 7 On the General Settings page, select the CAN FD support.



- 8 In the dialog tree, click General Options – Experimental Software to open the Experimental Software page.
- 9 On the Experimental Software page in the CAN Replay and real-time testing CANLib field, specify the maximum number of experimental messages which you want to use for real-time testing.



- 10 Click Apply to confirm your settings.
- 11 Click Create to create all the necessary files and close the dialog.
- 12 Click OK to confirm your settings and close the block dialog.

Result

The Simulink model is prepared for CAN FD message handling by RTT sequences.

Next steps

Build the real-time application, refer to [Enabling Real-Time Testing for dSPACE Platforms](#) on page 33.

Related topics

References

[RTICANMM ControllerSetup \(RTI CAN MultiMessage Blockset Reference !\[\]\(4729e517bc6a7cd81c8025b9646574fb_img.jpg\)\)](#)
[RTICANMM GeneralSetup \(RTI CAN MultiMessage Blockset Reference !\[\]\(90a2fb2f2c617b26262139ae4159c0a0_img.jpg\)\)](#)
[RTICANMM MainBlock \(RTI CAN MultiMessage Blockset Reference !\[\]\(40394d85fb59f1a516df36b5a2680ad2_img.jpg\)\)](#)

How to Use a Prepared RTT CAN Model with a SCALEXIO System

Objective

To make communication via the CAN bus available, you must build the signal chain for CAN communication in ConfigurationDesk using CAN function blocks.

Basics

For basic information on hardware assignment in ConfigurationDesk for SCALEXIO systems, refer to [Assigning Hardware Resources to Function Blocks \(ConfigurationDesk Real-Time Implementation Guide !\[\]\(3e2231b1ad3ca8da8658228c00dd08e0_img.jpg\)\)](#).

Precondition

The Simulink model must be prepared for CAN message handling, see [How to Prepare the Simulink Model for CAN Message Handling](#) on page 109.

Method

To use a prepared RTT CAN model with a SCALEXIO system

- 1 In the Model Browser, open the model's context menu and select **Analyze Model (Complete Analysis)**.

The Simulink model with the modeled CAN communication is opened in MATLAB. ConfigurationDesk analyzes the model's interfaces and the model components. ConfigurationDesk creates one configuration port block for every identified RTICANMM ControllerSetup block and gives it the same name. The Model Browser displays the created configuration port blocks.

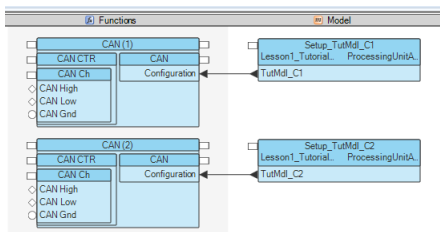
- 2 From the Model Browser, drag one of the configuration port blocks for the CAN communication to the graphical window.

The configuration port block is added to the signal chain. It is displayed in the model section of the graphical window. The configuration port of the configuration port block is named after the controller specified in the associated RTICANMM ControllerSetup block in the Simulink model.

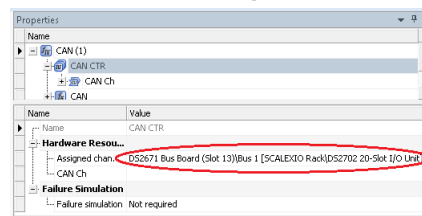
- 3 From the Function Browser, drag a CAN function block to the graphical window.

ConfigurationDesk adds the instantiated CAN function block to the signal chain.

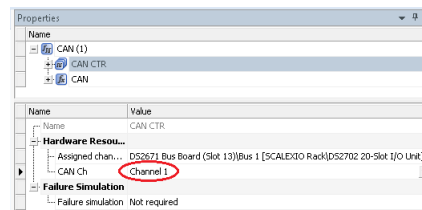
- 4 Map the Configuration function port of the CAN function block to the configuration port of the associated configuration port block.



- 5 To assign a hardware resource to be used with the CAN function block, perform the following steps for both CAN function blocks.
1. In the graphical window, select a CAN CTR electrical interface unit of a CAN function block.
 2. In the Properties Browser, select the hardware resource to be assigned to the block in the Assigned channel set property.



3. Select a suitable channel of the hardware resource in the CAN Ch property.



4. Repeat the previous steps with a different channel for the second CAN function block.
- 6 Check the hardware assignment and modify the settings if necessary.

Result

The model with implemented CAN communication is prepared for the build process.

Related topics

Basics

Assigning Hardware Resources to Function Blocks (ConfigurationDesk Real-Time Implementation Guide )
 Basics on the rttlib.canlib Module..... 106

Accessing the CAN Bus with the rttlib.canlib Module

Introduction

Before you send or receive CAN messages, you must access the CAN bus.


Example

The example shows how to get an object to access the CAN bus using the canlib module. The example is not ready-to-use, it shows only the relevant code.

```
from rttlib.canlib.controllers import canmmlib
MyCANBus = r"BusSystems/CAN/Chassis"
MyController = canmmlib.GetController(MyCANBus)
MyChannel = MyController.GetChannel()
```

Description

In the example the `canmmlib` module from the `rttlib` module is imported.

The `MyCANBus` variable contains the TRC path of the local board to the CAN bus. This is the path which is displayed in the variable tree of ControlDesk's Variables controlbar. The first two parts of the TRC path are always `BusSystems/CAN` when you access a CAN bus. The last part is the CAN bus name which is specified on the General Settings page of the RTICANMM MainBlock (see [General Settings Page \(RTICANMM MainBlock\)](#) (RTI CAN MultiMessage Blockset Reference )).

The `GetController()` method provides a Controller object of the specified CAN bus.

The `GetChannel()` method provides a Channel object which is finally used to send or receive CAN messages. As the controller has only one channel, the parameter of `GetChannel` can be omitted.

Next steps

Now you can send and receive CAN messages:

- [Sending CAN Messages with the rttlib.canlib Module](#) on page 118
- [Receiving CAN Messages with the rttlib.canlib Module](#) on page 121

Related topics

Basics

[Basics of the Variables Controlbar \(ControlDesk Variable Management !\[\]\(e50091943b385fe16d3277389202856f_img.jpg\)\)](#)

References

[GetChannel Method \(Real-Time Testing Library Reference !\[\]\(179f167ede0522ebb4ea025b3ad78ca7_img.jpg\)\)](#)

[GetController Method \(Real-Time Testing Library Reference !\[\]\(4a7b4ce770af8456e11a71f9565c8c2b_img.jpg\)\)](#)

Sending CAN Messages with the rttlib.canlib Module

Introduction

You can send raw data of a CAN message using the canlib module. You can trigger the transmission of a CAN message within a defined sampling step, but the time when the CAN message is transmitted depends on the actual CAN bus communication (i.e. a high bus load can delay the CAN message transmission).

Setting raw data

Raw data to be sent depends on the Data and DLC attributes of the message object. The Data attribute specifies the value of the raw data. The DLC attribute specifies how many bytes of raw data are set. It is not necessary to specify all eight bytes of a CAN message. If the Data value contains more bytes than specified with DLC, the left bytes of Data are ignored. If the Data value contains fewer bytes than specified with DLC, the lowest bytes of raw data are filled with 0.

The following table shows some examples. In some of them, the DLC value does not match the number of bytes specified with Data to describe the behavior of RTT.

Data Value	DLC Value	Raw Data (Byte 0 ... Byte 7)
0x123	2	01 23 00 00 00 00 00 00
0x0123	2	01 23 00 00 00 00 00 00
0x123	1	23 00 00 00 00 00 00 00
0x123	3	00 01 23 00 00 00 00 00
0x12345678	1	78 00 00 00 00 00 00 00
0x12345678	2	56 78 00 00 00 00 00 00
0x12345678	3	34 56 78 00 00 00 00 00
0x12345678	4	12 34 56 78 00 00 00 00

Setting the DLC greater than 8

Setting the DLC to a value greater than 8 is not allowed if the message format is not a CAN FD type.

Wrong example The following code shows a wrong example:

```
# Get experimental message
Msg1 = MyChannel.GetRawMessage()
# Initialize message1
Msg1.Format = canmmlib.canmmbase.lib.ftSTD
Msg1.DLC = 64 # CAUSES AN EXCEPTION
```

Correct example The following code shows a correct example:

```
# Get experimental message
Msg1 = MyChannel.GetRawMessage()
# Initialize message1
Msg1.Format = canmmlib.canmmbase.lib.ftFDSTD
Msg1.DLC = 64 # THIS IS ALLOWED
```

Non CAN FD message with a DLC greater than 8

Setting the message format to a non CAN FD type is not allowed if the message DLC is greater than 8.

Wrong example The following code shows a wrong example:

```
# Get experimental message
Msg1 = MyChannel.GetRawMessage()
# Initialize message1
Msg1.Format = canmlib.canmmbase.lib.ftFDSTD
Msg1.DLC = 64
Msg1.Format = canmlib.canmmbase.lib.ftSTD # CAUSES AN EXCEPTION
```

Correct example The following code shows a correct example:

```
# Get experimental message
Msg1 = MyChannel.GetRawMessage()
# Initialize message1
Msg1.Format = canmlib.canmmbase.lib.ftFDSTD
Msg1.DLC = 64
Msg1.DLC = 8
Msg1.Format = canmlib.canmmbase.lib.ftSTD # THIS IS ALLOWED
```

CAN FD not supported

Setting the message format to a CAN FD type is not allowed if the controller does not support CAN FD messages.

Wrong example The following code shows a wrong example:

```
# Access the bus
MyCANBus = r"BusSystems/CAN/Chassis"
MyController = canmlib.GetController(MyCANBus)
MyChannel = MyController.GetChannel()
# Get experimental message
Msg1 = MyChannel.GetRawMessage()
# Initialize message1
If MyController.IsCANFDSupportEnabled == 0:
    # CAN FD is NOT supported by this controller
    Msg1.Format = canmlib.canmmbase.lib.ftFDSTD # CAUSES AN EXCEPTION
```

Correct example The following code shows a correct example:

```
# Access the bus
MyCANBus = r"BusSystems/CAN/Chassis"
MyController = canmlib.GetController(MyCANBus)
MyChannel = MyController.GetChannel()
# Get experimental message
Msg1 = MyChannel.GetRawMessage()
# Initialize message1
If MyController.IsCANFDSupportEnabled == 1:
    # CAN FD is supported by this controller
    Msg1.Format = canmlib.canmmbase.lib.ftFDSTD # THIS IS ALLOWED
```

Example

The example shows how you can send messages on a CAN bus. The example is not ready-to-use, it shows only the code which is relevant for sending the messages. You must access the CAN bus beforehand (see [Accessing the CAN Bus with the rtlib.canlib Module](#) on page 117).

```

# Get experimental messages
Msg1 = MyChannel.GetRawMessage()
Msg2 = MyChannel.GetRawMessage()
Msg3 = MyChannel.GetRawMessage()
# Initialize message1
Msg1.Format = canmlib.canmmbaselib.ftSTD
Msg1.ID = 0x123
Msg1.DLC = 8
Msg1.TX.Data = 0x1020304050607080
# Initialize message2
Msg2.Format = canmlib.canmmbaselib.ftSTD
Msg2.ID = 0x124
Msg2.DLC = 3
Msg2.TX.Data = 0x10203
# Initialize message3
Msg3.Format = canmlib.canmmbaselib.ftEXT
Msg3.ID = 0x123456
Msg3.DLC = 8
Msg3.TX.Data = 0x8070605040302010
# Trigger the transmission of the messages with timeout
yield Msg1.TransmitGen(20)
yield Msg2.TransmitGen(20)
yield Msg3.TransmitGen(20)
# Trigger the transmission of the messages all together in
# one sampling step without waiting for the transmission
Msg1.Transmit()
Msg2.Transmit()
Msg3.Transmit()
yield None
yield None
# Delete the message objects
Msg1 = None
Msg2 = None
Msg3 = None

```

Description

The `GetRawMessage()` method reserves a free experimental message for the CAN message. This example requires three free experimental messages. The maximum number of allowed experimental messages is set in the `RTICANMM MainBlock`. Note that the maximum value is specified for all RTT sequences created on the real-time platform.

After the experimental messages are reserved, they are initialized. This example initializes message1 and message2 in standard frame format and message3 in the extended frame format. The following values are initialized:

Message	Data	DLC	Raw Data
0x123	0x1020304050607080	8	10 20 30 40 50 60 70 80
0x124	0x10203	3	01 02 03
0x123456	0x8070605040302010	8	80 70 60 50 40 30 20 10

After the messages are initialized, they are sent using the `TransmitGen(20)` method. The method waits until the message is transmitted. This means it is a blocking method and can take several sampling steps to finish. If it waits longer than 20 sampling steps, it exits without sending the message and raises the

canmmerror. The messages are transmitted one after another in different sampling steps.

After the first transmission, the messages are transmitted a second time. In this case the non-blocking `Transmit()` method is used. This triggers the transmission of all messages in one sampling step. However, it cannot be guaranteed that the messages are sent in the same sampling step.

Tip

You can read the `IsReady` attribute of the message object to check whether a message was sent to the CAN controller or not, for example:

- `Msg1.TX.IsReady = 1`: Data of Msg1 is sent to the CAN controller. The Msg1 can be configured for the next transmission.
- `Msg1.TX.IsReady = 0`: Data of Msg1 was not sent yet.

When the message objects are no longer used, they must be deleted to free the experimental messages for other RTT sequences.

Related topics

Basics

[Basics on the rttlib.canlib Module..... 106](#)
[canmmerror \(Real-Time Testing Library Reference !\[\]\(faf942dc3e59ce8eb64b4ac481eca7e0_img.jpg\)\)](#)

HowTos

[How to Prepare the Simulink Model for CAN Message Handling..... 109](#)

References

[GetRawMessage Method \(Real-Time Testing Library Reference !\[\]\(95b425611cbd2b8716a140cf67c81822_img.jpg\)\)](#)
[Transmit Method \(Real-Time Testing Library Reference !\[\]\(98475352b625a273242ad989dd0cabc3_img.jpg\)\)](#)
[TransmitGen Method \(Real-Time Testing Library Reference !\[\]\(116076dec7d8ea5879a528db6ccb1b4b_img.jpg\)\)](#)

Receiving CAN Messages with the rttlib.canlib Module

Introduction

You can receive raw data of a CAN message using the canlib module.

Example

The example shows a function definition which reads a CAN message with the `MessageID` ID and prints its data. This function definition is part of the `RTTSequence_Professional.py` demo, located in the demo archive `<MyDocuments>\dSPACE\Real-Time Testing\5.0\12_CAN`.

```

def ReceiveFunctionGen(MessageID):
    # Use global Channel object
    global Channel
    print(RTT_PREFIX + "Start receive of CAN messages.")
    try:
        # Create a raw messages
        Message = Channel.GetRawMessage()
        # Initialize the message data
        Message.Format = canmmlib.canmmbase.lib.ftSTD
        Message.ID = MessageID
        RXCounter = Message.RX.Counter
        while(1):
            Message.Receive()
            if RXCounter < Message.RX.Counter:
                # New message was received
                RXCounter = Message.RX.Counter
                print(RTT_PREFIX + "t = %f [ID = %d] Receive Data: 0x%x" \
                    % (Message.RX.TimeStamp, Message.ID, Message.RX.Data))
            yield None
        finally:
            # Clean up
            Message = None
    print(RTT_PREFIX + "Receive of CAN messages finished.")
    yield None

```

Description

The **Channel** object must be created outside the function definition. This is required for accessing the CAN bus. For details, refer to [Accessing the CAN Bus with the rttlib.canlib Module](#) on page 117.

Message = Channel.GetRawMessage() reserves an experimental message for this function definition.

After the message object is created, it is initialized. The message to be received has the standard frame format and the ID specified by the **MessageID** parameter.

The **RXCounter** variable is set to the value of **Message.RX.Counter**. The **Message.RX.Counter** is raised when a message with the specified ID is received. You can compare it with the old value to see whether a message was received.

The RTT sequence waits for the message with the specified message ID in an endless loop. Each time the message is received, its data is printed and the **RXCounter** is set to the new counter value.

In the final section, the message object is deleted to free the reserved experimental object.

Related topics

Basics

[Basics on the rttlib.canlib Module](#)..... 106

[canmmerror \(Real-Time Testing Library Reference !\[\]\(1d3a1175dd4902218e694b9c098adb83_img.jpg\)\)](#)

Examples

[Demo Examples of Using Real-Time Testing..... 22](#)

References

[GetRawMessage Method \(Real-Time Testing Library Reference !\[\]\(a03a7eb2f4046e1d3c76772003e549ea_img.jpg\)\)](#)

[Receive Method \(Real-Time Testing Library Reference !\[\]\(cbe2492b119e39e02a1dab2af4a4b296_img.jpg\)\)](#)

[ReceiveGen Method \(Real-Time Testing Library Reference !\[\]\(e474458956c9a37fbf9586ddb60a7fa1_img.jpg\)\)](#)

Handling CAN Messages Using the `rttlib.dscanapilib` Module

Introduction

You can handle CAN Messages using the `rttlib.dscanapilib` module.

Where to go from here

Information in this section

[Basics of the `rttlib.dscanapilib` Module..... 124](#)

Real-Time Testing provides the `rttlib.dscanapilib` module to receive and send CAN or CAN FD messages in raw data format.

[How to Prepare a CAN Channel for Using it with the `rttlib.dscanapilib` Module..... 125](#)

To use CAN channel with the `rttlib.dscanapilib` module, it must be prepared in `ConfigurationDesk`.

[Accessing the CAN Bus with the `rttlib.dscanapilib` Module..... 127](#)

Before you send or receive CAN messages, you must access the CAN channels.

[Sending CAN Messages with the `rttlib.dscanapilib` Module..... 129](#)

You can send raw data of a CAN message.

[Receiving CAN Messages with the `rttlib.dscanapilib` Module..... 132](#)

You can receive raw data of a CAN message.

[Unregistering CAN Channels with the `rttlib.dscanapilib` Module..... 136](#)

You must unregister a CAN channel when it is not required any longer.

Basics of the `rttlib.dscanapilib` Module

Introduction	Real-Time Testing provides the <code>dscanapilib</code> module to receive and send CAN or CAN FD messages in raw data format.
Real-Time Testing version	This feature is supported as of Real-Time Testing 3.0.
Supported simulation platforms	<p>You can use the <code>rttlib.dscanapilib</code> module on the following platforms:</p> <ul style="list-style-type: none"> ▪ SCALEXIO Processing Unit ▪ DS6001 Processor Board ▪ MicroAutoBox III ▪ VEOS <p>A hardware platform must have an I/O board or module with CAN controller.</p>
<code>rttlib.dscanapilib</code> module	<p>The <code>rttlib.dscanapilib</code> module contains all classes and methods which are necessary for handling CAN messages. The object model is analog to the API commands of the dSPACE CAN API, so you can reuse scripts written for the dSPACE CAN API to use them for real-time testing.</p> <p>Message format The <code>rttlib.dscanapilib</code> module supports raw data CAN and CAN FD messages in the standard or extended format.</p> <p>Unsupported classes and methods Most of the classes and methods of the <code>rttlib.dscanapilib</code> module are used identically to the classes and commands of the dSPACE CAN API. However, some classes and methods are not suitable for real-time testing and therefore not available in the <code>rttlib.dscanapilib</code> module:</p> <ul style="list-style-type: none"> ▪ <code>GetVendorInformation</code> (method of <code>dscanapilib</code> class) ▪ <code>SetEventNotification</code> (method of <code>dscanapilib</code> class) ▪ <code>ChannelSearchAttribute</code> class ▪ <code>VendorInfo</code> class <p>Reference information For reference information on the <code>rttlib.dscanapilib</code> module, refer to rttlib.dscanapilib Module (Real-Time Testing Library Reference).</p>
Implementing CAN bus communication	<p>The ConfigurationDesk application must contain CAN function block for each CAN channel that you want to use. Refer to How to Prepare a CAN Channel for Using it with the <code>rttlib.dscanapilib</code> Module on page 125.</p> <p>Before you send or receive CAN messages, you must access the CAN bus. Refer to Accessing the CAN Bus with the <code>rttlib.dscanapilib</code> Module on page 127.</p>

You can send CAN messages to the CAN bus. The CAN messages are specified as raw data. They can have the standard or extended frame format. Refer to [Sending CAN Messages with the `rttlib.dscanapilib` Module](#) on page 129.

You can read CAN messages which are transmitted on the CAN bus. The messages are read in raw format. They can have the standard or extended frame format. Refer to [Receiving CAN Messages with the `rttlib.dscanapilib` Module](#) on page 132.

Demos

Three demos for using the `canlib` module are available in the `RTTSequences\dscanapilib` folder of the demo `<MyDocuments>\dSPACE\Real-Time Testing\5.0\12_CAN`.

Name	Description
<code>RTTSequence_Basic.py</code>	Demonstrates how you can send a CAN message in standard frame format. A cyclic message is transferred every 50 ms (ten times).
<code>RTTSequence_Advanced.py</code>	Demonstrates how you can send CAN messages in standard and extended frame format. A standard CAN message is sent in the standard frame format and after a delay of one second in the extended frame format.
<code>RTTSequence_Professional.py</code>	Demonstrates how you can receive CAN messages in an RTT sequence. The functions for transmitting and receiving CAN messages are executed synchronously.

Limitation

You can access only the CAN or CAN FD controllers which are on the CPU where the RTT sequence is running. Accessing a CAN or CAN FD controller on a remote CPU in a multiprocessor system is not supported.

Related topics

Examples

[Demo Examples of Using Real-Time Testing..... 22](#)

How to Prepare a CAN Channel for Using it with the `rttlib.dscanapilib` Module

Objective

To use CAN channel with the `rttlib.dscanapilib` module, it must be prepared in `ConfigurationDesk`.

Basics

When you use the `rttlib.dscanapilib` module for handling CAN messages, the `ConfigurationDesk` application must contain a CAN function block for each

CAN channel you want to use. When you have used a channel for implementing a CAN bus communication in the model, the ConfigurationDesk application already has a CAN function block specified for this channel.

You can implement CAN bus communication on your dSPACE platform by using the RTI CAN MultiMessage Blockset, the Bus Manager, or V-ECU implementations.

- For further information on implementing CAN bus communication by using the RTI CAN MultiMessage Blockset, refer to [Modeling a CAN Bus Interface \(Model Interface Package for Simulink - Modeling Guide !\[\]\(756219e9389f679d57027482aa5cf5fc_img.jpg\)](#).
- For further information on implementing CAN bus communication by using the Bus Manager, refer to [ConfigurationDesk Bus Manager Implementation Guide !\[\]\(fcb77b2d9531d23794a07d244b7a89bc_img.jpg\)](#).
- For further information on implementing CAN bus communication by using V-ECU implementations, refer to [Special Aspects of V-ECU Implementations Containing CAN Controllers \(ConfigurationDesk Real-Time Implementation Guide !\[\]\(8175e06aff05874f50e11ffc448e6860_img.jpg\)](#).

It is possible to use a channel for Real-Time Testing which is not already used for an implementation of CAN bus communication. However, to be able to access this channel, you must insert a CAN function block into the ConfigurationDesk application and assign the channel to it. It is not necessary to connect the CAN function block within the signal chain, for example, a model port block.

Preconditions

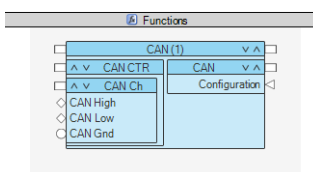
The CAN channel is not used for CAN bus communication or another bus communication.

Method

To prepare the ConfigurationDesk application for using `rtlib.dsacanapilib` module

- 1 Open the ConfigurationDesk application that is used to build the real-time application used for real-time testing.
- 2 From the Function Browser, drag a CAN function block to the graphical window.

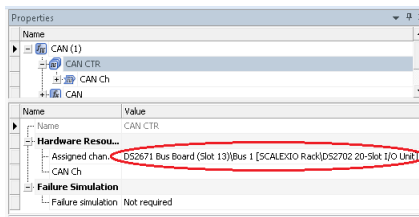
ConfigurationDesk adds the instantiated CAN function block to the signal chain.



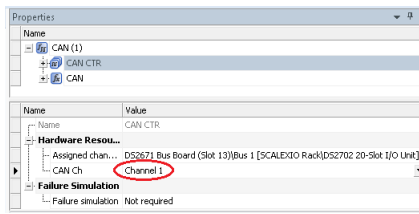
To assign a hardware resource to be used with the CAN function block, perform the following steps.

- 3 In the graphical window, select a CAN CTR electrical interface unit of a CAN function block.

- 4 In the Properties Browser, select the hardware resource to be assigned to the block in the Assigned channel set property.



- 5 Select a suitable channel of the hardware resource in the CAN Ch property.



- 6 Repeat the previous steps for further channels.

Result

You can register the channels for CAN communication with the `rttlib.dscanapilib` module.

Related topics

Basics

[Implementing Bus Communication in the Signal Chain Using the Bus Manager \(ConfigurationDesk Bus Manager Implementation Guide !\[\]\(d0262bbe9d2356661a2e89321dfcc781_img.jpg\)\)](#)
[Modeling a CAN Bus Interface \(Model Interface Package for Simulink - Modeling Guide !\[\]\(8572950e410320d7dd023da827ff014d_img.jpg\)\)](#)
[Special Aspects of V-ECU Implementations Containing CAN Controllers \(ConfigurationDesk Real-Time Implementation Guide !\[\]\(b2b6a2e56e47cc582ad4ec3c8f1864c0_img.jpg\)\)](#)

Accessing the CAN Bus with the `rttlib.dscanapilib` Module

Introduction

Before you send or receive CAN messages, you must access the CAN channels.

Example

The example shows the definition of a function which is used to get objects to access the CAN channels using the `dscanapilib` module. It shows only the relevant code for accessing the CAN channels.

```
from rttlib import utilities
from rttlib import scheduler
from rttlib import dscanapilib
```

```

# Module global variables
# Get variable object
ChannelInfoObjects = []
ChannelHandles = []
def initCAN(TimeOutSteps = 20):
    """
        The function shows how to initialize the CAN channels with
        the rttlib.dscanapilib.
    """
    global ChannelInfoObjects
    global ChannelHandles
    # Get the information about available CAN channels.
    ChannelInfoObjects = dscanapilib.GetAvailableChannels()
    yield None
    # Initialize every available CAN channel.
    for channel in ChannelInfoObjects:
        ChannelHandle = dscanapilib.RegisterChannel(channel.VendorName, \
                                                    channel.InterfaceName, \
                                                    channel.InterfaceSerialNumber, \
                                                    channel.ChannelIdentifier)

        # Add channel handle to global list.
        ChannelHandles.append(ChannelHandle)
        # Initialize the CAN channel.
        AccessPermission = None
        while(AccessPermission == None):
            AccessPermission = dscanapilib.InitChannel(ChannelHandle, \
                                                        dscanapilib.ctSTDXTD, \
                                                        20, True)

            TimeOutSteps -= 1
            if(TimeOutSteps <= 0):
                raise Exception("Could not initialize CAN channels.")
            yield None
        yield None
        dscanapilib.ActivateChannel(ChannelHandle)
        yield None

```

Description

The `dscanapilib` module from the `rttlib` library must be imported to get its classes and methods. The `GetAvailableChannels` method returns a list of all the available CAN channels. The list can be iterated, so that all the available CAN channels can be registered and initialized.

The CAN channels are initialized using the `InitChannel` method. The method can return three values:

- **None:** The channel is not yet initialized.
- **True:** The channel is initialized, the RTT sequence has access permission.
- **False:** The channel is initialized, the RTT sequence has no access permission.

The initialization can last several model steps. It is therefore done in a while loop by evaluating the return value of `InitChannel`. To avoid waiting too long for a channel initialization which may not be possible, a timeout is implemented.

When the code has finished, the `ChannelHandles` list contains all the channel handles for the available CAN channels.

Next steps

Now you can send and receive CAN messages:

- [Sending CAN Messages with the `rttlib.dscanapilib` Module](#) on page 129
- [Receiving CAN Messages with the `rttlib.dscanapilib` Module](#) on page 132

Final step

When a CAN channel is not used any longer, you must unregister it. Refer to [Unregistering CAN Channels with the `rttlib.dscanapilib` Module](#) on page 136.

Related topics**References**

[dscanapilib \(Real-Time Testing Library Reference !\[\]\(17413706fd4997a1a4bdf85c6864eee1_img.jpg\)\)](#)

Sending CAN Messages with the `rttlib.dscanapilib` Module

Introduction

You can send raw data of a CAN message using the `dscanapilib` module. You can trigger the transmission of a CAN message within a defined sampling step, but the time when the CAN message is transmitted depends on the actual CAN bus communication (i.e. a high bus load can delay the CAN message transmission).

Example

The following example shows the sending of CAN and CAN FD messages which have a standard or extended frame format. This example is an excerpt from the demo for the `dscanapilib` installed with Real-Time Testing.

```
from rttlib import utilities
from rttlib import scheduler
from rttlib import dscanapilib
# Module global variables
# Get variable object
ChannelInfoObjects = []
ChannelHandles = []
def TransmitMessagesGen(ChannelHandle, MessageList):
    dscanapilib.TransmitMessages(ChannelHandle, MessageList)
    yield None
def initCAN(TimeOutSteps = 20):
    """
    The function initializes the CAN channels
    """
    global ChannelInfoObjects
    global ChannelHandles
    # Get the information about available CAN channels.
    ChannelInfoObjects = dscanapilib.GetAvailableChannels()
    yield None
```

```

# Initialize every available CAN channel.
for channel in ChannelInfoObjects:
    ChannelHandle = dscanapilib.RegisterChannel(channel.VendorName, \
                                                channel.InterfaceName, \
                                                channel.InterfaceSerialNumber, \
                                                channel.ChannelIdentifier)

    # Add channel handle to global list.
    ChannelHandles.append(ChannelHandle)
    # Initialize the CAN channel.
    AccessPermission = None
    while(AccessPermission == None):
        AccessPermission = dscanapilib.InitChannel(ChannelHandle, \
                                                    dscanapilib.ctSTDXTD, \
                                                    20, True)

        TimeOutSteps -= 1
        if(TimeOutSteps <= 0):
            raise Exception("Could not initialize CAN channels.")
        yield None
    yield None
    dscanapilib.ActivateChannel(ChannelHandle)
    yield None
def advancedExample():
    """
    The function shows how to transmit a CAN message in the
    standard frame format, and, after a delay of one second, in the
    extended frame format.
    """
    try:
        # Create a raw message with
        # - CAN-ID 0x123
        # - Message Data Length 8
        # - Standard Identifier Format
        message = dscanapilib.CanMessage()
        # Initialize the message data in standard frame format
        message.CanIdentifierType = dscanapilib.ctSTD
        message.CanIdentifier = 0x123
        message.DLC = 8
        message.Data = [0x10, 0x20, 0x30, 0x40, 0x50, 0x60, 0x70, 0x80]
        # Trigger the message transmission and wait until the message
        # is actually sent and exactly one second has passed.
        yield scheduler.Parallel(TransmitMessagesGen(ChannelHandles[0], \
                                                    [message]), \
                                utilities.Wait(1.0))

        # Initialize message data in extended frame format
        message.CanIdentifierType = dscanapilib.ctXTD
        message.CanIdentifier = 0x123456
        message.DLC = 8
        message.Data = [0x10, 0x20, 0x30, 0x40, 0x50, 0x60, 0x70, 0x80]
        # Trigger the message transmission and wait until the message is sent.
        yield dscanapilib.TransmitMessages(ChannelHandles[0], [message])
    finally:
        # Clean up
        message = None
    yield None
def advancedExampleCANFD():
    """
    The function shows how to transmit a CAN FD message in the
    standard frame format, and, after a delay of one second, in the
    extended frame format.
    """

```

```

try:
    # Create a raw message with
    # - CAN-ID 0x123
    # - Message Data Length 64
    # - FD Standard Identifier Format
    message = dscanapilib.CanMessage()
    # Initialize the message data
    message.CanIdentifierType = dscanapilib.ctSTD
    message.Flags = dscanapilib.mfTXFD
    message.CanIdentifier = 0x123
    message.DLC = 15
    message.Data = [ 0x10, 0x11, 0x12, 0x13, 0x14, 0x15, 0x16, 0x17, \
                     0x18, 0x19, 0x1a, 0x1b, 0x1c, 0x1d, 0x1e, 0x1f, \
                     0x20, 0x21, 0x22, 0x23, 0x24, 0x25, 0x26, 0x27, \
                     0x28, 0x29, 0x2a, 0x2b, 0x2c, 0x2d, 0x2e, 0x2f, \
                     0x30, 0x31, 0x32, 0x33, 0x34, 0x35, 0x36, 0x37, \
                     0x38, 0x39, 0x3a, 0x3b, 0x3c, 0x3d, 0x3e, 0x3f, \
                     0x40, 0x41, 0x42, 0x43, 0x44, 0x45, 0x46, 0x47, \
                     0x48, 0x49, 0x4a, 0x4b, 0x4c, 0x4d, 0x4e, 0x4f]

    # Trigger the message transmission and wait until the message
    # is actually sent and exactly one second has passed
    yield scheduler.Parallel(TransmitMessagesGen(ChannelHandles[0], \
                                                  [message]), \
                             utilities.Wait(1.0))

    # Initialize the message data
    message.CanIdentifierType = dscanapilib.ctXTD
    message.CanIdentifier = 0x123456
    message.DLC = 13
    message.Data = [ 0x10, 0x11, 0x12, 0x13, 0x14, 0x15, 0x16, 0x17, \
                     0x18, 0x19, 0x1a, 0x1b, 0x1c, 0x1d, 0x1e, 0x1f, \
                     0x20, 0x21, 0x22, 0x23, 0x24, 0x25, 0x26, 0x27, \
                     0x28, 0x29, 0x2a, 0x2b, 0x2c, 0x2d, 0x2e, 0x2f]

    # Trigger the message transmission and wait until the message is sent
    yield dscanapilib.TransmitMessages(ChannelHandles[0], [message])
finally:
    # Clean up
    message = None
    yield None
def MainGenerator():
    """
        The function is the main real-time testing generator function.
        The name of the function is mandatory because it is the defined
        entry point for the script scheduler.
    """
    global ChannelInfoObjects
    global ChannelHandles
    yield initCAN()
    # CAN message transmission in different frame formats
    yield advancedExample()
    yield None
    # Check whether the CAN controller supports CAN FD.
    if(ChannelInfoObjects[0].ChannelCapabilities & dscanapilib.ccFD) == TRUE:
        # CAN FD message transmission in different frame formats
        yield advancedExampleCANFD()

```

Description

Several functions are defined in the example.

initCAN The `initCAN` function registers and initialize all the available CAN channels. For a description, refer to [Accessing the CAN Bus with the `rttlib.dscanapilib` Module](#) on page 127.

advancedExample The `advancedExample` function shows how to send CAN messages in standard and extended frame format. First a CAN message in standard frame format is specified and sent, after exactly one second, a CAN message in the extended frame format is specified and sent. This is implemented by using the `ParallelRace`. The `TransmitMessages` and `Wait` functions are executed in parallel using the `ParallelRace` method so that both functions must be finished before the following code is executed. Normally, the `Wait` function should require the longest execution time, which is exactly 1 second.

advancedExampleCANFD The `advancedExampleCANFD` function shows how to send CAN FD messages in standard and extended frame format. First a CAN FD message in standard frame format is specified and sent, after exactly one second, a CAN FD message in the extended frame format is specified and sent.

MainGenerator The `MainGenerator` function is the main generator function. It calls the initialization function and the function which transmits the CAN and CAN FD messages. Before the CAN FD messages are sent, it is checked whether the CAN channel supports CAN FD.

Related topics**Basics**

[Basics on the `rttlib.canlib` Module](#)..... 106

Examples

[Demo Examples of Using Real-Time Testing](#)..... 22

References

[CanMessage \(Real-Time Testing Library Reference !\[\]\(626ce8ac21792b9405bfddfea8e0c96a_img.jpg\)\)](#)
[TransmitMessages \(Real-Time Testing Library Reference !\[\]\(2b752d244c1fc411d86684a042d55b85_img.jpg\)\)](#)

Receiving CAN Messages with the `rttlib.dscanapilib` Module

Introduction

You can receive raw data of a CAN message using the `dscanapilib` module.

Example

The example shows function definition which transmit CAN and CAN FD messages and read them to print their data.

```

from rttlib import utilities
from rttlib import scheduler
from rttlib import dscanapilib
# Module global variables
rttPrefix = r" *RTT:* "
ChannelInfoObjects = []
ChannelHandles

def TransmitMessagesGen(ChannelHandle, MessageList):
    dscanapilib.TransmitMessages(ChannelHandle, MessageList)
    yield None

def initCAN(TimeOutSteps = 20):
    """
        The function initializes the CAN channels
    """
    global ChannelInfoObjects
    global ChannelHandles
    # Get the information about available CAN channels.
    ChannelInfoObjects = dscanapilib.GetAvailableChannels()
    yield None
    # Initialize every available CAN channel.
    for channel in ChannelInfoObjects:
        ChannelHandle = dscanapilib.RegisterChannel(channel.VendorName, \
            channel.InterfaceName, \
            channel.InterfaceSerialNumber, \
            channel.ChannelIdentifier)

        # Add channel handle to global list.
        ChannelHandles.append(ChannelHandle)
        # Initialize the CAN channel.
        AccessPermission = None
        while(AccessPermission == None):
            AccessPermission = dscanapilib.InitChannel(ChannelHandle, \
                dscanapilib.ctSTDXTD, \
                20, True)
            TimeOutSteps -= 1
            if(TimeOutSteps <= 0):
                raise Exception("Could not initialize CAN channels.")
            yield None
        yield None
        dscanapilib.ActivateChannel(ChannelHandle)
        yield None

def transmitFunctionGen(messageID):
    """
        The function transmits CAN messages.
    """
    global ChannelInfoObjects
    global ChannelHandles
    try:
        # Create a raw message with
        # - CAN-ID specified by parameter
        # - Message Data Length 4
        # - Standard identifier format
        message = dscanapilib.CanMessage()

```

```

# Initialize the message data
message.CanIdentifierType = dscanapilib.ctSTD
message.CanIdentifier = messageID
message.DLC = 4
message.Data = [0x10, 0x20, 0x30, 0x40]
# Trigger the message transmission and wait exactly for 50ms.
yield scheduler.Parallel(TransmitMessagesGen(ChannelHandles[0], \
    [message]), \
    utilities.Wait(0.05))

finally:
    # Clean up
    message = None
yield None

def transmitCANFDFunctionGen(messageID):
    """
    The function transmits CAN FD messages.
    """
    try:
        # Create a CAN message with
        # - CAN-ID specified by parameter
        # - Message Data Length 48
        # - FD Standard Identifier Format
        message = dscanapilib.CanMessage()
        # Initialize message data
        message.CanIdentifierType = dscanapilib.ctSTD
        message.CanIdentifier = messageID
        message.DLC = 14
        message.Data = [ 0x00, 0x11, 0x12, 0x13, 0x14, 0x15, 0x16, 0x17, \
            0x18, 0x19, 0x1a, 0x1b, 0x1c, 0x1d, 0x1e, 0x1f, \
            0x20, 0x21, 0x22, 0x23, 0x24, 0x25, 0x26, 0x27, \
            0x28, 0x29, 0x2a, 0x2b, 0x2c, 0x2d, 0x2e, 0x2f, \
            0x30, 0x31, 0x32, 0x33, 0x34, 0x35, 0x36, 0x37, \
            0x38, 0x39, 0x3a, 0x3b, 0x3c, 0x3d, 0x3e, 0x3f]
        # Trigger the message transmission and wait exactly for 50ms
        yield scheduler.Parallel(TransmitMessagesGen(ChannelHandles[0], \
            [message]), \
            utilities.Wait(0.05))

    finally:
        # Clean up
        message = None
    yield None

def receiveFunctionGen(messageID):
    """
    The function receives CAN messages.
    """
    try:
        while(1):
            rxList = dscanapilib.ReadReceiveQueue(ChannelHandles[0])
            # Check if List of received messages is not empty.
            if len(rxList) > 0:
                # New message was received
                print(rttPrefix + "t = %f [ID = %d] Receive Data: 0x%x" \
                    % (rxList[0].Timestamp, rxList[0].CanIdentifier, \
                    rxList[0].Data[0]))
            yield None
    finally:
        # Nothing Left to clean up.
        pass
    print(rttPrefix + "Receive of CAN messages finished.")
    yield None

```

```
def MainGenerator():
    """
    Function: MainGenerator
    The function is the main real-time testing generator function.
    The name of the function is mandatory because it is the defined
    entry point for the script scheduler.
    """
    global ChannelInfoObjects
    global ChannelHandles
    yield initCAN()
    messageID = 0x123
    yield scheduler.ParallelRace(transmitFunctionGen(messageID), \
                                receiveFunctionGen(messageID))
    # Check if CAN controller supports CAN FD.
    if(ChannelInfoObjects[0].ChannelCapabilities & dscanapilib.ccFD == True:
        yield scheduler.ParallelRace(transmitCANFDFunctionGen(messageID), \
                                    receiveFunctionGen(messageID))
    print(rttPrefix + "End of real-time test execution.")
```

Description

Several functions are defined in the example.

initCAN The `initCAN` function registers and initialize all the available CAN channels. For a description, refer to [Accessing the CAN Bus with the `rttlib.dscanapilib` Module](#) on page 127.

transmitFunctionGen The `transmitFunctionGen` function shows how to send CAN messages in standard frame format. For a description, refer to [Sending CAN Messages with the `rttlib.dscanapilib` Module](#) on page 129.

transmitCANFDFunctionGen The `transmitCANFDFunctionGen` function shows how to send CAN FD messages in standard frame format. For a description, refer to [Sending CAN Messages with the `rttlib.dscanapilib` Module](#) on page 129.

receiveFunctionGen The `receiveFunctionGen` function shows how to receive CAN messages. It uses the `ReadReceiveQueue` method to try to read the message from the first channel (`channelhandle[0]`). When a CAN message is received, some of its attributes are printed.

MainGenerator The `MainGenerator` function is the main generator function. It calls the initialization function and the functions which transmits and receives the CAN and CAN FD messages. Before the CAN FD messages are sent, it is checked whether the CAN channel supports CAN FD. The transmit and receive functions are executed in parallel using the `ParallelRace` method so that both functions must be finished before the following code is executed.

Related topics**Examples**

[Demo Examples of Using Real-Time Testing..... 22](#)

References

[CanMessage \(Real-Time Testing Library Reference !\[\]\(96cc62f861fdd6e50510c0224a756dff_img.jpg\)\)](#)
[ReadReceiveQueue \(Real-Time Testing Library Reference !\[\]\(e658400d40ca763c7cf4c8c420885c6a_img.jpg\)\)](#)

Unregistering CAN Channels with the `rttlib.dscanapilib` Module

Introduction

You must unregister a CAN channel when it is not required any longer. If a CAN channel remains registered, the dependencies in the driver cannot be cleared so that subsequent calls might fail or you might not get access permission.

Example

The example shows how to get unregister the CAN channels. The example is not ready-to-use, it shows only the relevant code.

```
from rttlib import dscanapilib
global ChannelHandles
# Unregister all the available channels
for ChannelHandle in ChannelHandles:
    dscanapilib.UnregisterChannel(ChannelHandle)
yield None
```

Description

The `UnregisterChannel` method of the `dscanapilib` module is called for each CAN channel that was registered before and is stored in the `ChannelHandles` variable list.

Related topics**References**

[UnregisterChannel \(Real-Time Testing Library Reference !\[\]\(4146d17f71dced09c6ad789cacceaa6d_img.jpg\)\)](#)

Implementing Communication via Ethernet

Introduction You can receive and transmit messages via Ethernet on a SCALEXIO system and VEOS V-ECU.

Where to go from here

Information in this section

[Basics on the ds Ethernet API Module.....](#) 137

You can transmit and receive frames via Ethernet in RTT sequences. Real-Time Testing provides the ds Ethernet API module for this.

[How to Use Ethernet with Real-Time Testing on a SCALEXIO Platform.....](#) 138

To make Ethernet communication and an Ethernet interface on SCALEXIO Processing Unit or DS6001 available for use with Real-Time Testing, you must build a signal chain that contains at least one Ethernet interface in ConfigurationDesk using Ethernet function blocks.

[Using Ethernet with Real-Time Testing on a VEOS Platform.....](#) 139

To make Ethernet communication and an Ethernet interface on VEOS available for use with Real-Time Testing, you must build an offline simulation application.

[Example of Sending and Receiving Frames via Ethernet.....](#) 140

The example demonstrates how you can send and receive frames via Ethernet.

Basics on the ds Ethernet API Module

Introduction You can transmit and receive frames via Ethernet in RTT sequences. Real-Time Testing provides the ds Ethernet API module for this.

Supported simulation platforms

You can use the ds Ethernet API module on the following platforms:

- SCALEXIO Processing Unit
- DS6001 Processor Board
- VEOS
- VEOS V-ECU

dsethernetapilib module

The **dsethernetapilib** module provides classes and methods to use Ethernet interface of the real-time platform:

- To register, initialize, and activate the Ethernet interfaces.
- To read all the information on the Ethernet interfaces.
- To transmit frames in raw format via the Ethernet interface.
- To receive frames in raw format via the Ethernet interface.

Related topics**References**

[rttlib.dsethernetapilib Module \(Real-Time Testing Library Reference !\[\]\(a870788d6ed9b8fd294b7654a8c8526b_img.jpg\)](#))

How to Use Ethernet with Real-Time Testing on a SCALEXIO Platform

Objective

To make Ethernet communication and an Ethernet interface on SCALEXIO Processing Unit or DS6001 available for use with Real-Time Testing, you must build a signal chain that contains at least one Ethernet interface in ConfigurationDesk using Ethernet Setup function blocks.

RTT demos

The RTT demos contain ConfigurationDesk backup projects for a SCALEXIO Processing Unit and DS6001 Processor Board. You can use these projects as basis. The RTT demos are installed in **C:\Program Files\Common Files\dspace\RealTimeTesting\<Version>\Demos**.

Method**To use Ethernet with Real-Time Testing on a SCALEXIO platform**

- 1** Open ConfigurationDesk.
- 2** To open a ConfigurationDesk backup project, go to the File ribbon and click **Open – Project + Application from Backup**.
The Open dialog opens.
- 3** In the dialog, open the **SampleExperiments** folder of the RTT demos and select the **TurnSignal_SCALEXIO_Cfg.ZIP** or **TurnSignal_DS6001_Cfg.ZIP** ConfigurationDesk backup project.
- 4** Open the Signal Chain view.
- 5** Drag an Ethernet Setup function block from the Function Browser to the Signal Chain.
- 6** If the registered hardware differs from the hardware topology of the ConfigurationDesk project, replace the hardware topology in the project by the registered hardware. For details, refer to [Managing Real-Time Hardware \(ConfigurationDesk Real-Time Implementation Guide !\[\]\(098e47036f78288d477e334896a43770_img.jpg\)](#)).

- 7 Assign a hardware Ethernet interface in the properties of the Ethernet Setup block. For details, refer to [Configuring Function Blocks \(ConfigurationDesk Real-Time Implementation Guide !\[\]\(6302aad5aed157b291fddf37b4870784_img.jpg\)](#)).
- 8 In the properties of the Ethernet Setup block, enter a valid local IP address, for example, 192.168.2.42.
- 9 To start the build process, go to the Home ribbon and click Build – Start.
- 10 To download the real-time application, go to the Home ribbon and click Real-Time Application – Load to Platform <Platform>.

Result

You can use the SCALEXIO platform with the downloaded real-time application for Ethernet communication with Real-Time Testing. For further information, refer to the 17_Ethernet RTT demo.

Related topics**Basics**

[Basics on the ds Ethernet API Module.....](#) 137

Using Ethernet with Real-Time Testing on a VEOS Platform

Introduction

To make Ethernet communication and an Ethernet interface on VEOS available for use with Real-Time Testing, you must build an offline simulation application.

Preconditions

You must have a V-ECU implementation container file (VECU file) where an Ethernet implementation is included.

Building OSA

You must import the VECU file in the VEOS Player and build the offline simulation application (OSA). The offline simulation application can be simulated in the VEOS Player. For details, refer to [How to Import V-ECU Implementations \(VEOS Manual !\[\]\(b792654f2cef9719eabeb6c5be00811e_img.jpg\)](#)).

Result

You can use the VEOS platform with the downloaded offline simulation application for Ethernet communication with Real-Time Testing. For further information, refer to the 17_Ethernet RTT demo.

Related topics**Basics**

[Basics on the ds Ethernet API Module.....](#) 137

Example of Sending and Receiving Frames via Ethernet

Introduction

The example demonstrates how you can send and receive frames via Ethernet.

Example

The following example shows how to send and receive raw frames via Ethernet. It shows only a part of the RTT sequence. For the complete example, see `RTTSequences\RTTSequence.py` in `<MyDocuments>\dSPACE\Real-Time Testing\5.0\17_Ethernet`.

```
#-----
# Import all classes from the rttlib module.
# This Python library provides all real-time testing modules.
# All imports must be defined in the global part of the script.
#-----
from rttlib import utilities
from rttlib import dssethernetapilib
#-----
# Module global variables
#-----
# Identifier for real-time testing print messages
rttPrefix = " *RTT:* "
# Global List to hold Ethernet interface information and handles.
InterfaceInfoObjects = []
InterfaceHandles = []
def initEthernet(TimeOutSteps = 60):
    """
    Function: initEthernet
    This function shows how to initialize the Ethernet interfaces with
    the rttlib.dssethernetapilib.
    """
    # Use global List objects.
    global InterfaceInfoObjects
    global InterfaceHandles
    # Retrieve the information about all available Ethernet interfaces.
    InterfaceInfoObjects = dssethernetapilib.GetAvailableInterfaces()
    yield None
    # Initialize all available Ethernet interfaces.
    for interface in InterfaceInfoObjects:
        InterfaceHandle = dssethernetapilib.RegisterInterface( \
            interface.AccessProviderName, \
            interface.InterfaceName, \
            interface.InterfaceSerialNumber, \
            interface.InterfaceIdentifier)
        # Add an interface handle to the global List.
        InterfaceHandles.append(InterfaceHandle)
```

```

# Initialization of Ethernet interface might last several model steps.
# When the function returns 'True' the initialization was successful.
# As long as the return value of InitInterface() is 'None' the interface
# is not initialized. A timeout is useful to avoid waiting too long
# for an interface initialization that might not be possible.
IsInitialized = None
while(IsInitialized == None):
    IsInitialized = dssethernetapilib.InitInterface(InterfaceHandle)
    TimeOutSteps -= 1
    if(TimeOutSteps <= 0):
        raise Exception("Could not initialize Ethernet interfaces.")
    yield None
yield None
# Activate valid Ethernet interface.
dssethernetapilib.ActivateInterface(InterfaceHandle)
yield None
def unregisterEthernet():
    """
    Function: unregisterEthernet
    This function shows how to unregister the Ethernet interfaces with
    the rttlib.dssethernetapilib.
    """
    # Use global ethernetLib object.
    global InterfaceHandles
    # Unregister all previously registered Ethernet interfaces.
    for InterfaceHandle in InterfaceHandles:
        dssethernetapilib.UnregisterInterface(InterfaceHandle)
    yield None
def basicExampleTransmitFrame():
    """
    Function: basicExampleTransmitFrame
    The basic example shows how to transmit an Ethernet frame.
    """
    # Use global object.
    global InterfaceHandles
    print(rttPrefix + "Start cyclic transmission of an Ethernet frame.")
    try:
        # Create an Ethernet frame object.
        # - type Ethernet
        # - raw data Length 1500
        Frame = dssethernetapilib.EthRawFrame()
        Frame.Header.FrameType = dssethernetapilib.ftETHERNET
        Frame.Header.RawDataLength = 1500
        RawDataList1500Bytes = [x%256 for x in range(0, 1500)]
        Frame.RawData = RawDataList1500Bytes
        # Send it every 50 ms, 10 times.
        for i in xrange(10):
            # Create transmit buffer.
            TransmitBuffer = dssethernetapilib.CreateBuffer(Frame.Length)
            # Trigger the frame transmission.
            yield dssethernetapilib.TransmitFrames(InterfaceHandles[0], \
                TransmitBuffer, [Frame])
            # Wait for exactly 50 ms.
            yield utilities.Wait(0.05)
    finally:
        # Clean up
        Frame = None
        TransmitBuffer = None
    print(rttPrefix + "Finished cyclic transmission of an Ethernet frame.")
    yield None

```

```

def basicExampleReceiveFrame():
    """
    Function: basicExampleReceiveFrame
    The basic example shows how to receive an Ethernet frame.
    """
    # Use global object.
    global InterfaceHandles
    print(rttPrefix + "Start receiving Ethernet messages.")
    # Receive buffer must be large enough to hold all expected frames.
    ReceiveBufferSize = 20000
    rxList = []
    try:
        # Create receive buffer.
        ReceiveBuffer = dssethernetapilib.CreateBuffer(ReceiveBufferSize)
        # Read all frames in the receive queue.
        rxList = dssethernetapilib.ReadFrames(InterfaceHandles[0], ReceiveBuffer)
        # Check if list of received frames is not empty.
        if len(rxList) > 0:
            print (rttPrefix + "%d frames received! "% len(rxList))
        else:
            print (rttPrefix + "No frames received!")
        yield None
    finally:
        # Nothing left to clean up.
        pass
    print(rttPrefix + "Finished receiving Ethernet frames.")
    yield None

def MainGenerator(*args):
    """
    Function: MainGenerator
    This function is the main real-time testing generator function.
    The name of the function is mandatory because it is the defined
    entry point for the script scheduler.
    """
    # Use global object.
    global InterfaceInfoObjects
    print(rttPrefix + "Start of real-time test execution on target platform.")
    # Initialize Ethernet interfaces.
    yield initEthernet()
    # Cyclic Ethernet frame transmission.
    yield basicExampleTransmitFrame()
    # Receiving Ethernet frames.
    yield basicExampleReceiveFrame()
    # Unregister Ethernet interfaces.
    yield unregisterEthernet()
    print(rttPrefix + "End of real-time test execution.")

```

Description

The `dssethernetapilib` module must be imported to get all the methods for using the Ethernet interface.

In the example several functions are defined: `initEthernet`, `basicExampleTransmitFrame`, `basicExampleReceiveFrame`, and `MainGenerator`.

The `initEthernet()` function initializes all the Ethernet interfaces of the real-time platform. At the beginning, the function retrieves the information about all Ethernet interfaces that are available using the `GetAvailableInterfaces()` method. After that, the interfaces are registered (`RegisterInterface()`)

method) and initialized (**InitInterface()** method). Because the initialization can require several model steps, a while loop is used to evaluate the return value of the **InitInterface()** method. The interfaces are activated after their initialization.

The **unregisterEthernet()** function frees all the dependencies of the selected Ethernet interfaces. The interface handles become invalid.

The **basicExampleTransmitFrame()** function transmits a frame via an Ethernet interface. The frame is transmitted 10 times every 50 ms. A transmit buffer is created (**CreateBuffer()** method) and the frame is transmitted (**TransmitFrames()** method).

The **basicExampleReceiveFrame** function receives frames via an Ethernet interface. To receive frames, a buffer must be created (**CreateBuffer()** method). Then the frames can be read from the receive queue (**ReadFrames()** method).

In **MainGenerator()**, the functions for initializing the Ethernet interfaces, transmitting frames, receiving frames, and unregistering the interfaces are called.

Related topics

Basics

[Basics on the ds Ethernet API Module..... 137](#)

References

[CreateBuffer Method \(Real-Time Testing Library Reference !\[\]\(d0262bbe9d2356661a2e89321dfcc781_img.jpg\)\)](#)
[GetAvailableInterfaces Method \(Real-Time Testing Library Reference !\[\]\(8572950e410320d7dd023da827ff014d_img.jpg\)\)](#)
[InitInterface Method \(Real-Time Testing Library Reference !\[\]\(b2b6a2e56e47cc582ad4ec3c8f1864c0_img.jpg\)\)](#)
[ReadFrames Method \(Real-Time Testing Library Reference !\[\]\(b51ca72c89286e93c23769c3302173c1_img.jpg\)\)](#)
[RegisterInterface Method \(Real-Time Testing Library Reference !\[\]\(5be36cd8971383ef0e304a7698e11a72_img.jpg\)\)](#)
[TransmitFrames Method \(Real-Time Testing Library Reference !\[\]\(cdd54a81b8b830d3c1fbf30edda522d6_img.jpg\)\)](#)
[UnregisterInterface Method \(Real-Time Testing Library Reference !\[\]\(83bd4e44f0a47db8128e320e2223ff83_img.jpg\)\)](#)

Implementing a Communication via a Serial Interface

Introduction

To send and receive data via a serial interface of the real-time platform in an RTT sequence, Real-Time Testing provides the rs232lib module.

Where to go from here

Information in this section

Basics of Working with the rs232lib Module..... 144

The rs232lib module provides functions for configuring serial interfaces and sending/receiving data within an RTT sequence.

Example of Sending Data via a Serial Interface..... 145

The example shows how you can send data via a serial interface in an RTT sequence.

Example of Receiving Data via a Serial Interface..... 147

The example shows how you can receive data via a serial interface in an RTT sequence.

Basics of Working with the rs232lib Module

Introduction

The rs232lib module provides functions for configuring serial interfaces and sending/receiving data within an RTT sequence.

Supported dSPACE boards

You can use the serial interfaces of the DS1006 Processor Board. Other boards are not supported.

Initializing/Configuring the serial interface

Before you can send or receive data via a serial interface, the interface must be initialized and configured.

You specify the controller/channel which is used for communication and the buffer size in the initialization function. The **OpenEx** function returns a handle for the used serial channel. This handle must be used for all other functions used for serial communication.

During configuration you specify the baud rate, word length, settings for parity calculation, and number of stop bits in the **SetConfig()** function.

Sending data

To send data, it is written to the transmit buffer of the serial interface. The data is then transmitted via the RS232 transceiver of the real-time board. The rs232lib

module provides functions for sending one byte (`Write()`) or a whole string (`WriteString()`).

Receiving data

Data that is received via the serial interface is copied to the receive buffer. You can read the number of bytes in the receive buffer using the `GetNumInBytes()` function. You read the data from the receive buffer using the `Read()` function.

Managing the serial interface

You can clear the receive and transmit buffer using the `PurgeComm()` function.

When the communication is finished, you can close the channel using the `Close()` function. After calling the function, the handle for the used serial channel is invalidated and can be deleted.

If you do not close the channel, the channel is still locked for other RTT sequences. If the RTT sequence is removed, the channel is closed automatically and can be used by other RTT sequences.

Limitations for the rs232lib module

The following limitations apply when the rs232lib module is used:

- If the real-time application uses a serial interface, you cannot use it with the rs232lib module in your RTT sequence.
- The buffer size is fixed to 64 bytes and cannot be changed during run time by the rs232lib.

Related topics

References

[rttlib.rs232lib Module \(Real-Time Testing Library Reference !\[\]\(95b425611cbd2b8716a140cf67c81822_img.jpg\)](#))

Example of Sending Data via a Serial Interface

Introduction

The example shows how you can send and receive data via a serial interface in an RTT sequence.

Example

The following example shows how to send a single byte and a string via a serial interface. It shows only a part of the RTT sequence. For the complete example, see `RTTSequences\RTTSequence_Transmit.py` in `<MyDocuments>\dSPACE\Real-Time Testing\5.0\13_RS232`.

```
from rttlib import rs232lib
from rttlib import utilities
```

```

#-----
# Module global variables
#-----
# Get variable object
CurrentTime = utilities.currentTime
# Set RS232 values
BoardType = rs232lib.ONBOARD      # Do not change: Reserved for future use
BoardIndex = 1                    # Do not change: Reserved for future use
PortIndex = 1                     # Do not change: Reserved for future use
Baudrate = 9600                   # RS232 baud rate
DataBits = 8                      # Data bits per RS232 frame
Parity = rs232lib.NO_PARITY       # Parity mode: NO, ODD, EVEN, MARK or SPACE
StopBits = 1.0                    # Extra bits at the end of a frame
#-----
# Controller initialization
#-----
Channel = rs232lib.OpenEx(BoardType, BoardIndex, PortIndex)
#-----
# Function: WaitGen
# The WaitGen function is a generator function which continues after the
# specified duration (in seconds).
#-----
def WaitGen(Duration):
    # Store the current time at the beginning of the function
    StartTime = CurrentTime.Value
    # Wait as long as the condition is true.
    while((StartTime + Duration) > CurrentTime.Value):
        yield None
#-----
# Function: TransmissionDemo
# This example demonstrates how to transmit a RS232 message.
#-----
def TransmissionDemo():
    # Use global Channel object
    global Channel
    # Configure controller
    yield rs232lib.SetConfig(Channel, Baudrate, DataBits, Parity, StopBits)
    ByteMessage = 0x64
    StringMessage = "SPACE"
    # send it every 500ms
    for i in xrange(20):
        # Transmit single byte
        rs232lib.Write(Channel, ByteMessage)
        # Transmit string
        rs232lib.WriteString(Channel, StringMessage)
        # Wait for exactly 500 ms
        yield WaitGen(0.5)
    yield None
#-----
# Function: MainGenerator
# This function is the main real-time testing generator function.
# The name of the function is mandatory because it is the defined
# entry point for the script scheduler.
#-----
def MainGenerator():
    # Cyclic RS232 message transmission
    yield TransmissionDemo()

```

Description

The `rs232lib` module must be imported to get all functions required for using the serial interface.

The global variables are defined (`BoardType ... StopBits`). These variables are used to open and configure the serial port for communication using the `OpenEx()` and `SetConfig()` functions. The `OpenEx()` function returns a handle object which must be used by all the functions using the serial port.

In `TransmissionDemo()`, the `Write()` function is used to send several bytes via the serial interface. The `WriteString()` function is used to send a string.

In `MainGenerator()`, the `TransmissionDemo()` function is called. Note that it is a generator function and must be called with `yield`.

Related topics**Examples**

[Demo Examples of Using Real-Time Testing..... 22](#)

References

[OpenEx Function \(Real-Time Testing Library Reference !\[\]\(870f5d5e9c0d57485634be3ecf52f3ca_img.jpg\)](#))
[SetConfig Function \(Real-Time Testing Library Reference !\[\]\(66b14d8ba452f6f18b47935355b6120a_img.jpg\)](#))
[Write Function \(Real-Time Testing Library Reference !\[\]\(bcb9bfd69e5b89da3d817cb72bfcfd1e_img.jpg\)](#))
[WriteString Function \(Real-Time Testing Library Reference !\[\]\(0eb6abbd70294475dc7cb6513507500d_img.jpg\)](#))

Example of Receiving Data via a Serial Interface

Introduction

The example shows how you can send and receive data via a serial interface in an RTT sequence.

Example

The following example shows how to receive data via a serial interface. It shows only a part of the RTT sequence. For the complete example, see `RTTSequences\RTTSequence_Receive.py` in `<MyDocuments>\dSPACE\Real-Time Testing\5.0\13_RS232`.

```
from rttlib import rs232lib
from rttlib import utilities
#-----
# Module global variables
#-----
# Get variable object
CurrentTime = utilities.currentTime
```

```

# Set RS232 values
BoardType = rs232lib.ONBOARD      # Do not change: Reserved for future use
BoardIndex = 1                    # Do not change: Reserved for future use
PortIndex = 1                     # Do not change: Reserved for future use
Baudrate = 9600                   # RS232 baud rate
DataBits = 8                      # Data bits per RS232 frame
Parity = rs232lib.NO_PARITY       # Parity mode: NO, ODD, EVEN, MARK or SPACE
StopBits = 1.0                    # Extra bits at the end of a frame
#-----
# Controller initialization
#-----
Channel = rs232lib.OpenEx(BoardType, BoardIndex, PortIndex)
#-----
# Function: ReceptionDemo
# This example demonstrates how to receive RS232 messages.
#-----
def ReceptionDemo(Duration = 20):
    # Configure controller
    yield rs232lib.SetConfig(Channel, Baudrate, DataBits, Parity, StopBits)
    ReceivedBytes = 0
    ReceivedData = None
    StartTime = CurrentTime.Value
    # Listen to the bus for 'Duration' seconds
    while((StartTime + Duration) > CurrentTime.Value):
        # Check receive buffer
        ReceivedBytes = rs232lib.GetNumInBytes(Channel)
        if ReceivedBytes > 0:
            # Read data from receive buffer
            ReceivedData = rs232lib.Read(Channel, ReceivedBytes)
        yield None
    yield None
#-----
# Function: MainGenerator
# This function is the main real-time testing generator function.
# The name of the function is mandatory because it is the defined
# entry point for the script scheduler.
#-----
def MainGenerator():
    # Start reception demo
    yield ReceptionDemo(Duration = 20)

```

Description

The `rs232lib` module must be imported to get all functions required for using the serial interface.

The global variables are defined (`BoardType ... StopBits`). These variables are used to open and configure the serial port for communication using the `OpenEx()` and `SetConfig()` functions. The `OpenEx()` function returns a handle object which must be used by all the functions using the serial port.

In the `ReceptionDemo()` function, the `GetNumInBytes()` function is used to check if data is received. When data is received, it is read by the `Read()` function.

In `MainGenerator()`, the `ReceptionDemo()` function is called. Note that it is a generator function and must be called with `yield`.

Related topics**Examples**

[Demo Examples of Using Real-Time Testing..... 22](#)

References

[GetNumInBytes Function \(Real-Time Testing Library Reference !\[\]\(74d4806277d7e73349d8e8c0897931e9_img.jpg\)\)](#)
[OpenEx Function \(Real-Time Testing Library Reference !\[\]\(5f42d2cd7ad901bc24e5d35a38c777fd_img.jpg\)\)](#)
[Read Function \(Real-Time Testing Library Reference !\[\]\(628bc0b1ef2b63d1fc4442fb794e3e78_img.jpg\)\)](#)
[SetConfig Function \(Real-Time Testing Library Reference !\[\]\(210e01d0c2c300cf4405442bfd570b4e_img.jpg\)\)](#)

Accessing Variables of a Simulation Application on a Remote Node

Introduction

An RTT sequence can access variables of a simulation application running on another node (processor board in a multiprocessor system, core on a multicore board, or a remote VPU in an offline simulation application).

Where to go from here

Information in this section

[Basics on Accessing Variables of a Simulation Application on a Remote Node..... 150](#)

You can access variables of a simulation application running on another node (processor board in a multiprocessor system, CPU on a multicore board, or a remote VPU in an offline simulation application).

[Notes on Accessing Variables of a Simulation Application on a Remote Node..... 153](#)

There are some points to note when you access variables of a remote node.

[Example of Accessing Variables of a Remote Node..... 156](#)

You can access the values of variables of a node other than the node where the RTT sequence is running.

Basics on Accessing Variables of a Simulation Application on a Remote Node

Introduction

You can access variables of a simulation application running on another node (processor board in a multiprocessor system, CPU on a multicore board, or a remote VPU in an offline simulation application).

Multiprocessor, multicore systems, and VPUs

Real-Time Testing supports multiprocessor, multicore systems, and multiple virtual processing units (VPUs). Real-Time Testing is performed in the same way in the system types, but the behaviors are different due to the different hardware topologies.

Multiprocessor system A multiprocessor system consists of several processor boards which are connected via Gigalinks or processing units (SCALEXIO) which are connected via IOCNET. Each processor board has 4 Gigalink ports that can be connected to other processor boards. The topology of the connection influences

the time that is required for accessing a remote variable because one sampling step is required for each node connection.

Multi-processing-unit system A multi-processing-unit system consists of several processing units (SCALEXIO) which are connected via IOCNET. A processing unit has IOCNET ports that can be used to connect to other processing units. The topology of the connection influences the time that is required for accessing a remote variable because one sampling step is required for each node connection.

Multicore system A multicore system is a processor board that has several, internally connected cores. Real-Time Testing can therefore always use the shortest connection between the cores to access a remote variable. In a SCALEXIO system, the connection used by Real-Time Testing is independent of the connection specified with ConfigurationDesk.

Offline simulation application An offline simulation application can contain multiple virtual processing units (VPUs), for example, virtual ECUs (V-ECUs). To access a variable of a remote VPU, one sampling step is required.

In the following description, processor boards, processor units, cores, and VPUs are called nodes. The connection between nodes is called a node connection.

Real-Time Testing version

Accessing a multi-processing-unit system with SCALEXIO Processing Units is supported as of Real-Time Testing 2.3.

Accessing a multi-processing-unit system based on DS1007 is supported as of Real-Time Testing 2.5.

Accessing variables of a remote node of a processing-unit system based on DS1007 is supported as of Real-Time Testing 2.6.

Accessing variables of remote VPUs is supported as of Real-Time Testing 4.0.

Accessing variables

An RTT sequence running on a node (local node) can transparently access variables of a real-time application running on another node (remote node) of a multiprocessor or multicore system. You have read/write access to these variables from the RTT sequence.

Local node The local node is the node where the RTT sequence is running.

Remote node The remote node is the node whose variable is accessed (a value is written to the variable or the value is read).

Note

You can only access variables of remote nodes. Other features of Real-Time Testing, for example, CAN message handling at a remote node, are not supported.

Routing variable values

The variable values are routed in the multiprocessor or multicore system via node connections. Real-Time Testing can allocate resources for up to 500 variables of one node connection. If more variables are created, an exception occurs. When an RTT sequence is deleted, Real-Time Testing automatically frees the allocated resource of the node connection.

Enabling real-time testing

Real-time testing must be enabled for each node in the multiprocessor or multicore system which is involved:

- Node on which the RTT sequence runs (local node)
- Node whose variables are accessed (remote node)
- All nodes in the network which are connected in the network between the two nodes

For information on how to enable real-time testing, refer to [Enabling Real-Time Testing for dSPACE Platforms](#) on page 33.

Accessing platforms

If you have Real-Time Testing 2.2 or earlier, you must access each individual node to perform the real-time tests:

- If you use the Real-Time Test Manager, refer to [How to Start the Real-Time Test Manager and Access a Platform](#) on page 176.
- If you manage RTT sequences using Python scripts, refer to [Creating and Starting RTT Sequences in Python Scripts](#) on page 182.

It is not necessary to access each node individually if you use the host software from dSPACE Release 2014-A and later and the real-time application for a SCALEXIO system was built with this release.

Accessing variables of a remote CPU

The syntax for accessing the variables of a remote CPU in a multiprocessor system is the same as for single-processor systems.

```
from rttlib import variable
Var = variable.Variable(VariableName)
```

The **VariableName** parameter must also contain the application name (name of the submodel) running on the node, for example, `r'masterAppl/Model Root/master/Clock/Out1'`, where **masterAppl** is the name of the submodel and **Model Root/master/Clock/Out1** is the name of the path and variable.

Tip

If you want to access a variable of the local node, you can optionally specify the name of the submodel. This makes it easy to use an RTT sequence on different nodes. For an example, refer to [Example of Accessing Variables of a Remote Node](#) on page 156.

Related topics**Basics**

[Creating Multi-Processing-Unit Applications With ConfigurationDesk \(ConfigurationDesk Real-Time Implementation Guide !\[\]\(666e09182d4cd268646ea700ea60dcdf_img.jpg\)\)](#)
[Distributing the Model for MP Systems \(RTI and RTI-MP Implementation Guide !\[\]\(1ef1ef0bf9af6c6996401964cf280f2d_img.jpg\)\)](#)
[Read/Write Access to Variables of the Simulation Application..... 62](#)

References

[Variable Class \(Real-Time Testing Library Reference !\[\]\(d66ff64371a51729ac8c1cdaa685ba6f_img.jpg\)\)](#)

Notes on Accessing Variables of a Simulation Application on a Remote Node

Introduction

You have to note some points when you access variables of a remote node.

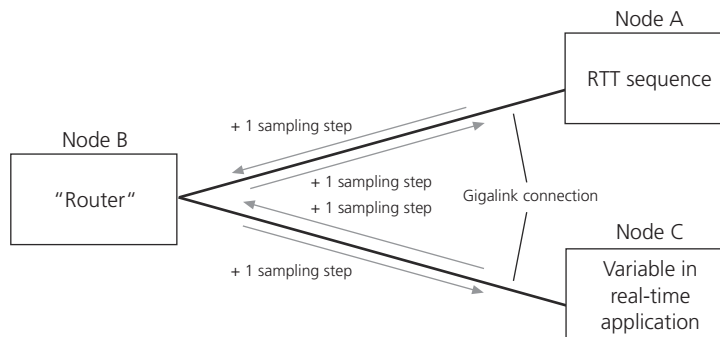
Latency when accessing variables

Accessing variables of a remote node has certain limitations. The interprocessor communication requires some time as the values must be routed through the system. One sampling step is required for each node connection through which the values must run (the routing path). Thus, the time delay is the number of passed node connections (length of the routing path) multiplied by the sampling step size.

Unpredictable latency The latency cannot be determined in some cases:

- If interprocessor communication is possible via different node connections, a minimized routing path is calculated automatically. Although the routing path is minimized, the actual routing path used between two nodes can be longer than expected. You can avoid this by not using a star topology by using a topology without loops.
- If your multiprocessor or multicore system has multiple step sizes, the delay time cannot be determined. For details on step size multiple, refer to [IPCx \(RTI and RTI-MP Implementation Reference !\[\]\(896151ec231b70900e969d67696ca48d_img.jpg\)\)](#).
- If the turnaround time on a node that the values are routed through is very low, transferring a variable value might not require a sampling step. Routing might be faster so that the latency cannot be determined (see also [Lost Variable Values](#) on page 222).

Example of the time delay in a multiprocessor system An RTT sequence runs on local node A. It writes to a variable in the submodel running on remote node C. The CPUs are connected via two Gigalink connections, see the following illustration.



The following table shows which value of the variable is valid at node A and node C respectively when reading it back on each node in the sampling steps after writing the new values.

Node A (local node)	17	17	17	17	42	42
Node C (remote node)	17	17	42	42	42	42
Sampling Step	1	2	3	4	5	6

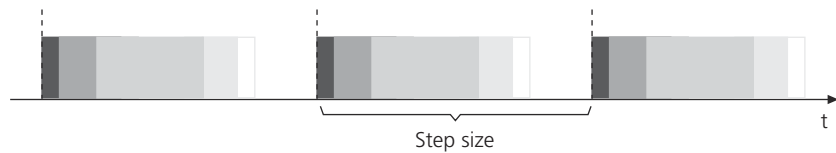
In the 1st sampling step, a value of 42 is written to a variable in the RTT sequence running on CPU A. Although the new value was written, the variable still contains the old value because this value is read from node C. It requires two sampling steps to route the changed value to node C, because the two nodes are separated by two Gigalink connections. Then the value is changed on node C. It takes two additional sampling steps for the changed value to become visible in the RTT sequence. In the meantime, the variable still has the old value in the RTT sequence.

Consistency of variable values

Normally all variables read from the same remote node have the same time delay, and the values are read in the same sampling step. However, if the turnaround time on a node that the values are routed through is very small, the values can become inconsistent. Some values can be transferred in the same sampling step, other values may be transferred in the following sampling step (see also [Lost Variable Values](#) on page 222).

Performance of variable value routing

The variable values are read from the communication buffer before the RTT sequences are executed in the PreComputation channel. They are written to the communication buffer after the RTT sequences are executed in the PostComputation channel. See the following illustration.



- Reading data from Communication buffer
- Execution of RTT sequences in PreComputation channel
- Execution of real-time application
- Execution of RTT sequences in PostComputation channel
- Writing data to Communication buffer

The execution and writing the variables depends on the number of variables. The execution times are different on the local node (where the RTT sequence is executed) and the remote node (where the variable is located). They depend mainly on the time that is required for the Gigalink access.

The following table shows the execution times for local and remote variables on a DS1006 running at 2.6 GHz. These are the combined times for the read phase and the write phase in the above illustration.

CPU	Number of Variables		
	10	100	500
Local node	11.9 μ s	101.5 μ s	502.0 μ s
Remote node	13.6 μ s	118.5 μ s	585.2 μ s

The following table shows the execution times for the internal Gigalink connection of a DS1006MC system with 2.8 GHz.

Node	500 Variables
Local node	26 μ s
Router node	97 μ s
Remote node	50 μ s

The following table shows the execution times for a SCALEXIO multicore system with 2.8 GHz.

Node	500 Variables
Local node	14 μ s
Router node	0 μ s
Remote node	11 μ s

Note that the SCALEXIO multicore system does not need a router CPU as it always uses a direct connection between the local and the remote node.

Writing to variables

Variables can be accessed from several RTT sequences running on different nodes. However, only one RTT sequence should write to a variable at a time.

Limitations in multiprocessor and multicore systems

- The length of the submodel name is restricted to 58 characters.
- The length of the variable name with its full path as the specified variable description file is restricted to 2048 characters.
- The number of variables per node connection is restricted to 500.
- You cannot read, modify and write a variable of a remote node in one sampling step. The number of required sampling steps depends on the number of node connections between the two nodes (the local node where the RTT sequence is running and the remote node whose variable is accessed).
- In the RTT sequence, values of variable objects located on remote nodes are updated in each sampling step. This increases the execution time of the RTT sequence.
- In large SCALEXIO systems with 55 nodes (in certain cases also below 55 nodes), you cannot access variables of remote nodes. For more information, refer to [Using Real-Time Testing in Large SCALEXIO Systems](#) on page 224.
- You can access only the CAN or CAN FD controllers which are on the CPU where the RTT sequence is running. Accessing a CAN or CAN FD controller on a remote CPU in a multiprocessor system is not supported.
- If a multiprocessor system is registered using ControlDesk, you must access each processor board individually (for example, calling `AccessBoard("192.168.0.15/MyApp")` and afterwards `AccessBoard("192.168.0.16/MyApp2")` and so on).
- Global variables cannot be used by RTT sequences running on different CPUs in a multiprocessor system.
- RTT remote variables must not be read in initialization phase of an RTT sequence because they might have incorrect values in this phase. If the topology distance of a remote variable is N , at least N simulation steps are required for a value to propagate through the network of computation nodes. This is especially important for the first access of the variable, since the variable's initialization value might be undefined.

Related topics

Basics

[Read/Write Access to Variables of the Simulation Application..... 62](#)

Example of Accessing Variables of a Remote Node

Introduction

You can access the values of variables of a node other than the node where the RTT sequence is running.

Demo files

The host PC Python script and RTT sequence shown below are parts of a demo in `<MyDocuments>\dSPACE\Real-Time Testing\5.0\14_Multiprocessor` archive. You can use the `StartDemo.bat` batch file to start the demo.

The demo experiment is in `SampleExperiments\TurnSignal_<platform>` (`<platform>` is platform type).

Host PC Python

The `RTTLoader.py` script loads the RTT sequence to the processor board. The following listing shows a part of the script.

```
def executeDemo(boardNames):
    """
    Function : executeDemo
    Execute part of the demo.
    """
    # Create new RTTManagerServer instance
    rttManager = rttmanagerlib.RealTimeTestManagerServer()
    # Generate byte code from the test script (RTT sequence)
    bcgFileName = rttManager.BCGServiceProvider.Generate(testScriptName,[])
    print("BCG file created: %s" % bcgFileName)
    board = rttManager.AccessBoard(boardNames[0])
    board_2 = rttManager.AccessBoard(boardNames[1])
    print("Boards '%s' and '%s' successfully connected." % (board.Name,
        board_2.Name))
    # Connect to sequence's event handle
    sequencesEvents = rttdemoutilities.RTTMSequencesEvents(board.Sequences)
    sequencesEvents_2 = rttdemoutilities.RTTMSequencesEvents(board_2.Sequences)
    # Set channel for RTT sequence
    Channel = rttmanagerlib.constants.scPostComputation
    try:
        # Load sequence to the real-time platform 1
        sequence = board.Sequences.Create(bcgFileName, SequenceChannel = Channel)
        print("Sequence '%s' on real-time platform '%s' created. " \
            % (bcgFileName, board.Name))
        # Load sequence to the real-time platform 2
        sequence_2 = board_2.Sequences.Create(bcgFileName, SequenceChannel = Channel)
        print("Sequence '%s' on real-time platform '%s' created. " \
            % (bcgFileName, board_2.Name))
        # Start the sequence on real-time platform 'ds100X'
        sequence.Run()
        print("Sequence on board %s started.\n" % board.Name)
        # Suspend host script for the execution time of the real-time testing
        # script
        numberOfSeconds = 11
        print("\nSuspend host script execution for %d seconds." \
            % numberOfSeconds)
        rttutilities.RTTSleep(numberOfSeconds)
        # Start the sequences on real-time platform 2
        sequence_2.Run()
        print("Sequence on board %s started.\n" % board_2.Name)
```

```

# Suspend host script for the execution time of the real-time testing
# script
numberOfSeconds = 11
print("\nSuspend host script execution for %d seconds." \
      % numberOfSeconds)
rttutilities.RTTSleep(numberOfSeconds)
finally:
    # Clear traceback object to free COM object
    sys.last_traceback = None
    # Clean up
if sequencesEvents:
    sequencesEvents.close()
    sequencesEvents = None
if sequencesEvents_2:
    sequencesEvents_2.close()
    sequencesEvents_2 = None
sequence = None
sequence_2 = None
board = None
rttManager = None

#-----
# Main Program
#-----
if __name__ == "__main__":
    if(3 == len(sys.argv)):
        # The script execution is started in python.exe.
        if "python.exe" in sys.executable.lower():
            # Python.exe does not call the COM initialization method.
            # Initialize the COM Libraries for the calling thread.
            pythoncom.CoInitialize()
        try:
            boardNames = [sys.argv[1], sys.argv[2]]
            print("\nThe real-time testing demonstration is starting...")
            executeDemo(boardNames)
            print("\nThe real-time testing demonstration was successfully\
                  " completed.")
        finally:
            # The script execution is started in python.exe.
            if "python.exe" in sys.executable.lower():
                # Python.exe does not call the COM uninitialization method.
                # Uninitialize the COM Libraries for the calling thread.
                pythoncom.CoUninitialize()
    else:
        if(1 == len(sys.argv)):
            print("No platform defined.\n")
        elif(2 == len(sys.argv)):
            print("No second platform defined.\n")
        else: print("Too much platforms defined.\n")

```

Description of host PC Python script

Starting the host PC Python script The host PC Python script must be started with the board name or IP address as an argument.

If you have a DS1006 multiprocessor system, start the host PC script as follows

```
RTTLoader.py <BoardName_1> <BoardName_2>
```

If you have a multicore or multiprocessor system based on a DS1007, DS1202, DS6001, SCALEXIO, or VEOS platform, start the host PC script as follows:

```
RTTLoader.py <ip1>/<application> <ip2>/<application2>
```

<ip1> and <ip2> are the IP addresses of the platforms (<ip1> and <ip2> are identical for a multicore platform), <application1> and <application2> are the names of the real-time applications

Accessing the boards The MP system name is **multiprocessor**. It consists of two processor boards. The following table shows the assignment between board and application.

Board Name	CPU	Application
DS1006	master	masterAppl
or DS1006_2	slave	slaveAppl

The Python script accesses the two boards of the multiprocessor system. The **AccessBoard** function requires the board name, for example, 'DS1006' or 'DS1006_2'.

The following table shows the assignment between board and application for a SCALEXIO platform.

Board Name	CPU	Application
Platform_2	master	masterAppl
Platform_3	slave	slaveAppl

The Python script accesses the two real-time applications of the SCALEXIO system. The **AccessBoard** function requires the IP address and the application name, for example, '192.168.0.15/masterAppl' or '192.168.0.15/slaveAppl'.

Using the RTT sequence The script generates a BCG file from the RTT sequence. The BCG file is created on both processor boards. The BCG file is started on the master CPU first. That means the master CPU is the local node and the slave CPU is the remote node. After 11 seconds the BCG file is started on the slave CPU. Then the slave CPU is the local node and the master CPU is the remote node.

RTT Sequence

The **RTTSequence.py** script shows how to create variables in the RTT sequence for accessing variables on a remote node. In the example, the RTT sequence is used twice. It is created on both nodes of the multiprocessor system (see host PC Python script above) because when real-time tests are executed, you must access each node which is involved.

```
#-----
# Import all classes from the rttlib module.
# This Python Library provides all real-time testing modules.
# ALL imports must be defined in the global part of the script.
#-----
from rttlib import variable
from rttlib import utilities
#-----
# Module imports
#-----
import sys
```

```

#-----
# Module global variables
#-----
# Create variable object on the node where the sequence is loaded.
currentTime = utilities.currentTime
if variable.IsA2L():
    # The variable description for this application is an A2L file.
    currentTimeMaster = variable.Variable(r'masterAppl()//masterAppl/SimulationTime')
    currentTimeSlave = variable.Variable(r'slaveAppl()//masterAppl/SimulationTime')
else:
    # Create variable object on node 'masterappl'.
    currentTimeMaster = variable.Variable(r'masterAppl/Model Root/master/Clock/Out1')
    # Create variable object on node 'slaveappl'.
    currentTimeSlave = variable.Variable(r'slaveAppl/Model Root/slave/Clock/Out1')
# Identifier for real-time testing print messages
rttPrefix = r" *RTT:* "
def MainGenerator(*args):
    """
    Function: MainGenerator
        This function is the main real-time testing generator function.
        The name of the function is mandatory because it is the defined
        entry point for the script scheduler.
    """
    print(rttPrefix + "Start of real-time test execution on target platform")
    # Read value of Simulink variable
    startTime = currentTime.Value
    # Output values of Simulink variables
    print(rttPrefix + "CurrentTime.Value = %.4f" % currentTime.Value)
    print(rttPrefix + "currentTimeMaster.Value = %.4f" % currentTimeMaster.Value)
    print(rttPrefix + "currentTimeSlave.Value = %.4f" % currentTimeSlave.Value)
    # Wait for 10 seconds.
    print(rttPrefix + "Wait for 10 seconds. ")
    # The condition of the while loop is checked in every sample step.
    while ((startTime + 10.0) > currentTime.Value):
        yield None
    # Output values of Simulink variables
    print(rttPrefix + "CurrentTime.Value = %.4f" % currentTime.Value)
    print(rttPrefix + "currentTimeMaster.Value = %.4f" % currentTimeMaster.Value)
    print(rttPrefix + "currentTimeSlave.Value = %.4f" % currentTimeSlave.Value)
    print(rttPrefix + "End of real-time test execution.")

```

Description of the RTT sequence

Creating variable objects The RTT sequence creates several variable objects on the local or remote node:

```
currentTime = utilities.currentTime
```

The **currentTime** variable is always located at the local node where the BCG file is created because no application name is specified in the parameter.

```
currentTimeMaster = variable.Variable(r'masterAppl/Model Root/master/Clock/Out1')
```

The **currentTimeMaster** variable is always located at the master CPU where the turnlamp application is running. It does not matter whether the master CPU is the local or remote node.

```
currentTimeSlave = variable.Variable(r'slaveAppl/Model Root/slave/Clock/Out1')
```


The `currentTimeSlave` variable is always located at the slave node, where the `slaveAppl` application is running. It does not matter whether the slave node is the local or remote node.

Output variable values The `MainGenerator()` function outputs the values of the variables to the Log Viewer.

Tip

You can read the time which the RTT sequence needs for running using attributes of the `Sequence` object. Refer to [Getting the Run Time of an RTT Sequence](#) on page 195.

If you do so, the `RunningTime` attribute has the value 10,000,500,000 after the RTT sequence is terminated. The RTT sequence waits 10 seconds (= 10,000,000,000 ns) caused by the implemented while-loop. An additional step is necessary to end the RTT sequence. This step is always necessary regardless of an implemented code.

Related topics

Basics

Basics on Accessing Variables of a Simulation Application on a Remote Node.....	150
Notes on Accessing Variables of a Simulation Application on a Remote Node.....	153

Examples

Demo Examples of Using Real-Time Testing.....	22
---	----

References

[rttlib.variable Module \(Real-Time Testing Library Reference !\[\]\(4b7a79268f6ba26c1471d4232fffa85a_img.jpg\)\)](#)

Tips and Tricks

Introduction

Below are some tips and tricks for implementing RTT sequences.

Writing Effective RTT Sequences

Introduction

The RTT sequences and the real-time application run on the same platform. It is therefore vital for the code of RTT sequence to save calculation power.

Using local variables

Use local variables instead of global variables. The Python interpreter can access local variables faster because it does not access them via the dictionary. Arguments of a function definition are local variables. It is not necessary to copy their values to a local variable explicitly.

xrange or range

Only for Python Version 2.7: If you program a **for** statement, use **xrange** instead of **range**. The **xrange** function returns an **xrange** iterator object instead of a whole list. This is an opaque sequence type which yields the same values as the corresponding list, without actually storing them all simultaneously. This makes the **xrange** function faster than the **range** function.

Calling generator functions sequentially

If you get peaks in the turnaround time when you call two generator functions consecutively, you should insert **yield None** between the function calls. Otherwise the code of the first generator function which comes after the last yield, and the code of the following generator function which comes before the first yield, are executed in one simulation step. **yield None** splits these parts so that they are executed in two simulation steps. This avoids an overrun.

Accessing execution times

You can access the execution time of the Real-Time Testing service function through the following C variables:

- **dsrtt_pre_comp_exec_time** contains the time used for all sequences in the PreComputation channel.
- **dsrtt_post_comp_exec_time** contains the time for all scripts in the PostComputation channel plus the time needed to initialize a currently downloading RTT sequence and to delete RTT sequence.

You must add the following lines to the user TRC file `_USR.TRC` to display the two variables in ControlDesk's Variables controlbar:

```
dsrttpre_comp_exec_time
{
  type: flt(64,IEEE)
  alias: "dsrttpre_comp_exec_time"
  desc: "execution time of PreComputation channel"
  flags: SYSTEM|READONLY
  unit: "s"
}
dsrttpost_comp_exec_time
{
  type: flt(64,IEEE)
  alias: "dsrttpost_comp_exec_time"
  desc: "execution time of PostComputation channel + initialization"
  flags: SYSTEM|READONLY
  unit: "s"
}
```

Sorting lists

When sorting lists with the `sort()` method, use the `key` function which is called only once per list item instead of the `comparison` function which is called several times.

Since sorting large lists can take a long time, sort only lists with ten elements or less.

Avoiding a timeout during initialization of an RTT sequence

The time available for initializing an RTT sequence is limited. If your RTT sequence requires more time for the initialization, for example, when long lists are created, a timeout occurs. In this case, you can increase the available initialization time. Refer to [Avoiding a Timeout During Initialization of an RTT Sequence](#) on page 70.

Avoiding calls to `eval()` and `exec()`

Calling the `eval()` and `exec()` commands will destroy the real-time behavior, because their arguments are parsed and compiled by the Python bytecode compiler. Parsing and compiling is performed at run-time and will therefore trigger large peaks in the turnaround time.

For example, you use `eval()` and `exec()` commands to refer to a variable by name:

```
value = eval("my_variable_" + i)
```

A better approach is:

```
local_dict = locals()
value = local_dict["my_variable_" + i]
```

Real-time capability of Python data structures

Large lists, tuples, or dictionaries cannot be run in real-time.

Using `collections.deque` instead of lists You can use the `deque` data type from the `collections` module instead of lists to save computation time and make

the data real-time capable. Deques (double-ended queues) are a generalization of stacks and queues. They support the thread-safe, memory-efficient **append** and **pop** methods from either side of the deque with approximately the same $O(1)$ performance, that is constant execution time independent of the queue length, in either direction. Refer to *deque objects* in the *Python Library Reference*.

Wiki.Python

Tip

For a complete overview of the performance tips, refer to <http://wiki.python.org/moin/PythonSpeed/PerformanceTips>.

Managing RTT Sequences

Introduction The following sections describe how you can execute RTT sequences on dSPACE platforms. They explain how to start and monitor one or more RTT sequences and provide information on how you can debug an RTT sequence.

Where to go from here

Information in this section

Basics on Managing RTT Sequences.....	166
Gives basic information on managing RTT sequences using the rttmanagerlib module and Real-Time Test Manager.	
Managing RTT Sequences Using the Real-Time Test Manager.....	170
Describes how you can handle the RTT sequences on the real-time platform using the Real-Time Test Manager. This is the easiest way, but you cannot use all the features for handling RTT sequences.	
Managing RTT Sequences in Python Scripts.....	181
Describes how to handle RTT sequences on the simulation platform using Python scripts. This lets you use all the features for handling RTT sequences, for example, event handling.	
Error Management.....	198
Introduction to the Message Reader API.....	201

Information in other sections

Basics on Managing RTT Sequences

Introduction

The following topics give basic information on managing RTT sequences using the rttmanagerlib module and Real-Time Test Manager.

Where to go from here

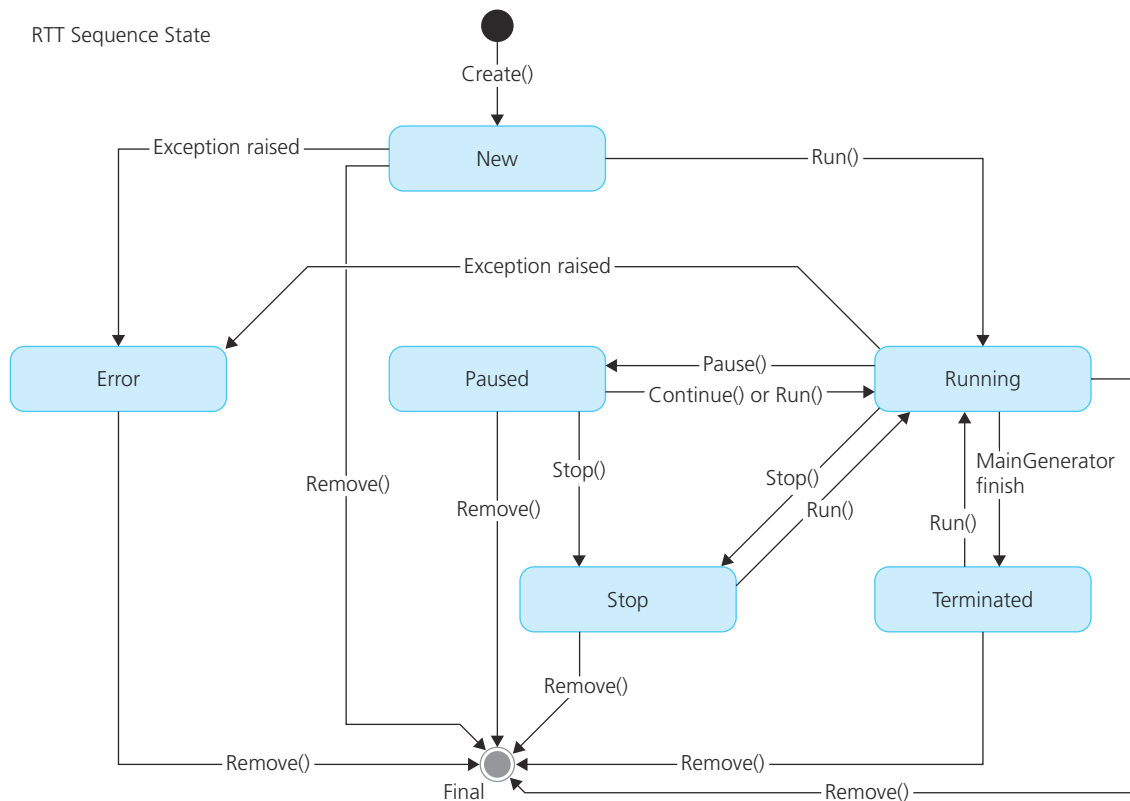
Information in this section

States of RTT Sequences.....	166
Showing the state diagram for RTT sequences.	
Basics on Executing RTT Sequences.....	167
Providing basic information on how RTT sequences are executed on the simulation platform.	

States of RTT Sequences

States

The following illustration shows the states of RTT sequences and the user commands which change the state of the RTT sequence. The commands are available in the Real-Time Test Manager and the rttmanagerlib module.



Paused, stopped, and terminated RTT sequences can be restarted with the same namespace. Refer to [Basics on Running RTT Sequences](#) on page 29.

Running and paused RTT sequences are automatically stopped (stop state) if the simulation model is stopped.

Related topics

Basics

[Managing RTT Sequences in Python Scripts..... 181](#)

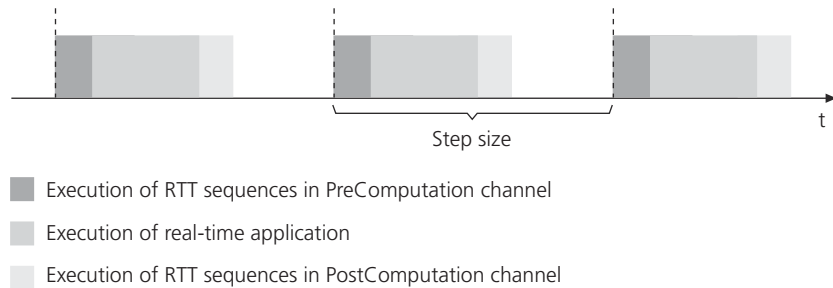
Basics on Executing RTT Sequences

Introduction

RTT sequences are executed in a time slice before the simulation model is calculated or afterwards. As several RTT sequences can be executed, the order is specified by priorities.

Channel

The channel specifies when an RTT sequence is executed in relation to the calculation of the model by the simulation application. See the following illustration.



PreComputation channel The RTT sequence is calculated before the simulation model within a sampling step.

PostComputation channel The RTT sequence is calculated after the simulation model within a sampling step.

Execution order in a channel

If RTT sequences are not independent of each other, the execution order may be important. The execution order within a channel is specified by the priorities of the RTT sequences. The priorities are specified when the RTT sequences are created.

If RTT sequences have the same priority, they are executed in the reverse download order: The most recently created RTT sequence is executed before older RTT sequences (within a sampling step).

Namespace

Each RTT sequence has its own namespace on the platform. To exchange data between different RTT sequences, you can use the `globalvariables` or the `dynamicvariables` module. Refer to [Using Variables Accessible by Several RTT Sequences](#) on page 62 and [Basics on Dynamic Variables](#) on page 77.

Real-Time Testing version check

Real-Time Testing checks all components for version compatibility when creating RTT sequences.

For real-time testing, a Python interpreter is integrated in the simulation platform. The version of this Python interpreter and the version of Python installed on the host PC for BCG file generation must be equal.

For more information on using matching Real-Time Testing versions, refer to [Using Different Versions of Real-Time Testing](#) on page 226.

If an RTT sequence (PY file) is compatible with the version of the Python interpreter on the simulation platform but its BCG file was generated for another Python version, you must generate its BCG file again. Refer to [BCGServiceProvider \(Real-Time Testing Library Reference\)](#).

Starting RTT sequences when the simulation model is stopped

RTT sequences are executed only if the simulation model is running. However, you can access the board and create RTT sequences if the simulation model is stopped. You can also start the RTT sequences but they are executed only if you start the simulation model. When you start the simulation, the RTT sequences in the **running** state are executed synchronously.

This means that RTT sequences may be in **running** state but are not executed.

Stopping the simulation model

If you stop the simulation model, all the RTT sequences in the **running** or **paused** state are also stopped (**stop** state).

Related topics

Basics

Creating and Starting RTT Sequences in Python Scripts.....	182
Features of Real-Time Testing.....	14

References

[rttlib.globalvariables Module \(Real-Time Testing Library Reference !\[\]\(cf531ed27e91483460120fcc057b3901_img.jpg\)](#))

Managing RTT Sequences Using the Real-Time Test Manager

Introduction

This section describes how you can handle the RTT sequences on the real-time platform using the Real-Time Test Manager. This is the easiest way, but you cannot use all the features for handling RTT sequences.

Where to go from here

Information in this section

[Basics on the Real-Time Test Manager..... 171](#)

The Real-Time Test Manager is a graphical user interface for handling RTT sequences.

[How to Customize the Screen Arrangement..... 172](#)

Instructions on creating a custom pane arrangement.

[How to Specify and Use a Filter..... 174](#)

If a table has a lot of entries, you can use a filter to reduce the number of entries displayed.

[How to Start the Real-Time Test Manager and Access a Platform..... 176](#)

Before you can execute your RTT sequences, you must start the Real-Time Test Manager and access the platform on which the test will be executed.

[How to Create a New RTT Sequence on the Platform..... 178](#)

When you are connected to the platform, you can start testing.

[How to Manage RTT Sequences on the Real-Time Platform..... 180](#)

You can manage the RTT sequences with the Real-Time Test Manager, which lets you run, pause, continue, stop, delete or reload an RTT sequence.

Information in other sections

[Managing RTT Sequences in Python Scripts..... 181](#)

Describes how to handle RTT sequences on the simulation platform using Python scripts. This lets you use all the features for handling RTT sequences, for example, event handling.

[Real-Time Test Manager Commands \(Real-Time Testing Library Reference \)](#)

Lists the commands of the Real-Time Test Manager available in the main menu and the context menus of the platform view and sequence list.

Basics on the Real-Time Test Manager

Introduction

The Real-Time Test Manager is a graphical user interface for handling RTT sequences.

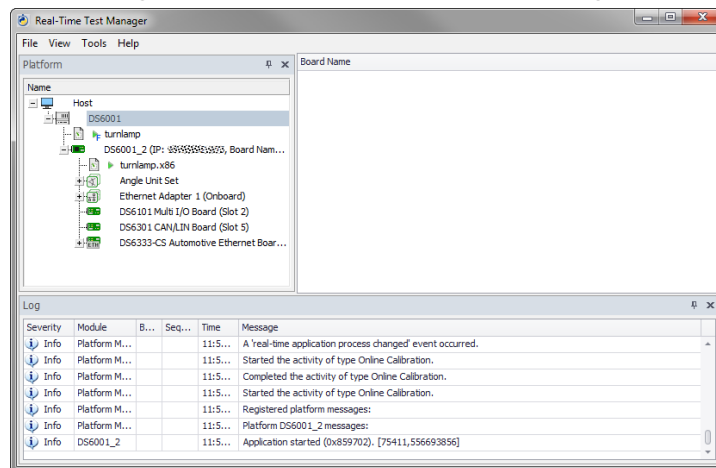
Features of the Real-Time Test Manager

The Real-Time Test Manager manages the RTT sequences of platforms. It can access any platforms that are suitable for real-time testing.

To download the real-time application to the platform, use a dSPACE tool with Platform Manager, for example, ControlDesk, as the Real-Time Test Manager cannot do this.

User interface of the Real-Time Test Manager

The following illustration shows the Real-Time Test Manager.



The Real-Time Test Manager has several working areas:

- Platform view: For accessing the platform for real-time testing
- Sequence list in Board Name window: Lists all the created RTT sequences and their settings (including global variables and source for data streaming) on the selected platform.
- Log Viewer: Displays information on the real-time test outputs of the Python print command used in the RTT sequence.

The menu commands are available in the main menu and context menu. For a description of the commands, refer to [Real-Time Test Manager Commands \(Real-Time Testing Library Reference\)](#).

Working with Real-Time Test Manager

You can manage RTT sequences on the real-time platform. See

- [How to Start the Real-Time Test Manager and Access a Platform](#) on page 176
- [How to Create a New RTT Sequence on the Platform](#) on page 178
- [How to Manage RTT Sequences on the Real-Time Platform](#) on page 180

Tip

The graphical user interface of Real-Time Test Manager supports the following features:

- You can modify the screen layout. Refer to [How to Customize the Screen Arrangement](#) on page 172.
- You can specify filters for some tables to reduce the number of entries displayed. Refer to [How to Specify and Use a Filter](#) on page 174.

Related topics**Basics**

[Implementing RTT Sequences.....](#) 43

How to Customize the Screen Arrangement

Objective

The screen arrangement defines which panes are displayed and how they are arranged. The first time you execute the application, it starts with its default screen arrangement, which you can modify.

Screen modifications

The screen arrangement contains information about:

- Display states and positions of the toolbars
- Display states and positions of the panes such as the Platform view
- Pane settings, such as the docking state

Saving the screen arrangement

All the modifications you made to the screen are automatically saved to the current screen arrangement when you exit the application. You cannot save them explicitly.

Resetting the screen arrangement






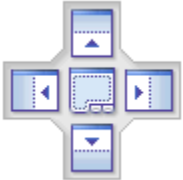
To discard all the modifications that you made in the screen arrangement, you can reset it to its default. Use the **Reset to Default** command from the **View** menu to make the user interface look like it did the first time you started the application.

Commands for customizing the screen arrangement

The application provides various commands, such as the **Floating** command, to modify the state of each pane. You can execute all of these commands quickly and flexibly via mouse. The following instructions describe how to do this.

Method**To customize the screen arrangement**

- 1 Move the mouse pointer onto the title bar of the pane whose position you want to change.
If you want to move a tabbed pane, you must select its tab instead of the title bar.
- 2 Drag the pane to another position while holding the left mouse button down.
The docking state of the pane is automatically changed to *floating* and the screen displays *docking stickers* that you can use to specify the new position.

Docking Sticker	Description
	The pane is docked to the top of your application's main window.
	The pane is docked to the bottom of your application's main window.
	The pane is docked to the left of your application's main window.
	The pane is docked to the right of your application's main window.
	The pane is docked to the top, bottom, left, or right of your application's working area.
	The pane is docked above, below, to the left, or to the right of the selected pane. If you drag the mouse onto the middle docking sticker, the pane is docked as a new page.

- 3 Move the mouse pointer onto a docking sticker. When the area of the new position is displayed, release the left mouse button.

Result

The component is moved to the new position in the user interface and docked to another component.

If you release the mouse button anywhere except on a docking sticker, the docking state of the pane remains floating.

Tip

If you want to change the order of pane tabs, you can drag them to new positions.

Related topics**HowTos**

[How to Specify and Use a Filter.....](#) 174

How to Specify and Use a Filter

Objective

If a table has a lot of entries, you can use a filter to reduce the number of entries displayed.

Possible methods

You can use the Filter Editor to specify a filter containing several conditions or use a simple autofilter based on the table entries.

- To specify a filter using the Filter Editor, refer to [Basics of the Filter Editor](#) on page 174 and [Method 1](#) on page 175.
- To use an autofilter, refer to [Method 2](#) on page 175.

Basics of the Filter Editor

A filter can consist of several conditions that are combined by logical operations. You can group conditions to create a multistage filter. In a condition, a column header of the table is compared with a specified value. An entry is displayed only if it fulfills the filter criteria. The following illustration shows an example.

Logical operator An operator that compares several conditions or condition groups. The result is either true or false. The Filter Editor provides four logical operators (AND, OR, NOR, NAND). Logical operators are displayed in red.

Relational operator An operator that compares two values. The Filter Editor provides relational operators for mathematical and string comparisons. Relational operators are displayed in green.

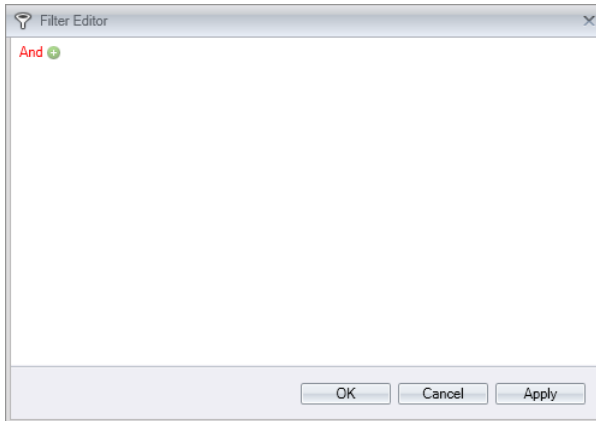
Column header The table header column that is used to build a filter condition.

Condition An expression that compares a value of the table column header with a specified value via a relational operator. The result is either true or false.

Condition group A group of two or more conditions that are combined by a logical operator. The result is either true or false. A condition group can also contain a condition group, so you can build a multistage filter.

Method 1**To specify and use a filter using the Filter Editor**

- 1 Click  to open the Filter Editor.




The Filter Editor opens. It contains a red word: And. This is a logical operator.

- 2 Click the red word.

The Filter Editor opens a menu for you use to change the logical operator or add a condition or condition group:

- To change the logical operator, select And, Or, NotAnd, or NotOr.
- To add a condition, select Add Condition.
- To add a condition group, select Add Group.

- 3 If you have added a condition, specify it:

- Click the blue word to select a column header.
- Click the green word to select a relational operator.
- Click the gray word(s) to specify the values.
- To delete a condition, click it first and then  or press - or Del.


- 4 Repeat the previous steps until the filter is complete.

- 5 Click Apply to apply the filter.

When the filter is active, only entries that meet the filter criteria are displayed. If you are not satisfied with the result, you can repeat the previous steps.

- 6 Click OK

The Filter Editor dialog is closed, and the filter is active. The active filter is displayed in the line below the table.

- 7 To clear the filter, clear the checkbox or click .

Cleared filters are not deleted. They are still available.

- 8 To use a defined filter, click the down arrow to open the filter list and select the filter.

Method 2**To specify and use a filter via autofilters**

- 1 Click the filter symbol in the column header.

A list of all the column entries opens.

- 2 Select the entries to display.
- 3 Click outside the list.

Result Fewer entries are displayed.

Related topics

HowTos


[How to Customize the Screen Arrangement.....](#) 172


How to Start the Real-Time Test Manager and Access a Platform


Objective Before you can execute your RTT sequences, you must start the Real-Time Test Manager and access the platform on which the test will be executed.

Registering platforms

You must register the platform on which the test will be executed in the Real-Time Test Manager. The Real-Time Test Manager is connected to the platform for creating and handling RTT sequences only if the platform is registered. This configuration is stored, so that you must register a platform only once. The next time you start the Real-Time Test Manager, it can be connected to registered platforms automatically. As this can lead in long timeouts of the startup process, you can disable this automatic search.

For further information on managing recent platform configurations, refer to [Manage Recent Platform Configuration](#) (Real-Time Testing Library Reference )

For further information on enabling/disabling the automatic search for registered platforms, refer to [General Properties](#) (Real-Time Testing Library Reference )

For further information on running a search for registered platforms manually, refer to [Refresh Platform Configuration](#) (Real-Time Testing Library Reference )

Preconditions

- The real-time application must be prepared for real-time testing. Refer to [Enabling Real-Time Testing for dSPACE Platforms](#) on page 33.
- In a multiprocessor system, all the real-time applications which are involved must be prepared, see [Basics on Accessing Variables of a Simulation Application on a Remote Node](#) on page 150.
- The real-time applications must be downloaded and started. This can be done using ControlDesk or ConfigurationDesk.

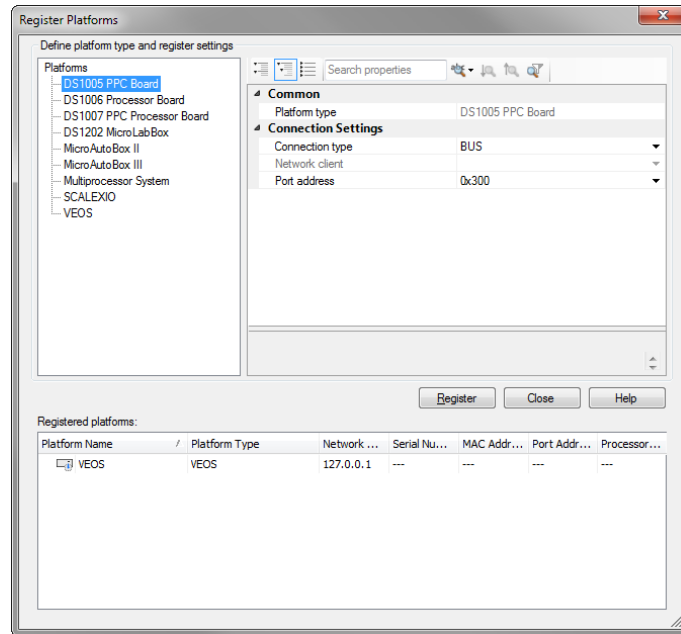
Method**To start the Real-Time Test Manager and access a platform**

- 1 From the Windows Start menu, select Programs – dSPACE Real-Time Testing – Real-Time Testing <x.y>, and click Real-Time Test Manager <x.y>.

The Real-Time Test Manager opens.

- 2 In the Tools menu, select Register Platforms.

The Register Platforms dialog opens.



- 3 In the Platforms list, select the type of the platform.
- 4 In the Connection Settings category, specify the connection settings. For details of the properties, refer to [Register Platforms \(Real-Time Testing Library Reference\)](#).
- 5 Click Register.
The platform is added to the Registered platforms list. If registration fails, check the communication settings or the connection from the host PC to the platform.
- 6 In the Platform view, open the context menu of the platform and select Connect.
If the connection fails, check whether the real-time applications run.

Result

The Real-Time Test Manager has access to the platform. If RTT sequences were already created on the platform, they are displayed. You can start creating new RTT sequences.

Related topics**HowTos**

[How to Create a New RTT Sequence on the Platform.....](#) 178

References

[Connect \(Real-Time Testing Library Reference !\[\]\(339a16584d5da0f0a3ca4e9ec17bf6a1_img.jpg\)\)](#)
[Manage Recent Platform Configuration \(Real-Time Testing Library Reference !\[\]\(e06a1d39938b2f5d7a2c3618fea4f77f_img.jpg\)\)](#)
[Refresh Platform Configuration \(Real-Time Testing Library Reference !\[\]\(23ac9e28f2600a1e787d149d7f76716a_img.jpg\)\)](#)
[Register Platforms \(Real-Time Testing Library Reference !\[\]\(ba1ec627dd10668218bdb3f2bf103f06_img.jpg\)\)](#)

How to Create a New RTT Sequence on the Platform

Objective

You must create a new RTT sequence for the real-time platform before you can start testing.

Preconditions

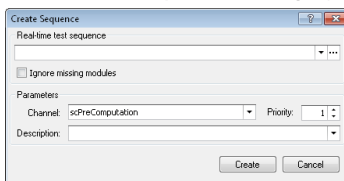
- The Real-Time Test Manager is running and you have access to the platform. Refer to [How to Start the Real-Time Test Manager and Access a Platform](#) on page 176.
- To generate a signed BCG file, you must have the developer version.
- To create a new RTT sequence for the real-time platform (i.e., download the BCG file to it), you must have the developer or operator version.


Method**To create a new RTT sequence on the platform****Tip**

To create an RTT sequence with the default settings, drag the RTT sequence file in PY or BCG file format to the real-time platform.

- 1 On the platform's context menu, select **Create Sequence**.

The **Create Sequence** dialog opens.



- 2 Select your RTT sequence file in the PY or BCG file format. Click  to create an RTT sequence that was already created or use the Browse button to browse to an RTT sequence.

You can select the Python script of the RTT sequence or an RTT sequence in the intermediate file format BCG. For more information on BCG files, refer to [Basics on the Real-Time Test Manager Server Interface](#) on page 182.

- 3 To ignore missing modules which are imported in the RTT sequence, select **Ignore missing modules**.
- 4 Specify the run parameter, see the following table.

Parameter	Description
Channel	<p>Time when the RTT sequence is executed</p> <ul style="list-style-type: none"> ▪ scPreComputation: The RTT sequence is executed before the simulation model is calculated by the real-time application. ▪ scPostComputation: The RTT sequence is executed after the simulation model is calculated by the real-time application. <p>For basic information, refer to Basics on Running RTT Sequences on page 29.</p> <p>It is possible to change the channel within the initialization phase of an RTT sequence, refer to SequenceProperties (Real-Time Testing Library Reference).</p>
Priority	<p>Priority of the RTT sequence in a range from 1 to 256, where 1 is the highest priority. The priority specifies the execution order of the RTT sequences. If RTT sequences have the same priority, they are executed in the reverse order in which they are downloaded to the real-time platform. The most recently created RTT sequence is then executed before older RTT sequences in a sampling step.</p>
Description	User-defined description of the RTT sequence

- 5 Click **Create**.

The RTT sequence is checked and created for the platform. The Real-Time Test Manager shows the RTT sequence in the list and the platform where it is created. It has the **new** status.

If an exception occurs during creation, it is displayed in the **Log Viewer**. The RTT sequence has the **error** status.

Result

The RTT sequence has been downloaded to the platform.

Related topics







Basics

[Basics on Executing RTT Sequences](#)..... 167

References

[Create Sequence \(Real-Time Testing Library Reference\)](#)

How to Manage RTT Sequences on the Real-Time Platform

Objective	You can manage the RTT sequences with the Real-Time Test Manager, which lets you run, pause, continue, stop, delete, or reload RTT sequences.
State of real-time application	The real-time application is not affected and does not change its state when you manage the RTT sequences.
Preconditions	The RTT sequences must be created on the real-time platform, see How to Create a New RTT Sequence on the Platform on page 178.
Method	<p>To manage one or more RTT sequences on the real-time platform</p> <ol style="list-style-type: none"> 1 In the Sequences list, select one or more RTT sequences whose status you want to change (use the Ctrl or Shift key for multiselection). 2 Open the context menu of one of the selected RTT sequences and choose a command: <ul style="list-style-type: none"> ▪ To start the RTT sequences: Run ▪ To pause the RTT sequences: Pause ▪ To continue the RTT sequences: Continue ▪ To stop the RTT sequences: Stop ▪ To reload the RTT sequences: Reload ▪ To open the RTT sequence for editing: Open. ▪ To delete the RTT sequences: Delete <p>For an overview of the states, refer to States of RTT Sequences on page 166.</p>
Result	The states of the selected RTT sequences are changed. If an RTT sequence uses data streaming, the source file is listed under the RTT sequence in the Sequence list. Global variables are listed on the Global Variable page.
Related topics	<p>References</p> <ul style="list-style-type: none"> Continue (Real-Time Testing Library Reference ) Delete (Real-Time Testing Library Reference ) Open (Real-Time Testing Library Reference ) Reload (Real-Time Testing Library Reference ) Run (Real-Time Testing Library Reference ) Stop (Real-Time Testing Library Reference )

Managing RTT Sequences in Python Scripts

Introduction

This section describes how you can handle RTT sequences on the simulation platform using Python scripts. This lets you use all the features for handling RTT sequences, for example, event handling.

Where to go from here

Information in this section

[Basics on the Real-Time Test Manager Server Interface..... 182](#)

Real-Time Test Manager Server interface (rttmanagerlib) is used in Python scripts to handle RTT sequences.

[Creating and Starting RTT Sequences in Python Scripts..... 182](#)

You can create and start RTT sequences on the simulation platform using the rttmanagerlib module.

[Starting RTT Sequences with Arguments in Python Scripts..... 185](#)

You can start RTT sequences with arguments on the simulation platform using the rttmanagerlib module.

[Controlling one RTT Sequence in Python Scripts..... 187](#)

You can manage an RTT sequence using the rttmanagerlib module. You can start, pause, continue, stop, or delete an RTT sequence.

[Controlling All the Created RTT Sequences in Python Scripts..... 189](#)

You can manage all the created RTT sequences using the rttmanagerlib module. You can start, pause, continue, or stop all the RTT sequences together in one step.

[Handling Events of RTT Sequences in Python Scripts..... 191](#)

You can implement event handling for events triggered by several states of the RTT sequences. The events can be evaluated in the Python script running on the host.

[Handling OnHostCall Events of an RTT Sequence in Python Scripts..... 193](#)

You can implement event handling for OnHostCall events triggered by an RTT sequence. The events can be evaluated in the Python script running on the host.

[Getting the Run Time of an RTT Sequence..... 195](#)

You can use attributes of a Sequence object to get the running or active time of an RTT sequence and the model step size.

Information in other sections

[Managing RTT Sequences Using the Real-Time Test Manager..... 170](#)

Describes how you can handle the RTT sequences on the real-time platform using the Real-Time Test Manager. This is the easiest way, but you cannot use all the features for handling RTT sequences.

[dSPACE Python Modules for Managing RTT Sequences \(Real-Time Testing Library Reference !\[\]\(d0a1791f26d167e866e44ebbf83efebe_img.jpg\)\)](#)

These modules are available on the host PC. You can use them to manage the RTT sequences and the Real-Time Test Manager Server.

Basics on the Real-Time Test Manager Server Interface

Introduction

The Real-Time Test Manager Server interface is a Python library. You can use it to program Python scripts which download and manage RTT sequences. These Python scripts run in a Python interpreter on the host PC of your dSPACE system.

Real-Time Test Manager Server interface

The Real-Time Test Manager Server interface is a Python library for managing real-time tests. The library lets you access the simulation platform, generate BCG files, and create the RTT sequence on the simulation platform. Refer to [rttmanagerlib Module \(Real-Time Testing Library Reference !\[\]\(10f8862fc183b400327470ea85afe9ae_img.jpg\)\)](#).

Related topics

References

[dSPACE Python Modules for Managing RTT Sequences \(Real-Time Testing Library Reference !\[\]\(d5d7044e5caf6907399af2dced8d6ff8_img.jpg\)\)](#)

Creating and Starting RTT Sequences in Python Scripts

Introduction

You can create and start RTT sequences on the simulation platform using the `rttmanagerlib` module.

Mandatory tasks

A simple start script must perform some mandatory tasks:

1. Initialize the Real-Time Test Manager Server interface.
2. Generate the BCG file of the RTT sequence.
3. Access the platform or platforms (in a multiprocessor system).
4. Create the RTT sequence on the platform.

To generate a signed BCG file, you must have the developer version.

5. Start the RTT sequence.

For details, refer to [Basics on the Real-Time Test Manager Server Interface](#) on page 182.

Example

The following Python script is for creating and starting an RTT sequence. It shows only how to start an RTT sequence from scratch.

```
#-----
# Import real-time testing modules
#-----
# This Python Library is used to handle the test script download.
import rttmanagerlib
# This Python Library provides utility functions.
import rttutilities
#-----
# Import this module for accessing the Python COM automation API.
#-----
import pythoncom
#-----
# Use the standard Python built-in Libraries for operation-system commands.
#-----
import os
import sys
#-----
# Module global variables
#-----
workingDir = os.path.dirname(os.path.realpath(__file__))
if os.path.isdir(workingDir) == 0:
    workingDir = os.getcwd()
workingDir = os.path.abspath(workingDir)
workingDir = workingDir[:workingDir.rfind("\\")]
testScriptName = workingDir + r'\RTTSequences\RTTSequence.py'
#-----
# Import real-time testing demo utilities.
#-----
import rttdemoutilities
def executeDemo(boardName):
    """
        Function : executeDemo
        Execute part of the demo.
    """
    # Create new RTTManagerServer instance
    rttManager = rttmanagerlib.RealTimeTestManagerServer()
    # Generate byte code from the test script (RTT sequence)
    bcgFileName = rttManager.BCGServiceProvider.Generate(testScriptName,[])
    print("BCG file created: %s" % bcgFileName)
    board = rttManager.AccessBoard(boardName)
    print("Board '%s' successfully connected." % board.Name)
    # Connect to sequence's event handle
    sequencesEvents = rttdemoutilities.RTTMSequencesEvents(board.Sequences)
    try:
        # Load a sequence to the real-time platform
        sequence = board.Sequences.Create(bcgFileName)
        print("Sequence '%s' on real-time platform created." % bcgFileName)
        # Start the sequence on the real-time platform
        sequence.Run()
        print("Sequence on real-time platform started.\n")
        # Suspend host script for the execution time of the real-time testing
        # script
        numberOfSeconds = 11
        print("\nSuspend host script execution for %d seconds." \
              % numberOfSeconds)
        rttutilities.RTTSleep(numberOfSeconds)
```

```

finally:
    # Clear traceback object to free COM object
    sys.last_traceback = None
    # Clean up if sequencesEvents:
    sequencesEvents.close()
    sequencesEvents = None
    sequence = None
    board = None
    rttManager = None

#-----
# Main Program
#-----
if __name__ == "__main__":
    if (2 == len(sys.argv)):
        # The script execution is started in python.exe.
        if "python.exe" in sys.executable.lower():
            # Python.exe does not call the COM initialization method.
            # Initialize the COM Libraries for the calling thread.
            pythoncom.CoInitialize()
        try:
            board = sys.argv[1]
            print("\nThe real-time testing demonstration is starting...")
            executeDemo(board)
            print("\nThe real-time testing demonstration was successfully "\
                  "completed.")
        finally:
            # The script execution is started in python.exe.
            if "python.exe" in sys.executable.lower():
                # Python.exe does not call the COM uninitialization method.
                # Uninitialize the COM Libraries for the calling thread.
                pythoncom.CoUninitialize()
    else:
        print("No platform defined.\n")

```

Description


The **FileName** variable specifies the RTT sequence to be started. The **UserSearchPath** specifies additional folders that are searched for user Python modules. You must adapt these settings to use the start script.



BCGServiceProvider.Generate() generates and signs the BCG file and saves it to the folder where the Python script of the RTT sequence is stored. If there is already a BCG file, the modification times of the Python script and the corresponding BCG file are compared. If the Python script is younger than the BCG file, a new BCG file is generated. Otherwise, no new BCG file is generated.

Board = rttm.AccessBoard(BoardName) accesses the simulation platform on which the real-time test is executed. The platform must be registered on the host PC and a real-time application with Real-Time Testing support enabled must be running on it. In the example, the platform name is specified via an argument (**sys.argv[1]**) when the script is started. **sys.argv[1]** must contain the board name, for example, **'127.0.0.1/MyApp'**. You can also specify the platform in the script. Refer to [AccessBoard Method \(Real-Time Testing Library Reference\)](#).

The script specifies **scPreComputation** as the channel. This means that the RTT sequence is executed before the simulation model is calculated by the simulation application. If the RTT sequence must be executed after the simulation

application is calculated by the real-time application, use the `scPostComputation` channel.

It is possible to change the channel within the initialization phase of an RTT sequence, refer to [SequenceProperties](#) (Real-Time Testing Library Reference ).

`Sequence = Board.Sequences.Create(BCGFileName, SequenceChannel = Channel)` creates the RTT sequence on the platform specified by the `Board` object. A Python object can be passed to the RTT sequence during creation. The object can be read by a method and can only be used in the sequence initialization phase. For details, refer to [Create Method](#) (Real-Time Testing Library Reference ) and [GetSequenceArgument Function](#) (Real-Time Testing Library Reference ).

`Sequence.Run()` starts the RTT sequence. Parameters can be passed to the RTT sequence when sequence execution starts. Refer to [Starting RTT Sequences with Arguments in Python Scripts](#) on page 185.

The variables are cleared in the `finally:` statement. Note that if the object of the `RealTimeTestManagerServer()` is cleared (`rttm` in the example), the Real-Time Test Manager Server is stopped and cannot be used any longer.

This example shows only how to start an RTT sequence. For more handling features, refer to [Controlling one RTT Sequence in Python Scripts](#) on page 187.

Related topics

Basics

Basics on Executing RTT Sequences.....	167
Starting RTT Sequences with Arguments in Python Scripts.....	185

References

[BCGServiceProvider Class Description](#) (Real-Time Testing Library Reference )
[rttmanagerlib Module](#) (Real-Time Testing Library Reference )

Starting RTT Sequences with Arguments in Python Scripts

Introduction

You can start RTT sequences with arguments on the simulation platform using the `rttmanagerlib` module.

Basics

- You can only start RTT sequences with arguments using the `rttmanagerlib` module. You cannot use the Real-Time Test Manager for this.
- The `MainGenerator(*args)` function must contain an `(*args)` parameter, which is the argument list passed from the host PC Python script containing all parameters for the RTT sequence.

If you do not want to pass arguments, use the `MainGenerator()` function with empty parentheses.

- You can pass only one parameter. If you want to pass multiple objects, use a list or a tuple.

Limitations for RTT sequences with arguments

- When starting an RTT sequence with an argument list, you also have to specify an arguments parameter for the `MainGenerator` function in your RTT sequence. Use `'def MainGenerator(*args)'` instead of `'def MainGenerator()'`.

If the parameter is missing and the RTT sequence is called with an argument list, an exception is raised with the unspecific error message `C API call failed`.

- The size of the argument is limited to 10 kbit when creating an RTT sequence on a DS1007, MicroLabBox, VEOS, V-ECU on VEOS, SCALEXIO, or DS6001 platform. On other platform types, the size of the argument is not restricted.

RTT sequence

The following Python script is a short example of an RTT sequence starting with arguments passed on from the host PC Python script. Refer to the demo archive `<MyDocuments>\dSPACE\Real-Time Testing\5.0\09_SequenceArguments` and see the `RTTSequence.py` file.

```
# Identifier for real-time testing print messages
RTT_PREFIX = r" *RTT:* "
def MainGenerator(*args):
    print RTT_PREFIX + "Start of real-time test execution on target platform."
    yield None
    if (len(args) > 0):
        print RTT_PREFIX + "Received argument: '%s'." %str(args[0])
        yield None
        print RTT_PREFIX + "Argument is of type '%s'." %type(args[0])
    else:
        print RTT_PREFIX + "No argument received."
    yield None
    print RTT_PREFIX + "End of real-time test execution."
    yield None
```

Host PC Python script

The following Python script is a short example of a host PC Python script starting an RTT sequence and passing on arguments to the RTT sequence at the same time. For the complete example, refer to the `RTTLoader.py` file of the demo.

```
...
def ExecuteDemo(BoardName):
    # Create new sequence
    RTTManager = rttmanagerlib.RealTimeTestManagerServer()
    # Generate byte code from the test script (RTT sequence)
    BcgFileName = RTTManager.BCGServiceProvider.Generate(TEST_SCRIPT_NAME,[])
    Board = RTTManager.AccessBoard(BoardName)
    print "Board '%s' successfully connected." %Board.Name
    # Connect to sequence's event handle
    SequencesEvents = rttidemoutilities.RTTMSequencesEvents(Board.Sequences)
```

```

try:
    # Load a sequence to the real-time platform
    Sequence = Board.Sequences.Create(BcgFileName)
    print "Sequence '%s' on real-time platform created." %BcgFileName
    # Create some data tuples
    DataSetCollection, MixedArgumentsCollection = CreateDemoData()
    for Collection in (DataSetCollection, MixedArgumentsCollection):
        for DataSet in Collection:
            # Start sequence with arguments.
            print "\nStart the sequence on the real-time platform with\n
                arguments."
            # Start the sequence on the real-time platform
            Sequence.Run(DataSet)
            print "Sequence '%s' on real-time platform started."\
                %Sequence.Name
            # Suspend host script for the execution time of the real-time\
testing script
            NumberOfSeconds = 5
            print "Suspend host script execution for %d seconds."\
                %NumberOfSeconds
            rttutilities.RTTSleep(NumberOfSeconds)
    ...

```

Description

In the example, the RTT sequence is started with an argument which is specified by the parameter of the `Run` method (`Sequence.Run(DataSet)`). The value of `DataSet` is therefore available in the `MainGenerator(*args)` function. If you want to have the value of an argument in the initialization phase of an RTT sequence, you must use the `create` method. A parameter of the `create` method used in the host PC Python script can be read by the `GetSequenceArgument` function in the RTT sequence. Refer to [Create Method \(Real-Time Testing Library Reference\)](#) and [GetSequenceArgument Function \(Real-Time Testing Library Reference\)](#).

Related topics

Basics

[Creating and Starting RTT Sequences in Python Scripts](#)..... 182

References

[rttmanagerlib Module \(Real-Time Testing Library Reference\)](#)

Controlling one RTT Sequence in Python Scripts

Introduction

You can manage an RTT sequence using the `rttmanagerlib` module. You can start, pause, continue, stop, or delete an RTT sequence. You can change the state of an RTT sequence independently of the state of the simulation application.

Status of real-time application	The simulation application is not affected when you manage the RTT sequences. The simulation application does not change its status.
Pause/continue	You can continue the execution of an RTT sequence at the point where it was paused.
Stopping	<p>When you stop an RTT sequence, you can restart or recreate it.</p> <p>Restarting starts the RTT sequence with the state it had when it stopped (see below (refer to Restarting on page 188)).</p> <p>Recreating starts and initializes the RTT sequence, refer to Creating and Starting RTT Sequences in Python Scripts on page 182.</p> <p>A stopped RTT sequence is not deleted from the simulation platform.</p>
Restarting	<p>An RTT sequence can be restarted if it has one of the following states:</p> <ul style="list-style-type: none"> ▪ Paused ▪ Stopped ▪ Terminated <p>When you restart an RTT sequence, it is not initialized again. Only the <code>MainGenerator(*args)</code> is restarted and the RTT sequence starts with the state it had when it stopped. This can lead to unexpected behavior.</p>
Deleting	<p>You can delete an RTT sequence from the simulation platform manually.</p> <p>All RTT sequences are automatically deleted when a simulation application is downloaded to the simulation platform.</p>
Example	<p>The following example shows how you can manage an RTT sequence.</p> <pre>import rttmanagerlib import rttutilities import pythoncom # The script execution is started in python.exe. if "python.exe" in sys.executable.lower(): # Python.exe does not call the COM initialization method. # Initialize the COM libraries for the calling thread. pythoncom.CoInitialize() # Specify file name of the RTT sequence FileName = r"D:\rtt\MyScript_rt.py" UserSearchPath = [r"D:\MyBaseLibraries", r"D:\MyBaseUtilities"] # Initialize the Real-Time Test Manager Server rttm = rttmanagerlib.RealTimeTestManagerServer() # Generating the BCG file BCGFileName = rttm.BCGServiceProvider.Generate(FileName, UserSearchPath) # Access the platform Board = rttm.AccessBoard("DS1006")</pre>

```

# Set channel for RTT sequence
Channel = rttmanagerlib.constants.scPreComputation
try:
    # Create the RTT sequence
    Sequence = Board.Sequences.Create(BCGFileName, SequenceChannel = Channel)
    print(Sequence.State)
    # Start the RTT Sequence
    Sequence.Run()
    print(Sequence.State)
    # Pause the RTT sequence, wait 2 s and then continue
    Sequence.Pause()
    print(Sequence.State)
    rttutilities.RTTSleep(2)
    Sequence.Continue()
    print(Sequence.State)
    # Stop the RTT sequence
    rttutilities.RTTSleep(2)
    Sequence.Stop()
    print(Sequence.State)
    # Remove all RTT sequences
    while(Board.Sequences.Count):
        Board.Sequences.Remove(0)
finally:
    Sequence = None
    Board = None
    rttm = None

# The script execution is started in python.exe.
if "python.exe" in sys.executable.lower():
    # Python.exe does not call the COM uninitialization method.
    # Uninitialize the COM Libraries for the calling thread.
    pythoncom.CoUninitialize()

```

Related topics

Basics

States of RTT Sequences..... 166

References

[RTTSleep Function Description \(Real-Time Testing Library Reference !\[\]\(0aff635c4179ba9e710b00f4b01d3b20_img.jpg\)\)](#)
[Sequence \(Real-Time Testing Library Reference !\[\]\(29658d981ebdf5edc259074cbf6110e0_img.jpg\)\)](#)
[Sequences \(Collection\) \(Real-Time Testing Library Reference !\[\]\(9b3d169a802e50e3425ebff869ff6250_img.jpg\)\)](#)

Controlling All the Created RTT Sequences in Python Scripts

Introduction

You can manage all the created RTT sequences using the `rttmanagerlib` module. You can start, pause, continue or stop all the RTT sequences together in one simulation step. You can change the state of RTT sequences independently of the state of the real-time application.

Controlling all the created RTT sequences

You can start, pause, continue, or stop all the created RTT sequences together in one step.

Note

You can only restart RTT sequences one after the other, not in one step.

Example

The following example shows how you can manage three RTT sequences in parallel.

```
import rttmanagerlib
import rttutilities
import pythoncom
# The script execution is started in python.exe.
if "python.exe" in sys.executable.lower():
    # Python.exe does not call the COM initialization method.
    # Initialize the COM Libraries for the calling thread.
    pythoncom.CoInitialize()
# Specify file name of the RTT sequence
FileName1 = r"D:\rtt\MyScript1_rt.py"
FileName2 = r"D:\rtt\MyScript2_rt.py"
FileName3 = r"D:\rtt\MyScript3_rt.py"
UserSearchPath = [r"D:\MyBaseLibraries", r"D:\MyBaseUtilities"]
# Initialize the Real-Time Test Manager Server
rttm = rttmanagerlib.RealTimeTestManagerServer()
# Generating the BCG file
BCGFileName1 = rttm.BCGServiceProvider.Generate(FileName1, UserSearchPath)
BCGFileName2 = rttm.BCGServiceProvider.Generate(FileName2, UserSearchPath)
BCGFileName3 = rttm.BCGServiceProvider.Generate(FileName3, UserSearchPath)
# Access the platform
Board = rttm.AccessBoard("DS1006")
# Set channel for RTT sequence
Channel = rttmanagerlib.constants.scPreComputation
try:
    # Download the RTT sequences
    Sequence1 = Board.Sequences.Create(BCGFileName1, SequenceChannel = Channel)
    Sequence2 = Board.Sequences.Create(BCGFileName2, SequenceChannel = Channel)
    Sequence3 = Board.Sequences.Create(BCGFileName3, SequenceChannel = Channel)
    # Start the RTT Sequence
    Board.Sequences.RunAll()
    # Pause the RTT sequences, wait 2 s and then continue
    Board.Sequences.PauseAll()
    rttutilities.RTTSleep(2)
    # Continue the RTT sequences
    Board.Sequences.ContinueAll()
    # Stop the RTT sequences
    rttutilities.RTTSleep(2)
    Board.Sequences.StopAll()
    # Remove all RTT sequences
    while(Board.Sequences.Count):
        Board.Sequences.Remove(0)
finally:
    Sequence = None
    Board = None
    rttm = None
```



```
# The script execution is started in python.exe.
if "python.exe" in sys.executable.lower():
    # Python.exe does not call the COM uninitialization method.
    # Uninitialize the COM Libraries for the calling thread.
    pythoncom.CoUninitialize()
```

Related topics

Basics

Controlling one RTT Sequence in Python Scripts.....	187
States of RTT Sequences.....	166

References

RTTSleep Function Description (Real-Time Testing Library Reference )
 Sequences (Collection) (Real-Time Testing Library Reference )

Handling Events of RTT Sequences in Python Scripts


Introduction

You can implement event handling for events triggered by the RTT sequences. The events can be evaluated in the Python script running on the host.

Event handling

You can implement event handling for the following events.

Event	Description
OnError	An error occurs in an RTT sequence.
OnStateChanged	The state of an RTT sequence changes. For more details on the states, refer to States of RTT Sequences on page 166.
OnWrite	The print command is used in the RTT sequence.
OnRemove	An RTT sequence is removed from the simulation platform.
OnCreate	A new RTT sequence is created.

If you create event handlers, they must use a specified method. For details on the method, refer to [SequencesEvents Class Description](#) (Real-Time Testing Library Reference )

Example

The following example shows how an event handler can be implemented. The event handler writes only information on the event to standard output.

```
import os
import rttmanagerlib
import rttutilities
```

```

State2Str = { rttmanagerlib.constants.sesNew : "New",
              rttmanagerlib.constants.sesTerminated : "Terminated",
              rttmanagerlib.constants.sesError : "Error",
              rttmanagerlib.constants.sesStopped : "Stopped",
              rttmanagerlib.constants.sesRunning : "Running",
              rttmanagerlib.constants.sesPaused : "Paused",
              }

class RTTSequencesEvents(rttmanagerlib._IRTSequencesEvents):
    def __init__(self, EventSource):
        # Call base class constructor to connect to event source
        rttmanagerlib._IRTSequencesEvents.__init__(self, EventSource)
    def OnError(self, Sequence):
        """Method OnError"""
        Sequence = rttmanagerlib.Sequence(Sequence)
        print("OnError: ", Sequence.Name)
    def OnStateChanged(self, Sequence, NewState):
        """Method OnStateChanged"""
        Sequence = rttmanagerlib.Sequence(Sequence)
        print("OnStateChanged: %s, new state: %s" % \
              (Sequence.Name, State2Str[NewState]))
    def OnWrite(self, Sequence, Output):
        """Method OnWrite"""
        Sequence = rttmanagerlib.Sequence(Sequence)
        print("OnWrite: %s, %s" % (Sequence.Name, Output))
    def OnRemove(self, Name):
        """Method OnRemove"""
        print("OnRemove: ", Name)
    def OnCreate(self, Sequence):
        """Method OnCreate"""
        Sequence = rttmanagerlib.Sequence(Sequence)
        print("OnCreate: ", Sequence.Name)

def main():
    SequencesEvents = None
    WorkingDir = os.path.dirname(sys.argv[0])
    if os.path.isdir(WorkingDir) == 0:
        WorkingDir = os.getcwd()
    RTTSequenceFileName = WorkingDir + r'\d_RTTSequenceTemplate.py'
    rttm = rttmanagerlib.RealTimeTestManagerServer()
    # Generate BCG file
    BcgFileName = rttm.BCGServiceProvider.Generate(RTTSequenceFileName,[])
    try:
        Board = rttm.AccessBoard("DS1006")
        # Connect to sequences event handle
        SequencesEvents = RTTSequencesEvents(Board.Sequences)
        # Create the new RTT Sequence on platform
        Sequence = Board.Sequences.Create(BcgFileName)
        # Start RTT sequence
        Sequence.Run()
        # Wait for events
        rttutilities.RTTSleep(1)
        # Pause RTT sequence
        Sequence.Pause()
        # Wait for events
        rttutilities.RTTSleep(1)
        # Continue RTT sequence
        Sequence.Continue()
        # Wait for events
        rttutilities.RTTSleep(1)
        # Stop RTT sequence
        Sequence.Stop()
    
```



```

# Wait for events
rttutilities.RTTSleep(1)
# Remove RTT sequence from platform
Sequence.Remove()
# Wait for events
rttutilities.RTTSleep(1)
Sequence = None

finally:
    if SequencesEvents:
        # Disconnect from sequences event handle
        SequencesEvents.close()
        SequencesEvents = None
        Board = None
        rttm = None

#-----
# module main block
#-----
if __name__ == '__main__':
    main()

```

Description

The `RTTMSequencesEvents` class handles the events. In this example it prints the event information to the standard output.

Related topics**Basics**

Exchanging Data Between RTT Sequences and Python Scripts Running on the Host	
PC.....	83
Handling OnHostCall Events of an RTT Sequence in Python Scripts.....	193

Handling OnHostCall Events of an RTT Sequence in Python Scripts

Introduction

You can implement event handling for OnHostCall events triggered by a single RTT sequence. The events can be evaluated in the Python script running on the host.

Event handling

You can implement event handling for the following event:

Event	Description
OnHostCall	The host script gets a Python data object from an RTT sequence. The return value is sent to the RTT sequence.

Only one client at a time can access the event handling. If another client tries to access the event handling, an exception is raised for that client.

If you create event handlers, they must use a specified method. For details on the method, refer to [SequenceEvents Class Description \(Real-Time Testing Library Reference !\[\]\(b4eeff342f60cc7bcd67d869b4fedca2_img.jpg\)](#)).

For information on how to raise events in an RTT sequence, refer to [Exchanging Data Between RTT Sequences and Python Scripts Running on the Host PC](#) on page 83.

Example

The following example shows how an event handler can be implemented. This is a part of a demo Python script. The complete Python script (RTTLoader.py) is in <MyDocuments>\dSPACE\Real-Time Testing\5.0\11_HostCalls. The appropriate RTT sequence for this demo (RTTSequence.py) is available in the same folder.

```
class RTTHostCallEvents(rttmanagerlib._IRTSequenceEvents):
    def __init__(self, Sequence, BoardName):
        # Call base class constructor to connect to event source
        rttmanagerlib._IRTSequenceEvents.__init__(self, Sequence)
        self.CurrentBoardName = BoardName
    def OnHostCall(self, *Data):
        """Method OnHostCall
        Parameters : *Data - Tuple - Tuple with different Python objects. The tuple
                     is filled in an RTT sequence.
        Description : This method is called if an OnHostCall event is received
                     in the current sequence.
        Return Value: ReturnResultToRT - List - List of different Python objects
                     which are sent back to the RTT sequence.
        Limitations : The return value must be restorable with a cPickle module.
        """
        # Initialize return value
        ReturnResultToRT = []
        # Implement individual reaction to data objects in host call events
        try:
            # The received argument *Data is a tuple. The tuple can be filled with
            # different Python objects.
            for Element in Data:
                if isinstance(Element, (str, unicode)):
                    # Print the content of the string coming from the real-time platform
                    print ON_HOST + str(Element)
                    ReturnResultToRT.append("String '%s' received on host." %Element)
                elif isinstance(Element, d_RTTSequence_DataExchange.HCTransferClass):
                    print ON_HOST + r"Received a '%s' object." %(Element.__class__.__name__)
                    ElementName = Element.GetName()
                    ElementValue = Element.GetValue()
                    # Start any program with automation interface, e.g. rtplib
                    # Check the received data with the rtplib
                    RtplibValue = None
                    RtplibValue = GetValueFromRtplib(self.CurrentBoardName, ElementName)
                    # Check result
                    if RtplibValue != ElementValue:
                        raise Exception, ON_HOST + "Different values of data object ('%s') and Rtplib ('%s')." \
                                                    %(ElementValue, RtplibValue)
                    else:
                        print ON_HOST + "Data object was successfully tested."
                        # Set new data value
                        Element.Value = ElementValue + 1
                        print ON_HOST + "Set new data object value '%s' and send data object back to RTT sequence." \
                                                    %Element.Value
                        # Send element with new value back to real-time platform
                        ReturnResultToRT.append(Element)
                        # Send element with a comment back to real-time platform
                        ReturnResultToRT.append(r"Data object successfully proved on host.")
```

```

        # All objects which are not of string type or DataExchange objects.
        else:
            print "%s '%s' of type '%s'." %(ON_HOST, Element, type(Element))
    except:
        print rttutilities.GetTraceBackString()
    # The return arguments are sent back to the calling sequence running
    # on the real-time platform (and now waiting for the answer from the host).
    return ReturnResultToRT

```

Description

The `OnHostCall` function definition in the `RTTSequenceEvents` class handles the host call events of an RTT sequence. The `Data` parameter contains the Python objects which are sent from the RTT sequence. The `ReturnResultToRT` variable contains the objects which are sent back to the RTT sequence. In this example, the event handler writes only information on the event to standard output.

To be able to use the `OnHostCall` events, the Python script must be connected to the sequence event handle.

```

# Initialize events
HostCallEvent = None
...
# Connect to sequence event handle
HostCallEvent = RTTHostCallEvents(Sequence, BoardName)

```

Related topics

Basics

[Handling Events of RTT Sequences in Python Scripts..... 191](#)

Getting the Run Time of an RTT Sequence

Introduction

You can use attributes of a `Sequence` object to get the running or active time of an RTT sequence and the model step size.

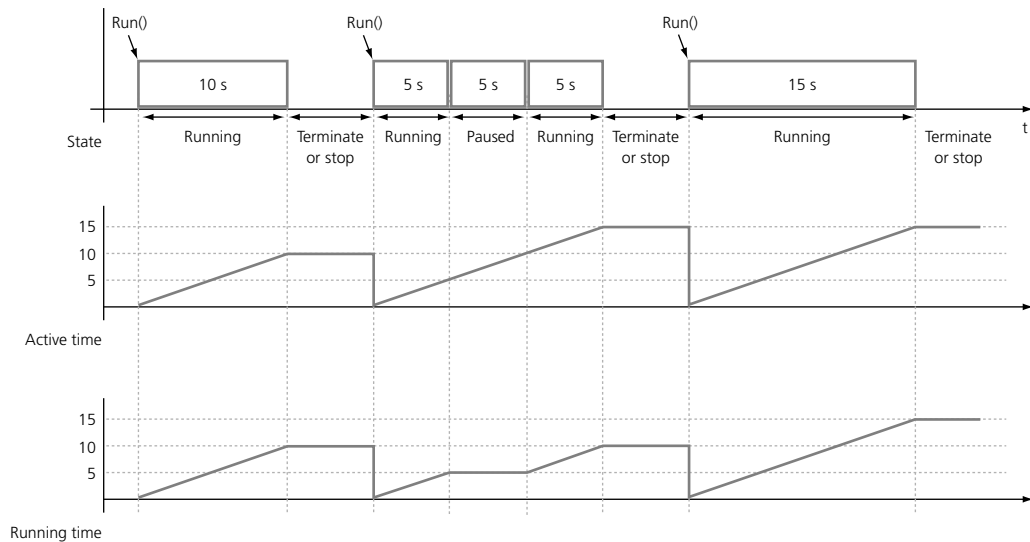
Basics

The `Sequence` object has attributes to get information about the run time of the corresponding RTT sequence and the model step size.

ActiveTime The `ActiveTime` attribute contains the time in which an RTT sequence is active. That is the time when it is in Running and Paused state. However, the value can only be read when the RTT sequence is in Running state. The value is set to 0 when it is started (`Sequence.Run()` or `Sequences.RunAll()`).

RunningTime The `RunningTime` attribute contains the time in which the RTT sequence is in the Running state. The value is set to 0 when it is started (`Sequence.Run()` or `Sequences.RunAll()`).

The following illustration shows the difference between the **ActiveTime** and **RunningTime** attributes when an RTT sequence is executed.



The first curve shows the times when the RTT sequence is executed and its states. The second and third curve show the values of the **ActiveTime** and **RunningTime** attributes.

Preparing the Python script

To read the attributes, you must cast the **Sequence** object of the **SequencesEvents** class. The following example shows the necessary adaption. The first listing shows a part of the original Python code of the **SequencesEvents** class:

```
import rttmanagerlib
class RTTSequencesEvents(rttmanagerlib._IRTSequencesEvents):
    def __init__(self, EventSource):
        # Call base class constructor to connect to event source
        rttmanagerlib._IRTSequencesEvents.__init__(self, EventSource)
    def OnError(self, Sequence):
        """Method OnError"""
        Sequence = rttmanagerlib.IRTSequence(Sequence)
```

The second listing shows the adapted Python code of the **SequencesEvents** class which allows the using the attributes (changed code is written in bold):

```
import rttmanagerlib
class RTTSequencesEvents(rttmanagerlib._IRTSequencesEvents):
    def __init__(self, EventSource):
        # Call base class constructor to connect to event source
        rttmanagerlib._IRTSequencesEvents.__init__(self, EventSource)
    def OnError(self, Sequence):
        """Method OnError"""
        Sequence = rttmanagerlib.IRTSequence1(Sequence)
        print(Sequence.ActiveTime)
```

Without this casting you can still use parameters or functions older than Real-Time Testing 1.8.1.

Example

If you use the variables introduced above, you can measure the execution time of the following simple RTT sequence.

```
from rttlib import utilities
def MainGenerator():
    yield utilities.Wait(10)
```

Although the RTT sequence has only a `Wait` function, which should wait for exactly 10 seconds, the measurement will provide a value of 10.0005 seconds. The difference derives from one step which is executed between the last `yield` and the end of the `MainGenerator`. This part of the RTT sequence can be empty, nevertheless it requires one step to execute.

Related topics**Basics**

[States of RTT Sequences..... 166](#)

References

[Sequence Class Description \(Real-Time Testing Library Reference !\[\]\(8bba887393ca45b761e5cb49e755e762_img.jpg\)](#))
[SequencesEvents Class Description \(Real-Time Testing Library Reference !\[\]\(b898b980f2d860cdb0237afbc3664529_img.jpg\)](#))
[Wait Function \(Real-Time Testing Library Reference !\[\]\(489b6f540446f926b6e5cda90c9ff8a8_img.jpg\)](#))

Error Management

Where to go from here

Information in this section

[Debugging RTT Sequences](#)..... 198

If an error occurred in RTT sequences, you get some information for debugging.

[Example of Debugging an RTT Sequence](#)..... 198

Shows how you can debug an RTT sequence running on a simulation platform.

Debugging RTT Sequences

Introduction

If an error occurred in RTT sequences, you get some information for debugging.

Traceback

If an RTT sequence has a syntax error, the error message includes a stack traceback showing the file name and code line where the error occurred.

Exception

You can implement exception handling, refer to [Implementing an Exception Handling](#) on page 73.

Related topics

Examples

[Example of Debugging an RTT Sequence](#)..... 198

Example of Debugging an RTT Sequence

Introduction

This example shows how you can debug an RTT sequence which runs on a simulation platform. An error is purposely inserted in an RTT sequence, and the example shows how the error can be found.

Demo files

The example is based on an application demo. All the required files are in the demo folder.

Loader script You can find the Python script for downloading in the Python folder of <MyDocuments>\dSPACE\Real-Time Testing\5.0\08_Stimulus\08_Stimulus.zip.

RTT sequence You can find the RTT sequence in the RTTSequences folder of <MyDocuments>\dSPACE\Real-Time Testing\5.0\08_Stimulus\08_Stimulus.zip.

Model You can find the model and the corresponding experiment used in this example in the <MyDocuments>\dSPACE\Real-Time Testing\5.0\SampleExperiments\TurnSignal_<platform> folder prepared for the platform types.

Method

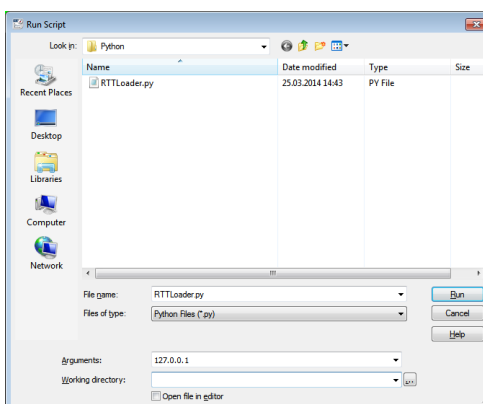
To debug an RTT sequence

- 1 In the ControlDesk's Source Code Editor, open the RTTSequence.py file (path see above).
- 2 Insert the error in the RTT sequence. Change the name of a real-time variable to WrningLightSwitch (44th code line):

```
WarningLightSwitch = variable.Variable(r'Model Root/WrningLightSwitch[0]
1]/Value')
```

- 3 Open the experiment of the TurnSignal_<platform>, TurnSignal_<platform>.CDP (path see above).
- 4 To create the RTT sequence, you must execute the loader of the RTT sequence. Go to the Automation ribbon and choose Interpreter – Run Script and specify the run parameter:

Parameter	Value
File name	Python\RTTLoader.py
Argument	Platform type: DS1006, DS1401, IP address of SCALEXIO or VEOS



Because the error is inserted, an exception is raised. See the Interpreter window:

```

The real-time testing demonstration is starting...
RTG file created: C:\Users\BerndB\Documents\dspace\Real-Time Testing\2.3\08_Stimulus\RTTSequences\RTTSequence.
Board '127.0.0.1' successfully connected.
+SEQUENCES EVENT+ New RTT sequence 'RTTSequence_2' created.
Traceback (most recent call last):
  File "C:\Users\BerndB\Documents\dspace\Real-Time Testing\2.3\08_Stimulus\Python\RTTLoader.py", line 143, in <
    executeDemo(board)
  File "C:\Users\BerndB\Documents\dspace\Real-Time Testing\2.3\08_Stimulus\Python\RTTLoader.py", line 90, in ex
    sequence = board.Sequences.Create(logFileName)
  File "dspace Internal Module RealTimeTestManagerServer:rttmanagerlib23.py", line 425, in Create
  File "C:\Program Files (x86)\Python27\lib\site-packages\win32com\client\__init__.py", line 459, in _ApplyType
    self._oleobj_.InvokeTypes(dispatch, 0, wFlags, retType, argTypes, *args)
com_error: (-2147352567, 'Exception occurred.', (0, None, u'\nRTT Traceback:\n File "C:\Users\BerndB\Docume
  
```

5 Double-click the traceback error message.

It is automatically scrolled to the incorrect code line in the Source Code Editor.

```

# Module global variables
# Create variable objects for accessing simlink signals
currentTime = utilities.currentTime
WarningLightSwitch = variable.Variable('Model Root/WarningLightSwitch/1/Value')
TurnSignalLeft = variable.Variable('Model Root/TurnSignalLeft/1/Value')
BatteryVoltage = variable.Variable('Model Root/BatteryVoltage/1/Value')
TurnSignalLever = variable.Variable('Model Root/TurnSignalLever/1/1/Value')

# Identifier for real-time testing print messages
rttPrefix = " " + "RTT:" + " "

def resetGen():
    """
    Function: resetGen
    The function sets the variables to their default values.
    """
    WarningLightSwitch.Value = 0
    TurnSignalLeft.Value = 20000
    BatteryVoltage.Value = 12.0
    TurnSignalLever.Value = 0
    yield None

def stimulateTurnSignalLeftGen(numberOfSeconds):
    """
    Function: stimulateTurnSignalLeftGen
    The function shows how to use an external Python module.
    """
    # All functions must be generator functions
    yield resetGen()
  
```

6 Correct the RTT sequence and save the file.

7 Start the loader of the RTT sequence again (see step 4).

Result

The RTT sequence runs without errors.

Related topics

Examples

Demo Examples of Using Real-Time Testing..... 22

Introduction to the Message Reader API

Where to go from here	Information in this section
	Reading dSPACE Log Messages via the Message Reader API..... 201 You can read log messages of the dSPACE Log via the Message Reader API.
	Supported dSPACE Products and Components..... 203 Provides an overview of all dSPACE products and components whose messages you can access via the Message Reader API.
	Example of Reading Messages with Python..... 203 You can read the log messages via Python. You can combine multiple filters to display only messages according to your specifications.
	Example of Reading Messages with C#..... 205 You can read the log messages via C#. You can combine multiple filters to display only messages according to your specifications.

Reading dSPACE Log Messages via the Message Reader API

Introduction	You can read log messages of the dSPACE Log via the Message Reader API.
dSPACE Log	<p>The dSPACE Log is a collection of errors, warnings, information, questions, and advice issued by all dSPACE products and connected systems over more than one session.</p> <p>The dSPACE Log is saved as a collection of binary message log files. These files are created when a dSPACE product is running. A single run of a dSPACE product is called a <i>log session</i>.</p> <div> <p>Note</p> <p>If the maximum file size for the binary message log file is reached, messages at the beginning of the dSPACE Log might get deleted. Contact dSPACE Support to solve this.</p> </div>
Message Reader API	You can use the Message Reader API to access all binary message log files of the dSPACE Log. You can combine multiple filters to display only log messages according to your specifications. For example, you can configure the Message Reader API to display only log messages from a specific dSPACE product.

The Message Reader API is available as of dSPACE Release 2020-A. For information on the dSPACE products and components that support the Message Reader API, refer to [Supported dSPACE Products and Components](#) on page 203.

dSPACE.Common.MessageReader.dll The Message Reader API is implemented by the `dSPACE.Common.MessageReader.dll` file. It is located in the `bin` subfolder of the installation folder of each dSPACE product that supports the Message Reader API.

Supported dSPACE Releases

The Message Reader API lets you access log messages written by dSPACE products since dSPACE Release 2016-B.

Message Reader API change in dSPACE Release 2021-A

There is a migration issue specific to the Message Reader API. The issue occurs if you use the API with Python. The issue was caused by the migration to Python 3.9/pythonnet 2.5.3 with dSPACE Release 2021-A.

There is no migration issue to consider if you use the API with C#.

Specifying a product filter As of dSPACE Release 2021-A, the `Products` property of the `MessageReaderSettings` class can no longer be used to set the list of products for which to filter in the log sessions. The Message Reader API provides the `SetProducts` method for this purpose.

The following table shows how to specify a product filter before and after migration:

Using Message Reader API of ...	
... dSPACE Release 2020-B and Earlier (Python 3.6)	... dSPACE Release 2021-A and Later (Python 3.9)
<pre># Specify products whose messages to read: Settings = MessageReaderSettings() Settings.Products.Add('ControlDesk') Settings.Products.Add('AutomationDesk')</pre>	<pre># Specify products whose messages to read: Settings = MessageReaderSettings() Settings.SetProducts(['ControlDesk', 'AutomationDesk'])</pre>

Related topics

Basics

[Supported dSPACE Products and Components](#)..... 203

Examples

[Example of Reading Messages with C#](#)..... 205
[Example of Reading Messages with Python](#)..... 203

References

[MessageReaderSettings Class \(Real-Time Testing Library Reference\)](#)

Supported dSPACE Products and Components

Supported dSPACE products and components

You can use the Message Reader API to access messages from the following dSPACE products and components:

- ASM KnC
- AutomationDesk
- Bus Manager (stand-alone)
- cmdloader
- ConfigurationDesk
- Container Management
- ControlDesk
- dSPACE AUTOSAR Compare
- dSPACE XIL API .NET Implementation
- Firmware Manager
- ModelDesk
- MotionDesk
- Real-Time Testing
- RTI Bypass Blockset
- SYNECT client
- SystemDesk
- TargetLink Property Manager
- VEOS

Related topics

Basics

[Reading dSPACE Log Messages via the Message Reader API..... 201](#)

Example of Reading Messages with Python

Introduction

You can read the log messages via Python by using the `c1r` module. You can combine multiple filters to display only messages according to your specifications.

Referencing a message reader assembly

You have to reference a `dSPACE.Common.MessageReader.dll` assembly. For information on the location of the assembly, refer to [dSPACE.Common.MessageReader.dll](#) on page 202.

In the following examples it is assumed that the dSPACE Installation Manager is installed and that the message reader assembly is installed in `C:\Program Files\Common Files\dSPACE\InstallationManager\bin`.

The following code references and imports the message reader assembly.

```
# Insert path of message log file access assembly:
import sys
AssemblyPath = r'C:\Program Files\Common Files\dSPACE\InstallationManager\bin'
if not sys.path.count(AssemblyPath):
    sys.path.insert(1, AssemblyPath)

# Add reference to assembly and import it:
import clr
clr.AddReference('dSPACE.Common.MessageReader')
from dSPACE.Common.MessageHandler.Logging import *
```

Reading all messages

The following example reads all existing message log files and prints all messages via Python. It is assumed that the message reader assembly is referenced and imported. Refer to [Referencing a message reader assembly](#) on page 203.

```
# Create message reader and print text of each message:
Reader = MessageReader(None)
for Message in Reader.ReadMessages():
    print(Message.MessageText)
Reader.Dispose()
```

Filtering messages by severity, product, and session


The following example reads and prints messages with a severity of `Error`, `SevereError`, or `SystemError`. Also, only messages of the last sessions of `ControlDesk` and `AutomationDesk` are read and printed. It is assumed that the message reader assembly is referenced and imported. Refer to [Referencing a message reader assembly](#) on page 203.

```
# Define error severities:
SEVERITY_ERROR = 3
SEVERITY_SEVERE_ERROR = 4
SEVERITY_SYSTEM_ERROR = 5

# Configure products and sessions whose messages to read:
Settings = MessageReaderSettings()
Settings.MaximalSessionCount = 1
Settings.SetProducts(['ControlDesk', 'AutomationDesk'])

# Create message reader and print text of each error message:
Reader = MessageReader(Settings)
for Message in Reader.ReadMessages():
    # Print error messages only:
    if Message.Severity == SEVERITY_ERROR or \
        Message.Severity == SEVERITY_SEVERE_ERROR or \
        Message.Severity == SEVERITY_SYSTEM_ERROR:
        print('%s: %s' % (Message.Session.ProductName, Message.MessageText))
Reader.Dispose()
```

Note

The ReadMessages method returns an enumerator which must either read all messages or must be disposed when no longer used. It is not possible to use two enumerators interleaved, only one enumerator may read messages at a time. Refer to [MessageReader Class \(Real-Time Testing Library Reference\)](#) .

Filtering messages by time

Times are given by .NET DateTime objects. Times are given as UTC times (Coordinated Universal Time). You can obtain the current UTC time by `System.DateTime.UtcNow`.

The following example reads all messages after a certain start time. It is assumed that the message reader assembly is referenced and imported. Refer to [Referencing a message reader assembly](#) on page 203.

```
import System
Settings = MessageReaderSettings()
Settings.MessageTimeAfter = System.DateTime.UtcNow # Read messages after now

# Create message reader and print time and text of each message:
Reader = MessageReader(Settings)
for Message in Reader.ReadMessages():
    print('%s: %s' % (Message.UtcTimeStamp, Message.MessageText))
Reader.Dispose()
```

Related topics**Basics**

[Reading dSPACE Log Messages via the Message Reader API](#)..... 201
[Supported dSPACE Products and Components](#)..... 203

References

[MessageReaderSettings Class \(Real-Time Testing Library Reference\)](#) 

Example of Reading Messages with C#

Introduction

You can read the log messages via C#. You can combine multiple filters to display only messages according to your specifications.

Referencing a message reader assembly

You have to reference a `dSPACE.Common.MessageReader.dll` assembly. For information on the location of the assembly, refer to [dSPACE.Common.MessageReader.dll](#) on page 202.

Reading all messages

The following example reads all existing message log files and prints the messages:

```
using dSPACE.Common.MessageHandler.Logging;
...

// Create message reader and print text of each message:
using (MessageReader reader = new MessageReader(null))
{
    foreach (message in reader.ReadMessages())
    {
        Console.WriteLine(message.MessageText);
    }
}
```

Filtering messages by severity, product, and session

The following example reads and prints messages with a severity of **Error**, **SevereError**, or **SystemError**. Also, only messages of the last sessions of **ControlDesk** and **AutomationDesk** are read and printed.

```
using dSPACE.Common.MessageHandler.Logging;
...

// Read the Last Log sessions of ControlDesk and AutomationDesk only:
MessageReaderSettings settings = new MessageReaderSettings();
settings.MaximalSessionCount = 1;
settings.Products.Add("ControlDesk");
settings.Products.Add("AutomationDesk");

using (MessageReader reader = new MessageReader(settings))
{
    foreach (ILogMessage message in reader.ReadMessages())
    {
        // Print error messages only:
        if (message.Severity == Severity.Error
            || message.Severity == Severity.SevereError
            || message.Severity == Severity.SystemError)
        {
            Console.WriteLine(message.Session.ProductName + ": " + message.MessageText);
        }
    }
}
```

Note


The `ReadMessages` method returns an enumerator which must either read all messages or must be disposed when no longer used. It is not possible to use two enumerators interleaved, only one enumerator may read messages at a time. Refer to [MessageReader Class \(Real-Time Testing Library Reference\)](#).

Related topics

Basics

Reading dSPACE Log Messages via the Message Reader API.....	201
Supported dSPACE Products and Components.....	203

References

MessageReaderSettings Class (Real-Time Testing Library Reference )
--

Test Scenario Demos

Example of Testing ECUs for Turn Signals and Warning Signals Control

Introduction	The demo shows how Real-Time Testing can be used to test the functionality of ECUs which control the turn signals and warning signals of a vehicle. The tests use several features of Real-Time Testing.
Preconditions	To work with the demo, a simulation platform is required. The demo can be used with any platform which is supported by Real-Time Testing.
Installation	<p>To access the demo, it must be copied to a work folder, see Demo Examples of Using Real-Time Testing on page 22.</p> <p>The demos are in <MyDocuments>\dSPACE\Real-Time Testing\5.0\15_TurnSignalTests\15_TurnSignalTests.zip.</p>
Test candidates	<p>The ECUs, which controls the turn signals and warning signal of a vehicle are tested. The control algorithms of the ECUs are implemented as SoftECUs in a Simulink model. The inputs of the ECUs are operating elements for the driver: The position of the turn-signal level (right, off, left) and the warning light switch (off, on). The outputs of the ECUs are the signals for the turn lights and a warning light indicator.</p> <p>The ECUs are implemented with malfunctions, for example, a malfunction if the battery voltage is below a certain voltage level. The tests should detect these malfunctions.</p>

Test specification

To test the ECUs, some tests are specified:

Test Number	Test	Description
0	T010	Turn signal left The test checks whether the left signals are activated when the turn-signal level is on the left position.
1	T020	Turn signal right The test checks whether the right signals are activated when the turn-signal level is on the right position.
2	T030	Warning light The test checks whether all signals are activated when the warning switch is on.
3	T040	Warning light together with turn signal The test checks whether pressing the warning switch has a higher priority than activating the turn-signal level.
4	T050	Reaction time of turn signal lever The test checks whether the time span between activating the turn-signal level and activating the lights is lower than 100 ms.
5	T060	Reaction time of warning light The test checks whether the time span between activating the warning light switch and activating the lights is lower than 50 ms.
6	T070	Monkey test for turn signal lever The test checks whether the left and right signals are activated correctly when the turn-signal level is activated several times in a randomized sequence.
7	T075	Monkey test for turn signal lever and warning light switch The test checks whether the signals are activated correctly when the turn-signal level and the warning light switch are activated several times in a randomized sequence. The warning signal must always have priority to the turn signals.
8	T080	Run tests T010...T075 at different battery voltages The test repeats the previous tests at different battery voltages ($V_{\text{Bat}} = 7 \text{ V} \dots 15 \text{ V}$). The signals must always be activated correctly. This test cannot be executed if the host PC and the real-time platform are connected via network.
9	T090	Test turn signals during engine start phase The test checks whether the signals are activated correctly when the engine is started.

Implemented malfunctions

The tests should find the following malfunction which are implemented intentionally.

Failure ID	Description
F010	If the turn-signal level is switched between its positions fast, the turn-on time of the signals are longer than specified. To be detected by T070

Failure ID	Description
F020	If the turn-signal level is switched between its positions fast, all signals are activated at the same time. This is only allowed if the warning light switch is on. To be detected by T070
F030	At a battery voltage below 8.2 V, the receiver of the front and rear ECUs have sporadically malfunction. To be detected by T080, T090
F040	At a battery voltage below 7.2 V, the transmitter of the central ECU does not work. No signal is activated. To be detected by T080, T090

Starting the tests

Batch files give a convenient way to start the tests. The batch files are in the **15_TurnSignalTests** folder. The **StartTurnSignalTests.bat** files do the following:

- Lets you select the platform type.
- Displays a short instruction how to start the experiment.
- Provides a menu for selecting a test
- Starts Python.exe with the Python script which starts the selected test (RTT sequence)

Overview of the test

All files required for the tests are in the **TurnSignalTests** folder and below.

Model and ControlDesk experiment All the files required for the experiment are in the **SampleExperiment\TurnSignal_<platform>** folders. The folders contain the Simulink model, all files of the ControlDesk experiment, and the real-time application built for the platform.

RTT management Python scripts manage the tests (RTT sequences) and create the test reports. They are in the **Tests** folder.

File	Description
RTTTurnSignalTests.py	Main script for turn signal tests. Execute this script with proper command line arguments to run the turn signal tests. RTTTurnSignalTests PlatformType BoardName TestNr PlatformType shows whether the registered hardware is a single-processor system ("SP"), a multicore system ("MC") or a multiprocessor system ("MP"). BoardName is the name of the real-time board, for example, DS1006. TestNr is the number of the test to be executed (see list above).
rttfailuremessages.py	Provides a dictionary with possible failure messages. Due to performance reason, creating large failure messages within RTT sequences is avoided. All the possible failure messages are stored in this dictionary. If a failure occurs, the RTT sequence adds only the key of a failure message to the test result. The value of the message is then resolved on the host.

File	Description
rttturnsignalreport.py	Provides utility functions and classes for writing the test results.
rttturnsignalutilities.py	Provides utility functions and classes for managing RTT sequences, for example, loading and removing.

RTT sequences The tests are implemented in RTT sequences. The scripts are in the `RTTSequences\RTTSequences` folder. The names of the RTT sequences correspond to the test specification listed above. The `commonlibs` subfolder contains some modules that are used in several RTT sequences.

File	Description
commonutilities.py	Containing useful functions and classes for writing RTT sequences. These functions and classes are independent of the turn signal model.
modelvariables.py	Providing variable objects for accessing variables of the turn signal model. The paths for these variables are stored in <code>signalmapping.py</code> .
signalmapping.py	Providing a mapping for the model variables of the turn signal model. Tip: Use the alias names for variable paths stored in the <code>SignalMap</code> dictionary to create variable objects. If the model is changed, you must only adjust this file.
turnsignalutilities.py	Providing some frequently used functions for testing the turn signal model.

The `data` subfolder contains a MAT file that is used for a monkey test (T075).

Related topics

Examples

[Demo Examples of Using Real-Time Testing.....22](#)

Troubleshooting

Introduction	If a problem related to Real-Time Testing comes up, you can refer to this collection of possible malfunctioning scenarios and information on solving the problem.
Where to go from here	<div>Information in this section<div>Troubleshooting for RTT Sequences.....214 Problems can occur in the RTT sequences.</div><div>Troubleshooting for Host PC Python Scripts.....226 Problems can occur in the Python scripts which run on the host PC to manage RTT sequences.</div></div>

Troubleshooting for RTT Sequences

Introduction

Problems can occur in the RTT sequences. The following topics can help you to find solutions.

Where to go from here

Information in this section

Peaks in Turnaround Time.....	215
Problem: There are peaks in the turnaround time.	
A TestOverrun Exception Occurs.....	216
Problem: A TestOverrun exception occurs when an RTT sequence is executed.	
The Output of Print Is Lost.....	217
Problem: The output of print command is lost.	
No Output in Running State.....	217
Problem: RTT sequences have the running state but give no output.	
Timeout During Initialization.....	217
Problem: A timeout occurs during initialization of an RTT sequence.	
Exceptions Occur During Initialization.....	218
Problem: Exceptions occur during initialization of an RTT sequence which uses data streaming.	
Exceptions Occur During Data Streaming.....	218
Problem: Exceptions occur during the run of data streaming.	
Problem when Using Too Many Data Streams.....	219
When replaying numerical data, problems can occur due to too many data streams used.	
The RTT Sequence Does Not Run in Real-Time.....	220
Problem: A task overrun is ignored and the RTT sequence does not run in real-time.	
Timing Problem When Removing Objects in a Loop.....	221
Problem: When RTT sequences are removed, an error message occurs.	
Variables Cannot Be Accessed After Migrating to MATLAB R2011a or Later.....	221
Problem: An RTT sequence accesses variables of a real-time application. After migrating the corresponding real-time model to MATLAB R2011a or later and rebuilding the real-time application, the RTT sequence cannot access some variables.	
Lost Variable Values.....	222
Problem: When an RTT sequence writes a value to a variable of a remote node in a multiprocessor or multicore system, the value is occasionally lost.	

Waiting for Barrier Timed Out.....	223
Problem: In a VEOS MC system, the error message "(ds_barrier_wait): waiting for barrier timed out app_no <n>" occurs.	
Exception Occurs When Creating a Variable Object.....	224
Problem: An RTT sequence aborts and an exception occurs: "... VariableManagerError: Reading from TRZ array failed. Position index ..."	
Memory Leaks Caused by Circular References.....	224
Problem: An "out of memory" exception occurs on the real-time platform.	
Using Real-Time Testing in Large SCALEXIO Systems.....	224
Problem: In large SCALEXIO systems with 55 nodes (in certain circumstances also below 55 nodes), you cannot access remote variables.	

Information in other sections

Writing Effective RTT Sequences.....	162
The RTT sequences and the real-time application run on the same hardware. It is therefore absolutely essential that the code of RTT sequence is effective to save calculation power.	
Avoiding a Timeout During Initialization of an RTT Sequence.....	70
The time available for initializing an RTT sequence is limited. If your RTT sequence requires more time for the initialization, for example, when long lists are created, a timeout occurs. In this case you can increase the available initialization time.	

Peaks in Turnaround Time

Problem	There are peaks in the turnaround time.
Description	<p>RTT sequences can cause small peaks (a few microseconds) in the turnaround time. Peaks caused by the RTT sequence can occur, if</p> <ul style="list-style-type: none"> ▪ A generator function begins (first call) or ends (last call). ▪ There is a branch in an If-Else clause, where different computations are performed in each branch.
Solution	<p>You can optimize your RTT sequence:</p> <ul style="list-style-type: none"> ▪ Insert <code>yield None</code> to distribute the execution of the RTT sequence across several simulation steps. ▪ Shift time-consuming operations into the initialization section.

- Use local variables instead of global variables in functions.
- Do not use the `print()` command.
- Do not use the `eval()` or `exec()` commands. These commands trigger the execution of the Python bytecode compiler which can lead to large peaks in the turnaround time.

Tip

For more performance tips, refer to <http://wiki.python.org/moin/PythonSpeed/PerformanceTips>.

A TestOverrun Exception Occurs

Problem

A TestOverrun exception occurs when an RTT sequence is executed, see the following example.

```
"TestOverrunError: Test overrun occurred in interpreter main
loop. (Python/Python/ceval.c 1070)
RTT Traceback:
File "D:\Work\d_RTTSequence.py", line 60, in MainGenerator"
```

Description

A TestOverrun exception occurs if the execution time of RTT sequences within a PreComputation or a PostComputation channel is too long. The TestOverrunError is used to avoid a Task Overrun error message in the real-time application so that its execution is not affected.

The exception appears when the execution time of an RTT sequence is too long.

Solution

Check the code line of the RTT sequence where the exception occurs.

If the exception occurs in the initialization phase, you can increase the computation time per sampling step reserved for the initialization of an RTT sequence, see [SetImportTimeslice Function \(Real-Time Testing Library Reference\)](#).

If the exception does not occur in the initialization phase, interrupt the execution by inserting a `yield None`.


The Output of Print Is Lost

Problem	The output of print command is lost.
Description	If you use the print command in the RTT sequence, some of its output can be lost. The output is copied to a ring buffer first. If the host is too busy to read the data from the ring buffer, older data may be overwritten. The size of the ring buffer is currently around 4 kbit.
Solution	Reduce the amount of data which is printed.

No Output in Running State

Problem	RTT sequences have the running state but give no output.
Description	RTT sequences can only be executed if the real-time application is running. If the real-time application is stopped, you can create and start RTT sequences. These RTT sequences are in the running state although they are actually not executed. They are executed when the real-time application is started.
Solution	Start the real-time application.

Timeout During Initialization

Problem	A timeout occurs during initialization of an RTT sequence.
Description	The time available for initializing an RTT sequence is limited. If your RTT sequence requires more time for the initialization, a timeout occurs.
Solution	<ul style="list-style-type: none"> Adapt the initialization time by changing the initialization time slices. Refer to SetImportTimeslice Function (Real-Time Testing Library Reference ). Stop the real-time application and load and run the RTT sequence.

The initialization phase of the RTT sequence is executed, but the script itself (`MainGenerator()` function) is not yet executed. It is executed when the real-time application starts.

Related topics

Basics

[Avoiding a Timeout During Initialization of an RTT Sequence..... 70](#)

Exceptions Occur During Initialization

Problem

Exceptions occur during initialization of an RTT sequence which uses data streaming.

Description

The RTT exception `DataStreamError: "Could not pickle data for DSRPC_RSM_GET_DATA_STREAM_INFO"` occurs in the RTT sequence's init phase.

Solution

Set an import timeslice:

```
utilities.SetImportTimeslice(...)
```

Related topics

Basics

[Data Replay in RTT Sequences..... 87](#)

Exceptions Occur During Data Streaming

Problem

Exceptions occur during the run of data streaming:

"`DataStreamError: No data in receive buffer of data stream 'DataStream_0'.`"

Description

Several reasons are possible:

- No connection to the host PC, for example, the Real-Time Test Manager Server does not run.
- The host PC is too busy to deliver data in time.
- The real-time application's background task has not enough computing time.

Data streaming is performed in the background task of the real-time application. If the turnaround time of the real-time application requires the whole computing time, data cannot be transferred in time.

Solution

- You can use the data streaming demo to check the connection to the host PC. Note that you must adapt the time of the `RTTSleep()` method.
- Check the working load of the host PC.
- Check whether there is enough time for processing the background task.

Related topics

Basics

[Data Replay in RTT Sequences.....87](#)

Problem when Using Too Many Data Streams

Problem

When replaying numerical data, problems can occur due to too many data streams used. One of the following error messages is displayed.

Error Message	Meaning
Timeout: After waiting for 10 seconds, the hardware service did not deliver any data.	The data streams do not start.
DataStreamError: No data in receive buffer of data stream 'DataStream_0'.	The data streams do not receive sufficient data.

Description

When numerical data is replayed on a simulation platform, all signals from the same time base are combined to one data stream from the host PC to the platform.

The following example shows how the time bases of the signals relate to the number of data streams. The tables represent measurements from MF4 or MAT files with different time bases.

Time	0	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9
Signal 1	2	5	3	4	1	8	3	2	7	4

Time	0	0.2	0.4	0.6	0.8
Signal 2	6	7	1	3	8

In this example, the signals are measured using two different time bases. This means, that the data is streamed to the platform using two data streams.

In the following example, the same signals are measured using the same time base.

Time	0	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9
Signal 1	2	5	3	4	1	8	3	2	7	4
Signal 2	6	6	7	7	1	1	3	3	8	8

Because the same time base is used for both signals, the data is streamed to the platform using only one data stream.

Tip

- When working with measurements from an MF4 file, the time base relates to the measurement raster. The measurement raster is displayed by the *group name* in the MF4 file.
- When working with measurements from an MAT file, the segment signal's *X-vector* is used as the time base.

The number of data streams is limited and depends on the platform hardware.

Solution

Reduce the number of data streams by using measurement data files (MF4 or MAT) with the same time base.

The RTT Sequence Does Not Run in Real-Time

Problem

A task overrun is ignored and the RTT sequence does not run in real-time.

Description

If task overruns are ignored, the RTT sequences are not executed in real-time and the **Current time** variable is delayed. Refer to [RTT sequence behavior on task overrun](#) on page 30.

Solution

In the real-time model, clear **Ignore overrun** and select **Stop simulation**. Refer to [RTI Task Configuration Dialog \(RTI and RTI-MP Implementation Reference\)](#) or [RTI Task Configuration Dialog \(Multiprocessor Setup Dialog\) \(RTI and RTI-MP Implementation Reference\)](#).

Timing Problem When Removing Objects in a Loop

Problem When RTT sequences are removed, an error message occurs with a message text similar to

“This object is invalid. The object was removed. Processor board/system ...”

Description Due to a timing problem, it can happen that invalid objects are accessed when RTT sequences are removed in a loop. The following shows an code example.

```
while 0 != Board.Sequences.Count:
    s = Board.Sequences[0]
    if s.State != rttmanagerlib.constants.sesNew:
        print("Invalid state for sequence %s: %i." %(s.Name, s.State))
        raise Exception("Invalid state exception.")
    print("Removing ", s.Name)
    Board.Sequences.Remove(0)
```

Solution Add an RTTSleep method after the Remove method, see the example

```
...
Board.Sequences.Remove(0)
rttutilities.RTTSleep(0.015)    # 15 ms
```

Related topics

References

[RTTSleep Function Description \(Real-Time Testing Library Reference !\[\]\(8bba887393ca45b761e5cb49e755e762_img.jpg\)\)](#)

Variables Cannot Be Accessed After Migrating to MATLAB R2011a or Later

Problem An RTT sequence accesses variables of a real-time application. After migrating the corresponding real-time model to MATLAB R2011a or later and rebuilding the real-time application, the RTT sequence cannot access some variables.

Description As of MATLAB R2011a, the Lookup and Lookup (2D) Simulink blocks are replaced by Lookup (nD) blocks. As a consequence, the TRC file entries for Lookup and Lookup (2D) blocks (replaced by Lookup (nD) blocks) change. If an RTT sequence has accessed such variables, the paths to the variables become invalid.

Solution Use the new paths to the variables in the RTT sequence.

Related topics

Basics

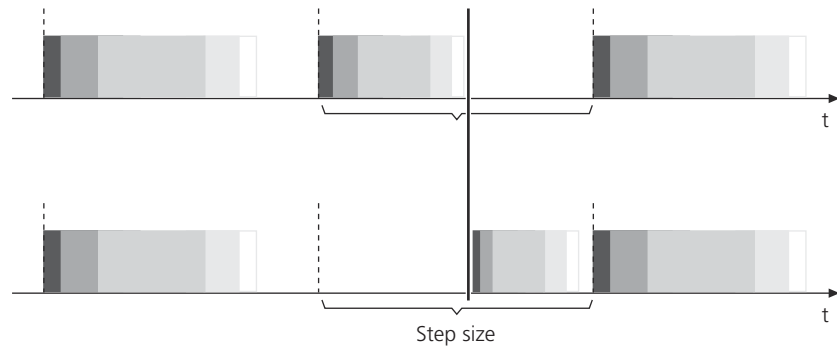
[Read/Write Access to Variables of the Simulation Application..... 62](#)

Lost Variable Values

Problem When an RTT sequence writes a value to a variable of a remote node in a multiprocessor or multicore system, the value is occasionally lost.

Description

The communication implemented in Real-Time Testing assumes that the model tasks of different nodes in multiprocessor and multicore systems are synchronous. This is true in most cases. However, it can happen that a task on a node is delayed due to a task having a higher priority. The following illustration visualizes the times when the communication buffer is read and written and times when the RTT sequences and real-time application is executed. The upper graph is the local node, the lower graph is the remote node.



- Reading data from Communication buffer
- Execution of RTT sequences in PreComputation channel
- Execution of real-time application
- Execution of RTT sequences in PostComputation channel
- Writing data to Communication buffer

In the illustration, a new value is written to the communication buffer (upper graph) although the old value has not been read by the remote node (lower graph) in the same sampling step. The remote node reads the value, the value is read that normally should be read in the following sampling step. The value of the current sampling step is lost.

Solution

Avoid the delay of model tasks:

- Avoid a model task being moved by a model task with a higher priority.
- For SCALEXIO systems, you can specify a sample time offset in ConfigurationDesk. If the value is too large or the execution time of the model task is small, the model task can be moved.

Related topics

Basics

[Accessing Variables of a Simulation Application on a Remote Node..... 150](#)

Waiting for Barrier Timed Out

Problem

For Real-Time-Testing 3.5 and lower:

In a VEOS MC system, the following warning message occurs:
(ds_barrier_wait): "Could not complete RTT Synchronization of MC-application sub-application <SubApplicationName>. Possible reason: The user specified a delay for the start of a sub-application. Synchronization will be disabled for this sub-application. If you are using RTT, variable access to this sub-application is out of sync."

Description

The warning message occurs when the application <SubApplicationName> cannot synchronize with other applications. When a remote node is accessed, a value might occasionally be lost. Refer to [Lost Variable Values](#) on page 222.

This can happen for different reasons:

- An application crashed.
- The V-ECU application was started with a time delay.

Related topics

Basics

[Accessing Variables of a Simulation Application on a Remote Node..... 150](#)

Exception Occurs When Creating a Variable Object

Problem	An RTT sequence aborts and an exception occurs: "... VariableManagerError: Reading from TRZ array failed. Position index ..."
Description	A variable object cannot be created if a structured workspace variable is an array of structures (for example, <code>Tunable Parameters/MyStruct[1].subelem1</code>) and one of the other workspace variables begins with the same prefix (for example, <code>Tunable Parameters/MyStructTEST</code>).
Solution	Rename the workspace variable so that its name starts with another prefix. For example, if the name of an array of structure is <code>Tunable Parameters/MyStruct[1].subelem1</code> , rename the other workspace variable to <code>Tunable Parameters/My2StructTEST</code> .

Memory Leaks Caused by Circular References

Problem	An "out of memory" exception occurs on the real-time platform.
Description	An out-of-memory exception can be caused by circular references of objects. The memory allocated for such objects cannot automatically be freed after removing the RTT sequence.
Solution	Real-Time Testing provides a function that forces the system to free these objects. Refer to SetCallGCAfterRemovingAllSequences Function (Real-Time Testing Library Reference) .

Using Real-Time Testing in Large SCALEXIO Systems

Problem	In large SCALEXIO systems with 55 nodes (in certain cases also below 55 nodes), you cannot access variables of remote nodes.
Description	To read and write variables of remote nodes, i.e., nodes in other application processes, their data is routed via the link boards of the SCALEXIO system. In

such large SCALEXIO systems, the buffer on the link boards is too small to manage the data.

Solution

Accessing remote variables is not possible. However, you can use real-time testing locally in application processes:

1. In ConfigurationDesk, open the Build Configuration table.
2. Select an application process.
3. In the Properties Browser, select Local real-time testing access to activate real-time testing for this application process, otherwise clear it.
4. Repeat the previous steps for the other application processes.

Troubleshooting for Host PC Python Scripts

Introduction Problems can occur in the Python scripts which run on the host PC to manage RTT sequences.

Where to go from here

Information in this section

Using Sleep() Function.....	226
Problem: If you want to execute a sleep method in your Python script, you must not use the Python sleep method.	
Using Different Versions of Real-Time Testing.....	226
Problem: The version of Real-Time Testing used on the simulation platform differs from the version used on the host PC.	
Error Message: RPC Server is Unavailable.....	228
Problem: The host script execution stops because a exception is thrown: "The RPC server is unavailable".	

Using Sleep() Function

Problem If you want to execute a sleep method in your Python script, you must not use the Python sleep method.

Solution Use the `RTTSleep` method from the `rttutilities` module, see [RTTSleep Function Description \(Real-Time Testing Library Reference !\[\]\(3211b5d1d968fc1665909b34f9f16010_img.jpg\)](#)).

Related topics

Basics

Writing Effective RTT Sequences.....	162
--	---------------------

Using Different Versions of Real-Time Testing

Problem The version of Real-Time Testing used on the simulation platform differs from the version used on the host PC.

Description

The version used on the simulation platform and on the host PC must be compatible. They are always compatible if the version numbers are the same. For example, if a real-time platform is running Real-Time Testing 3.0, the RTT sequences cannot be managed from a host PC with an active Real-Time Testing 3.2 installation.

In PHS-bus based systems and MicroAutoBox, Real-Time Testing is integrated in the real-time application. When the real-time application is built, the version of Real-Time Testing is used that is part of the active RCP & HIL installation. Note that this version can differ from the Real-Time Testing version that is active on the host PC.



On the other simulation platforms (SCALEXIO, DS6001, DS1007, MicroLabBox, MicroAutoBox III, and VEOS), Real-Time Testing is integrated into their firmware. You must therefore use on the host PC the version of Real-Time Testing that is compatible with the the firmware version of the platform.

The following table shows the dSPACE Release and Real-Time testing version on the host PC and the compatible firmware versions integrated in the platforms.

Host PC		Compatible Firmware Version				
dSPACE Release	Real-Time Testing Version	SCALEXIO	MicroAutoBox III	DS1202 MicroLabBox	DS1007	VEOS
RLS2021-A	5.0	5.1 5.0	5.1	2.16	3.16	5.2
RLS2020-B	4.4	5.0	5.0	2.14	3.14	5.1
RLS2020-A	4.3	4.6	4.6	2.12	3.12	5.0
RLS2019-B	4.2	4.5	4.5	2.10	3.10	4.5
RLS2019-A	4.1	4.4	–	2.8	3.8	4.4
RLS2018-B	4.0	4.3	–	2.6	3.6	4.3
RLS2018-A	3.4	4.2	–	2.4	3.4	4.2
RLS2017-B	3.3	4.1	–	2.2	3.2	4.1
RLS2017-A	3.2	4.0	–	2.0	3.0	4.0
RLS2016-B	3.1	3.5	–	1.7	2.6	3.7
RLS2016-A	3.0	3.4	–	1.5	2.4	3.6
RLS2015-B	2.6	3.3	–	1.3	2.2	3.5
RLS2015-A	2.5	3.2	–	–	2.0	3.4
RLS2014-B	2.4	3.1	–	–	–	3.3
RLS2014-A	2.3	3.0	–	–	–	3.2
RLS2013-B	2.2	2.3	–	–	–	3.1

You can get the firmware version on the Properties controlbar of a platform in ConfigurationDesk or ControlDesk.

Solution

1. For platforms where Real-Time Testing is integrated in the real-time application, rebuild the real-time application using the version of the active Real-Time Testing installation.
2. Install or activate a compatible version of Real-Time Testing on the host PC as used in the real-time application or the firmware.
 - If the version is already installed on the host PC, activate it using dSPACE Installation Manager. For details of the activation, refer to [How to Activate a Single dSPACE Installation](#) ([Managing dSPACE Software Installations](#) ).
 - If the version is not installed on the host PC, install a compatible version of Real-Time Testing. For details of the installation procedure, refer to [Installing dSPACE Software Products](#) ([Installing dSPACE Software](#) ).

Related topics

Basics

[Hardware and Software Requirements](#)..... 18

Error Message: RPC Server is Unavailable

Problem

The host script execution stops because the following exception is thrown: "The RPC server is unavailable".

Description

The RealTimeTestManagerServer object can be invalid. This can be caused by timing problems when the server is started and released at frequent intervals.

Solution

Add the sleep command (`rttutilities.RTTSleep`) between the generation of the BCG code and download of the RTT sequence.

Related topics

References

[RTTSleep Function Description](#) ([Real-Time Testing Library Reference](#) )

Limitations

Introduction Some limitations apply for Real-Time Testing.

Where to go from here

Information in this section

[General Limitations for Real-Time Testing.....](#)229
Some limitations apply for Real-Time Testing.

[Limitations When Using Real-Time Testing.....](#)230
You must note some limitations when using the features of Real-Time Testing.

[Limitations for Platforms.....](#)234
You must note some limitations for specific platforms when using them for Real-Time Testing.

General Limitations for Real-Time Testing

Introduction Some limitations apply for Real-Time Testing.

General limitations

- Some limitations apply to Real-Time Testing.
- RTT sequences cannot be executed on the host PC.
 - RTT sequences are completely executed on the simulation platform.
 - The output of print commands in the RTT sequence can be lost if the host PC is too busy to catch all the data.
 - A model compatible with Real-Time Testing must have at least one timer task executed with the base sample time. Real-Time Testing cannot be executed in a task driven by an interrupt block.

- When executing an RTT sequence, the latencies can be erroneous because of the inaccurate representation of floating point numbers. For details on the floating point representation and examples, refer to *B. Floating Point Arithmetic: Issues and Limitations* in the *Python reference*.
- DS1007 and MicroLabBox only: When the real-time application is started from the flash memory of a DS1007 or MicroLabBox, Real-Time Testing is not possible.
- Real-Time Testing does not support Fast Tasks.
- MicroAutoBox III, SCALEXIO, DS6001, and VEOS only: Only characters of the local code page have to be used in RTT sequences. The local code page is cp-1252 in Western languages or cp-932 for the Japanese language, for example. You can always use ASCII.

Related topics

Basics

[Basics on Accessing Variables of a Simulation Application on a Remote Node..... 150](#)

Limitations When Using Real-Time Testing

Introduction

You must note some limitations when using the features of Real-Time Testing.

Limitations for variables

The following limitations apply to the variables of the real-time model.

- If a label and a parameter in a subsystem have the same name, it is not clear without ambiguity which of them is accessed by the RTT sequence.
- Labels cannot be distinguished from parameters if they have the same name and are on the same level of the TRC tree.
- Mask and workspace parameters are not supported.
- When using ControlDesk the following limitations apply:
 - Scalar variables containing ['number'] at the end of the variable name are not supported.
 - Variable names containing a slash "/" are not supported.
- Accessing matrices or vectors of a Simulink model as a list is not supported. However, you can access the elements individually, see the following example in ControlDesk.

```
fifth_element = variable.Variable(r"Platform()//ModelRoot/Subsystem1/Array[0][4]")
```

Where **Platform** is the name of the platform that is specified when the platform is registered.

- A TRC file contains the `simState` variable to read or set the simulation state of the application. The `simState` variable must not be modified in RTT sequences. Modifying this variable may lead to an unpredictable behavior of the RTT sequence.
- If multiple structured workspace variables are in the **Tunable Parameters** TRC group, there are certain cases where subelements of one of these structures cannot be used as a variable object in a RTT Sequence:
 - A variable object cannot be created if a structured workspace variable is an array of structures (for example, `TunableParameters/MyStruct[1].subelem1`) and one of the other workspace variables begins with the same prefix (for example, `TunableParameters/MyStructTEST`).
- If you use 64-bit integer variables (INT64 and UINT64), note the following limitations.
 - If you use the scaling attribute in the variable description, 64-bit integer variables are converted into the double data type. As a result, the range of valid values is restricted to $\pm 2^{52}-1$. Values outside the range differ from the real values. When you write values outside this range, an integer overflow can occur and might remain undetected. This can cause the signedness of the value being written to flip. This special case can occur if the value is close to the range limit of the 64-bit integer variable after applying the scaling function.

Limitations for host calls

The following limitations apply when host calls are used:

- Host calls can transfer built-in Python data types (int, float, strings, dictionaries, etc.) and user-defined classes. However, it is currently not possible to transfer new-style classes which are derived from the object. Use old-style classes instead.
- To return a user-defined class object to an RTT sequence, the corresponding class definition has to be imported during the initialization phase. It is not possible to load the class definition when required. This would require an import operation during the RTT sequence's real-time execution and thus increase the likelihood of an overrun. Refer to [Exchanging Data Between RTT Sequences and Python Scripts Running on the Host PC](#) on page 83.
- If the call of `hostcall.Hostcall()` is part of the `scheduler.ParallelRace()` generator object, the host call can be aborted while it is still in process. An aborted host call can block other pending host calls, so do not use host calls in `ParallelRace()` constructs.
- For DS1006-based modular systems connected to the host PC via Ethernet, host calls are not recommended, because it provides limited performance. The recommended connection type is bus. Use a dSPACE link board for the host PC.

Limitations for RTT sequences with arguments

- When starting an RTT sequence with an argument list, you also have to specify an arguments parameter for the `MainGenerator` function in your RTT sequence. Use `'def MainGenerator(*args)'` instead of `'def MainGenerator()'`.

If the parameter is missing and the RTT sequence is called with an argument list, an exception is raised with the unspecific error message **C API call failed**.

- The size of the argument is limited to 10 kbit when creating an RTT sequence on a DS1007, MicroLabBox, VEOS, V-ECU on VEOS, SCALEXIO, or DS6001 platform. On other platform types, the size of the argument is not restricted.

Limitations in multiprocessor and multicore systems

- The length of the submodel name is restricted to 58 characters.
- The length of the variable name with its full path as the specified variable description file is restricted to 2048 characters.
- The number of variables per node connection is restricted to 500.
- You cannot read, modify and write a variable of a remote node in one sampling step. The number of required sampling steps depends on the number of node connections between the two nodes (the local node where the RTT sequence is running and the remote node whose variable is accessed).
- In the RTT sequence, values of variable objects located on remote nodes are updated in each sampling step. This increases the execution time of the RTT sequence.
- In large SCALEXIO systems with 55 nodes (in certain cases also below 55 nodes), you cannot access variables of remote nodes. For more information, refer to [Using Real-Time Testing in Large SCALEXIO Systems](#) on page 224.
- You can access only the CAN or CAN FD controllers which are on the CPU where the RTT sequence is running. Accessing a CAN or CAN FD controller on a remote CPU in a multiprocessor system is not supported.
- If a multiprocessor system is registered using ControlDesk, you must access each processor board individually (for example, calling `AccessBoard("192.168.0.15/MyApp")` and afterwards `AccessBoard("192.168.0.16/MyApp2")` and so on).
- Global variables cannot be used by RTT sequences running on different CPUs in a multiprocessor system.
- RTT remote variables must not be read in initialization phase of an RTT sequence because they might have incorrect values in this phase. If the topology distance of a remote variable is **N**, at least **N** simulation steps are required for a value to propagate through the network of computation nodes. This is especially important for the first access of the variable, since the variable's initialization value might be undefined.

Limitations for data replay

- For DS1006-based modular systems connected to the host PC via Ethernet, data replay is not recommended, because it provides limited performance. The recommended connection type is a bus. Use a DS817 Link Board (PC) for the host PC.
- The performance of a DS817 Link Board depends on the architecture of the host PC. A DS817 has a PCI host interface. In newer PC models, the performance of the PCI interface has decreased. This leads to a reduced bandwidth for host communication of a connected DS817.

- For SCALEXIO real-time PCs of the first generation (based on the i7-860 processor) that have the Linux-based operating system, the performance is slightly reduced at the Linux-based operating system.
- To replay data, a DataStreaming Server is created at the host PC. The DataStreaming Server handles the data streaming for the corresponding RTT sequence. If an exception occurs during the initialization of the DataStreaming Server, it cannot be created and there is no exception or other error message transmitted to the RTT sequence. If the RTT sequence is started anyway, another exception is displayed, because the DataStreaming server is missing and data cannot be streamed.
- Exceptions detected during data replay cannot be handled using a try-except statement in the RTT sequence if they occur on the host PC. For example, if the difference between consecutive values of the 'Time' variable is lower than the step size of the model, an exception occurs. The DataStreaming Server passes this exception to the RTT sequence, but it cannot be handled because it occurred on the host PC. You can recognize exceptions occurred on the host PC by means of the missing "RTT Traceback" keyword and missing line number in the message text.
- It is not recommended to set up more than 8 data streams in parallel, even if they contain only one signal and one time vector.
- The path for MAT files used in RTT sequences, for example, for data replay, must not include non-ascii characters, i.e., characters that are not part of the local code page of the PC which is used for the test.
- Writing to target variables on the simulation platform is limited. Refer to [Variable Class \(Real-Time Testing Library Reference\)](#).
- The supported data type for MAT file vectors is 'double'.
- An instance of a `datastream.MatFile` object cannot be started by the `Replay` method twice in parallel.
- An instance of a `datastream.MDFFile` object cannot be started by the `Replay` method twice in parallel.

Limitations for the rs232lib module

The following limitations apply when the rs232lib module is used:

- If the real-time application uses a serial interface, you cannot use it with the rs232lib module in your RTT sequence.
- The buffer size is fixed to 64 bytes and cannot be changed during run time by the rs232lib.

Related topics

References

[ParallelRace Generator Function \(Real-Time Testing Library Reference\)](#)

Limitations for Platforms

Introduction

You must note some limitations for specific platforms when using them for Real-Time Testing.

Limitations for SCALEXIO

The following Real-Time Testing modules are not supported for a SCALEXIO system with a SCALEXIO Processing Unit or DS6001 Processor Board:

- rttlib.rs232lib (sending and receiving data via an RS232 interface)

Multi-client access of several Real-Time Test Manager servers running on different PCs to the same real-time application on the same time is not allowed.

Limitations for DS1007

- For Real-Time Testing 2.5: In DS1007 multiprocessor systems, it is not possible that RTT sequences running on a node access TRC variables of remote nodes.
- The following Real-Time Testing modules are not supported for a DS1007:
 - rttlib.rs232lib (sending and receiving data via an RS232 interface)
- Real-time testing is not possible if the real-time application is started from the flash memory.

Limitations for MicroLabBox

- The following Real-Time Testing modules are not supported for MicroLabBox:
 - rttlib.rs232lib (sending and receiving data via an RS232 interface)
- Real-time testing is not possible if the real-time application is started from the flash memory.

Limitations for MicroAutoBox III

- Sending and receiving CAN messages is not supported.

Limitations for VEOS

- To work with VEOS 3.0 in connection with Real-Time Testing (RTT) 2.0, you must install at least Patch 2 for VEOS 3.0. Patch 2 is available on the dSPACE Release 2013-A DVD in the \Updates\VEOS folder. The most recent patch for VEOS 3.0 is available at <http://www.dspace.com/go/PatchesVEOS>.
- The following Real-Time Testing modules are not supported for VEOS:
 - rttlib.rs232lib (sending and receiving data via an RS232 interface)
 - rttlib.canmmlib (sending and receiving CAN messages)
- If several environment models are used and one environment model has another task rate than the others, a synchronized task execution is not guaranteed. A task might read a value in a simulation step which is actually intended for the subsequent simulation step. This can lead to the same behavior as described in [Lost Variable Values](#) on page 222.
- The following limitations are valid for V-ECU on VEOS:
 - Only CHARACTERISTICS and MEASUREMENTS (read-only) are supported
 - Only scalar variables can be accessed.

- Only linear conversion is supported.
- The PIL mode of V-ECUs is not supported.
- Groups in A2L files are not supported.
- Only pointers with a size of 4 bytes are supported.
- V-ECUs are not supported on SCALEXIO.

A

- accessing
 - CAN bus (rttlib.canlib) 117
 - CAN bus (rttlib.dscanapilib) 127
- accessing platform
 - Real-Time Test Manager 176
- accessing variable of remote node
 - example 156
- accessing variables
 - multicore system 150
 - multiprocessor system 150
 - offline simulation system 150
- application areas
 - for real-time testing 19
- ASAM General Expression Syntax 65
- ASAM MDF file
 - using data in RTT sequences 101

B

- Bus Navigator 107

C

- CAN bus
 - accessing (rttlib.canlib) 117
 - accessing (rttlib.dscanapilib) 127
- CAN FD messages 108
- CAN message
 - handling 106
 - receiving (rttlib.canlib) 121
 - receiving (rttlib.dscanapilib) 132
 - sending (rttlib.canlib) 118
 - sending (rttlib.dscanapilib) 129
- CAN message format 106
- CAN message handling
 - preparing the model 109
- canlib
 - demos 107
- canlib module 107
- Common Program Data folder 10
- continuing
 - one RTT sequence 187
 - several RTT sequences 189
- creating RTT sequence 178
 - using Python script 182
- customizing the screen arrangement 172

D

- data exchange
 - RTT sequences <-> host PC 83
- data replay
 - ASAM MDF (MF4) files 90
 - MAT files 87
- data streaming
 - ASAM MDF (MF4) files 90
 - MAT files 87
 - replay mode 93
- debugging RTT sequences
 - basics 198

- example 198
- demo archive 22
- demo installation 22
- demos
 - canlib 107
 - dscanapilib 125
- docking sticker 172
- Documents folder 10
- DS1006
 - enabling Real-Time Testing 34
- DS1006 multiprocessor system
 - enabling Real-Time Testing 36
- DS6001
 - enabling Real-Time Testing 37
- dscanapilib
 - demos 125
- dscanapilib module 124
- dsETHERNETapilib module
 - basics 137
- dSPACE log
 - printing message 69
- dynamic variables 77
 - example 80

E

- enabling Real-Time Testing 34
- encoding
 - RTT sequence 58
- Ethernet
 - sending frames 140
- Ethernet communication 137
- event handling 191
 - single RTT sequence 193
- example
 - ECU test 209
- exception handling 73
- execution order 29
- execution order of RTT sequences 167, 168
- experimental message 106

F

- file name
 - RTT sequence 27
- filter
 - specifying 174

G

- generator function 28
 - basics 46
 - nesting 47
 - Parallel() 53
 - ParallelRace() 54
 - terminating 51
- global variables 61
- globalvariables 62

H

- handling
 - CAN message 106

- RTT sequences (using RTT Manager) 180
- handling events 191
 - single RTT sequence 193
- handling exceptions 73
- host calls 83
- host PC Python scripts 182

L

- limitations
 - general 229
 - platforms 234
 - using Real-Time Testing 230
- Local Program Data folder 10
- local variables 61
- Log Viewer 171

M

- MainGenerator() 28, 59
- MAT file
 - using data in RTT sequences 98
- message
 - printing 69
- MF4 file
 - using data in RTT sequences 101
- MicroAutoBox
 - enabling Real-Time Testing 34
- MicroAutoBox III
 - enabling Real-Time Testing 38
- multicore system
 - accessing variables 150
- multiprocessor system
 - accessing variables 150

O

- offline simulation system
 - accessing variables 150
- output of variables 68

P

- Parallel() generator function 53
- ParallelRace() generator function 54
- pausing
 - one RTT sequence 187
 - several RTT sequences 189
- Platform view 171
- PostComputation channel 168
- PreComputation channel 168
- preparing Simulink model
 - for CAN message handling 109
- print function 68
- programming tips 162
- Python interpreter 26
- Python version 15

R

- Real-Time Test Manager 26
 - accessing platform 176
 - starting 176

- user interface 171
- Real-Time Test Manager interface 182
- Real-Time Test Manager Server 26
- real-time testing
 - software components 25
- Real-Time Testing
 - firmware version of platform 226
 - hardware requirements 18
 - software requirements 19
 - version 226
- Real-Time Testing version
 - compatibility 226
- receiving
 - CAN message (rttlib.canlib) 121
 - CAN message (rttlib.dscanapilib) 132
- replay mode 93
- RTI CAN MultiMessage Blockset 106
- RTT sequence 27
 - controlling 187
 - creating 178
 - file name 27
 - properties 60
- RTT sequences
 - execution order 168
 - implementing 43
 - managing 165
 - running 167
 - version check 168
- rttmanagerlib 182

S

- SCALEXIO Processing Unit
 - enabling Real-Time Testing 37
- sending
 - CAN message (rttlib.canlib) 118
 - CAN message (rttlib.dscanapilib) 129
 - Ethernet frames 140
- Sequence list 171
- serial interface communication
 - basics 144
 - receiving data 147
 - sending data 145
- Several RTT sequences
 - controlling 189
- simulation application
 - accessing variables 62
- simulation stop 30
- software components
 - of real-time testing 25
- specifying
 - filter 174
- starting
 - one RTT sequence 187
 - Real-Time Test Manager 176
 - several RTT sequences 189
- starting RTT sequence
 - using Python script 182
- starting RTT sequence with arguments
 - using Python scripts 185
- states of RTT sequences 166
- stimulating variables

- using ASAM MDF (MF4) files 101
- using MAT files 98
- stopping
 - one RTT sequence 187
 - several RTT sequences 189

T

- template 59
- terminating a generator function 51
- TestOverrunError 30
- traceback 198
- troubleshooting
 - for host Python script 226
 - for RTT sequences 214

U

- user interface
 - Real-Time Test Manager 171
- using
 - local and global variables 61
 - variables
 - global in several RTT sequences 62
- using standard Python modules 75
- UTF-8 coding 58

V

- variables
 - in RTT sequences 61
 - monitoring 68
 - of simulation application 62
- VEOS
 - enabling Real-Time Testing 41
- VEOS compatibility 41
- version
 - Python 15
 - Real-Time Testing 226

W

- Wait function 74
- watcherlib 65
- workflow for real-time testing 31

Y

- yield 27