

Simulink® Real-Time™

User's Guide



MATLAB® & SIMULINK®

R2022b



How to Contact MathWorks



Latest news: www.mathworks.com
Sales and services: www.mathworks.com/sales_and_services
User community: www.mathworks.com/matlabcentral
Technical support: www.mathworks.com/support/contact_us



Phone: 508-647-7000



The MathWorks, Inc.
1 Apple Hill Drive
Natick, MA 01760-2098

Simulink® Real-Time™ User's Guide

© COPYRIGHT 1999–2022 by The MathWorks, Inc.

The software described in this document is furnished under a license agreement. The software may be used or copied only under the terms of the license agreement. No part of this manual may be photocopied or reproduced in any form without prior written consent from The MathWorks, Inc.

FEDERAL ACQUISITION: This provision applies to all acquisitions of the Program and Documentation by, for, or through the federal government of the United States. By accepting delivery of the Program or Documentation, the government hereby agrees that this software or documentation qualifies as commercial computer software or commercial computer software documentation as such terms are used or defined in FAR 12.212, DFARS Part 227.72, and DFARS 252.227-7014. Accordingly, the terms and conditions of this Agreement and only those rights specified in this Agreement, shall pertain to and govern the use, modification, reproduction, release, performance, display, and disclosure of the Program and Documentation by the federal government (or other entity acquiring for or through the federal government) and shall supersede any conflicting contractual terms or conditions. If this License fails to meet the government's needs or is inconsistent in any respect with federal procurement law, the government agrees to return the Program and Documentation, unused, to The MathWorks, Inc.

Trademarks

MATLAB and Simulink are registered trademarks of The MathWorks, Inc. See www.mathworks.com/trademarks for a list of additional trademarks. Other product or brand names may be trademarks or registered trademarks of their respective holders.

Patents

MathWorks products are protected by one or more U.S. patents. Please see www.mathworks.com/patents for more information.

Revision History

September 1999	First printing	New for Version 1 (Release 11.1)
November 2000	Online only	Revised for Version 1.1 (Release 12)
June 2001	Online only	Revised for Version 1.2 (Release 12.1)
September 2001	Online only	Revised for Version 1.3 (Release 12.1+)
July 2002	Online only	Revised for Version 2 (Release 13)
June 2004	Online only	Revised for Version 2.5 (Release 14)
August 2004	Online only	Revised for Version 2.6 (Release 14+)
October 2004	Online only	Revised for Version 2.6.1 (Release 14SP1)
November 2004	Online only	Revised for Version 2.7 (Release 14SP1+)
March 2005	Online only	Revised for Version 2.7.2 (Release 14SP2)
September 2005	Online only	Revised for Version 2.8 (Release 14SP3)
March 2006	Online only	Revised for Version 2.9 (Release 2006a)
May 2006	Online only	Revised for Version 3.0 (Release 2006a+)
September 2006	Online only	Revised for Version 3.1 (Release 2006b)
March 2007	Online only	Revised for Version 3.2 (Release 2007a)
September 2007	Online only	Revised for Version 3.3 (Release 2007b)
March 2008	Online only	Revised for Version 3.4 (Release 2008a)
October 2008	Online only	Revised for Version 4.0 (Release 2008b)
March 2009	Online only	Revised for Version 4.1 (Release 2009a)
September 2009	Online only	Revised for Version 4.2 (Release 2009b)
March 2010	Online only	Revised for Version 4.3 (Release 2010a)
September 2010	Online only	Revised for Version 4.4 (Release 2010b)
April 2011	Online only	Revised for Version 5.0 (Release 2011a)
September 2011	Online only	Revised for Version 5.1 (Release 2011b)
March 2012	Online only	Revised for Version 5.2 (Release 2012a)
September 2012	Online only	Revised for Version 5.3 (Release 2012b)
March 2013	Online only	Revised for Version 5.4 (Release 2013a)
September 2013	Online only	Revised for Version 5.5 (Release 2013b)
March 2014	Online only	Revised for Version 6.0 (Release 2014a)
October 2014	Online only	Revised for Version 6.1 (Release 2014b)
March 2015	Online only	Revised for Version 6.2 (Release 2015a)
September 2015	Online only	Revised for Version 6.3 (Release 2015b)
March 2016	Online only	Revised for Version 6.4 (Release 2016a)
September 2016	Online only	Revised for Version 6.5 (Release 2016b)
March 2017	Online only	Revised for Version 6.6 (Release 2017a)
September 2017	Online only	Revised for Version 6.7 (Release 2017b)
March 2018	Online only	Revised for Version 6.8 (Release 2018a)
September 2018	Online only	Revised for Version 6.9 (Release 2018b)
March 2019	Online only	Revised for Version 6.10 (Release 2019a)
September 2019	Online only	Revised for Version 6.11 (Release 2019b)
March 2020	Online only	Revised for Version 6.12 (Release 2020a)
September 2020	Online only	Revised for Version 7.0 (Release 2020b)
March 2021	Online only	Revised for Version 7.1 (Release 2021a)
September 2021	Online only	Revised for Version 7.2 (Release 2021b)
March 2022	Online only	Revised for Version 8.0 (Release 2022a)
September 2022	Online only	Revised for Version 8.1 (Release 2022b)

1	Introduction	
	Simulink Real-Time Product Description	1-2
	Speedgoat Target Computers and I/O Hardware	1-3

Model Architectures

2	FPGA Models	
	Speedgoat FPGA Support with HDL Workflow Advisor	2-2
	Speedgoat Simulink-Programmable I/O Module Support	2-2
	Prepare for FPGA Workflow	2-2
	Interrupt Configuration	2-4

3	Functional Mock-up Units and Simulink Real-Time	
	Apply Functional Mock-up Units by Using Simulink Real-Time	3-2
	Compile Source Code for Functional Mock-up Units	3-3
	Implement the FMU Block in Model	3-3
	Compile FMU File That Contains Source Code	3-3

4	Third-Party Calibration Support	
	Calibrate Real-Time Application	4-2
	Prepare ASAP2 Data Description File	4-3
	Initial Setup	4-6
	Set Up Parameters	4-7

Set Up Signals	4-7
Set Up Lookup Tables	4-8
Generate Data Description File	4-9
Calibrate Parameters with Vector CANape	4-10
Prepare Project	4-10
Prepare Device	4-10
Configure Signals and Parameters	4-10
Measure Signals and Calibrate Parameters	4-11
Vector CANape Limitations	4-12
Troubleshoot Vector CANape Operation	4-13
What This Issue Means	4-13
Try This Workaround	4-13
Calibrate Parameters with ETAS Inca	4-14
Prepare Database	4-14
Prepare Project	4-14
Prepare Workspace	4-14
Prepare Experiment	4-14
Configure Signals and Parameters	4-15
Measure Signals and Calibrate Parameters	4-15
ETAS Inca Limitations	4-16
Troubleshoot ETAS Inca Operation	4-17
What This Issue Means	4-17
Try This Workaround	4-17

ASAM XIL API Support

5

Install the Simulink Real-Time Support Package for ASAM XIL	
Standard	5-2
Prerequisites for Using ASAM XIL API	5-2
Classes and Methods of ASAM XIL API	5-4
MAPort Class	5-4
ECUMPort Class	5-5
ECUCPort Class	5-6
Capture Class	5-6
CapturingFactory Class	5-7
CapturingResult Class	5-7
MAPORTFactory Class	5-7
SignalFactory Class	5-8
SignalGeneratoryFactory Class	5-10
SignalGenerator Class	5-10

Real-Time Application Environment

6

Select Default Target Computer	6-2
Select Default Target Computer	6-2
Command-Line Interface and Target Computer	6-2
Targets Object and Target Computers	6-2
Set Up Target Computer Ethernet Connection	6-3
Connect Ethernet Cables	6-3
Configure Ethernet Address	6-3
Related Ethernet Configuration Topics	6-4
Target Computer Update, Reboot, and Startup Application	6-5
Update Software	6-5
Reboot Target Computer	6-5
Select Startup Application	6-5

Signals and Parameters

7

Signal Monitoring Basics	7-3
Monitor Signals by Using Simulink Real-Time Explorer	7-4
Instrument a Stateflow Subsystem	7-5
Animate Stateflow Charts with Simulink External Mode	7-7
Signal Tracing Basics	7-8
Export and Import Signals in Instrument by Using Simulink Real-Time Explorer	7-9
Save Signals to Disk	7-9
Get MATLAB Code for Signals	7-9
Trace Signals by Using Simulink External Mode	7-11
Set Up for External Mode Simulation	7-11
Set Stop Time and Simulate	7-12
Data Logging with Simulation Data Inspector (SDI)	7-14
Parameter Tuning and Data Logging	7-18
Trace or Log Data with the Simulation Data Inspector	7-21
Set Up Model for Logging	7-21

Set Up Simulation Data Inspector	7-21
View Simulation Data	7-22
External Mode Usage	7-25
Signal Logging and Streaming Basics	7-26
How Application is Run Affects Signals Logged	7-26
File Logging and Streaming Workflow	7-26
Tune Parameters by Using Simulink Real-Time Explorer	7-30
Set Up the Simulation Data Inspector	7-30
View Initial Parameter Values	7-31
Modify Parameter Values	7-31
Tune Parameters by Using MATLAB Language	7-33
Access Parameters by Using Application Object	7-33
Tune Parameters by Using Simulink External Mode	7-35
Tune Parameters by Using Block Diagram	7-35
Tune Parameters by Using Hold Updates and Update All Parameters	7-35
Save and Reload Parameters by Using the MATLAB Language	7-37
Save Current Set of Real-Time Application Parameters	7-38
Load Saved Parameters to Real-Time Application	7-38
View or Edit Parameter Values in Parameter Set	7-39
Add or Update Startup Parameter Set for Application	7-40
Tunable Block Parameters and Tunable Global Parameters	7-41
Tunable Parameters	7-41
Inlined Parameters	7-42
Tune Global Parameters by Using External Mode	7-42
Tune Global Parameters by Using Simulink Real-Time Explorer ...	7-42
Tune Global Parameters by Using MATLAB Language	7-42
Tune Inlined Parameters by Using Simulink Real-Time Explorer ..	7-44
Configure Model to Tune Inlined Parameters	7-44
Initial Value	7-45
Updated Value	7-47
Tune Inlined Parameters by Using MATLAB Language	7-48
Tune Inlined Parameter	7-48
Tune Parameter Structures by Using Simulink Real-Time Explorer	7-49
Create Parameter Structure	7-49
Replace Block Parameters with Parameter Structure Fields	7-50
Save and Load Parameter Structure	7-50
Tune Parameters in a Parameter Structure	7-51
Tune Parameter Structures by Using MATLAB Language	7-52
Create Parameter Structure	7-52
Save and Load Parameter Structure	7-53
Replace Block Parameters with Parameter Structure Fields	7-53
Tune Parameters in a Parameter Structure	7-53

Define and Update Inport Data	7-55
Required Files	7-55
Map Inport to Use Square Wave	7-55
Update Inport to Use Sawtooth Wave	7-57
Define and Update Inport Data by Using MATLAB Language	7-60
Required Files	7-60
Map Inport to Use Square Wave	7-60
Update Inport to Use Sawtooth Wave	7-61
Stimulate Root Inport by Using MATLAB Language	7-63
Inport Data Mapping Limitations	7-65
Display and Filter Hierarchical Signals and Parameters	7-66
Hierarchical Display	7-66
Filtered Display	7-67
Sorted Display	7-68
Troubleshoot Signals Not Accessible by Name	7-70
What This Issue Means	7-70
Try This Workaround	7-70
Troubleshoot Parameters Not Accessible by Name	7-72
What This Issue Means	7-72
Try This Workaround	7-72
Troubleshoot Instance-Specific Parameters Not Saved	7-73
What This Issue Means	7-73
Try This Workaround	7-73
Internationalization Issues	7-74

Execution Modes

8

Execution Modes	8-2
------------------------------	-----

Real-Time Application Execution

9

Working with the Target Computer Command Line

Control Real-Time Application at Target Computer Command Line	9-2
--	-----

Execute Target Computer RTOS Commands at Target Computer Command Line	9-3
--	------------

Tuning Performance

10	<table> <tr> <td>CPU Overload</td> <td>10-2</td> </tr> <tr> <td>Monitor CPU Overload Rate</td> <td>10-3</td> </tr> <tr> <td>Execution Profiling for Real-Time Applications</td> <td>10-7</td> </tr> <tr> <td>Reduce Build Time for Simulink Real-Time Referenced Models ..</td> <td>10-13</td> </tr> </table>	CPU Overload	10-2	Monitor CPU Overload Rate	10-3	Execution Profiling for Real-Time Applications	10-7	Reduce Build Time for Simulink Real-Time Referenced Models ..	10-13
CPU Overload	10-2								
Monitor CPU Overload Rate	10-3								
Execution Profiling for Real-Time Applications	10-7								
Reduce Build Time for Simulink Real-Time Referenced Models ..	10-13								

External Code Integration

11	<table> <tr> <td>External Code Integration of Libraries and C/C++ Code with Simulink Real-Time Models</td> <td>11-2</td> </tr> <tr> <td> Considerations for Integrating Third-Party Libraries and External Code into Simulink Real-Time</td> <td>11-2</td> </tr> <tr> <td> Value of Upgrading Your C/C++ Code for Integration into Simulink Real-Time</td> <td>11-2</td> </tr> <tr> <td> Approaches for C/C++ Code Integration into Simulink Real-Time</td> <td>11-3</td> </tr> <tr> <td> Build Libraries from Source Code for Simulink Real-Time</td> <td>11-3</td> </tr> <tr> <td> External Code Integration for S-Functions and Simulink Real-Time</td> <td>11-4</td> </tr> <tr> <td> Hello World! Example External Code Integration for Simulink Real-Time</td> <td>11-5</td> </tr> <tr> <td> Additional C/C++ Project for Simulink Real-Time</td> <td>11-7</td> </tr> </table>	External Code Integration of Libraries and C/C++ Code with Simulink Real-Time Models	11-2	Considerations for Integrating Third-Party Libraries and External Code into Simulink Real-Time	11-2	Value of Upgrading Your C/C++ Code for Integration into Simulink Real-Time	11-2	Approaches for C/C++ Code Integration into Simulink Real-Time	11-3	Build Libraries from Source Code for Simulink Real-Time	11-3	External Code Integration for S-Functions and Simulink Real-Time	11-4	Hello World! Example External Code Integration for Simulink Real-Time	11-5	Additional C/C++ Project for Simulink Real-Time	11-7
External Code Integration of Libraries and C/C++ Code with Simulink Real-Time Models	11-2																
Considerations for Integrating Third-Party Libraries and External Code into Simulink Real-Time	11-2																
Value of Upgrading Your C/C++ Code for Integration into Simulink Real-Time	11-2																
Approaches for C/C++ Code Integration into Simulink Real-Time	11-3																
Build Libraries from Source Code for Simulink Real-Time	11-3																
External Code Integration for S-Functions and Simulink Real-Time	11-4																
Hello World! Example External Code Integration for Simulink Real-Time	11-5																
Additional C/C++ Project for Simulink Real-Time	11-7																

Simulation Data Inspector

12	<table> <tr> <td>View Data in the Simulation Data Inspector</td> <td>12-2</td> </tr> <tr> <td> View Logged Data</td> <td>12-2</td> </tr> <tr> <td> Import Data from the Workspace or a File</td> <td>12-3</td> </tr> <tr> <td> View Complex Data</td> <td>12-5</td> </tr> <tr> <td> View String Data</td> <td>12-6</td> </tr> <tr> <td> View Frame-Based Data</td> <td>12-9</td> </tr> <tr> <td> View Event-Based Data</td> <td>12-9</td> </tr> <tr> <td>Import Data from a CSV File into the Simulation Data Inspector</td> <td>12-11</td> </tr> <tr> <td> Basic File Format</td> <td>12-11</td> </tr> </table>	View Data in the Simulation Data Inspector	12-2	View Logged Data	12-2	Import Data from the Workspace or a File	12-3	View Complex Data	12-5	View String Data	12-6	View Frame-Based Data	12-9	View Event-Based Data	12-9	Import Data from a CSV File into the Simulation Data Inspector	12-11	Basic File Format	12-11
View Data in the Simulation Data Inspector	12-2																		
View Logged Data	12-2																		
Import Data from the Workspace or a File	12-3																		
View Complex Data	12-5																		
View String Data	12-6																		
View Frame-Based Data	12-9																		
View Event-Based Data	12-9																		
Import Data from a CSV File into the Simulation Data Inspector	12-11																		
Basic File Format	12-11																		

Multiple Time Vectors	12-11
Signal Metadata	12-12
Import Data from a CSV File	12-13
Microsoft Excel Import, Export, and Logging Format	12-16
Basic File Format	12-16
Multiple Time Vectors	12-16
Signal Metadata	12-17
User-Defined Data Types	12-19
Complex, Multidimensional, and Bus Signals	12-21
Function-Call Signals	12-21
Simulation Parameters	12-22
Multiple Runs	12-22
Configure the Simulation Data Inspector	12-24
Logged Data Size and Location	12-24
Archive Behavior and Run Limit	12-25
Incoming Run Names and Location	12-26
Signal Metadata to Display	12-27
Signal Selection on the Inspect Pane	12-28
How Signals Are Aligned for Comparison	12-28
Colors Used to Display Comparison Results	12-29
Signal Grouping	12-29
Data to Stream from Parallel Simulations	12-30
Options for Saving and Loading Session Files	12-30
Signal Display Units	12-30
How the Simulation Data Inspector Compares Data	12-32
Signal Alignment	12-32
Synchronization	12-33
Interpolation	12-34
Tolerance Specification	12-34
Limitations	12-36
Save and Share Simulation Data Inspector Data and Views	12-37
Save and Load Simulation Data Inspector Sessions	12-37
Share Simulation Data Inspector Views	12-38
Share Simulation Data Inspector Plots	12-38
Create Simulation Data Inspector Report	12-39
Export Data to the Workspace or a File	12-40
Export Video Signal to an MP4 File	12-41
Inspect and Compare Data Programmatically	12-43
Create a Run and View the Data	12-43
Compare Two Signals in the Same Run	12-44
Compare Runs with Global Tolerance	12-45
Analyze Simulation Data Using Signal Tolerances	12-46
Limit the Size of Logged Data	12-49
Limit the Number of Runs Retained in the Simulation Data Inspector Archive	12-49
Specify a Minimum Disk Space Requirement or Maximum Size for Logged Data	12-49
View Data Only During Simulation	12-50
Reduce the Number of Data Points Logged from Simulation	12-50

Execution with MATLAB Scripts

Real-Time Application Objects and Options in the MATLAB Interface

13

Target and Application Objects	13-2
Control Real-Time Application by Using Objects	13-2
Use Real-Time Application Object Functions	13-3

Simulink Real-Time Instruments and Instrument Panel Apps

14

Add Instruments to Real-Time Application from Simulink Model	14-2
Instrumentation Apps for Real-Time Applications	14-5
Create App Designer Instrument Panels by Using App Generator	14-6
Tip About MLDATX and SLX Files	14-8
Create App Designer Instrument Panels by Using Simulink Real-Time Components	14-9
Create Standalone Instrument Panel App by Using Application Compiler	14-14

Automated Test with Simulink Test

15

Test Real-Time Application in Simulink Test	15-2
--	-------------

Parameter Tuning and Data Logging	16-2
Tune Decimation for File Log Data Without Model Rebuild	16-5
Concurrent Execution on Simulink Real-Time	16-11
Add App Designer App to Inverted Pendulum Model	16-18
Basic App Designer App for Real-Time Application Interface	16-22
Create Instrument Panel for Testing Highway Lane Following Controller	16-25
Connect Triggered Subsystem by Using Thread Trigger	16-32
EtherCAT Protocol with Beckhoff Analog IO Slave Devices EL3062 and EL4002	16-33
EtherCAT Protocol with Beckhoff Digital IO Slave Devices EL1004 and EL2004	16-38
EtherCAT Protocol Motor Velocity Control with Accelnet Drive	16-43
EtherCAT Protocol Motor Position Control with Accelnet Drive	16-48
Generate ENI Files for EtherCAT Devices	16-53
EtherCAT Protocol Detect Network Failure and Reset	16-59
EtherCAT Protocol Sequenced Writing SoE Slave Configuration Variables	16-64
EtherCAT Protocol Sequenced Writing CoE Slave Configuration Variables	16-69
Simple ASCII Encoding/Decoding Loopback Test (with Baseboard Blocks)	16-74
ASCII Encoding/Decoding Loopback Test	16-75
ASCII Encoding/Decoding Loopback Test (with Baseboard Blocks)	16-76
ASCII Encoding/Decoding Resync Loopback Test	16-78

ASCII Encoding/Decoding Resync Loopback Test (with Baseboard Blocks)	16-79
Binary Encoding/Decoding Loopback Test	16-81
Binary Encoding/Decoding Loopback Test (with Baseboard Blocks)	16-82
Binary Encoding/Decoding Resync Loopback Test	16-84
Binary Encoding/Decoding Resync Loopback Test (with Baseboard Blocks)	16-85
Target to Development Computer Communication by Using TCP	16-87
Target to Host Transmission by Using UDP	16-90
Apply 802.1Q VLAN Tag by Using Ethernet Send and Receive Blocks	16-94
Apply Simulink Real-Time Model Template to Create Real-Time Application	16-99
Insert Event into Execution Profiling Stream	16-101
Control Real-Time Application by Using C# Code	16-103
Run Real-Time Application by Using Python Script	16-105
Hello World! Example External Code Integration for Simulink Real-Time	16-109
Control Color of Lamp on Instrument Panel	16-112
Configure Input and Output Ports for Bit Packing and Unpacking	16-115
Run Real-Time Simulation of Permanent Magnet Synchronous Motor	16-118
Apply Persistent Variables in Real-Time Applications	16-123
Communicate with Data Distribution Service (DDS) Middleware	16-126

Troubleshooting

17

Troubleshooting Basics	17-2
-------------------------------------	-------------

Troubleshoot Missing Real-Time Tab	17-4
What This Issue Means	17-4
Try This Workaround	17-4
Troubleshoot Communication Failure Through Firewall (Windows) ...	17-5
What This Issue Means	17-5
Try These Workarounds	17-6
Troubleshoot Cannot Load Shared Object on Target Computer	17-13
What This Issue Means	17-13
Try This Workaround	17-13
Troubleshoot Signal Data Logging from Nonvirtual Bus, Fixed-Point, and Multidimensional Signals	17-15
What This Issue Means	17-15
Try This Workaround	17-15
Troubleshoot Signal Data Logging from Inport in Referenced Model .	17-17
What This Issue Means	17-17
Try This Workaround	17-18
Troubleshoot Signal Data Logging from Inport in Referenced Model in Test Harness	17-19
What This Issue Means	17-19
Try This Workaround	17-19
Troubleshoot Signal Data Logging from Send and Receive Blocks	17-21
What This Issue Means	17-21
Try This Workaround	17-21
Troubleshoot Signals for Streaming or File Logging	17-22
What This Issue Means	17-22
Try These Workarounds	17-22
Troubleshoot Folder Names with Spaces or Special Characters Halt Model Builds	17-23
What This Issue Means	17-23
Try This Workaround	17-23
Troubleshoot Model Links to Static Libraries or Shared Objects	17-24
What This Issue Means	17-24
Try This Workaround	17-24
Troubleshoot Build Error for Accelerator Mode	17-26
What This Issue Means	17-26
Try This Workaround	17-26
Troubleshoot Long Build Times for Real-Time Application	17-27
What This Issue Means	17-27
Try This Workaround	17-27
Troubleshoot Working with Persistent Variables	17-29
What This Issue Means	17-29
Try This Workaround	17-29

Troubleshoot Unsatisfactory Real-Time Performance	17-30
What This Issue Means	17-30
Try This Workaround	17-30
Troubleshoot Overloaded CPU from Executing Real-Time Application	17-32
What This Issue Means	17-32
Try This Workaround	17-32
Troubleshoot Gaps in Streamed Data	17-34
What This Issue Means	17-34
Try This Workaround	17-34
Find Simulink Real-Time Support	17-35
Install Simulink Real-Time Software Updates	17-36

Introduction

Simulink Real-Time Product Description

Perform rapid control prototyping and hardware-in-the-loop testing

Simulink Real-Time and Speedgoat® take you from simulation to rapid control prototyping (RCP) and hardware-in-the-loop (HIL) testing in a single click. The products connect to electronic control units and physical systems with MATLAB® and Simulink.

You can create, control, and instrument real-time applications that run on Speedgoat real-time target computers directly from your Simulink model or with the MATLAB API and App Designer. You can simulate and test control designs and the dynamics of electric motors, electric vehicles and powertrains, wind turbines, power converters, battery management systems, robots and manipulators, autonomous systems, and other devices.

Speedgoat Target Computers and I/O Hardware

Speedgoat target computers are real-time computers fitted with a set of I/O hardware, Simulink programmable FPGAs, and communication protocol support. Speedgoat target computers are optimized for use with Simulink Real-Time and fully support the HDL Coder™ workflow.

Connect a development computer to a Speedgoat target computer that meets your requirements: form factor, performance, I/O interface, and protocol interface. Speedgoat target computer systems come with:

- I/O and protocol interfaces, an Intel® CPU, and optional FPGA hardware, configured and ready to use
- I/O cables, terminal boards, Simulink driver blocks, documentation, and a loopback wiring harness that facilitates acceptance testing for each I/O module
- The Simulink Real-Time RTOS preinstalled on the target computer

Hardware-In-the-Loop (HIL) Simulators and Rugged Units for Controls (RCP), DSP, and Vision Prototyping



Model Architectures

FPGA Models

- “Speedgoat FPGA Support with HDL Workflow Advisor” on page 2-2
- “Interrupt Configuration” on page 2-4

Speedgoat FPGA Support with HDL Workflow Advisor

Use Simulink Real-Time and HDL Coder to implement Simulink algorithms and configure I/O functionality on Speedgoat Simulink-Programmable I/O modules. For an example that shows the development workflow for FPGA I/O modules, see “FPGA Programming and Configuration on Speedgoat Simulink-Programmable I/O Modules” (HDL Coder).

When you open the HDL Workflow Advisor in HDL Coder and run the Simulink Real-Time FPGA I/O workflow, you generate a Simulink Real-Time interface subsystem. The subsystem mask controls the block parameters. Do not edit the parameters directly. The FPGA I/O board block descriptions are for informational purposes only.

Speedgoat Simulink-Programmable I/O Module Support

Speedgoat Simulink-Programmable I/O modules are part of Speedgoat target computer systems. To run the Simulink Real-Time FPGA I/O workflow, install the Speedgoat I/O Blockset and the Speedgoat HDL Coder Integration Packages. You can then choose the **Target platform** and run the workflow to generate a Simulink Real-Time interface subsystem. To see the documentation for the integration packages, enter this command at the MATLAB command prompt.

```
speedgoat.hdlc.doc
```

To learn about	See links
The integration packages and how you can install them.	See Speedgoat - HDL Coder Integration Packages.
Speedgoat I/O modules that are supported with the HDL Workflow Advisor.	See Speedgoat Real-Time FPGA Application Support from HDL Coder.
Applications and use cases	See Common Use Cases and Applications.
Supported interfaces for various types of I/O connectivity and protocols as well as fundamental functionality such as PCIe read/write and DMA.	See Supported Interfaces.
Provided examples for all supported I/O modules and functionality	See Speedgoat I/O Examples.

Prepare for FPGA Workflow

To work with FPGAs in the Simulink Real-Time environment, install:

- HDL Coder and Simulink Real-Time.
- Xilinx® design tools with specific tool and version listed in “HDL Language Support and Supported Third-Party Tools and Hardware” (HDL Coder). You must also set up the path to the tool by using the `hdlsetuptoolpath` function.
- Speedgoat I/O Blockset and the Speedgoat HDL Coder Integration Packages.
- Speedgoat FPGA I/O module in the Speedgoat target machine.

You can use the workflow in HDL Coder to generate HDL code for your FPGA target device.

See Also

Related Examples

- “FPGA Programming and Configuration on Speedgoat Simulink-Programmable I/O Modules” (HDL Coder)

More About

- “HDL Language Support and Supported Third-Party Tools and Hardware” (HDL Coder)
- “Tool Setup” (HDL Coder)

External Websites

- www.speedgoat.com

Interrupt Configuration

Simulink Real-Time software schedules the real-time application by using either the internal timer of the Speedgoat target machine (default) or an interrupt from an I/O board. You can use your Speedgoat FPGA board to generate an interrupt. You can:

- Schedule execution of the real-time application based on this interrupt (synchronous execution). For this method, you must generate the interrupt periodically.
- Execute a designated subsystem in your real-time application (asynchronous execution).

To use FPGA-based interrupts, set up and configure the FPGA domain and Simulink Real-Time domain models. For more information, see “Speedgoat Target Computers and I/O Hardware” on page 1-3.

See Also

Functional Mock-up Units and Simulink Real-Time

- “Apply Functional Mock-up Units by Using Simulink Real-Time” on page 3-2
- “Compile Source Code for Functional Mock-up Units” on page 3-3

Apply Functional Mock-up Units by Using Simulink Real-Time

After you create a model that contains an FMU block, you can build and download the model to a target computer by using Simulink Real-Time. These limitations apply:

- Simulink Real-Time supports FMU blocks for Co-Simulation mode. Simulink Real-Time does not support FMU blocks for Model Exchange mode.
- Simulink Real-Time does not support FMU blocks within a referenced model. FMU blocks must be at the top level of the model.
- Simulink Real-Time generates a mask dialog box that contains both numeric-valued and string-valued parameters. Simulink Real-Time generates code for only numeric-valued parameters.

To convert a Simulink model that contains FMU blocks to a Simulink Real-Time model, set the model configuration parameters to values compatible with real-time execution:

- In the **Code Generation** pane, set **System target file** to `slrealtime.tlc`.
- In the **Solver** pane:
 - Set **Type** to Fixed-step.
 - Set **Fixed-step size** to a step size compatible with the real-time requirements of your model.
- Generate a shared object SO file by using the QNX® Neutrino® tools for the FMU. For more information, see `slrealtime.fmu.compileFMUSources`.

You can then build and download the model to a target computer and run the real-time application. This process loads the required FMU binary files on the target computer. For more information about creating the FMU files, see “Compile Source Code for Functional Mock-up Units” on page 3-3.

To open an example model that contains FMU blocks running in Simulink Real-Time, in the MATLAB Command Window, type:

```
open_system(fullfile(matlabroot, 'toolbox', 'slrealtime', ...  
'examples', 'slrt_ex_vanderpol'))
```

See Also

FMU

More About

- “Import FMUs”

External Websites

- <https://fmi-standard.org/>

Compile Source Code for Functional Mock-up Units

When you build a model that includes FMU blocks, you must compile the FMU source code by using the QNX Neutrino compiler `qcc` or `q++`. This compiler creates shared object SO files that you include in the FMU. This process makes sure that the FMU contains the code to run on a Simulink Real-Time target computer. For more information, see “Apply Functional Mock-up Units by Using Simulink Real-Time” on page 3-2.

Implement the FMU Block in Model

To implement the `vanDerPol` block in the Simulink model by using the FMU, specify the FMU name for the block. Open the model `slrt_ex_vanderpol`, double-click the FMU block `vanDerPol`, and select the `vanDerPol.fmu` file for the FMU name block parameter.

Build the model, load the real-time application on the target computer, and run the real-time application.

Compile FMU File That Contains Source Code

The `slrealtime.fmu.compileFMUSources` function compiles an FMU file that contains source code. The process outputs an FMU file and Simulink Real-Time binary file in the same folder as the input FMU file and appends an `_slrt` suffix to the output file name. This example selects an FMU file to compile and overwrites previous compiler output.

```
% create variable to provide path and file name
my_file = ['C:\work\my_fmu_work\', 'vanDerPol.fmu']
% compile the FMU file and overwrite previous output
slrealtime.fmu.compileFMUSources(my_file, 'overwriteBinary', true)
```

See Also

FMU | `slrealtime.fmu.compileFMUSources`

More About

- “Import FMUs”

External Websites

- <https://fmi-standard.org/>

Third-Party Calibration Support

- “Calibrate Real-Time Application” on page 4-2
- “Prepare ASAP2 Data Description File” on page 4-3
- “Calibrate Parameters with Vector CANape” on page 4-10
- “Vector CANape Limitations” on page 4-12
- “Troubleshoot Vector CANape Operation” on page 4-13
- “Calibrate Parameters with ETAS Inca” on page 4-14
- “ETAS Inca Limitations” on page 4-16
- “Troubleshoot ETAS Inca Operation” on page 4-17

Calibrate Real-Time Application

Simulink Real-Time supports interaction with third-party calibration tools such as Vector CANape (www.vector.com) and ETAS Inca (www.etas.com). Use these tools for:

- Parameter display and tuning
- Calibration data saving, restoring, and swapping by page
- Signal value streaming

These tools run in XCP client mode. Simulink Real-Time emulates an electronic control unit (ECU) operating in XCP server mode. To enable a real-time application to work with the third-party software:

- Configure the third-party software to communicate with the real-time application as an ECU.
- Provide a standard TCP/IP physical layer between the development and target computers. Simulink Real-Time supports third-party calibration software only through UDP protocol.
- Generate a real-time application with signal and parameter attributes that are consistent with A2L (ASAP2) file generation. See “Configure Model Data Elements for ASAP2 File Generation”.
- Use the build process to generate `model.a2l` (ASAP2) files that the software can load into its database. The generated file contains signal and parameter access information for the real-time application and XCP-related sections and memory addresses.

If your model includes referenced models, the build creates a `model.a2l` file for the real-time application and separate `refmodel.a2l` files for each referenced model.

Note You cannot configure third-party software for calibration with only the A2L files that Simulink Coder™ generates. These files do not contain XCP-related sections and memory addresses. Simulink Real-Time adds this information during the build process.

See Also

More About

- “Configure Model Data Elements for ASAP2 File Generation”
- “Prepare ASAP2 Data Description File” on page 4-3
- “Calibrate Parameters with Vector CANape” on page 4-10
- “Calibrate Parameters with ETAS Inca” on page 4-14
- “XCP Client Mode”

External Websites

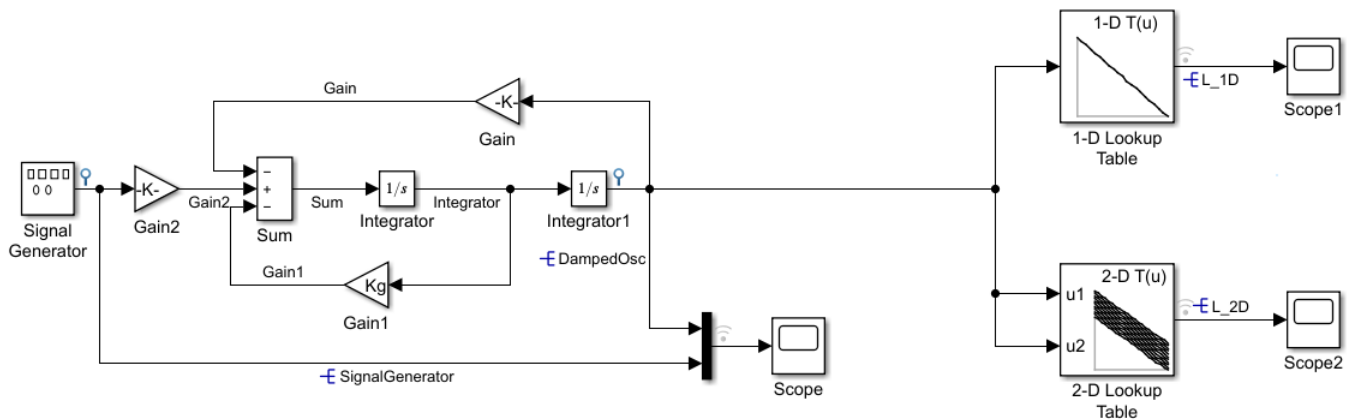
- www.vector.com
- www.etas.com

Prepare ASAP2 Data Description File

This example shows how to configure a Simulink Real-Time model so that the build generates an ASAP2 (A2L) data description file for the real-time application. The real-time application models a damped oscillator that feeds into 1-D and 2-D lookup tables, which invert and rescale the input waveform.

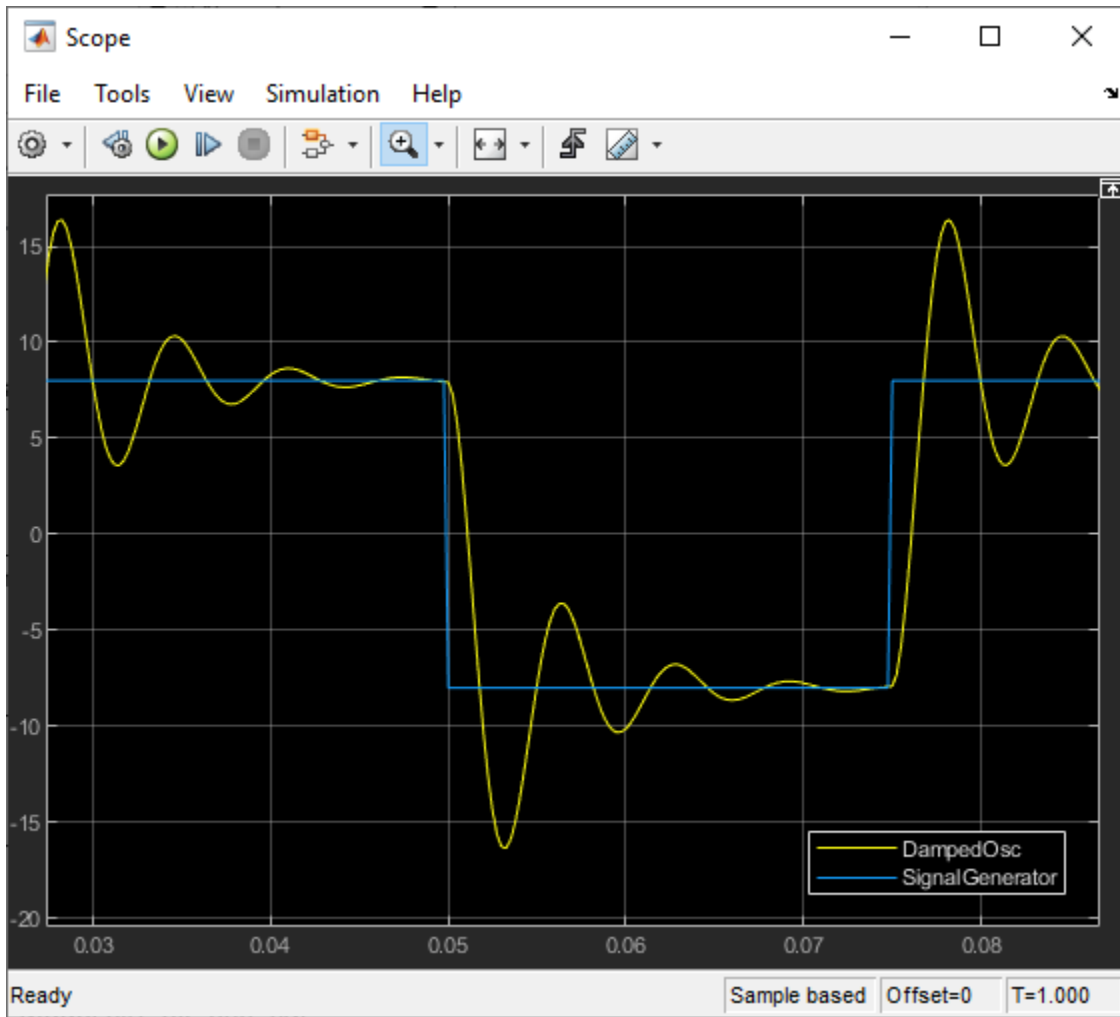
This example uses model `slrt_ex_osc_cal`. To open the model, in the MATLAB Command Window, type:

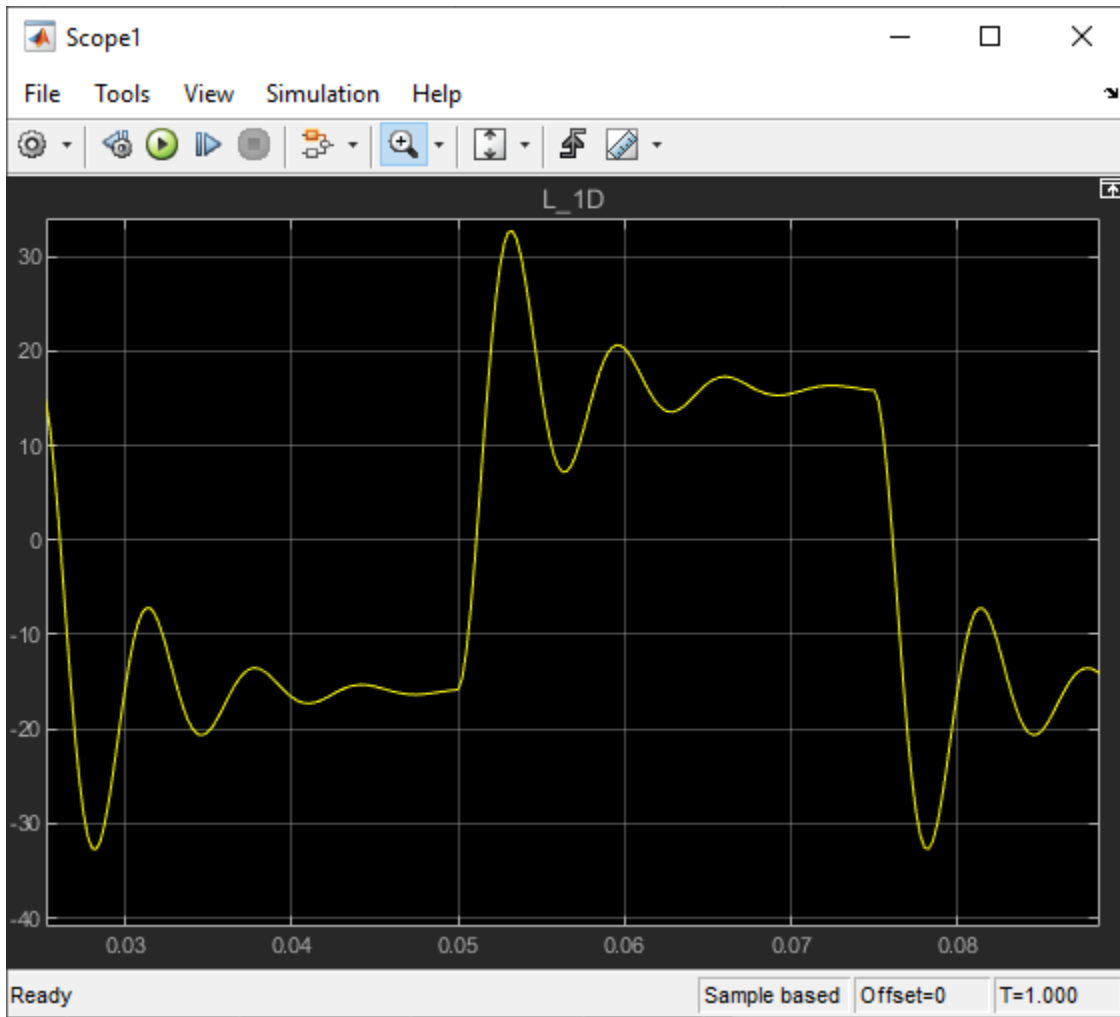
```
open_system(fullfile(matlabroot, 'toolbox', 'slrealtime', ...
    'examples', 'slrt_ex_osc_cal'))
```

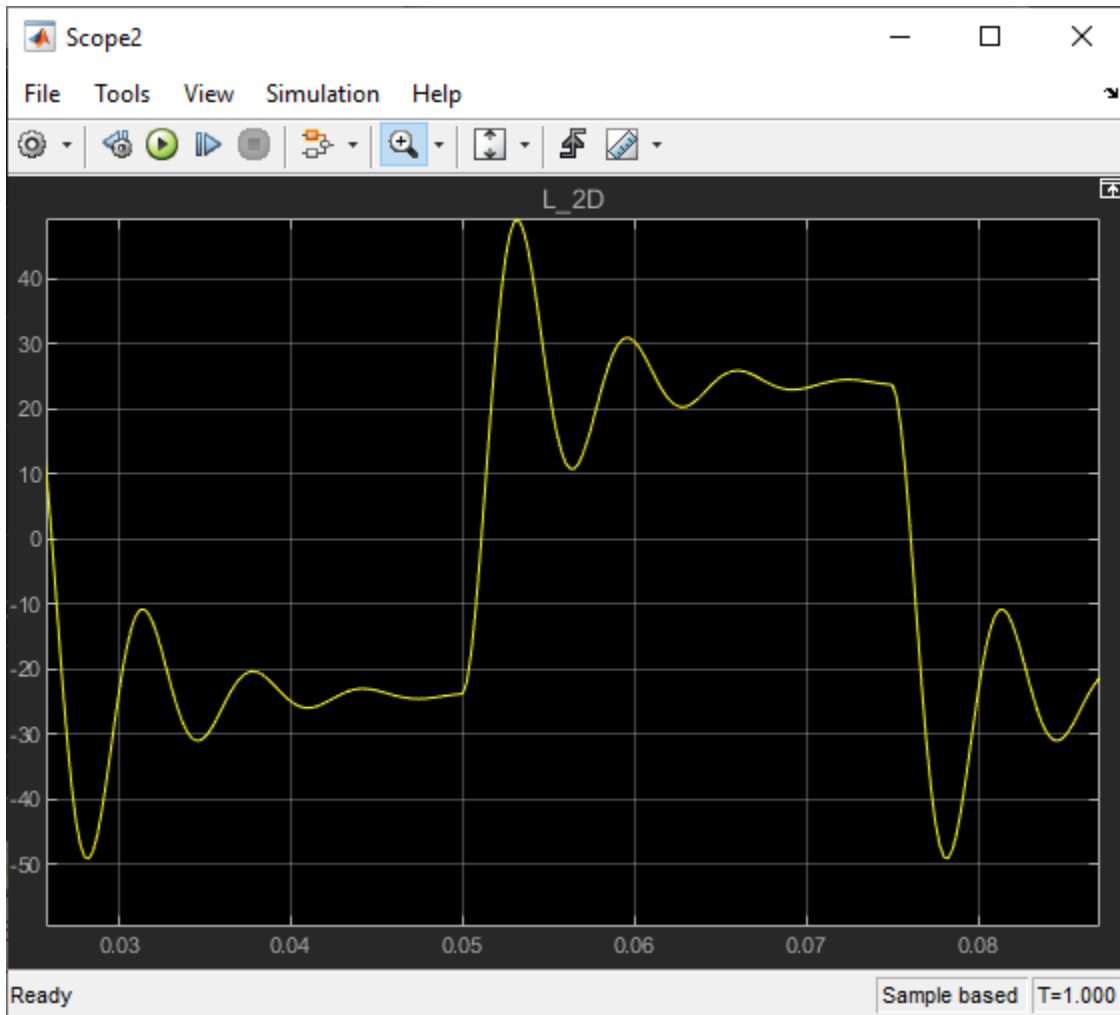


Model `slrt_ex_osc_cal`
Simulink Real-Time example model

Copyright 1999-2020 The MathWorks, Inc.







Calibration of parameters reduces the ringing in signals DampedOsc, L_1D, and L_2D.

Initial Setup

Open the model and check for model data.

1 Open slrt_ex_osc_cal

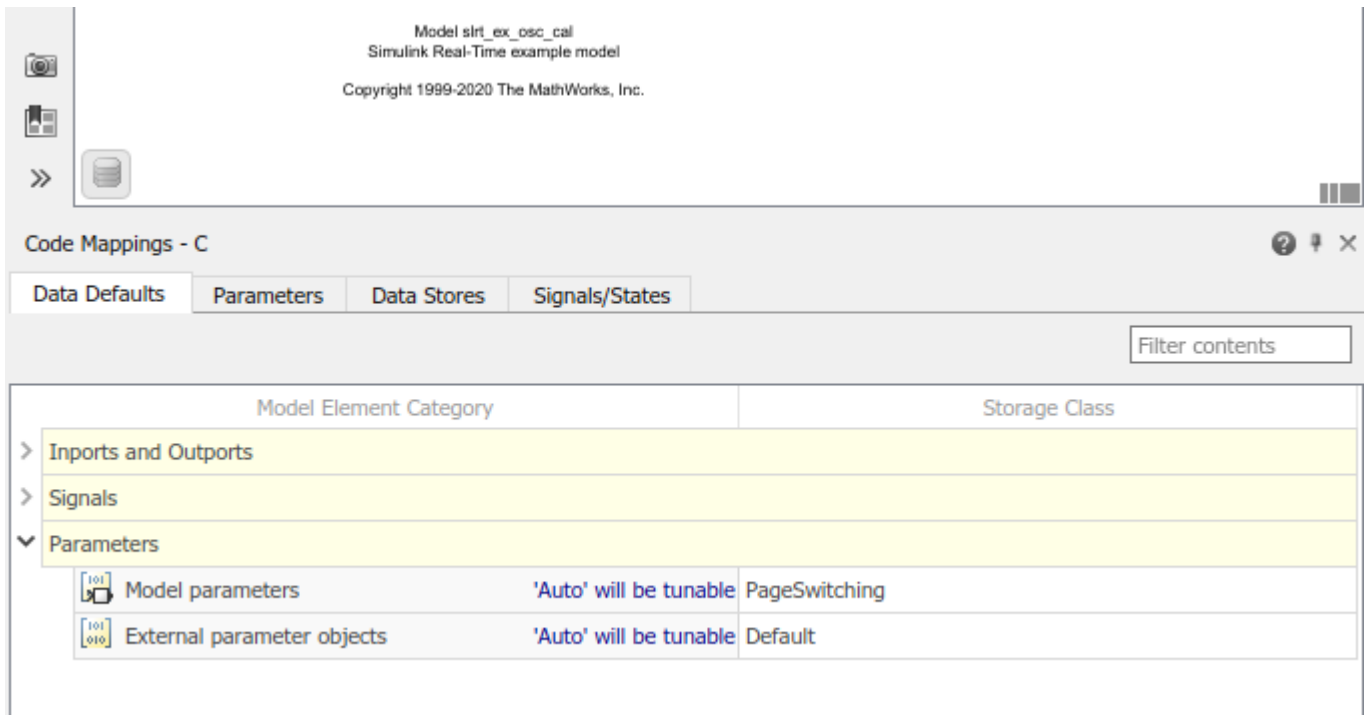
```
open_system(fullfile(matlabroot, 'toolbox', 'slrealtime', ...
    'examples', 'slrt_ex_osc_cal'))
```

The **Model Workspace** variables contain these functions:

- Kg — Parameter object for the Gain1 block
- DampedOsc, SignalGenerator, L_1D, L_2D — Signal objects for output signals
- LUT_1D_obj, LUT_2D_obj — 1-D and 2-D lookup tables data respectively
- SignalGenerator — Test input data

2 Set the **Default parameter behavior** configuration parameter to Tunable.



- 3 In the **Code Mappings Editor - C** in **Data Defaults**, specify the storage class as PageSwitching for **Model parameters** under **Parameters**.



Note The model default setting for parameters sets the storage class as PageSwitching.

Set Up Parameters


Set up parameter tuning by using Simulink parameter objects.

- 1 In `slrt_ex_osc_cal`, on the **Modeling** tab, click **Design > Model Explorer** .
- 2 Select **Model Workspace** in the **Model Hierarchy** pane.
- 3 Make sure that the Kg parameter object exists and has these properties:
 - **Value** — 400
 - **Data type** — double
- 4 If the parameter object does not exist, add it. On the toolbar, click the **Add Simulink Parameter** button .
- 5 Open `slrt_ex_osc_cal/Gain1`.
- 6 Make sure that you have set the **Gain** value to the parameter object Kg.

Set Up Signals

As a best practice, set up signal viewing by using Simulink signal objects.

- 1 In `slrt_ex_osc_cal`, on the **Modeling** tab, click **Design > Model Explorer** .

- 2 Select **Model Workspace** in the **Model Hierarchy** pane.
- 3 Make sure that the `DampedOsc` signal object exists and has these properties:
 - **Minimum** — -10
 - **Maximum** — 10
 - **Data type** — double
- 4 Make sure that the `SignalGenerator` signal object exists and has these properties:
 - **Minimum** — -10
 - **Maximum** — 10
 - **Data type** — double
- 5 Make sure that the `L_1D` signal object exists and has these properties:
 - **Minimum** — -15
 - **Maximum** — 15
 - **Data type** — double
- 6 Make sure that the `L_2D` signal object exists and has these properties:
 - **Minimum** — -15
 - **Maximum** — 15
 - **Data type** — double
- 7 If a signal does not exist, add it. On the toolbar, click the **Add Simulink Signal** button .
- 8 For each signal, open its Properties dialog box.
- 9 Make sure that you selected the **Signal name must resolve to Simulink signal object** and the **Test point** check boxes.

Set Up Lookup Tables

The example model contains 1-D and 2-D lookup tables.

- 1 Open the block parameters for the 1-D Lookup Table block.
- 2 In the **Table and Breakpoints** pane, verify these settings:
 - **Number of table dimensions** — 1
 - **Data specification** — Lookup table object
 - **Name** — LUT_1D_obj
- 3 Open the block parameters for the 2-D Lookup Table block.
- 4 In the **Table and Breakpoints** pane, check these settings:
 - **Number of table dimensions** — 2
 - **Data specification** — Lookup table object
 - **Name** — LUT_2D_obj

To view the contents of the lookup tables, click **Edit table and breakpoints**, and then click **Plot > Mesh**.

Generate Data Description File

- 1 On the **REAL-TIME** tab, select **RUN ON TARGET > Build Application**. The build produces a file named `slrt_ex_osc_cal_slrt.mdatx` in the working folder containing A2L file.
- 2 To retrieve the A2L file and update target IP address in the A2L file, use `extractASAP2` command.
- 3 Or, on the **C CODE** tab, select **Share > Generate Calibration Files**. Use the tool to generate the required version of ASAP2 file. For more information about using the tool, see “Generate ASAP2 and CDF Calibration Files”.
- 4 Connect to the target by using a third-party calibration tool.

See Also

n-D Lookup Table | `coder.asap2.export`

More About

- “Generate ASAP2 and CDF Calibration Files”
- “Calibrate Parameters with Vector CANape” on page 4-10
- “Calibrate Parameters with ETAS Inca” on page 4-14

External Websites

- www.vector.com
- www.etas.com

Calibrate Parameters with Vector CANape

This example shows how to view signals and tune parameters by using Vector CANape. You must have already completed the steps in “Prepare ASAP2 Data Description File” on page 4-3.

You also must be familiar with the Vector CANape user interface. For information about the user interface, see the vendor documentation (www.vector.com).

Prepare Project

- 1 Build and download the real-time application `slrt_ex_osc_cal`.
- 2 Start the real-time application by selecting **REAL-TIME > RUN ON TARGET > Start Application**.
- 3 Disconnect the connection from MATLAB:

```
tg = slrealtime
disconnect(tg)
```

You can now connect to third-party calibration tools.

- 4 Open Vector CANape.
- 5 Create a Vector CANape project with project name `slrt_ex_osc_cal`.

Accept the default folder.

Prepare Device

- 1 From the extracted `slrt_ex_osc_cal.a2l`, create an XCP device named `slrt_ex_osc_cal_slrt`.

Do not configure dataset management.

- 2 Select your local computer Ethernet adapter as the Ethernet channel.
- 3 Accept the remaining defaults.
- 4 Upload data from the device.

Configure Signals and Parameters

- 1 Open device `slrt_ex_osc_cal_slrt`, and then open `slrt_ex_osc_cal.a2l`.
- 2 Add signals `DampedOsc`, `SignalGenerator`, `L_1D`, and `L_2D` in separate display windows.
- 3 To make the waveform easier to evaluate, set the time and *y*-axis scaling.

For example, try the following settings for `DampedOsc`:

- *y*-axis min home value — -25
- *y*-axis max home value — 25
- Min home time-axis value — 0 s
- Max home time-axis value — 0.1 s
- Time duration — 0.1 s

- 4 Open the measurement list.
- 5 To set the required sample time for a signal, open the measurement properties for the signal. Select the required sample time from the measurement mode list.

The default sample time is the base sample time.

- 6 Add a graphic control on parameter Kg.

Measure Signals and Calibrate Parameters

- 1 Start the Vector CANape measurement.
- 2 To shorten the ring time on DampedOsc, L_1D, and L_2D, set parameter Kg to 800.
- 3 As required, toggle between calibration RAM active and inactive.

When using the **Run on Target** button on the **Simulink Editor Real-Time** tab to run the simulation, there is a time lag of a couple of seconds between the start of the real-time application on the target computer and the connect model operation on the development computer. If you are examining signals at or very close to application start, consider using the step-by-step approach to connect model and then using an SSH connection (for example, using PuTTY) start the real-time application. For more information, see “Execute Real-Time Application in Simulink External Mode by Using Step-by-Step Commands” and “Execute Target Computer RTOS Commands at Target Computer Command Line” on page 9-3.

See Also

More About

- “Prepare ASAP2 Data Description File” on page 4-3
- “Vector CANape Limitations” on page 4-12
- “Troubleshoot Vector CANape Operation” on page 4-13

External Websites

- www.vector.com

Vector CANape Limitations

For Vector CANape, the Simulink Real-Time software does not support:

- Connecting MATLAB to the target computer while using Vector CANape.
- Loading, starting, or stopping the real-time application by using Vector CANape commands.

To load, start, or stop the real-time application on the target computer, use the target computer command-line interface. For example:

```
slrealtime load --AppName app_name
```

```
slrealtime start
```

```
slrealtime stop
```

For more information, see “Target Computer Command-Line Interface”.

- Vector CANape flash programming.
- Multiple simultaneous Vector CANape connections to a single target computer.

Event mode data acquisition has the following limitations:

- Every piece of data that the Simulink Real-Time software adds to the event list slows the real-time application. The amount of data that you can observe depends on the model sample time and the speed of the target computer. It is possible to overload the target computer CPU to where data integrity is reduced.
- You can trace only signals and scalar parameters. You cannot trace vector parameters.

Troubleshoot Vector CANape Operation

My third-party calibration tool (Vector CANape) is not working with the real-time application.

What This Issue Means

You can use the Vector CANape tool to view signals and tune parameters in the real-time application. For more information, see the steps in “Prepare ASAP2 Data Description File” on page 4-3. In addition to the limitations listed in “Vector CANape Limitations” on page 4-12, there are various issues that can prevent the operation of this tool.

Try This Workaround

For Vector CANape tool issues, try these workarounds.

Simulation Data Inspector in Use

Simulation Data Inspector and the third-party calibration tools (Vector CANape and ETAS Inca) are mutually exclusive. If you use the Simulation Data Inspector to view signal data, you cannot use the calibration tools. If you use the calibration tools, you cannot use the Simulation Data Inspector to view signal data.

Client Cannot Connect

Check the IP address of the target computer associated with the model and compare it to the address stored in the ASAP2 file.

ASAP2 File Out of Date

When you rebuild a Simulink Real-Time application, update the ASAP2 file loaded in the calibration tool with the new version of the file. The ASAP2 file is valid only until the next time that you build the application.

See Also

More About

- “Prepare ASAP2 Data Description File” on page 4-3
- “Vector CANape Limitations” on page 4-12

External Websites

- MathWorks Help Center website
- www.vector.com

Calibrate Parameters with ETAS Inca

This example shows how to view signals and tune parameters by using ETAS Inca. You must have already completed the steps in “Prepare ASAP2 Data Description File” on page 4-3.

You also must be familiar with the ETAS Inca user interface. For information about the user interface, see the vendor documentation (www.etas.com).

Prepare Database

- 1 Build and download real-time application `slrt_ex_osc_cal`.
- 2 Start the real-time application by selecting **REAL-TIME > RUN ON TARGET > Start Application**.
- 3 Disconnect the connection from MATLAB:

```
tg = slrealtime
disconnect(tg)
```

You can then connect to third-party calibration tools.

- 4 Open ETAS Inca.
- 5 Add an ETAS Inca database by using the folder named `SLRTDatabase`.
- 6 Add subfolders named `Experiment`, `Project`, and `Workspace`.

Prepare Project

- 1 Under folder `Project`, add an ECU project.
- 2 When prompted, select A2L file `slrt_ex_osc_cal.a2l`, which is extracted from the project file. Ignore the prompt for a HEX file.

If you change and rebuild the real-time application, delete the ECU project and recreate it with the new A2L file.

Prepare Workspace

- 1 Under folder `Workspace`, add workspace `slrt_ex_osc_cal_wksp`.
- 2 Add project `slrt_ex_osc_cal_slrt` to workspace `slrt_ex_osc_cal_wksp`.
- 3 When prompted, add an Ethernet system XCP device to the workspace.
- 4 Configure the XCP device and initialize it. Autoconfigure the ETAS network.
- 5 To upload data from the device hardware, use enhanced operations on memory pages.

Data is uploaded from the real-time application on the target computer.

Prepare Experiment

- 1 Under folder `Experiment`, add experiment `slrt_ex_osc_cal_exp`.
- 2 Add experiment `slrt_ex_osc_cal_exp` to workspace `slrt_ex_osc_cal_wksp`.

Configure Signals and Parameters

- 1 Start experiment `slrt_ex_osc_cal_exp`.
- 2 To create graphic controls for the variables, add variables `Kg`, `DampedOsc`, `SignalGenerator`, `L_1D`, and `L_2D`.
- 3 Add YT oscilloscopes for `DampedOsc`, `SignalGenerator`, `L_1D`, and `L_2D`.
- 4 For each signal, set the sample time to the base sample time of the real-time application (250 μ s).

Measure Signals and Calibrate Parameters

- 1 Start the ETAS Inca measurement.
- 2 To shorten the ring time on `DampedOsc`, `L_1D`, and `L_2D`, set parameter `Kg` to 800.
- 3 As required, toggle between the reference page and the working page.

When using the **Run on Target** button on the **Simulink Editor Real-Time** tab to run the simulation, there is a time lag of a couple of seconds between the start of the real-time application on the target computer and the connect model operation on the development computer. If you are examining signals at or very close to application start, consider using the step-by-step approach to connect model and then using an SSH connection (for example, using PuTTY) start the real-time application. For more information, see “Execute Real-Time Application in Simulink External Mode by Using Step-by-Step Commands” and “Execute Target Computer RTOS Commands at Target Computer Command Line” on page 9-3.

See Also

More About

- “Prepare ASAP2 Data Description File” on page 4-3
- “ETAS Inca Limitations” on page 4-16
- “Troubleshoot ETAS Inca Operation” on page 4-17

External Websites

- www.etas.com

ETAS Inca Limitations

For ETAS Inca, the Simulink Real-Time software does not support:

- Connecting MATLAB to the target computer while using ETAS Inca.
- Loading, starting, or stopping the real-time application by using ETAS Inca commands.

To load, start, or stop the real-time application on the target computer, use the target computer command-line interface. For example:

```
slrealtime load --AppName app_name
```

```
slrealtime start
```

```
slrealtime stop
```

For more information, see “Target Computer Command-Line Interface”.

- ETAS Inca flash programming.
- Multiple simultaneous ETAS Inca connections to a single target computer.
- Tunability of parameters with `ExportedGlobal` storage class when the model has other parameters with `PageSwitching` storage class. As a work around you can:
 - Place all the parameters you want to tune in model workspace. Or
 - Change the default mapping for storage class from `PageSwitching` to `default`. The `PageSwitching` storage class is not used, and the page switching functionality is not available.

Event mode data acquisition has the following limitations:

- Every piece of data that the Simulink Real-Time software adds to the event list slows the real-time application. The amount of data that you can observe depends on the model sample time and the speed of the target computer. It is possible to overload the target computer CPU to where data integrity is reduced.
- You can trace only signals and scalar parameters. You cannot trace vector parameters.

Troubleshoot ETAS Inca Operation

Investigate issues that can occur when ETAS Inca controls a real-time application.

My third-party calibration tool (ETAS Inca) is not working with the real-time application.

What This Issue Means

You can use the ETAS Inca tool to view signals and tune parameters in the real-time application. For more information, see the steps in “Prepare ASAP2 Data Description File” on page 4-3. In addition to the limitations listed in “ETAS Inca Limitations” on page 4-16, there are various issues that can prevent the operation of this tool.

Try This Workaround

For ETAS Inca tool issues, try these workarounds.

Simulation Data Inspector in Use

Simulation Data Inspector and the third-party calibration tools (Vector CANape and ETAS Inca) are mutually exclusive. If you use the Simulation Data Inspector to view signal data, you cannot use the calibration tools. If you use the calibration tools, you cannot use the Simulation Data Inspector to view signal data.

Client Cannot Connect

Check the IP address of the target computer associated with the model and compare it to the address stored in the ASAP2 file.

ASAP2 File Out of Date

When you rebuild a Simulink Real-Time application, update the ASAP2 file loaded in the calibration tool with the new version of the file. The ASAP2 file is valid only until the next time that you build the application.

Cannot Disable Freeze Mode

Remove the dataset file from the target file system and reset the parameters to the original values specified in your model. The dataset file is named `flashdata_model_name.dat`.

Transport Layer Failure

When a transport layer failure occurs, ETAS Inca can display this message:

```
ERROR: Transport Layer Failure, Inconsistent MsgCounter
```

This error appears in ETAS Inca when the incorrect setting is used for 'Counter Consistency Mode'. Make sure that the 'Counter Consistency Mode' is set to 'one counter for all CT0s+DT0s' in the hardware settings for your experiment.

See Also

More About

- “Prepare ASAP2 Data Description File” on page 4-3
- “ETAS Inca Limitations” on page 4-16
- “Troubleshoot ETAS Inca Operation” on page 4-17

External Websites

- MathWorks Help Center website
- www.etas.com

ASAM XIL API Support

- “Install the Simulink Real-Time Support Package for ASAM XIL Standard” on page 5-2
- “Classes and Methods of ASAM XIL API” on page 5-4

Install the Simulink Real-Time Support Package for ASAM XIL Standard

Simulink Real-Time supports a subset of the ASAM XIL API. This API enables you to define ports for test cases. To use these APIs in Simulink Real-Time, install the Simulink Real-Time XIL API support package by using the Add On Explorer. For a list of support ASAM XIL APIs in the support package, see “Classes and Methods of ASAM XIL API” on page 5-4.

Prerequisites for Using ASAM XIL API

To enable support for the ASAM XIL API, install the Simulink Real-Time XIL API support package. This support package implements the ASAM XIL API standard for Simulink Real-Time target computers.

The Simulink Real-Time Support Package for ASAM XIL Standard implements the ASAM XIL standard API for Simulink Real-Time target computers. Using this C# API, you can run real-time hardware-in-the-loop (HIL) tests on a Simulink Real-Time target computer by using test cases created from any test automation software with the XIL framework. Also, you can use the support package to develop a custom XIL test framework for Simulink Real-Time.

- 1 In MATLAB, select **Home > Add-Ons > Get Add-Ons** and install the Simulink Real-Time XIL API support package.
- 2 After support package installation, verify that the manifest file `MathWorksXILServer.imf` that is located under `C:\ProgramData\ASAM\XIL\Implementation` provides the correct Assembly path.
- 3 Register MATLAB as the automation server. Share the MATLAB session. In the MATLAB Command Window, type:

```
comserver('register','User','current');  
enableservice('AutomationServer', true);
```
- 4 Build the model. The real-time application MLDATX file is required for setting up test cases.
- 5 Create a configuration file for the test bench by using the `createPortConfigureFile` function.

After installing the support package, the PDF documentation for the support package is available in the support package folder.

```
cd(fullfile(matlabshared.supportpkg.getSupportPackageRoot,...  
    'toolbox','slrealtime','xil'))
```

See Also

`createPortConfigureFile`

Related Examples

- “Control Real-Time Application by Using C# Code” on page 16-103

More About

- “Classes and Methods of ASAM XIL API” on page 5-4

External Websites

- ASAM XIL

Classes and Methods of ASAM XIL API

In this section...
"MAPort Class" on page 5-4
"ECUMPort Class" on page 5-5
"ECUCPort Class" on page 5-6
"Capture Class" on page 5-6
"CapturingFactory Class" on page 5-7
"CapturingResult Class" on page 5-7
"MAPORTFactory Class" on page 5-7
"SignalFactory Class" on page 5-8
"SignalGeneratoryFactory Class" on page 5-10
"SignalGenerator Class" on page 5-10

To interface with test cases, the Simulink Real-Time XIL API support package supports a subset of the ASAM XIL API. The tables include API methods that you can use with the support package.

The Simulink Real-Time XIL API support package supports XIL stimulation STI/STZ for v2.0-2.2.

MAPort Class

Class	Method	Introduced in Support Package Version
MAPort	CheckVariableNames(variableNames :A_UNICODE2STRING[]) :A_UNICODE2STRING[]	1.0
MAPort	Configure(config :MAPortConfig, forceConfig :A_BOOLEAN) :void	1.0
MAPort	GetDataTypes(variableName :A_UNICODE2STRING) :DataType	1.0
MAPort	GetVariableInfo(variableName :A_UNICODE2STRING) :MAPortVariableInfo	1.0
MAPort	IsReadable(variableName :A_UNICODE2STRING) :A_BOOLEAN	1.0
MAPort	IsWritable(variableName :A_UNICODE2STRING) :A_BOOLEAN	1.0
MAPort	LoadConfiguration(filepath :A_UNICODE2STRING) :MAPortConfig	1.0
MAPort	StartSimulation() :void	1.0
MAPort	StopSimulation() :void	1.0
MAPort	getConfiguration() :MAPortConfig	1.0
MAPort	getState() :MAPortState	1.0
MAPort	getTaskInfos() :TaskInfo[]	1.0
MAPort	getTaskNames() :A_UNICODE2STRING[]	1.0
MAPort	getVariableNames() :A_UNICODE2STRING[]	1.0
MAPort	Dispose	1.0

Class	Method	Introduced in Support Package Version
MAPort	Disconnect	1.0
MAPort	IBaseValue Read(string variableName);	1.1
MAPort	void Write(string variableName, IBaseValue value);	1.1
MAPort	ICapture CreateCapture(string taskName)	1.1
MAPort	void MAPort::DownloadParameterSets(IList<string> filepaths)	1.2

ECUMPort Class

Class	Method	Introduced in Support Package Version
ECUMPort	CheckVariableNames	1.0
ECUMPort	Configure	1.0
ECUMPort	CreateCapture	1.0
ECUMPort	GetDataType	1.0
ECUMPort	GetMeasuringVariables	1.0
ECUMPort	GetVariableInfo	1.0
ECUMPort	IsReadable	1.0
ECUMPort	LoadConfiguration	1.0
ECUMPort	Read	1.0
ECUMPort	SetMeasuringVariables	1.0
ECUMPort	StartMeasurement	1.0
ECUMPort	StopMeasurement	1.0
ECUMPort	getConfiguration	1.0
ECUMPort	getState	1.0
ECUMPort	getTaskInfos	1.0
ECUMPort	getTaskNames	1.0
ECUMPort	getVariableNames	1.0
ECUMPort	Disconnect	1.0
ECUMPort	Dispose	1.0

ECUCPort Class

Class	Method	Introduced in Support Package Version
ECUCPort	CalculateRefPageCRC	1.0
ECUCPort	CalculateWorkPageCRC	1.0
ECUCPort	CheckVariableNames	1.0
ECUCPort	Configure	1.0
ECUCPort	GetDataType	1.0
ECUCPort	GetVariableInfo	1.0
ECUCPort	IsReadable	1.0
ECUCPort	IsWriteable	1.0
ECUCPort	LoadConfiguration	1.0
ECUCPort	NumberOfPages	1.0
ECUCPort	Read	1.0
ECUCPort	StartOnlineCalibration	1.0
ECUCPort	StopOnlineCalibration	1.0
ECUCPort	SwitchToRefPage	1.0
ECUCPort	SwitchToWorkPage	1.0
ECUCPort	Write	1.0
ECUCPort	getConfiguration	1.0
ECUCPort	getState	1.0
ECUCPort	getVariableNames	1.0
ECUCPort	Disconnect	1.0
ECUCPort	Dispose	1.0

Capture Class

Class	Method	Introduced in Support Package Version
Capture	Fetch(whenFinished :A_BOOLEAN) :CaptureResult	1.0
Capture	Start(writer :CaptureResultWriter) :void	1.0
Capture	getCaptureResult() :CaptureResult	1.0
Capture	getState() :CaptureState	1.0
Capture	setVariables(variableNames :A_UNICODE2STRING[]) :void	1.0

CapturingFactory Class

Class	Method	Introduced in Support Package Version
CapturingFactory	CreateCaptureResult	1.0
CapturingFactory	ICaptureResultMDFWriter CapturingFactory::CreateCaptureResultMDFWriter()	1.2
CapturingFactory	ICaptureResultMDFWriter CapturingFactory::CreateCaptureResultMDFWriterByFileName(string fileName)	1.2

CapturingResult Class

Class	Method	Introduced in Support Package Version
CaptureResult	void CaptureResult::Save(ICaptureResultWriter writer)	1.2

MAPORTFactory Class

Class	Method	Introduced in Support Package Version
MAPortFactory	CreateMAPort	1.0
MAPortFactory	CreateMAPortBreakpoint	See note.
MAPortFactory	CreateMAPortBreakpoint2	See note.

Note The signature for the CreateMAPortBreakpoint method is incorrect in ASAM XIL v2.1.0. The signature for the CreateMAPortBreakpoint2 is the corrected version of the method and is contained in ASAM XIL v2.1.1.

SignalFactory Class

Class	Method	Introduced in Support Package Version
SignalFactory	CreateConstSegment():IConstSegment	1.1
SignalFactory	CreateConstSegment(IConstSymbol duration, IWatcher stopTrigger, ISymbol value):IConstSegment	1.1
SignalFactory	CreateDataFileSegment():IDataFileSegment	1.1
SignalFactory	CreateDataFileSegmentByParameters(string fileName, string timeVectorName, string dataVectorName, string channelSource, string channelPath, string groupName, string groupSource, string groupPath, IConstSymbol duration, InterpolationTypes interpolation, IConstSymbol start, IWatcher stopTrigger):IDataFileSegment	1.1
SignalFactory	CreateExpSegment():IExpSegment	1.1
SignalFactory	CreateExpSegmentBySymbols(IConstSymbol duration, ISymbol start, ISymbol stop, IWatcher stopTrigger, ISymbol tau):IExpSegment	1.1
SignalFactory	CreateIdleSegment():IIdleSegment:IIdleSegment	1.1
SignalFactory	CreateIdleSegmentByDuration(IConstSymbol duration, IWatcher stopTrigger):IIdleSegment	1.1
SignalFactory	CreateLoopSegment():ILoopSegment	1.1
SignalFactory	CreateLoopSegmentByLoopCount(ulong loopCount):ILoopSegment	1.1
SignalFactory	CreateNoiseSegment():INoiseSegment	1.1
SignalFactory	CreateNoiseSegmentBySymbols(IConstSymbol duration, ISymbol mean, ISymbol sigma, IConstSymbol seed, IWatcher stopTrigger):INoiseSegment	1.1
SignalFactory	CreateOperationSegment():IOperationSegment	1.1
SignalFactory	CreateOperationSegmentBySignalSegmentsAndOperationTypes(ISignalSegment leftSegment, ISignalSegment rightSegment, OperationTypes operation):IOperationSegment	1.1
SignalFactory	CreatePulseSegment():IPulseSegment	1.1
SignalFactory	CreatePulseSegmentBySymbols(IConstSymbol duration, ISymbol offset, ISymbol amplitude, ISymbol period, ISymbol dutyCycle, ISymbol phase, IWatcher stopTrigger):IPulseSegment	1.1
SignalFactory	CreateRampSegment():IRampSegment	1.1

Class	Method	Introduced in Support Package Version
SignalFactory	CreateRampSegmentBySymbols(IConstSymbol duration, ISymbol start, ISymbol stop):IRampSegment	1.1
SignalFactory	CreateRampSlopeSegment():IRampSlopeSegment	1.1
SignalFactory	CreateRampSlopeSegmentBySymbols(IConstSymbol duration, ISymbol offset, ISymbol slope, IWatcher stopTrigger):IRampSlopeSegment	1.1
SignalFactory	CreateSawSegment():ISawSegment	1.1
SignalFactory	CreateSawSegmentBySymbols(IConstSymbol duration, ISymbol offset, ISymbol amplitude, ISymbol period, ISymbol dutyCycle, ISymbol phase, IWatcher stopTrigger):ISawSegment	1.1
SignalFactory	CreateSegmentSignalDescription():ISegmentSignalDescription	1.1
SignalFactory	CreateSegmentSignalDescriptionByName(string name):ISegmentSignalDescription	1.1
SignalFactory	CreateSignalDescriptionSet():ISignalDescriptionSet	1.1
SignalFactory	CreateSignalDescriptionSetByReader(ISignalDescriptionSetReader reader):ISignalDescriptionSet	1.1
SignalFactory	CreateSignalDescriptionSetSTIReaderByFileName(string fileName):ISignalDescriptionSetSTIReader	1.1
SignalFactory	CreateSignalDescriptionSetSTIWriterByFileName(string fileName):ISignalDescriptionSetSTIWriter	1.1
SignalFactory	CreateSignalDescriptionSetSTZReaderByFileName(string fileName):ISignalDescriptionSetSTZReader	1.1
SignalFactory	CreateSignalDescriptionSetSTZWriterByFileName(string fileName):ISignalDescriptionSetSTZWriter	1.1
SignalFactory	CreateSignalValueSegment():ISignalValueSegment	1.1
SignalFactory	CreateSignalValueSegmentByValueAndInterpolation(ISignalValue value, InterpolationTypes interpolation):ISignalValueSegment	1.1
SignalFactory	CreateSineSegment():ISineSegment	1.1

Class	Method	Introduced in Support Package Version
SignalFactory	CreateSineSegmentBySymbols(IConstSymbol duration, ISymbol offset, ISymbol amplitude, ISymbol period, ISymbol phase, IWatcher stopTrigger):ISineSegment	1.1

SignalGeneratoryFactory Class

Class	Method	Introduced in Support Package Version
SignalGeneratorFactory	CreateSignalGenerator():ISignalGenerator	1.1
SignalGeneratorFactory	CreateSignalGeneratorSTIReader():ISignalGeneratorSTIReader	1.1
SignalGeneratorFactory	CreateSignalGeneratorSTIReaderByFileName(string fileName):ISignalGeneratorSTIReader	1.1
SignalGeneratorFactory	CreateSignalGeneratorSTIWriter():ISignalGeneratorSTIWriter	1.1
SignalGeneratorFactory	CreateSignalGeneratorSTIWriterByFileName(string fileName):ISignalGeneratorSTIWriter	1.1
SignalGeneratorFactory	CreateSignalGeneratorSTZReader():ISignalGeneratorSTZReader	1.1
SignalGeneratorFactory	CreateSignalGeneratorSTZReaderByFileName(string fileName):ISignalGeneratorSTZReader	1.1
SignalGeneratorFactory	CreateSignalGeneratorSTZWriter():ISignalGeneratorSTZWriter	1.1
SignalGeneratorFactory	CreateSignalGeneratorSTZWriterByFileName(string fileName):ISignalGeneratorSTZWriter	1.1

SignalGenerator Class

Class	Method	Introduced in Support Package Version
SignalGenerator	Load(ISignalGeneratorReader reader)	1.1
SignalGenerator	Save(ISignalGeneratorWriter writer)	1.1

Class	Method	Introduced in Support Package Version
SignalGenerator	Assignments	1.1
SignalGenerator	SignalDescriptionSet	1.1
SignalGenerator	State	1.1
SignalGenerator	DestryOnTarget()	1.1
SignalGenerator	Dispose()	1.1
SignalGenerator	LoadToTarget()	1.1
SignalGenerator	Pause()	1.1
SignalGenerator	Start()	1.1
SignalGenerator	Stop()	1.1
SignalGenerator or STIReader	Load(out ISignalGenerator signalGenerator)	1.1
SignalGenerator or STIWriter	Save(ISignalGenerator signalGenerator)	1.1
SignalGenerator or STZReader	Load(out ISignalGenerator signalGenerator)	1.1
SignalGenerator or STZWriter	Save(ISignalGenerator signalGenerator)	1.1

See Also

createPortConfigureFile

More About

- “Install the Simulink Real-Time Support Package for ASAM XIL Standard” on page 5-2

External Websites

- ASAM XIL

Real-Time Application Setup

Real-Time Application Environment

- “Select Default Target Computer” on page 6-2
- “Set Up Target Computer Ethernet Connection” on page 6-3
- “Target Computer Update, Reboot, and Startup Application” on page 6-5

Select Default Target Computer

When you start Simulink Real-Time Explorer for the first time, it opens a default target computer node, TargetPC1. You can configure this node for a target computer, then connect the node to the target computer. You can add other target computer nodes and designate one of them as the default target computer.

Select Default Target Computer

To set a target computer node as the default.

Select a nondefault target computer node from the **Targets Tree** panel in Simulink Real-Time Explorer.

In the **Target Configuration** tab, select the **Default** checkbox.

If you delete a default target computer node, the target computer node preceding it becomes the default target computer node. The last target computer node becomes the default target computer node and you cannot delete it.

Command-Line Interface and Target Computer

To use the Simulink Real-Time command-line interface to work with the target computer, you must indicate the target computer with which the command is interacting. If you do not identify a particular target computer, the Simulink Real-Time software uses the default target computer.

Targets Object and Target Computers

The Targets object manages collective and individual target computer environments. For more information, see “Set Up Target Computer Ethernet Connection” on page 6-3.

When you call the Targets object `getTargetSettings` function without arguments, the constructor gets the real-time environment settings for the default target computer.

```
my_tgs = slrealtime.Targets();  
my_tgs_settings = getTargetSettings(my_tgs);
```

When you call the Target object `slrealtime` function without arguments, the constructor uses the link properties of the default target computer to communicate with the target computer.

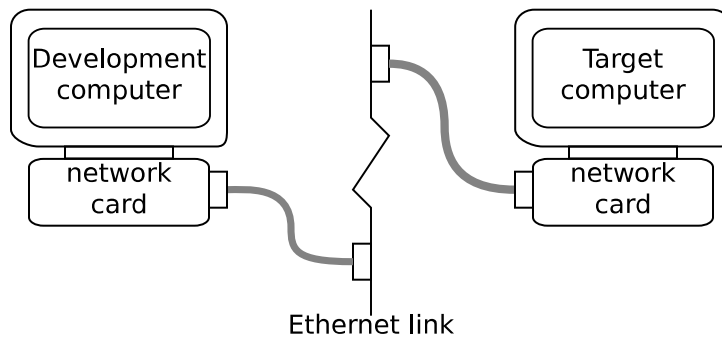
```
tg = slrealtime;
```

See Also

Simulink Real-Time Explorer | Targets | `getDefaultTargetName` | `setDefaultTargetName` | Target | `slrealtime`

Set Up Target Computer Ethernet Connection

To install PCI bus Ethernet protocol interface hardware in your Speedgoat target computer, see the Speedgoat website at www.speedgoat.com.



Connect Ethernet Cables

To configure the target computer Ethernet hardware:

If the target computer already contains one or more Ethernet cards, to get a list of these Ethernet cards, see your Speedgoat target machine documentation.

Assign a static IP address to the target computer Ethernet card by using Simulink Real-Time Explorer.

Unlike the target computer, the development computer network adapter card can have a dynamic host configuration protocol (DHCP) address and can be accessed from the network. Configure the DHCP server to reserve static IP addresses to prevent these addresses from being assigned to other systems.

Connect your target computer Ethernet card to your LAN by using an unshielded twisted-pair (UTP) cable.

You can directly connect your computers by using a crossover UTP cable with RJ45 connectors. Both computers must have static IP addresses. If the development computer has a second network adapter card, that card can have a DHCP address.

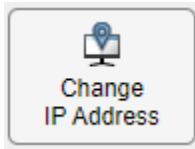
Configure Ethernet Address

To build and download a real-time application by using the installed Ethernet card, first specify the environment properties for the development and target computers. Before you start, ask your system administrator for the following information for your target computer IP address and Subnet mask address. This procedure sets up Ethernet protocol for the default target computer TargetPC1:

Open Simulink Real-Time Explorer. In the Simulink Editor, on the Real-Time tab, click **Prepare > SLRT Explorer**. Or, in the MATLAB Command Window, type `slrtExplorer`.

In the Simulink Real-Time Explorer **Targets Tree** panel, select target computer TargetPC1.

On the **Target Configuration** tab, click the **Change IP Address** button.



Configure the **New IP Address** and **New Netmask** fields in the Configure Target Computer IP Address dialog box. Click **OK**.

Click the **Disconnected** label, toggling it to **Connected**.

Related Ethernet Configuration Topics

You can also configure the target computer Ethernet protocol by using MATLAB commands. For more information, see the `Target`s object functions and examples.

See Also

Simulink Real-Time Explorer | `slrtExplorer`

More About

- “Target Computer Settings”
- “Enable Development Computer Communication (Windows)”
- “Enable Development Computer Communication (Linux)”

Target Computer Update, Reboot, and Startup Application

With Simulink Real-Time Explorer, you can update the target computer RTOS software, reboot the target computer, and configure a startup application that runs each time you start the target computer.

Update Software

To update the target computer software:

Open Simulink Real-Time Explorer.

In the **Targets Tree** panel, select target computer TargetPC1.

To update the target computer RTOS software, click the **Update Software** button.

Click the **Disconnected** label, toggling it to **Connected**.

Reboot Target Computer

To reboot the target computer:

Open Simulink Real-Time Explorer.

In the **Targets Tree** panel, select target computer TargetPC1.

To reboot the target computer, click the **Reboot** button.

Click the **Disconnected** label, toggling it to **Connected**.

Select Startup Application

To configure a startup real-time application:

Open Simulink Real-Time Explorer.

In the **Targets Tree** panel, select target computer TargetPC1.

To load a real-time application on the target computer, click the **Load Application** button.

After you load the application, select the application from the **Applications on target computer** list and select the **Startup App** check.box. The next time the target computer starts or reboots, the application runs on startup.

See Also

update | reboot | setStartupApp

Signals and Parameters

Important prototyping tasks include:

- Changing parameters in your real-time application while it is running
- Viewing the resulting signal data
- Checking the results

The Simulink Real-Time software includes command-line and graphical user interfaces to complete these tasks.

- “Signal Monitoring Basics” on page 7-3
- “Monitor Signals by Using Simulink Real-Time Explorer” on page 7-4
- “Instrument a Stateflow Subsystem” on page 7-5
- “Animate Stateflow Charts with Simulink External Mode” on page 7-7
- “Signal Tracing Basics” on page 7-8
- “Export and Import Signals in Instrument by Using Simulink Real-Time Explorer” on page 7-9
- “Trace Signals by Using Simulink External Mode” on page 7-11
- “Data Logging with Simulation Data Inspector (SDI)” on page 7-14
- “Parameter Tuning and Data Logging” on page 7-18
- “Trace or Log Data with the Simulation Data Inspector” on page 7-21
- “External Mode Usage” on page 7-25
- “Signal Logging and Streaming Basics” on page 7-26
- “Tune Parameters by Using Simulink Real-Time Explorer” on page 7-30
- “Tune Parameters by Using MATLAB Language” on page 7-33
- “Tune Parameters by Using Simulink External Mode” on page 7-35
- “Save and Reload Parameters by Using the MATLAB Language” on page 7-37
- “Tunable Block Parameters and Tunable Global Parameters” on page 7-41
- “Tune Inlined Parameters by Using Simulink Real-Time Explorer” on page 7-44
- “Tune Inlined Parameters by Using MATLAB Language” on page 7-48
- “Tune Parameter Structures by Using Simulink Real-Time Explorer” on page 7-49
- “Tune Parameter Structures by Using MATLAB Language” on page 7-52
- “Define and Update Inport Data” on page 7-55
- “Define and Update Inport Data by Using MATLAB Language” on page 7-60
- “Stimulate Root Inport by Using MATLAB Language” on page 7-63
- “Inport Data Mapping Limitations” on page 7-65
- “Display and Filter Hierarchical Signals and Parameters” on page 7-66
- “Troubleshoot Signals Not Accessible by Name” on page 7-70
- “Troubleshoot Parameters Not Accessible by Name” on page 7-72

- “Troubleshoot Instance-Specific Parameters Not Saved” on page 7-73
- “Internationalization Issues” on page 7-74

Signal Monitoring Basics

Signal monitoring acquires real-time signal data without time information during real-time application execution. There is a minimal additional load on the real-time tasks.

You can monitor signals by using:

- Simulink Real-Time Explorer and the Simulation Data Inspector
- MATLAB language and the `Instrument` object
- Simulink external mode and a Scope block

For more information, see **Simulation Data Inspector** and “How Application is Run Affects Signals Logged” on page 7-26.

See Also

`Instrument` | `Scope`

More About

- “Display and Filter Hierarchical Signals and Parameters” on page 7-66
- “View Data in the Simulation Data Inspector”
- “Troubleshoot Signals Not Accessible by Name” on page 7-70

Monitor Signals by Using Simulink Real-Time Explorer

This procedure uses the model `slrt_ex_osc`. You must have already completed this setup:

Open model `slrt_ex_osc`. Set property **Stop time** to `inf`. On the Simulink Editor **Real-Time** tab, select **Run on Target > Stop Time** and set the **Stop Time** to `inf`.


Connect to the target computer. Toggle the **Disconnected** indicator to **Connected**.

Build and download the real-time application to the target computer. Click **Run on Target**.

Open Simulink Real-Time Explorer. Click **Prepare > SLRT Explorer**.

To monitor a signal in the real-time application, in Simulink Real-Time Explorer, click **Load Application**. Select the `slrt_ex_osc` application from the **Applications on target computer** list and click **Load**.

Click the **Signals** tab.

Select the signals to monitor from the list, and then click **Add to signals in instrument** . To monitor signals, click **Start Streaming**. To display the monitored signal values, click **View Values**.

To start the real-time application, click **Start**.

To view the signals in the Simulation Data Inspector, click **Data Inspector**.

To stop execution, click **Stop**.

See Also

More About

- “Export and Import Signals in Instrument by Using Simulink Real-Time Explorer” on page 7-9
- “Display and Filter Hierarchical Signals and Parameters” on page 7-66
- “Troubleshoot Signals Not Accessible by Name” on page 7-70

Instrument a Stateflow Subsystem

A Simulink Real-Time model that uses Stateflow blocks can provide visual confirmation that your chart behaves as expected when you simulate the model or run the real-time application.

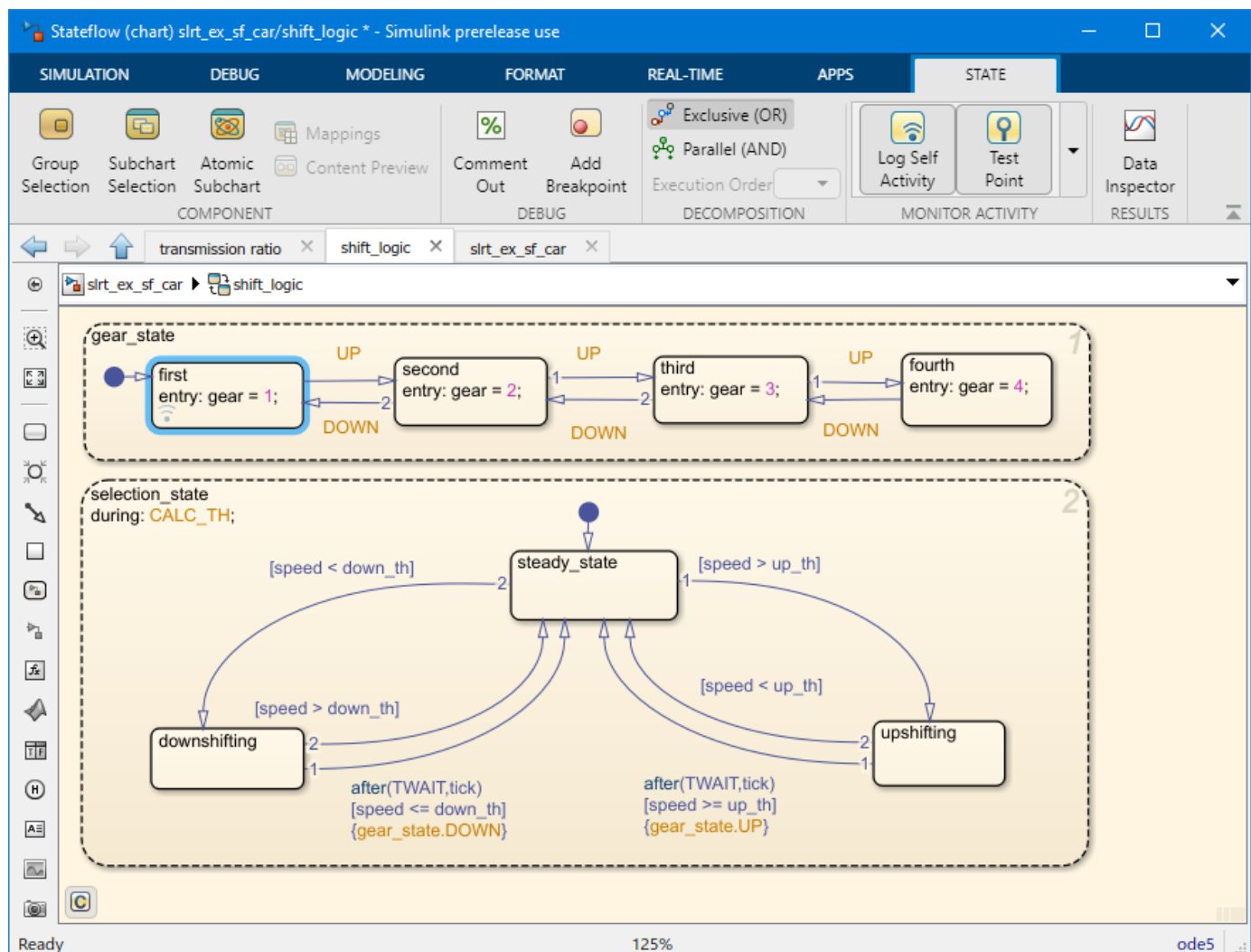
This procedure uses the model `slrt_ex_sf_car`. To open the model and its related MAT file, in the MATLAB Command Window, type:

```
open_system(fullfile(matlabroot, 'toolbox', 'slrealtime', ...
    'examples', 'slrt_ex_sf_car'))
load(fullfile(matlabroot, 'toolbox', 'slrealtime', ...
    'examples', 'slrt_ex_user_inputs.mat'), '-mat')
```

To make Stateflow states available in the Simulation Data Inspector, select them and mark them for **Log Self Activity**.

Open the `slrt_ex_sf_car` model.

Double-click the `shift_logic` chart.



In the **gear_state** chart, select the **first** state

Click the **Log Self Activity** button and the **Test Point** button.

Repeat steps 3-4 for **gear_state** values **second**, **third**, and **fourth**.

Build and download the real-time application to the target computer. On the **Real-Time** tab, click **Run on Target**.

Monitor Stateflow states by using the Simulation Data Inspector. For more information, see “View Data in the Simulation Data Inspector” and “View State Activity by Using the Simulation Data Inspector” (Stateflow).

See Also

More About

- “View Data in the Simulation Data Inspector”
- “View State Activity by Using the Simulation Data Inspector” (Stateflow)
- “Animate Stateflow Charts with Simulink External Mode” on page 7-7

Animate Stateflow Charts with Simulink External Mode

The Simulink Real-Time software supports the animation of Stateflow charts in your model to provide visual confirmation that your chart behaves as expected. You must be familiar with the use of Stateflow animation. For more information on Stateflow animation, see “Animate Stateflow Charts” (Stateflow).

You must have already configured the Stateflow states for animation in the model. If you have not, see “Animate Stateflow Charts” (Stateflow). This example uses model `slrt_ex_sf_car`. To open the model and load its related MAT file, in the MATLAB Command Window, type:

```
open_system(fullfile(matlabroot, 'toolbox', 'slrealtime', ...
    'examples', 'slrt_ex_sf_car'))
load(fullfile(matlabroot, 'toolbox', 'slrealtime', ...
    'examples', 'slrt_ex_user_inputs.mat'), '-mat')
```

Open the external mode control panel. In the Simulink Editor, in the **Real-Time** tab, click **Prepare > Control Panel**.

Select **Signal & Triggering**.

In the **Trigger** section of the **External Signal & Triggering** window:

- a** To direct the trigger to re-arm after the trigger event completes, set **Mode** to normal.
- b** To select the number of base rate steps for which external mode uploads data after a trigger event, in the **Duration** box, enter 5.
- c** To direct data upload to begin immediately after the trigger event, select the **Arm when connecting to target** check box.

Click **Apply**. For more information about signal and triggering options, see “Configure Host Monitoring of Target Application Signal Data”.

Connect to the target computer. On the **Real-Time** tab, toggle the **Disconnected** indicator to **Connected**.

Build and download the model to the target computer. On the **Real-Time** tab, click **Run on Target**.

The simulation begins to run. You can observe the animation by opening the Stateflow Editor for your model.

To stop the simulation, on the **Real-Time** tab, click **Stop**.

See Also

More About

- “Animate Stateflow Charts” (Stateflow)
- “Configure Host Monitoring of Target Application Signal Data”
- “Simulink External Mode Interface”

Signal Tracing Basics

Signal tracing acquires signal and time data from a real-time application. While the real-time application is running, you can visualize the data on the target computer by using the Simulation Data Inspector. You can upload the data from a File Log block to the development computer and display it using the Simulation Data Inspector.

You trace signals by marking the signals for logging or connecting the signals to File Log blocks. View the signals by using Simulink Real-Time Explorer, Simulink external mode, and the Simulation Data Inspector. For more information, see **Simulation Data Inspector** and “How Application is Run Affects Signals Logged” on page 7-26.

See Also

More About

- “Display and Filter Hierarchical Signals and Parameters” on page 7-66
- “View Data in the Simulation Data Inspector”
- “Troubleshoot Signals Not Accessible by Name” on page 7-70

Export and Import Signals in Instrument by Using Simulink Real-Time Explorer

When testing a complex model with many signals, you frequently must select signals for tracing or monitoring from multiple parts and levels of the model hierarchy.

Save Signals to Disk

You can make this task easier by using Simulink Real-Time Explorer to select the signals in instrument and save the list of signals to disk.

Open model `slrt_ex_osc`.

Connect to the target computer. On the Simulink Editor **Real-Time** tab, toggle the **Disconnected** indicator to **Connected**.


Build and download the real-time application to the target computer. Click **Run on Target**.

Open Simulink Real-Time Explorer. Click **Prepare > SLRT Explorer**.


To add signals to the signals in instrument, export the list of signals, and import the list of signals.

In Simulink Real-Time Explorer, click **Load Application**. Select the `slrt_ex_osc` application from the **Applications on target computer** list and click **Load**.

Click the **Signals** tab.

Select the signals to monitor from the list and then click **Add to signals in instrument** .

To export the list, click **Export instrument to file** . Name the files and click **Save**.

To remove signals from the signals in the instrument, select the signals in the list, and then click **Remove signals from instrument** .


To import the list, click **Import instrument from file** . Select the file and click **Open**.


Get MATLAB Code for Signals

When developing an App Designer application or an m-script that connects to a real-time application, it is helpful to have MATLAB code for the signals in the instrument. This code provides access to signals in an Instrument object (or instrumented signals), which are signals that are configured for streaming signal data from a real-time application. To generate this code from the **Signals in Instrument**:

In Simulink Real-Time Explorer, click **Load Application**. Select the `slrt_ex_osc` application from the **Applications on target computer** list and click **Load**.

Click the **Signals** tab.

Select the signals to monitor from the list, and then click **Add to signals in instrument** .

To create MATLAB code for the signals in the instrument, click **Generate MATLAB code to create Instrument programmatically** . An editor window opens in MATLAB and displays the code for the signals in the Instrument.

See Also

[Instrument](#) | [addSignal](#) | [connectLine](#) | [connectScalar](#) | [validate](#)

More About

- “Monitor Signals by Using Simulink Real-Time Explorer” on page 7-4
- “Display and Filter Hierarchical Signals and Parameters” on page 7-66

Trace Signals by Using Simulink External Mode

You can use Simulink external mode to establish a communication channel between your Simulink block diagram and your real-time application. The block diagram becomes a user interface to your real-time application. Simulink scopes can display signal data from the real-time application, including from models referenced inside a top model. You can control which signals to upload through the External Signal & Triggering dialog box. See “Select Signals to Upload” and “TCP/IP or Serial External Mode Control Panel”.

If using external mode simulation with the model serving as the interface to the real-time application and the model contains referenced models, use the Simulation Data Inspector to log signal data. Do not use Floating Scope or Scope Viewer blocks to display signals in the referenced models for external mode simulation.

Note Do not use Simulink external mode while Simulink Real-Time Explorer is running. Use only one interface to control the real-time application.

Set Up for External Mode Simulation

This procedure uses model `slrt_ex_osc`. This model contains a Simulink Scope block. To set up triggering for the external mode simulation:

Open model `slrt_ex_osc`.

Open the external mode control panel. In the Simulink Editor, on the **Real-Time** tab, click **Prepare > Control Panel**.

In the external mode control panel, click **Signal & Triggering**.

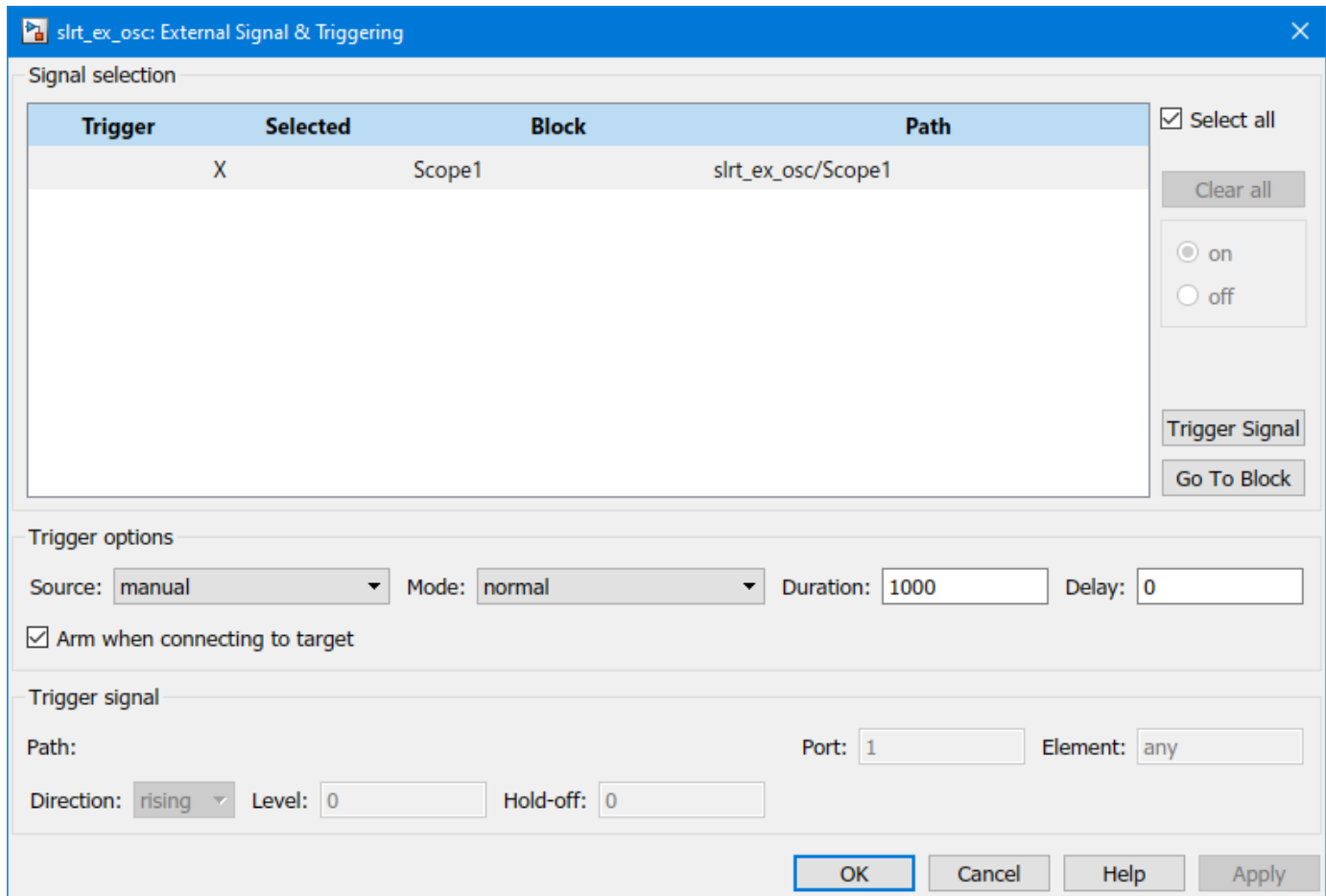
In the External Signal & Triggering dialog box, set the **Source** parameter to `manual`.

Set the **Mode** parameter to `normal`. In this mode, the scope acquires data continuously.

Select the **Arm when connecting to target** check box.

In the **Delay** box, enter `0`.

In the **Duration** box, enter the number of samples for which external mode is to log data, for example, `1000`. The External Signal & Triggering dialog box looks like this figure.



Click **Apply**, and then **Close**. In the External Mode Control Panel dialog box, click **OK**.

Set Stop Time and Simulate

To set the stop time and run the simulation:

In the Simulink toolbar, increase the simulation stop time to, for example, 50.

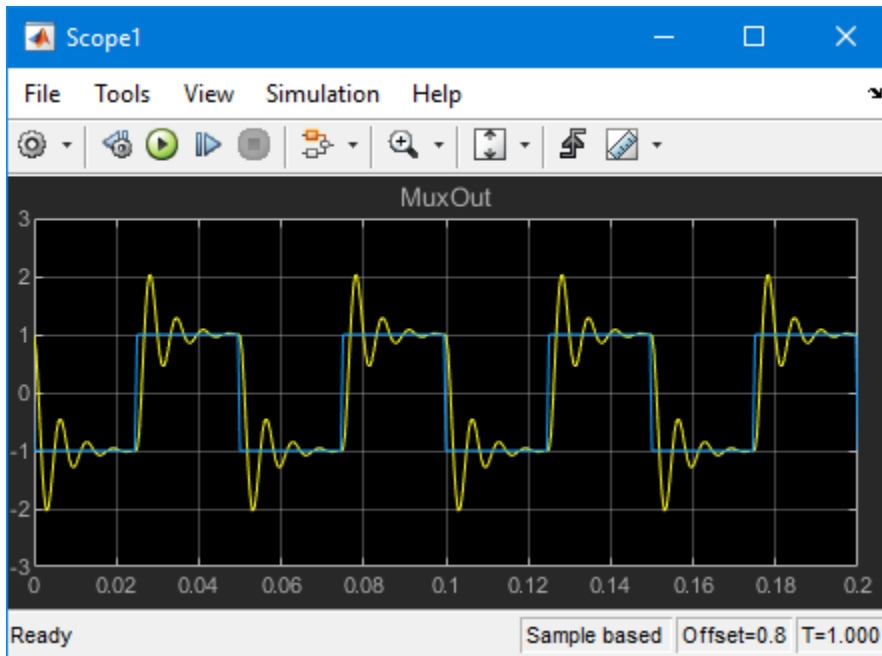
Save the model as `ex_slrt_ext_osc`. On the **Simulation** tab, from **Save**, click **Save As**.

If a scope window is not displayed for the Scope block, double-click the Scope block.

Connect to the target computer. On the **Real-Time** tab, toggle the **Disconnected** indicator to **Connected**.

Build and download the real-time application to the target computer. Click **Run on Target**.

The real-time application begins running on the target computer. The Scope window displays plotted data.



To stop the simulation, on the **Real-Time** tab click **Stop**.

See Also

Data Logging with Simulation Data Inspector (SDI)

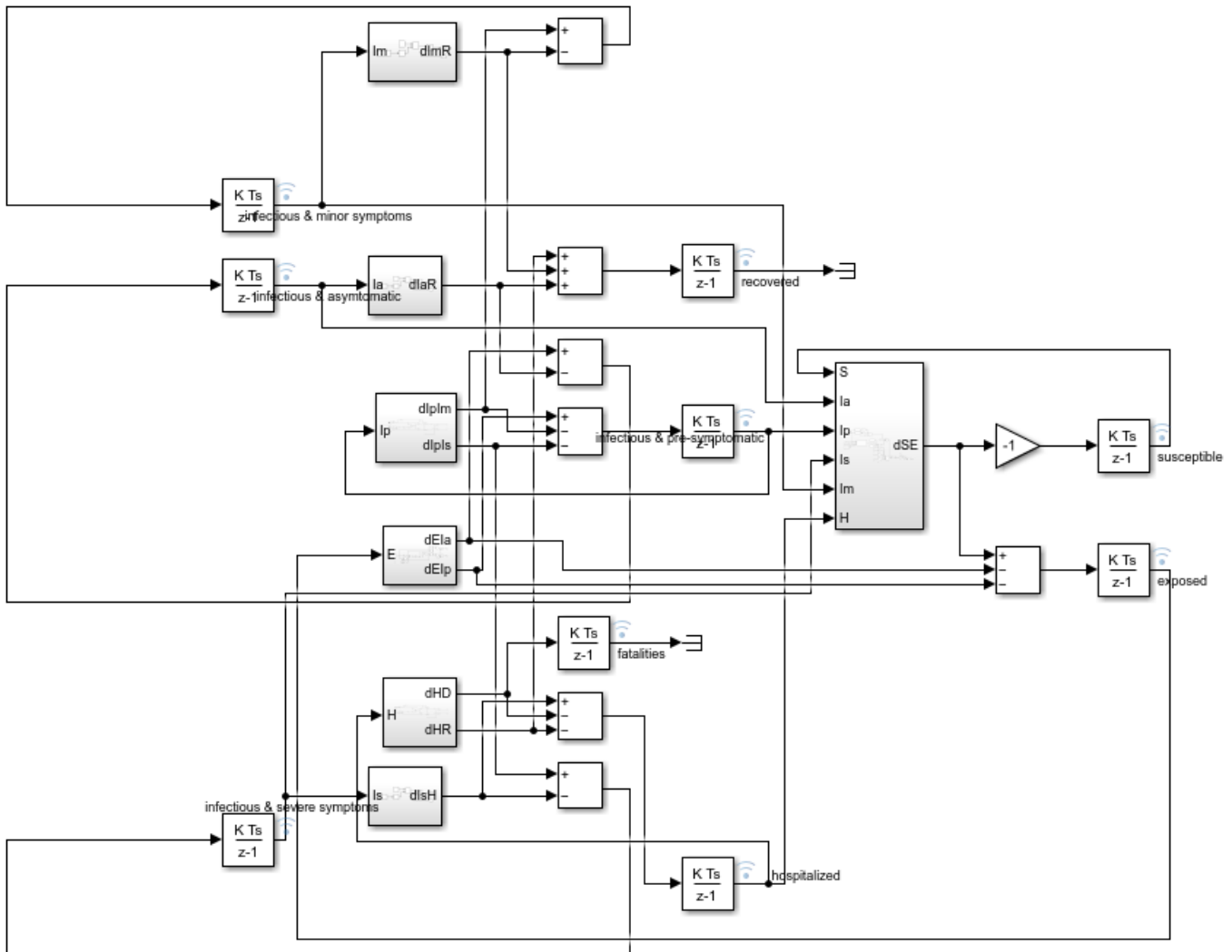
This example shows how to use a Simulink® Real-Time™ log of signal data and the Simulation Data Inspector. Signals are logged during model execution. At the end of the run, the Simulation Data Inspector interface displays the signal. This example show how to get the signals from the Simulation Data Inspector interface by using the command line.

Open, Build, and Download Model

Open the model `slrt_ex_soc_dist`. This model calibrates the control efforts through social distancing on an infectious disease outbreak.

Open the model.

```
model = 'slrt_ex_soc_dist';  
mdlOpen = 0;  
systems = find_system('type', 'block_diagram');  
if all(~strcmp(model, systems))  
    mdlOpen = 1;  
    open_system(fullfile(matlabroot, 'toolbox', 'slrealtime', 'examples', 'slrt_ex_soc_dist.slx'));  
end
```



Model slrt_ex_soc_dist
 Simulink Real-Time example model
 Copyright 2020 The MathWorks, Inc.

Build the model and download to the target computer:

- Configure for a non-Verbose build.
- Build and download application.

```
set_param(model, 'RTWVerbose', 'off');
evalc('slbuild(model)');
```

- Close the model it is open.

```
if (mdlOpen)
    bdclose(model);
end
```

Run Model to Evaluate Effects of No Social Distancing During Outbreak

Using the Simulink Real-Time object variable, `tg`, load and start the model, and modify model parameters.

```
tg = slrealtime;
load(tg,model);
setparam(tg,','soc_dist_level',1);
setparam(tg,','thresh_int_level',1);
start(tg);
while ~strcmp(tg.status,'stopped')
    pause(5);
end
stop(tg);
```

Update Parameters and Re-evaluate Effect of Social Distancing During Outbreak

Using the Simulink Real-Time object variable, `tg`, load and start the model, and modify model parameters

```
tg = slrealtime;
load(tg,model);
setparam(tg,','soc_dist_level',0.2);
setparam(tg,','thresh_int_level',0.2);
start(tg);
while ~strcmp(tg.status,'stopped')
    pause(5);
end
stop(tg);
```

Display Signals in Simulation Data Inspector

To view the plotted signal data, open the Simulation Data Inspector.

```
Simulink.sdi.view
```

Retrieve and Plot Signal Data from Simulation Data Inspector

You can also retrieve the signal data from the SDI and plot the data by using these commands.

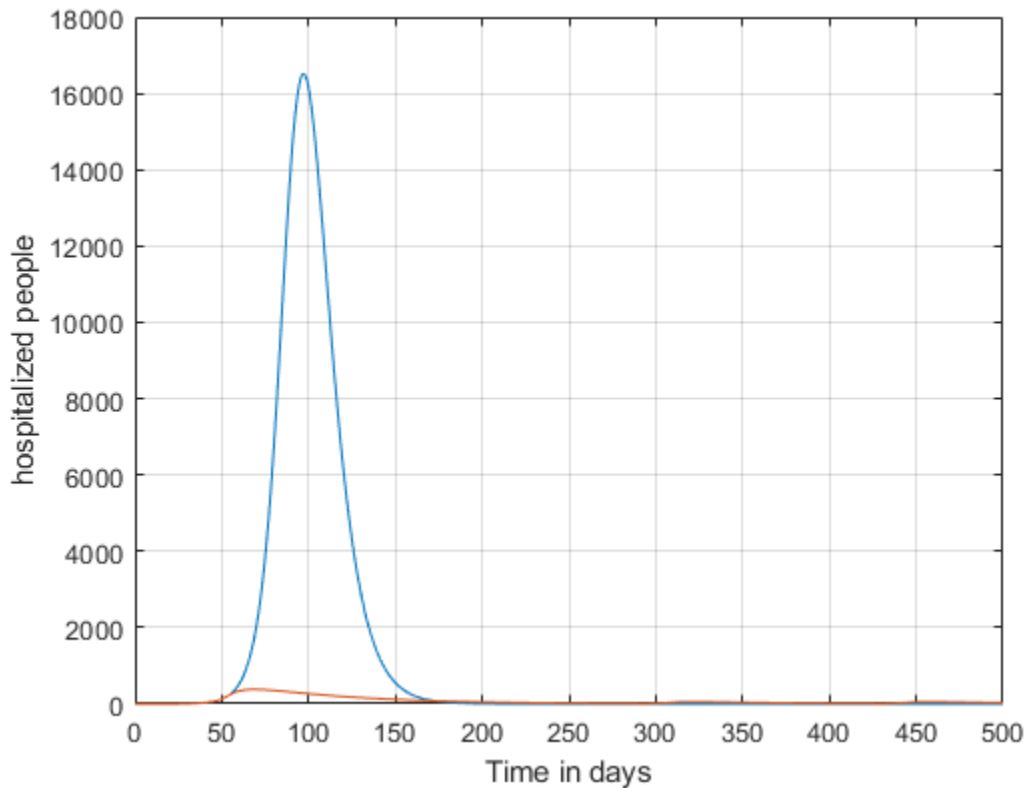
- Get all the runs
- Get the run information
- Get the signal.
- Get the signal objects.
- Take only infectious group values.
- Plot the signals.

The result shows that social distancing can reduce the number of hospitalized people

```
runIds = Simulink.sdi.getAllRunIDs();

for i = 1:length(runIds)
    run = Simulink.sdi.getRun(runIds(i));
    signalID = run.getSignalIDsByName('hospitalized');
    if ~isempty(signalID)
        signalObj = Simulink.sdi.getSignal(signalID);
    end
end
```

```
    signalArray(:,i) = signalObj.Values(:,1).Data;  
    timeValues = 100*(signalObj.Values(:,1).Time);  
    plot(timeValues,signalArray);  
    drawnow;  
end  
end  
  
grid on;  
xlabel('Time in days'); ylabel('hospitalized people');
```



See Also

slrtTETMonitor | SLRT Overload Options

More About

- “Trace or Log Data with the Simulation Data Inspector” on page 7-21
- Simulation Data Inspector

Parameter Tuning and Data Logging

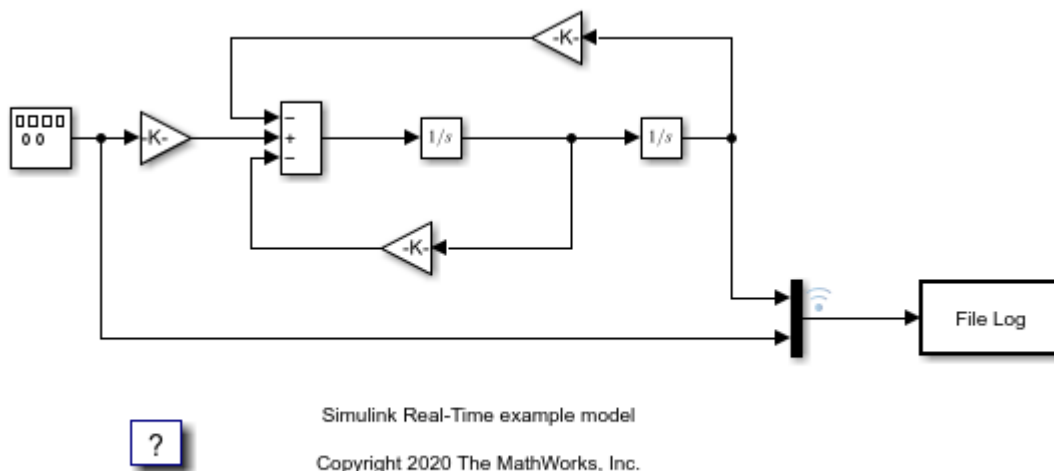
This example shows how to use real-time parameter tuning and data logging with Simulink® Real-Time™. After the example builds the model and downloads the real-time application, `slrt_ex_param_tuning`, to the target computer, the example executes multiple runs with the gain 'Gain1/Gain' changed (tuned) before each run. The gain sweeps from 0.1 to 0.7 in steps of 0.05.

The example uses the data logging capabilities of Simulink Real-Time to capture signals of interest during each run. The logged signals are uploaded to the development computer and plotted. A 3-D plot of the oscillator output versus time versus gain is displayed.

Open, Build, and Download Model to the Target Computer

Open the model, `slrt_ex_param_tuning`. The model configuration parameters select the `slrealtime.tlc` system target file as the code generation target. Building the model creates a real-time application, `slrt_ex_param_tuning.mldatx`, that runs on the target computer.

```
model = 'slrt_ex_param_tuning';
open_system(fullfile(matlabroot, 'toolbox', 'slrealtime', 'examples', model));
```



Build the model and download the real-time application, `slrt_ex_param_tuning.mldatx`, to the target computer.

- Configure for a non-Verbose build.
- Build and download application.

```
set_param(model, 'RTWVerbose', 'off');
set_param(model, 'StopTime', '0.2');
evalc('slbuild(model)');
tg = slrealtime;
load(tg, model);
```

Run Model, Sweep 'Gain' Parameter, Plot Logged Data

This code accomplishes several tasks.

Task 1: Create Target Object

Create the MATLAB® variable, `tg`, that contains the Simulink Real-Time target object. This object lets you communicate with and control the target computer.

- Create a Simulink Real-Time target object.
- Set stop time to 0.2s.

Task 2: Run the Model and Plot Results

Run the model, sweeping through and changing the gain (damping parameter) before each run. Plot the results for each run.

- If no plot figure exist, create the figure.
- If the plot figure exist, make it the current figure.

Task 3: Loop over damping factor z

- Set damping factor (Gain1/Gain).
- Start run of the real-time application.
- Store output data in `outp`, `y`, and `t` variables.
- Plot data for current run.

Task 4: Create 3-D Plot (Oscillator Output vs. Time vs. Gain)

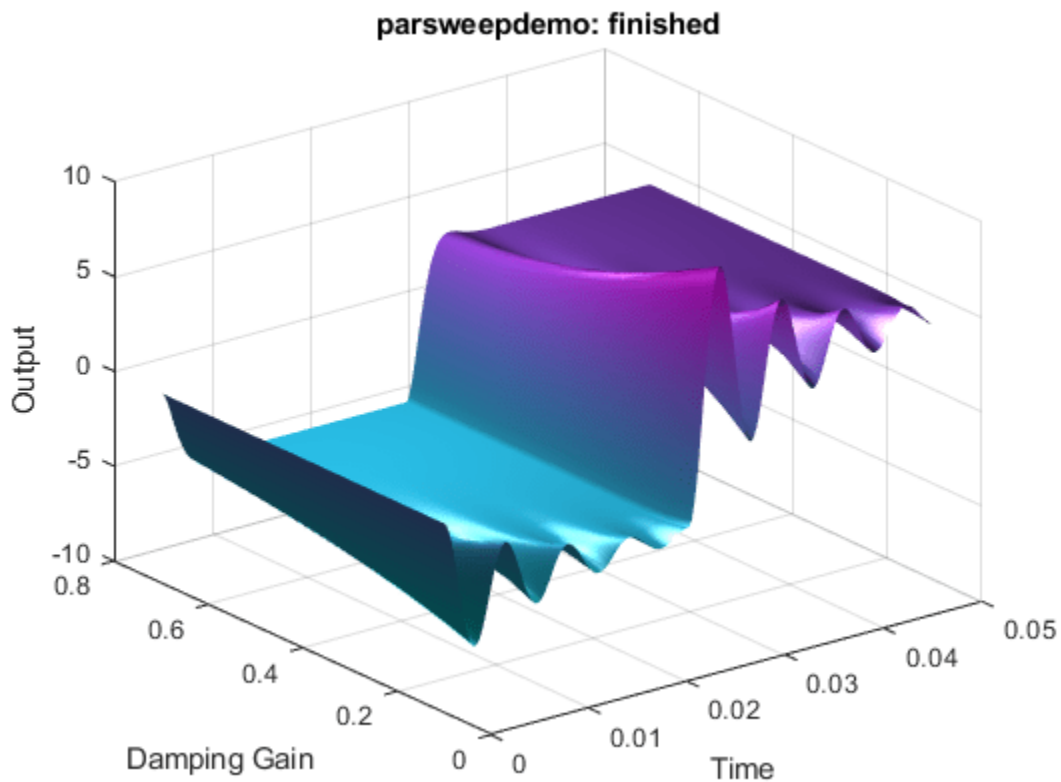
- Loop over damping factor.
- Create a plot of oscillator output versus time versus gain.
- Create 3-D plot.

```
figh = findobj('Name', 'parsweepdemo');
if isempty(figh)
    figh = figure;
    set(figh, 'Name', 'parsweepdemo', 'NumberTitle', 'off');
else
    figure(figh);
end
y = []; flag = 0;
for z = 0.1 : 0.05 : 0.7
    if isempty(find(get(0, 'Children') == figh, 1))
        flag = 1;
        break;
    end
    load(tg,model);
    tg.setparam([model '/Gain1'],'Gain',2 * 1000 * z);
    tg.start('AutoImportFileLog',true, 'ExportToBaseWorkspace', true);
    pause(0.4);
    outp = logsOut{1}.Values;
    y     = [y,outp.Data(:,1)];
    t     = outp.Time;
    plot(t,y);
    set(gca, 'XLim', [t(1), t(end)], 'YLim', [-10, 10]);
    title(['parsweepdemo: Damping Gain = ', num2str(z)]);
    xlabel('Time'); ylabel('Output');
    drawnow;
end
```

```

if ~flag
    delete(gca);
    surf(t(1 : 200), 0.1 : 0.05 : 0.7, y(1 : 200, :));
    colormap cool
    shading interp
    h = light;
    set(h, 'Position', [0.0125, 0.6, 10], 'Style', 'local');
    lighting gouraud
    title('parsweepdemo: finished');
    xlabel('Time'); ylabel('Damping Gain'); zlabel('Output');
end

```



Close Model

When done, close the model.

```
close_system(model,0);
```

See Also

slrtTETMonitor

More About

- “Trace or Log Data with the Simulation Data Inspector” on page 7-21
- Simulation Data Inspector

Trace or Log Data with the Simulation Data Inspector

With the Simulation Data Inspector and Simulink Real-Time, you can trace signal data by streaming signal data directly to the Simulation Data Inspector or by logging signal data by recording it through a File Log block. If streaming signal data directly, you view the output in real time as the application produces it.

The application can produce more data than the target computer can transmit in real time to the development computer. Data accumulates in the network buffer, and, if the buffer fills up, the RTOS drops data points. Streaming signal data directly does not support decimation or limit data points.

To avoid dropped data points caused by network buffer overruns, you can use logging through a File Log block. When logging, you connect signals to File Log blocks in the model. In the real-time application, these blocks store data for the buffered signals on the target computer. At the end of execution, the real-time application transmits the data to the development computer for display in the Simulation Data Inspector. You can then view the most important signals immediately and view the buffered signals afterward.

Logging signal data through a File Log block supports decimation or limit data points and supports conditional block execution semantics. Some examples are logging signal data by enabling data logging for a signal inside a for-iterator, function-call, or enabled/triggered subsystem. For more information, see **Simulation Data Inspector** and “How Application is Run Affects Signals Logged” on page 7-26.

Set Up Model for Logging

To set up the model for logging signal data:

Open `s_lrt_ex_osc`.

Select the MuxOut output signal, place your cursor over the signal, and select **Enable Data Logging**.

Tip Consider whether to configure the **Logging sample time** in the **Instrumentation Properties** for the logged signal. Use this property to set a lower sample time for signals that go into the Simulation Data Inspector while the simulation is running. Configuring this property can help makes the Simulation Data Inspector more responsive and easier to use.

Double-click the File Log block. The **Decimation** value is 1.

Set Up Simulation Data Inspector

To set up the Simulation Data Inspector:

Open the Simulation Data Inspector ()

Click **Layout** ()


Select two horizontal displays.

View Simulation Data

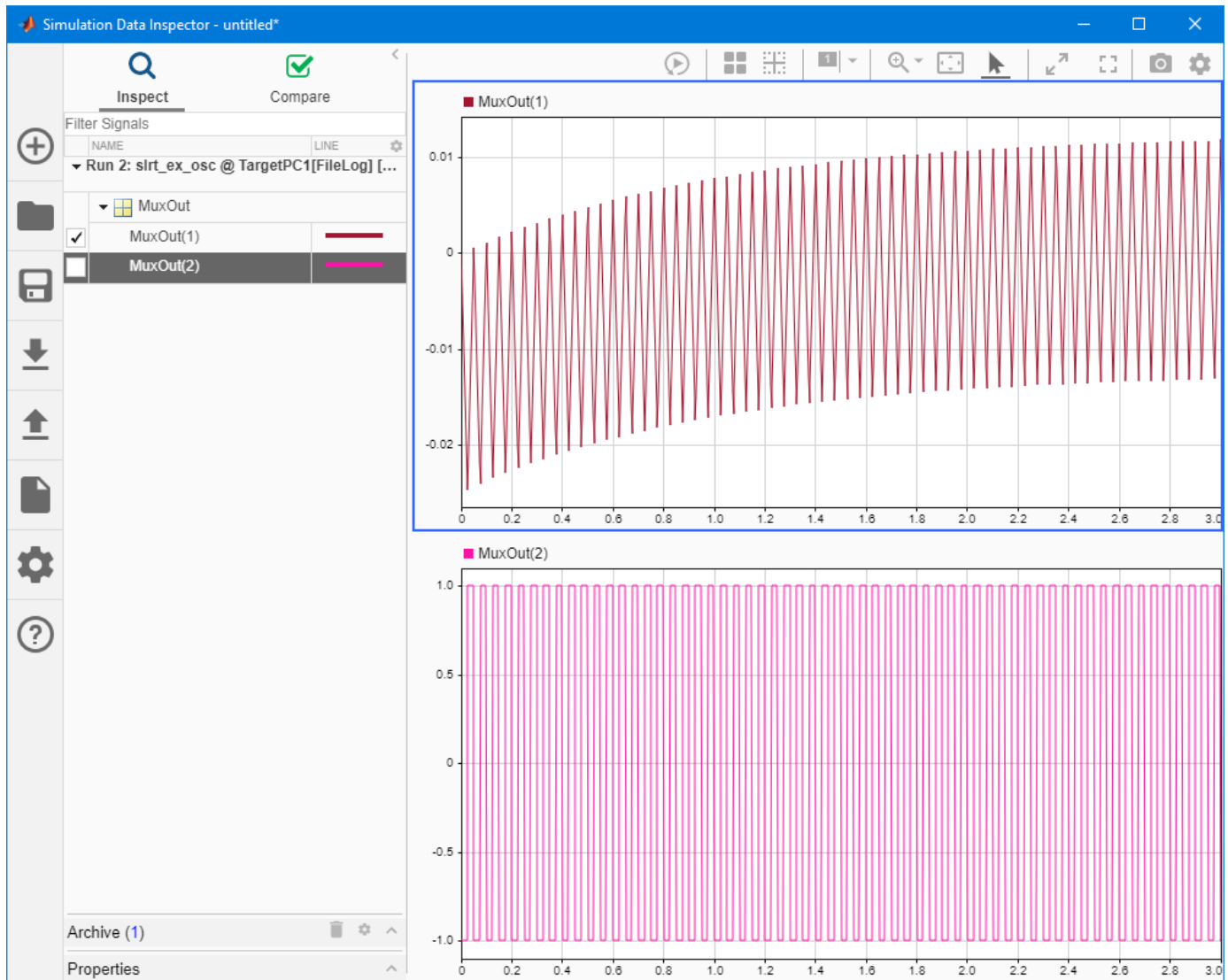
To view the simulation data:

Build and download `slrt_ex_osc`.

Start real-time execution.

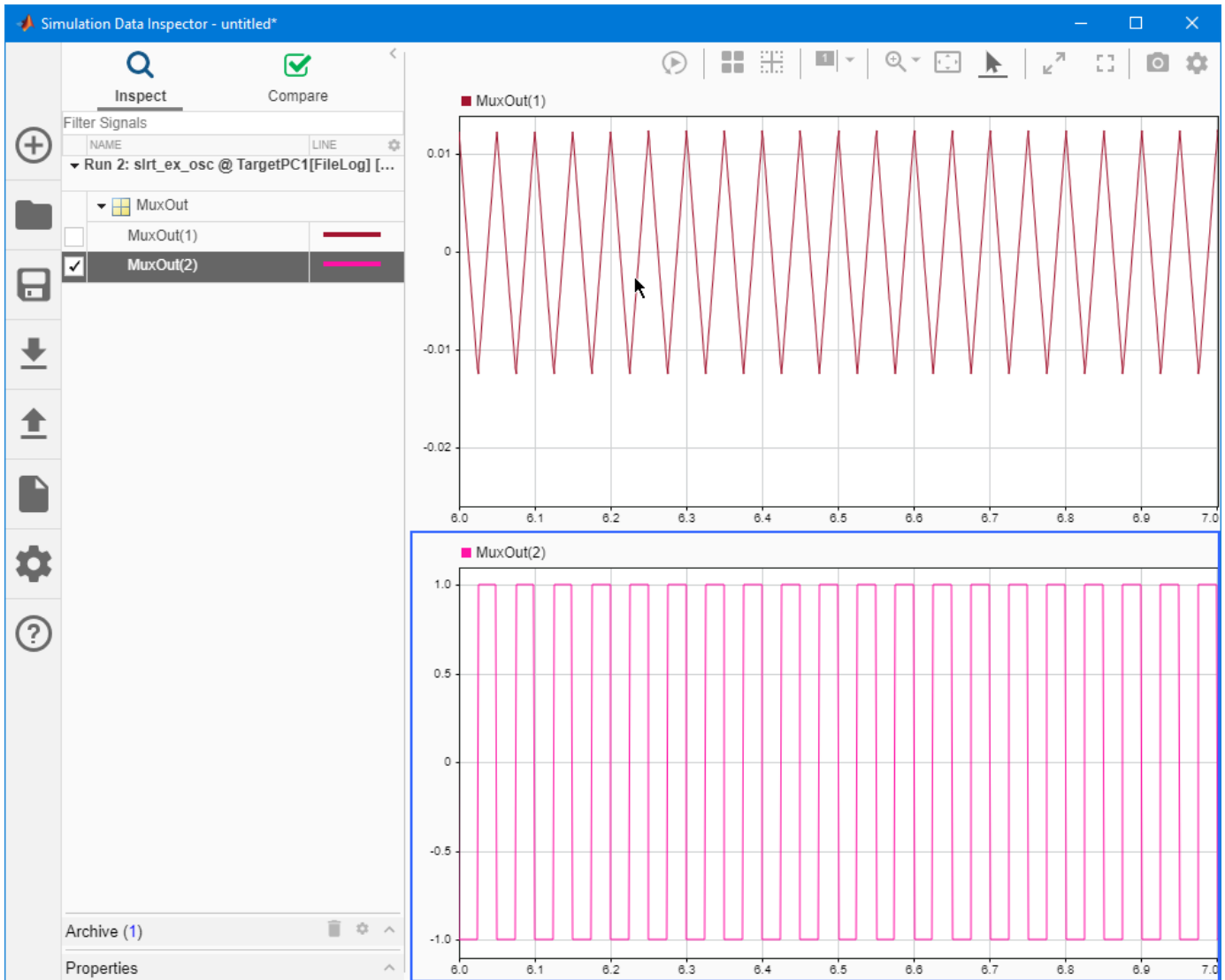
When the Simulation Data Inspector button glows , click the top display and select the Sum output signal.

Click in the bottom display and select the Mux output signals.

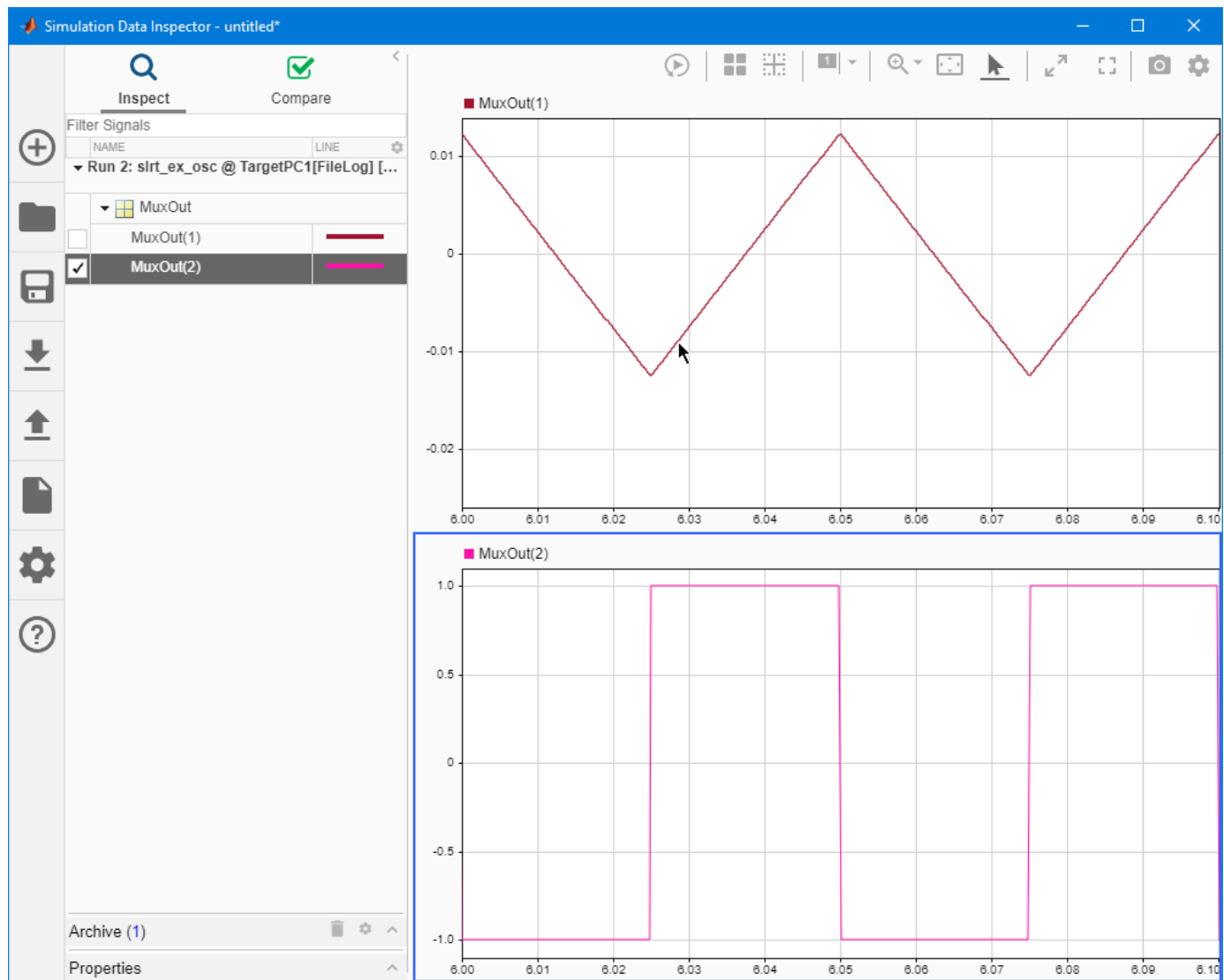


Stop real-time execution.

When the Sum output appears, click **Fit to View** (🔍 ↻).



To zoom in on a time segment of interest, for example, 10.0–10.1 s, click **Zoom in Time** (🔍) and use the mouse and mouse wheel.



To save the Simulation Data Inspector session as an MLDATX file, click **Save**.

See Also

More About

- “Data Logging with Simulation Data Inspector (SDI)” on page 7-14
- Simulation Data Inspector

External Mode Usage

When setting up signal triggering (Source set to signal), explicitly specify the element number of the signal in the **Trigger signal:Element** box. If the signal is a scalar, enter a value of 1. If the signal is a wide signal, enter a value from 1 to 10. When uploading Simulink Real-Time signals to Simulink scopes, do not enter Last or Any in this box.

The **Direction:Holdoff** value does not affect the Simulink Real-Time signal uploading feature.

See Also

More About

- “Trace Signals by Using Simulink External Mode” on page 7-11
- “Trace or Log Data with the Simulation Data Inspector” on page 7-21
- “Simulink External Mode Interface”

Signal Logging and Streaming Basics

Signal logging acquires signal data during a real-time run and stores it on the target computer. After you stop the real-time application, you transfer the data from the target computer to the development computer for analysis. You can plot and analyze the data, and later save it to a disk on the development computer.

Simulink Real-Time signal logging samples at the base sample time. You can log signals to the Simulation Data Inspector by:

- Mark signals for immediate logging to the Simulation Data Inspector.
- Connect signals to File Log blocks for buffered logging to the Simulation Data Inspector.

With regards to logging:

- Simulink Real-Time Explorer works with multidimensional signals in column-major format.
- Some signals are not observable.

Like signal logging, signal streaming also acquires signal data during a real-time run on the target computer. But, unlike signal logging that uses a File Log block or signals marked for logging, signal streaming uses an instrument that you add to the real-time application. You add signals to the instrument by using the **Real-Time** tab in the Simulink Editor or by selecting signals for streaming in the Simulink Real-Time Explorer. The streaming signal data transfers from the target computer to the development computer while the real-time application is running.

How Application is Run Affects Signals Logged

The **Run on Target** button provides slightly different data logging support than running the real-time application by using the `start(tg)` command:

- When you run the real-time application by using the `start(tg)` command, only signals marked for data logging or connected to a File Log block are logged to the Simulation Data Inspector.
- When you run the real-time application by using the **Run on Target** button on the real-time tab in the Simulink Editor or the **Start** button in the Simulink Real-Time Explorer, signals marked for logging, signals connected to File Log blocks, and signals connected to Scope blocks are logged to the Simulation Data Inspector.

File Logging and Streaming Workflow

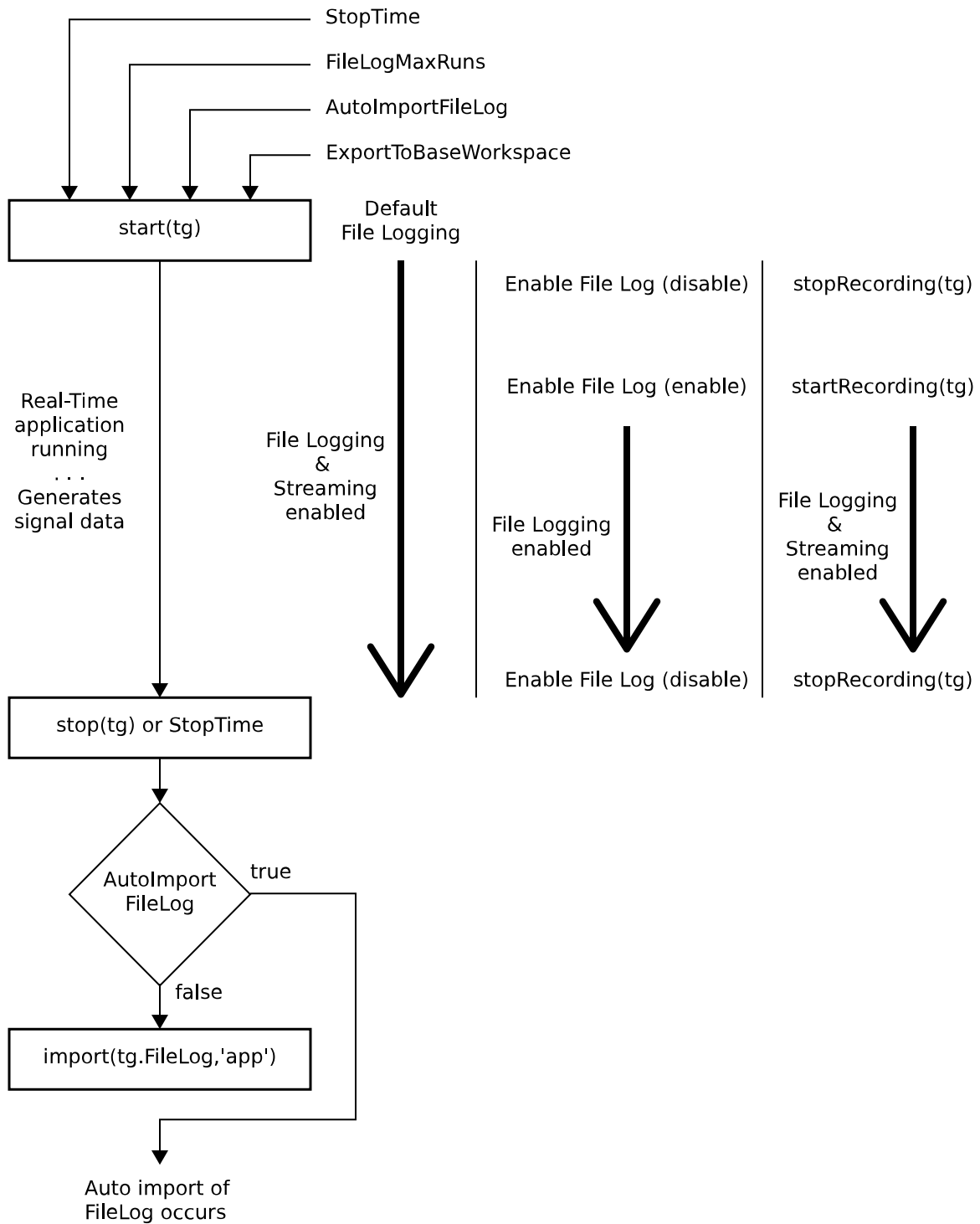
You can get signal data into the Simulation Data Inspector through logging by using a File Log block or through streaming by marking a signal for logging in the model or selecting a signal for streaming in the Simulink Real-Time Explorer.

Signal logging through a File Log block provides options that let you control:

- The number of file logs that are retained on the target computer
- Whether file log data is auto imported into the Simulation Data Inspector
- Whether file log data is exported into the base workspace

You can configure these options by using the option for the real-time application `start(tg)` function or by using the **Run in Real-Time** selection in the Simulink Real-Time Explorer or the Simulink

Editor. The **File Logging and Streaming Workflow** figure shows how these options configure operation of the real-time application `start(tg)` function. Where `startRecording` and `stopRecording` appear in the figure, you can use either these functions or the corresponding **Start Recording** and **Stop Recording** buttons on the **Real-Time** tab in the Simulink Editor or in the Simulink Real-Time Explorer.



File Logging and Streaming Workflow

While the real-time application is running, you can control file logging from File Log blocks:

- Default logging logs signal data for the entire simulation run.
- Enable or disable file logging by using the Enable File Log block in the model. If the model includes an Enable File Log, the `startRecording` function and `stopRecording` function control only streaming, not logging.
- Enable or disable file logging by using the `startRecording` function or `stopRecording` function. These function also enable or disable streaming. Alternatively, you can use the **Start Recording** button and **Stop Recording** button on the **Real-Time** tab in the Simulink Editor or in the Simulink Real-Time Explorer.

After file logging stops, which occurs:

- By using the `stop(tg)` function
- By `StopTime` expiring
- By using the `stopRecording(tg)` function or **Stop Recording** button

The configuration of the `AutoImportFileLog` option selects whether file log data is auto imported into the Simulation Data Inspector or whether you use the `import(tg.FileLog)` function to import the data.

Auto import of the file log is handled differently by the workflows in the **File Logging and Streaming Workflow** figure:

- For **all the workflows**, the auto import operation occurs when the real-time application stops.
- For the **recording** workflow, the auto import operation also occurs when the `stopRecording` function is called.

See Also

File Log | Enable File Log | `import` | `start` | `stop` | `startRecording` | `stopRecording`

Related Examples

- “Data Logging with Simulation Data Inspector (SDI)” on page 7-14

More About

- “Troubleshoot Signals Not Accessible by Name” on page 7-70
- “Display and Filter Hierarchical Signals and Parameters” on page 7-66

Tune Parameters by Using Simulink Real-Time Explorer

You can use Simulink Real-Time Explorer to change parameters in your real-time application while it is running or between runs. You do not need to rebuild the Simulink model, set the Simulink interface to external mode, or connect the Simulink interface with the real-time application.

This procedure uses the model `slrt_ex_osc`.

Set Up the Simulation Data Inspector

Before tuning parameter values, set up the Simulation Data Inspector:

Open the Simulation Data Inspector ()

Click **Layout** ()

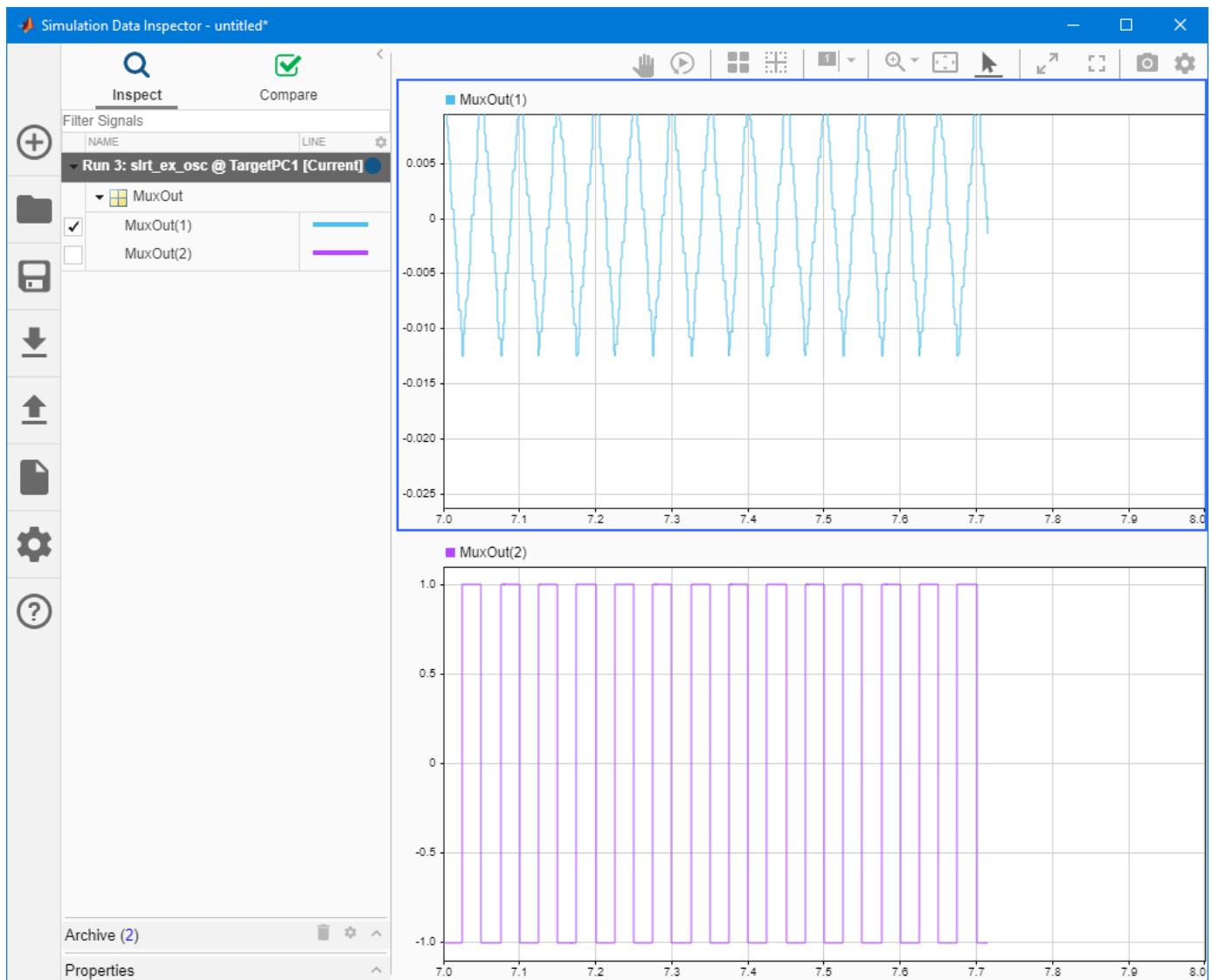
Select two horizontal displays.

Open model `slrt_ex_osc`. Set property **Stop time** to `inf`. In the Simulink Editor, on the **Real-Time** tab, select **Run on Target > Stop Time** and set **Stop Time** to `inf`.

Connect to the target computer. Toggle the **Disconnected** indicator to **Connected**.

Build and download the real-time application to the target computer. Click **Run on Target**.

In the Simulation Data Inspector, drag the **MuxOut(1)** signal to the top display and drag the **MuxOut(2)** signal to the bottom display.



View Initial Parameter Values

To view the initial parameter values:

Open Simulink Real-Time Explorer. On the **Real-Time** tab, click **Prepare > SLRT Explorer**.

Select the **Parameters** tab. The tab lists parameters Amplitude, Frequency, A, and C with their values.

Modify Parameter Values

To update a parameter value:

Select the parameter value for the Amplitude parameter and change the value to 0.5.

Select the parameter value for the Frequency parameter and change the value to 15.

After each change, the signal display in the Simulation Data Inspector alters to match the effect of the parameter change. You can change multiple parameters at the same time by using The **Hold Updates** button. For more information, see “Tune Parameters by Using Hold Updates and Update All Parameters” on page 7-35.

See Also

More About

- “Simulink Real-Time Operation Modes”
- “Display and Filter Hierarchical Signals and Parameters” on page 7-66
- “Tune Parameters by Using Hold Updates and Update All Parameters” on page 7-35
- “Troubleshoot Parameters Not Accessible by Name” on page 7-72

Tune Parameters by Using MATLAB Language

To change block parameters, you can use the MATLAB functions. With these functions, you do not need to set the Simulink interface to external mode or connect the Simulink interface with the real-time application.

You can download parameters to the real-time application while it is running or between runs. You can change parameters in your real-time application without rebuilding the Simulink model and change them back to their original values by using Simulink Real-Time functions.

Note Simulink Real-Time does not support parameters of multiword data types.

Access Parameters by Using Application Object

This procedure uses the Simulink model `slrt_ex_osc`. You must have already created and downloaded the real-time application to the default target computer.

To create the target object and application object, in the MATLAB Command Window, type:

```
tg = slrealtime('TargetPC1');
app = slrealtime.Application('slrt_ex_osc');
```

The `Parameters` property of the Application object is a structure that includes a `BlockPath` and `BlockParameterName` for each parameter. To display the parameter name of the first of parameter in the real-time application, in the MATLAB Command Window, type:

```
app.Parameters(1).BlockParameterName
```

To change the gain for the Gain1 block, type:

```
pt = setparam(tg, 'Gain1', 'Gain', 800)
```

The `setparam` method returns a structure that stores the source information, the previous value, and the new value.

When you change parameters, the changed parameters in the target object are downloaded to the real-time application. The development computer displays this message:

```
pt =
    Source: {'Gain1' 'Gain'}
  OldValues: 400
  NewValues: 800
```

The real-time application runs. The plot frame updates the signals for the active scopes.

Stop the real-time application. In the Command Window, type:

```
stop(tg)
```

To reset to the previous values, type:

```
pt = setparam(tg, pt.Source{1}, pt.Source{2}, pt.OldValues)
pt =
```

```
Source: {'Gain1' 'Gain'}  
OldValues: 800  
NewValues: 400
```

See Also

More About

- “Simulink Real-Time Operation Modes”
- “Troubleshoot Parameters Not Accessible by Name” on page 7-72

Tune Parameters by Using Simulink External Mode

To connect your Simulink model to your real-time application, you use Simulink external mode simulation. The model becomes a user interface to your real-time application. Set up the Simulink interface in external mode to establish a communication channel between your Simulink model and your real-time application.

In Simulink external mode, when you change parameters in the Simulink model, Simulink downloads those parameters to the real-time application while it is running. You can change parameters in your program without rebuilding the Simulink model to create a new real-time application.

Note Simulink Real-Time does not support parameters of multiword data types.

Tune Parameters by Using Block Diagram

After you download your real-time application to the target computer, you can connect your Simulink model to the real-time application. This procedure uses the Simulink model `slrt_ex_osc`. You must have already built and downloaded the real-time application for that model.

Open model `slrt_ex_osc`.

Connect to the target computer. On the **Real-Time** tab, toggle the **Disconnected** indicator to **Connected**.

Build and download the real-time application to the target computer. Click **Run on Target**.

The real-time application begins running on the target computer.

From the Simulation block diagram, double-click the block labeled **Gain1**

In the Block Parameters: Gain1 parameter dialog box, in the **Gain** text box, enter 800. Click **OK**.

When you change a MATLAB variable and click **OK**, the changed parameters in the model are downloaded to the real-time application.

To stop the simulation, click **Stop**.

Disconnect to the target computer. Toggle the **Connected** indicator to **Disconnected**.

The Simulink model is disconnected from the real-time application. If you then change a block parameter in the Simulink model, the real-time application does not change.

Tune Parameters by Using Hold Updates and Update All Parameters

By using the **Hold Updates** button, you can tune multiple parameters and apply the tuning changes at once by using **Update All Parameters**, instead of tuning one parameter at a time. This example uses model `slrt_ex_osc`.

Open model `slrt_ex_osc`. in the MATLAB Command Window, type:

```
open_system(fullfile(matlabroot, 'toolbox', 'slrealtime', ...  
'examples', 'slrt_ex_osc'))
```

In the Simulink Editor, on the **Real-Time** tab, click **Run on Target**.

Click **Prepare > Hold Updates**. The editor holds parameter updates until you click **Hold Updates** again.

To set parameter values, you can set values either by clicking each block or by using the Model Data Editor in the base workspace.

On the **Real-Time** tab, click **Prepare > Signal Table**.

In the Model Data Editor, click the **Parameters** tab. Modify parameters values in the Model Data Editor in the base workspace.

Click **Prepare > Update All Parameters**.

To stop the simulation before it ends, click **Stop**.

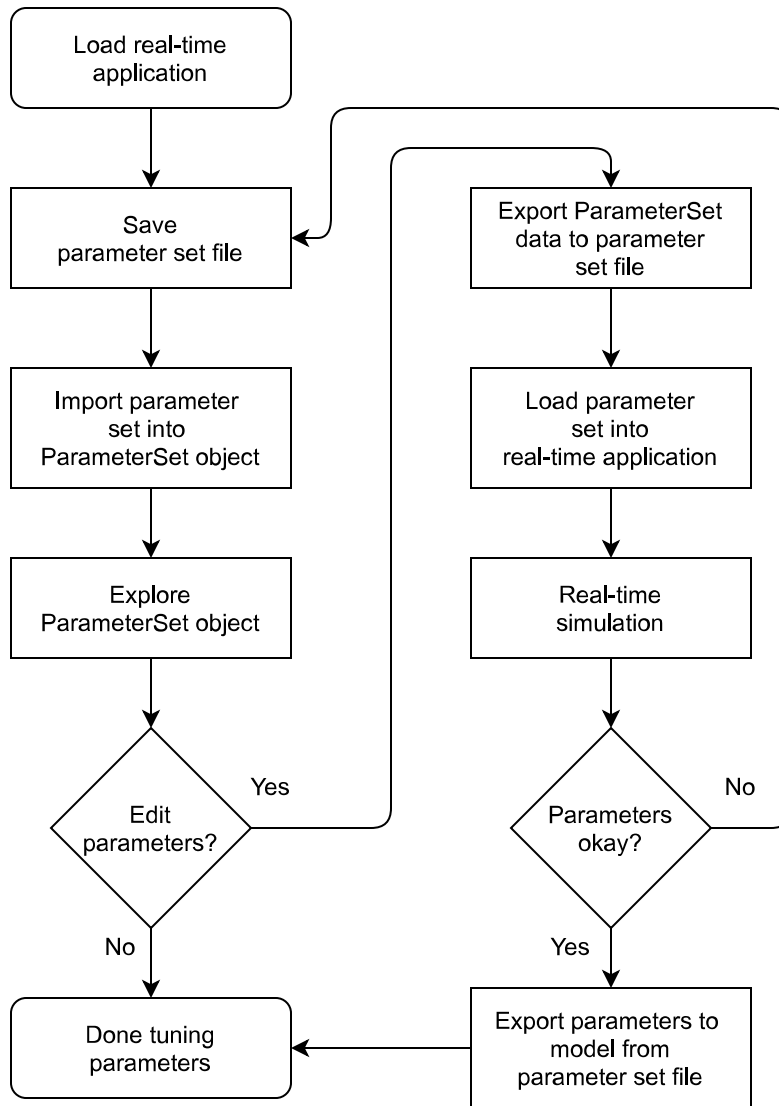
See Also

More About

- “Simulink Real-Time Operation Modes”
- “Troubleshoot Parameters Not Accessible by Name” on page 7-72

Save and Reload Parameters by Using the MATLAB Language

After you load a real-time application that has parameter values, you can save those values to a parameter set file on the target computer. You can then later reload these parameter values to the same real-time application. To ease the process of tuning parameters, use the `ParameterSet` object workflow. For example code that demonstrates this workflow, see the `ParameterSet` object.



When your parameter set values are tuned, you can:

- Export the values from the parameter values to the model by using `exportToModel` function.
- Save the `ParameterSet` object as a MAT file and share this MAT file with other developers.
- Add the parameter set into the real-time application MLDATX file and set the parameter set as the startup parameter set by using the `addParamSet` and `updateStartupParameterSet` functions.

You can save parameters from your real-time application while the real-time application is running or between runs. You can save and restore parameters in your real-time application without rebuilding

the Simulink model. Load parameters to the same real-time application from which you saved the parameter file. If you attempt to load a parameter file to a different real-time application or to a real-time application that has changed since the parameter set was created, the load issues an error.

You can use the `syncWithApp` function to synchronize an out-of-sync parameter set object with the specified real-time application. This function synchronizes the parameter name-value pairs and synchronizes the model checksum saved in the parameter set object with the real-time application. After synchronizing, the parameter set that you saved from the original application can be loaded into the most updated application on the target computer.

You save and restore parameters by using the target object methods `saveParamSet` and `loadParamSet`.

Requirements:

- Create a Target object named `tg` connected to the target computer.
- Load a real-time application on the target computer.
- There are parameters to save from the application.

Save Current Set of Real-Time Application Parameters

To save a set of parameters from a real-time application to a parameter set file, use the `saveParamSet` function. The real-time application can be loaded or running.

This example uses the model `slrt_ex_osc_outport`. To open this model, in the MATLAB Command Window, type:

```
open_system((fullfile(matlabroot,'toolbox','slrealtime', ...  
    'examples','slrt_ex_osc_outport')))
```

Select a descriptive file name for the parameters. For example, use the model name in the file name.

In the MATLAB Command Window, type:

```
% build model and load real-time application  
mdlName = 'slrt_ex_osc_outport';  
slbuild(mdlName);  
tg = slrealtime('TargetPC1');  
load(tg,mdlName);  
  
% save parameter set to file  
paramSetName = 'outportTypes';  
saveParamSet(tg,paramSetName);
```

The Simulink Real-Time software creates a parameter set file named `outportTypes` on the target computer.

Load Saved Parameters to Real-Time Application

To load a parameter set file of saved parameters to a real-time application, use the `loadParamSet` function. Load parameters to the same real-time application from which you save the parameter set file. If you attempt to load a parameter file to a different real-time application or to the a real-time application that has changed since the parameter set was created, the load issues an error. This

example uses the model `slrt_ex_osc_outport`. You must have a parameters file saved from an earlier run of `saveParamSet` to perform this procedure.

To open this model, in the MATLAB Command Window, type:

```
open_system((fullfile(matlabroot,'toolbox','slrealtime', ...
    'examples','slrt_ex_osc_outport')))
```

From the collection of parameter set files on the target computer, select the one that contains the parameter values to load. To get a list of available parameter set files, use the `listParamSet` function.

In the Command Window, type:

```
% load real-time application
mdlName = 'slrt_ex_osc_outport';
tg = slrealtime('TargetPC1');
load(tg,mdlName);

% load parameter set file
paramSetName = 'outportTypes';
loadParamSet(tg,paramSetName);
```

The Simulink Real-Time software loads the parameter values into the real-time application. For an example that shows how to get the parameter values from a `ParameterSet` object and use the object in the `loadParamSet` function, see `loadParamSet`.

View or Edit Parameter Values in Parameter Set

To view or edit parameters in a parameter set, use the `ParameterSet` object workflow. For more information about this workflow, see “Save and Reload Parameters by Using the MATLAB Language” on page 7-37.

Build the model and load the real-time application.

```
mdlName = 'slrt_ex_osc_outport';
slbuild(mdlName);
tg = slrealtime('TargetPC1');
load(tg,mdlName);
```

Save the parameter set to a file.

```
paramSetName = 'outportTypes';
saveParamSet(tg,paramSetName);
```

Import the parameter set into a `ParameterSet` object on the development computer.

```
myParamSet = importParamSet(tg,paramSetName);
```

Open the `ParameterSet` in the Simulink Real-Time Parameter Explorer UI. In the explorer, you can view and edit the parameter values in the object.

```
explorer(myParamSet);
```

After tuning the parameters, export the modified parameter set to the target computer and load the parameters into the real-time application.

```
exportParamSet(tg,myParamSet);  
loadParamSet(tg,myParamSet.filename);
```

Add or Update Startup Parameter Set for Application

You can add a `ParameterSet` object to a real-time application by using the `addParamSet` function. After adding one or more `ParameterSet` objects to an application by using the `addParamSet` function, you can choose which of these parameter sets is loaded into the real-time application on startup by using the `updateStartupParameterSet` function.

This example loads a real-time application, imports the parameters into a `ParameterSet` object, adds the `ParameterSet` object to a real-time application, and selects the parameter set as the startup parameter set for the application.

```
load(tg,mdlName);  
paramSetName = 'outportTypes';  
saveParamSet(tg,paramSetName);  
myParamSet = importParamSet(tg,paramSetName);  
addParamSet(app_object,myParamSet);  
updateStartupParameterSet(app_object,myParamSet);
```

See Also

`delete` | `explorer` | `exportToModel` | `set` | `syncWithApp` | `exportParamSet` | `getParam` | `getParameters` | `importParamSet` | `listParamSet` | `loadParamSet` | `saveParamSet` | `setparam` | `addParamSet` | `updateStartupParameterSet` | `Application` | `ParameterSet` | `Target`

More About

- “Tune Parameters by Using Simulink Real-Time Explorer” on page 7-30
- “Tune Parameters by Using MATLAB Language” on page 7-33
- “Tune Parameters by Using Simulink External Mode” on page 7-35
- “Troubleshoot Instance-Specific Parameters Not Saved” on page 7-73

Tunable Block Parameters and Tunable Global Parameters

To change the behavior of a real-time application, you can tune Simulink Real-Time tunable parameters. In Simulink external mode, you can change the parameters directly in the block or indirectly by using MATLAB variables to create tunable global parameters. Simulink Real-Time Explorer and the MATLAB language enable you to change parameter values and MATLAB variables as your real-time application is executing.

Note Simulink Real-Time does not support parameters of multiword data types.

Tunable Parameters

Simulink Coder defines two kinds of parameters that can be modified during execution: tunable block parameters and tunable global parameters. Simulink Real-Time support for tunable parameters includes:

- Variables for block parameters that are present in the top model workspace or MATLAB base workspace. These variables are tunable global parameters.
- Literal expressions for block parameters that are present in the top model workspace or data dictionary. These expressions are tunable block parameters.
- Instance-specific block parameters that are present in referenced models. These parameters are tunable global parameters.

Tunable Block Parameters

A tunable block parameter is a literal expression in the top model workspace or data dictionary that you reference in a Simulink block dialog box.

Suppose that you assign the value 5/2 to the **Amplitude** parameter of a Signal Generator block. **Amplitude** is a tunable parameter.

Tunable Global Parameter

A tunable global parameter is a variable in the top model workspace or MATLAB base workspace that you reference in a Simulink block dialog box. Suppose that you enter **A** in the **Amplitude** parameter of a Signal Generator block. Variable **A** is a tunable parameter. You can tune the values of MATLAB variables that are grouped in a parameter structure. For example:

Assign a parameter structure that contains the field `Ampl` to variable **A**.

Enter `A.Ampl` in the **Amplitude** parameter of a Signal Generator block.

Change the amplitude of the signal generator by tuning the value of `A.Ampl` in the MATLAB workspace during simulation.

Tunable Global Parameters from Referenced Models

There are some limitations on tuning parameters in referenced models. For more information about using instance-specific block parameters and using model arguments to configure these, see:

- “Limitations for Block Parameter Tunability in Generated Code”

- “Specify Instance-Specific Parameter Values for Reusable Referenced Model”
- “Parameterize a Referenced Model Programmatically”

Inlined Parameters

To optimize execution efficiency, you can change the **Default parameter behavior** option from Tunable to Inlined on the **Code Generation > Optimization** pane.

You cannot tune inlined block parameters. You can define a tunable global parameter or `Simulink.Parameter` object, enter it in the parameter field in the block dialog box, and tune the MATLAB variable or object.

For more information about inlined parameters, see Default parameter behavior.

Tune Global Parameters by Using External Mode

In external mode, Simulink Real-Time connects your Simulink model to your real-time application. The block diagram becomes a user interface for the real-time application.

You can change a block parameter value during execution in the block dialog box. When you click **OK**, Simulink transfers the new value to the real-time application. For more information, see “Tune Parameters by Using Simulink External Mode” on page 7-35.

You can change a tunable global parameter during execution by assigning a new value to the MATLAB workspace. You must then explicitly command Simulink to transfer the data. Do one of the following:

- Press **Ctrl+D**.
- On the **Real-Time** tab, click **Prepare > Signal Table**. On the **Parameters** tab, edit the parameters and click **Update Diagram**.

Tune Global Parameters by Using Simulink Real-Time Explorer

During real-time execution, Simulink Real-Time Explorer becomes a user interface for the real-time application.

To access a block parameter value, navigate to the block in the Explorer model hierarchy. You can change the value in a text entry box in the parameter window. When you apply the new value, Simulink Real-Time transfers the new value to the real-time application. For more information, see “Tune Parameters by Using Simulink Real-Time Explorer” on page 7-30.

You can access a tunable global parameter at the top level of the model hierarchy. Change it the same way as you would a tunable block parameter.

You can use Simulink Real-Time Explorer instrument panels to tune block parameters and global parameters.

Tune Global Parameters by Using MATLAB Language

To change the values of tunable block parameters and tunable global parameters during execution, use the Simulink Real-Time command `setparam`. For more information, see “Tune Parameters by Using MATLAB Language” on page 7-33.

These code examples use the model `slrt_ex_osc`. To change a block parameter value, use a nonempty block path and the parameter name. For example, to change the amplitude of the signal generator:

```
slbuild(slrt_ex_osc);
tg = slrealtime('TargetPC1');
load(tg, 'slrt_ex_osc')
start(tg);
setparam(tg, 'Signal Generator', 'Amplitude', 4.57)
```

To change a tunable global parameter, use the variable name. For example, to change the amplitude of the signal generator via the parameter structure field `A.Ampl`:

```
slbuild(slrt_ex_osc);
tg = slrealtime('TargetPC1');
load(tg, 'slrt_ex_osc')
start(tg);
setparam(tg, '', 'A.Ampl', 4.57)
```

See Also

[setparam](#) | [getparam](#)

More About

- “Tune Inlined Parameters by Using Simulink Real-Time Explorer” on page 7-44
- Default parameter behavior
- “Specify Source for Data in Model Workspace”
- “Troubleshoot Parameters Not Accessible by Name” on page 7-72
- “Tune and Experiment with Block Parameter Values”
- “Share and Reuse Block Parameter Values by Creating Variables”
- “How Generated Code Stores Internal Signal, State, and Parameter Data”
- “Preserve Variables in Generated Code”

Tune Inlined Parameters by Using Simulink Real-Time Explorer

This procedure describes how you can tune inlined parameters through the Simulink Real-Time Explorer.

Note Simulink Real-Time does not support parameters of multiword data types.

The procedure starts with the Simulink model `slrt_ex_osc_inlined`. To open the model, in the MATLAB Command Window, type:

```
open_system(fullfile(matlabroot, 'toolbox', 'slrealtime', ...  
'examples', 'slrt_ex_osc_inlined'))
```

Configure Model to Tune Inlined Parameters

This procedure makes the `Amplitude` parameter of the Signal Generator block tunable.

Open model `slrt_ex_osc_inlined`.

In the Simulink Editor, select the input to the Scope block and mark it for data logging by using the Simulation Data Inspector.

Select the blocks containing the parameters that you want to tune. To represent the amplitude, use the variable `A`.

- a** Double-click the Signal Generator block, and then enter `A` for the `Amplitude` parameter. Click **OK**.
- b** Assign a constant to variable `A`. In the MATLAB Command Window, type:

```
A = 4
```

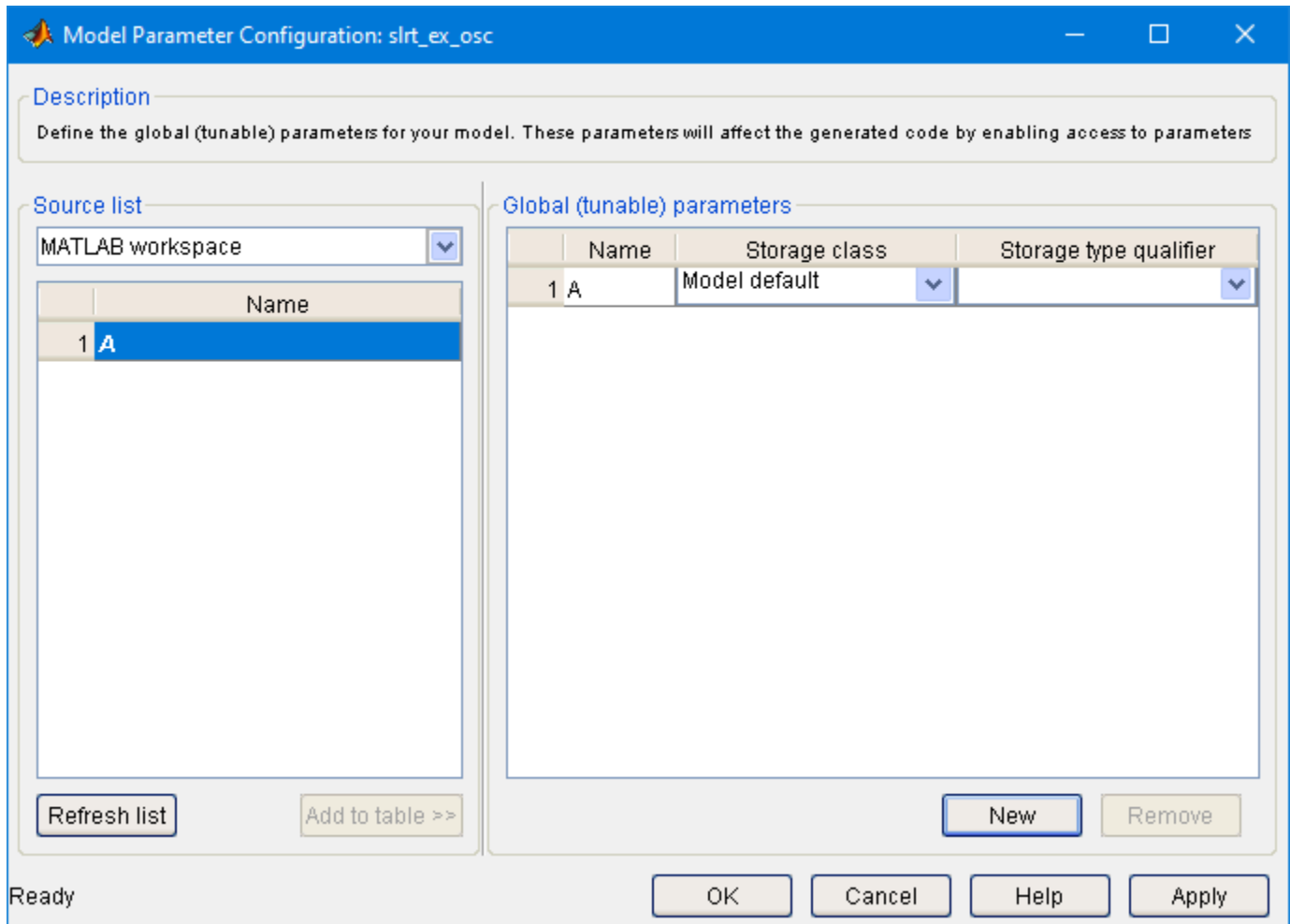
The value is displayed in the MATLAB workspace.

Open the Configuration Parameters dialog box. On the **Real-Time** tab, click **Hardware Settings**.

Select **Code Generation > Optimization > Default parameter behavior > Inlined**.

Click **Configure**. The Model Parameter Configuration dialog box opens. The MATLAB workspace contains the constant you assigned to `A`.

Select the line that contains your constant. Click **Add to table**.



Click **Apply**, and then click **OK**.

In the Configuration Parameters dialog box, click **Apply**, and then **OK**.

Save the model as `slrt_ex_osc_inlined`. On the **Simulation** tab, from **Save**, click **Save As**. For example, save it as `slrt_ex_osc_inlined`.

Build and download the model to your target computer. On the **Real-Time** tab, click **Run on Target**.

Initial Value

This procedure assumes that you have completed the steps in “Configure Model to Tune Inlined Parameters” on page 7-44.

Open Simulink Real-Time Explorer. On the **Real-Time** tab, click **Prepare > SLRT Explorer**.

Load the `slrt_ex_osc_inlined` real-time application. Click **Load Application**, select the application, and click **Load**.

Set the application stop time to **inf**.

To start execution, click **Start**.

In the **Applications** pane, expand both the real-time application node and the **Model Hierarchy** node.

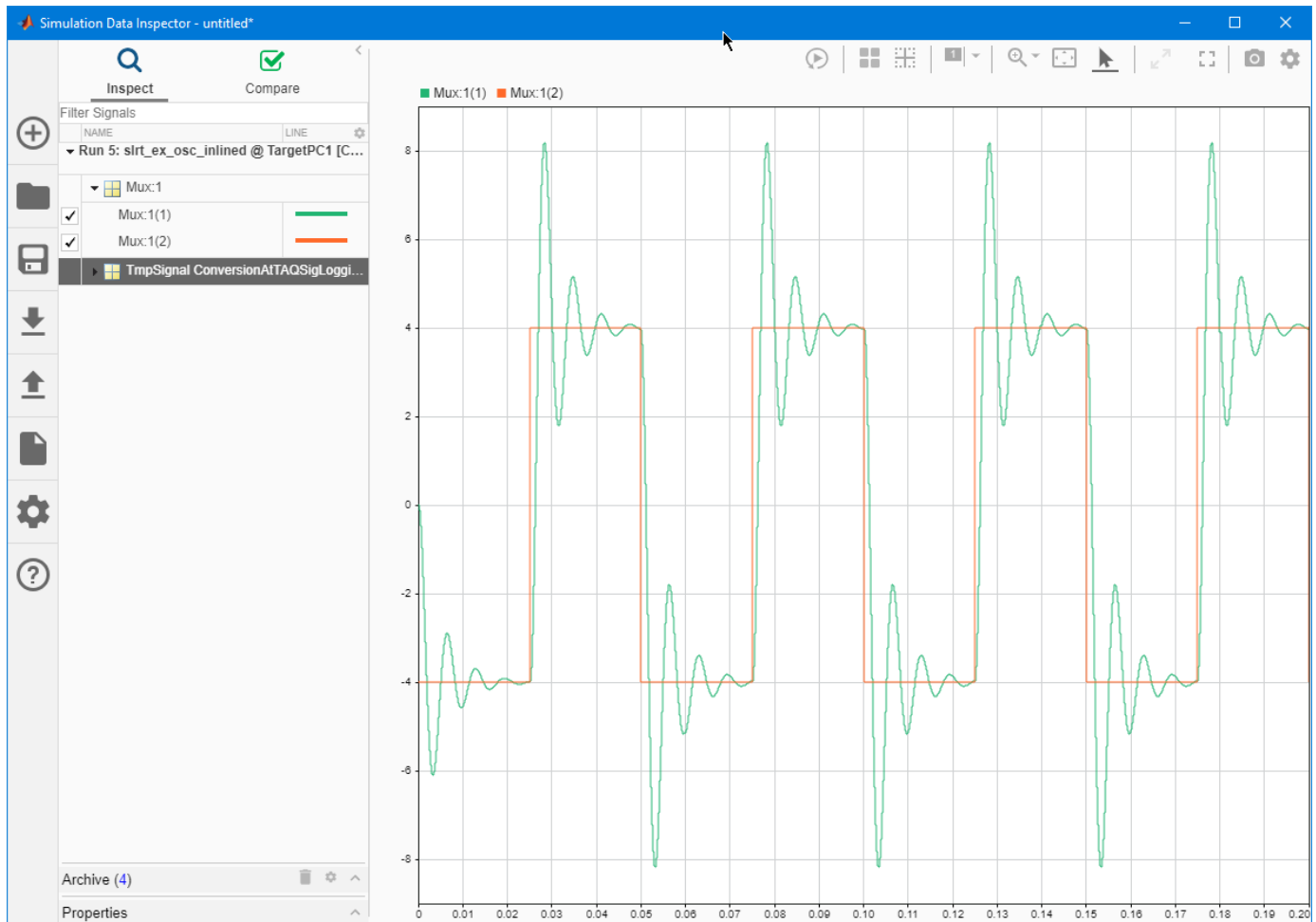
Select the **Parameters** tab.

The screenshot shows the Simulink Real-Time Explorer window. The interface includes a toolbar with buttons for 'Load Application', 'Start', 'Stop Time', 'Hold Updates', 'Update All Parameters', and 'REVIEW RESULTS'. The 'Parameters' tab is selected, displaying a table of parameters available to tune on the target computer.

Block Path	Name	Value	Type	Size
	A	4	double	[1 1]

At the bottom of the interface, the status bar indicates 'LOADED: slrt_ex_osc_inlined'.

Open the Simulation Data Inspector and view the signals you marked for signal logging. On the **Real-Time** tab, click **Data Inspector**.



Updated Value

This procedure assumes that you have completed the steps in “Initial Value” on page 7-45.

Change the value of the MATLAB variable A to 2. In Simulink Real-Time Explorer, type 2 into the **Value** box, and then press **Enter**.

The Simulation Data Inspector display changes to show the new signal amplitude.

To stop execution, click **Stop**.

See Also

More About

- “Tune Inlined Parameters by Using MATLAB Language” on page 7-48
- “Display and Filter Hierarchical Signals and Parameters” on page 7-66
- “Troubleshoot Parameters Not Accessible by Name” on page 7-72

Tune Inlined Parameters by Using MATLAB Language

You can tune inlined parameters through the MATLAB interface.

Note Simulink Real-Time does not support parameters of multiword data types.

You must have already built and downloaded the model `slrt_ex_osc_inlined`. To open this model, in the MATLAB Command Window, type:

```
open_system(fullfile(matlabroot, 'toolbox', 'slrealtime', ...  
'examples', 'slrt_ex_osc_inlined'))
```

Tune Inlined Parameter

With the real-application `slrt_ex_osc_inlined` already running, you can tune inlined parameter A by using the `setparam` function.

Save the following code in a MATLAB file. For example, `change_inlineA`.

```
A = 4;  
open_system(fullfile(matlabroot, 'toolbox', 'slrealtime', ...  
'examples', 'slrt_ex_osc_inlined'));  
slbuild('slrt_ex_osc_inlined');  
tg = slrealtime;  
load(tg, 'slrt_ex_osc_inlined');  
setparam(tg, 'A', 2);
```

Execute that MATLAB file. Type:

```
change_inlineA
```

To see the new parameter value, type:

```
getparam(tg, 'A')
```

See Also

More About

- “Troubleshoot Parameters Not Accessible by Name” on page 7-72

Tune Parameter Structures by Using Simulink Real-Time Explorer

In this section...

“Create Parameter Structure” on page 7-49

“Replace Block Parameters with Parameter Structure Fields” on page 7-50

“Save and Load Parameter Structure” on page 7-50

“Tune Parameters in a Parameter Structure” on page 7-51

To reduce the number of workspace variables you must maintain and avoid name conflicts, you can group closely related parameters into structures. See “Organize Related Block Parameter Definitions in Structures”.

In this example, the initial model `slrt_ex_osc` has four parameters that determine the shape of the output waveform.

Block	Parameter	Structure Field Expression	Initial Value
Signal Generator	Freq	<code>spkp.sg_freq</code>	20
Gain	Gain	<code>spkp.g_gain</code>	1000^2
Gain1	Gain	<code>spkp.g1_gain</code>	$2*0.2*1000$
Gain2	Gain	<code>spkp.g2_gain</code>	1000^2

Create Parameter Structure

This procedure groups some closely related parameters into structures.

Open model `slrt_ex_osc`, and save a copy of the model to a working folder.

Open the Base Workspace in the Model Explorer. On the **Modeling** tab, click **Base Workspace**.


Click **Add Simulink Parameter** .

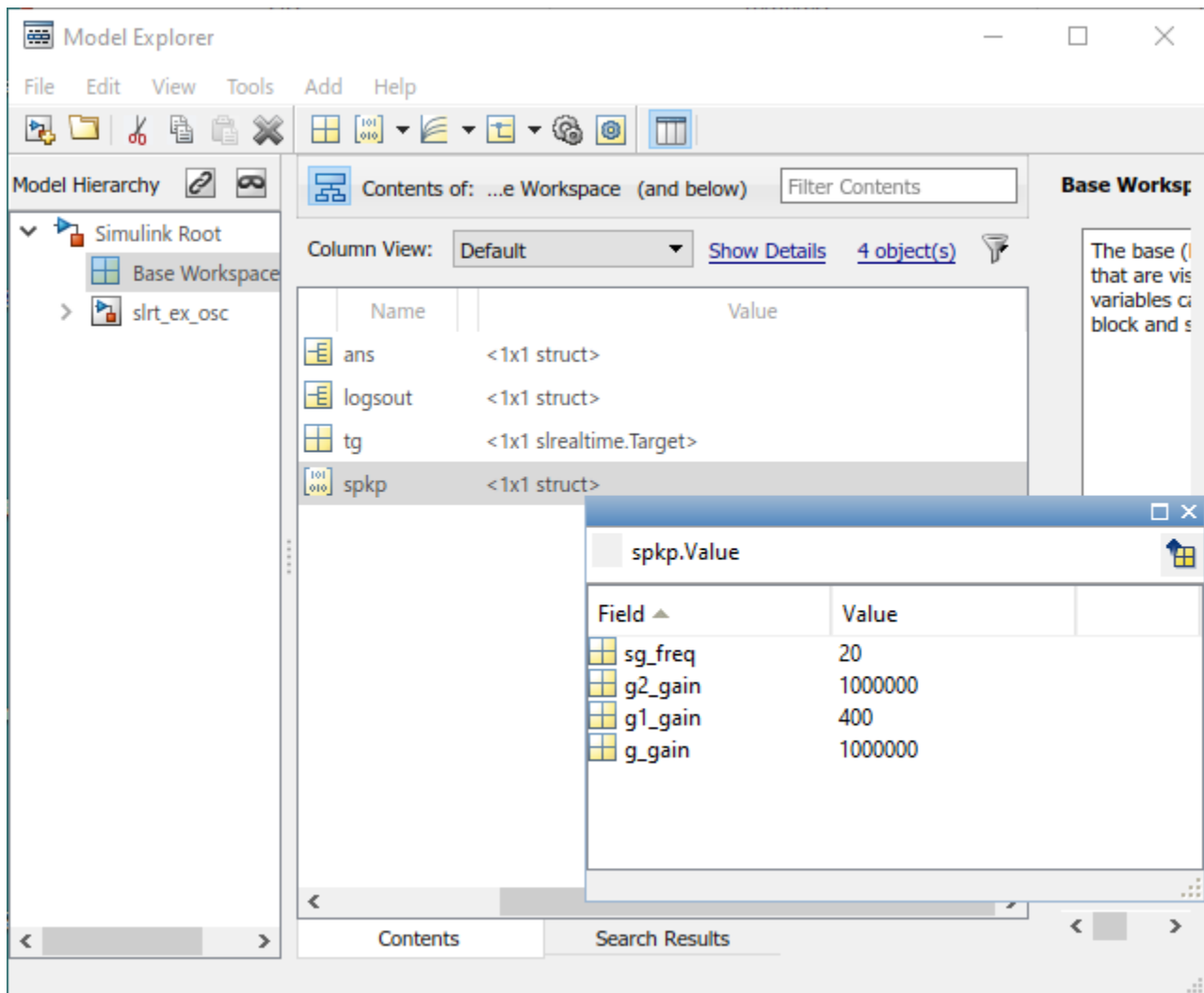
In the **Name** column, type the name `spkp`.

In the **Storage class** field, select `ExportedGlobal`.

In the **Value** field, type as one line:

```
struct('sg_freq',20, 'g2_gain',1000^2, 'g1_gain',2*0.2*1000, 'g_gain',1000^2)
```

The field values duplicate the literal values in the dialog boxes. To change the field values, in row `spkp`, click the **Value** cell and click **Edit** .



Click **Apply**.

Save the model as `slrt_ex_osc_struct`. On the **Simulation** tab, from **Save**, click **Save As**.

Replace Block Parameters with Parameter Structure Fields

- 1 In the Signal Generator block, replace the value of parameter **Frequency** with `spkp.sg_freq`.
- 2 In the Gain block, replace the value of parameter **Gain** with `spkp.g_gain`.
- 3 In the Gain1 block, replace the value of parameter **Gain** with `spkp.g1_gain`.
- 4 In the Gain2 block, replace the value of parameter **Gain** with `spkp.g2_gain`.

Save and Load Parameter Structure

- 1 In Model Explorer, right-click row `spkp`.
- 2 Click **Export selected** and save the variable as `slrt_ex_osc_struct.mat`.

To load the parameter structure when you open the model, add a `load` command to the `PreLoadFcn` callback. To remove the parameter structure from the workspace when you close the model, add a `clear` command to the `CloseFcn` callback. For more information, see “Model Callbacks”.

Tune Parameters in a Parameter Structure

If you have not completed the steps in “Create Parameter Structure” on page 7-49, “Replace Block Parameters with Parameter Structure Fields” on page 7-50, and “Save and Load Parameter Structure” on page 7-50, you can start by using the completed model.

To open the model, in the MATLAB Command Window, type:

```
open_system(fullfile(matlabroot, 'toolbox', 'slrealtime', ...  
'examples', 'slrt_ex_osc_struct'));  
load(fullfile(matlabroot, 'toolbox', 'slrealtime', ...  
'examples', 'slrt_ex_osc_struct.mat'));
```

Build and download the model to your target computer.

Open Simulink Real-Time Explorer. In the **Real-Time** tab, click **Prepare > SLRT Explorer**.

Set the real-time application **Stop Time** to **Inf**.

Click the **Parameters** tab.

Start the real-time application.

Open the Simulation Data Inspector and view the signals from the real-time application.

In the **Values** text box for `spkp(1).g1_gain`, change the value to **800** and press **Enter**.

Observe the change to the signals in the Simulation Data Inspector.

Stop the real-time application.

See Also

More About

- “Organize Related Block Parameter Definitions in Structures”
- “Display and Filter Hierarchical Signals and Parameters” on page 7-66
- “Model Callbacks”

Tune Parameter Structures by Using MATLAB Language

In this section...

“Create Parameter Structure” on page 7-52

“Save and Load Parameter Structure” on page 7-53

“Replace Block Parameters with Parameter Structure Fields” on page 7-53

“Tune Parameters in a Parameter Structure” on page 7-53

To reduce the number of workspace variables you must maintain and avoid name conflicts, you can group closely related parameters into structures. See “Organize Related Block Parameter Definitions in Structures”.

In this example, the initial model `slrt_ex_osc` has four parameters that determine the shape of the output waveform.

Block	Parameter	Structure Field Expression	Initial Value
Signal Generator	Freq	<code>spkp.sg_freq</code>	20
Gain	Gain	<code>spkp.g_gain</code>	1000^2
Gain1	Gain	<code>spkp.g1_gain</code>	$2*0.2*1000$
Gain2	Gain	<code>spkp.g2_gain</code>	1000^2

Create Parameter Structure

This procedure groups some closely related parameters into structures.

Open model `slrt_ex_osc` and save a copy to a working folder.

To create a parameter structure, in the MATLAB Command Window, enter:

```
kp = struct(...
    'sg_freq', 20, ...
    'g2_gain', 1000^2, ...
    'g1_gain', 2*0.2*1000, ...
    'g_gain', 1000^2)
```

```
kp =
```

```
struct with fields:
```

```
sg_freq: 20
g2_gain: 1000000
g1_gain: 400
g_gain: 1000000
```

To make the parameter structure tunable on the target computer:

```
spkp = Simulink.Parameter(kp);
spkp.StorageClass = 'ExportedGlobal';
spkp.Value
```



```
ans =
    struct with fields:
    sg_freq: 20
    g2_gain: 1000000
    g1_gain: 400
    g_gain: 1000000
```

Save and Load Parameter Structure

To save the parameter structure `spkp` for later use, type:

```
save 'slrt_ex_osc_struct.mat', 'spkp'
```

To load the parameter structure when you open the model, add a `load` command to the `PreLoadFcn` callback. To remove the parameter structure from the workspace when you close the model, add a `clear` command to the `CloseFcn` callback. For more information, see “Model Callbacks”.

Replace Block Parameters with Parameter Structure Fields

- 1 In the Signal Generator block, replace the value of parameter **Frequency** with `spkp.sg_freq`.
- 2 In the Gain block, replace the value of parameter **Gain** with `spkp.g_gain`.
- 3 In the Gain1 block, replace the value of parameter **Gain** with `spkp.g1_gain`.
- 4 In the Gain2 block, replace the value of parameter **Gain** with `spkp.g2_gain`.

Tune Parameters in a Parameter Structure

If you have not completed the steps in “Create Parameter Structure” on page 7-52, “Replace Block Parameters with Parameter Structure Fields” on page 7-53, and “Save and Load Parameter Structure” on page 7-53, you can start by using the completed model.

To open the model, in the MATLAB Command Window, type:

```
open_system(fullfile(matlabroot, 'toolbox', 'slrealtime', ...
'examples', 'slrt_ex_osc_struct'));
load(fullfile(matlabroot, 'toolbox', 'slrealtime', ...
'examples', 'slrt_ex_osc_struct.mat'));
```

Build and download the model to the target computer.

```
slbuild('slrt_ex_osc_struct');
tg = slrealtime('TargetPC1');
load(tg, 'slrt_ex_osc_struct');
```

Set stop time to `inf`.

```
setStopTime(tg, inf);
```

Sweep the **Gain** value of the Gain1 block from 200 to 800.

```
start(tg);
for g = 200 : 200 : 800
    setparam(tg, '', 'spkp.g1_gain', g);
```

```
        pause(1);  
end  
stop(tg);
```

View the signals sent to the Scope block in the Simulation Data Inspector.

```
Simulink.sdi.view;
```

See Also

More About

- “Organize Related Block Parameter Definitions in Structures”
- “Model Callbacks”

Define and Update Inport Data

In this section...

“Required Files” on page 7-55

“Map Inport to Use Square Wave” on page 7-55

“Update Inport to Use Sawtooth Wave” on page 7-57

You can create root-level input ports and use the Root Inport Mapper to define input data. You can update the input data without rebuilding the model by using the MATLAB language.

Required Files

This procedure has these file dependencies:

- `slrt_ex_osc_inport` — Damped oscillator that takes its input data from input port In1 and sends its multiplexed output to output port Out1. To open this model, in the MATLAB Command Window, type:

```
open_system(fullfile(matlabroot, 'toolbox', 'slrealtime', ...
    'examples', 'slrt_ex_osc_inport'))
```

- `slrt_ex_inport_square.mat`— One second of output from a Signal Generator block that is configured to output a square wave. To load this data, in the MATLAB Command Window, type:

```
load(fullfile(matlabroot, 'toolbox', 'slrealtime', ...
    'examples', 'slrt_ex_inport_square.mat'))
```

- `slrt_ex_inport_sawtooth.mat` — One second of output from a Signal Generator block that is configured to output a sawtooth wave. To load this data, in the MATLAB Command Window, type:

```
load(fullfile(matlabroot, 'toolbox', 'slrealtime', ...
    'examples', 'slrt_ex_inport_sawtooth.mat'))
```

Before starting this procedure, navigate to a working folder.

Map Inport to Use Square Wave

This procedure uses the Root Inport Mapper.

Open model `slrt_ex_osc_inport` and save a copy to a working folder.

Load `slrt_ex_inport_square.mat` and assign `square` to a temporary workspace variable for use with the Root Inport Mapper.

```
waveform = square;
```

Double-click input port In1.

Clear **Interpolate data**, and then click **Connect Inputs**.

This example chooses not to interpolate the data because the time steps in the dataset are identical to the sample time in the model. If the model were to be run with a different sample time, consider whether to enable interpolation.

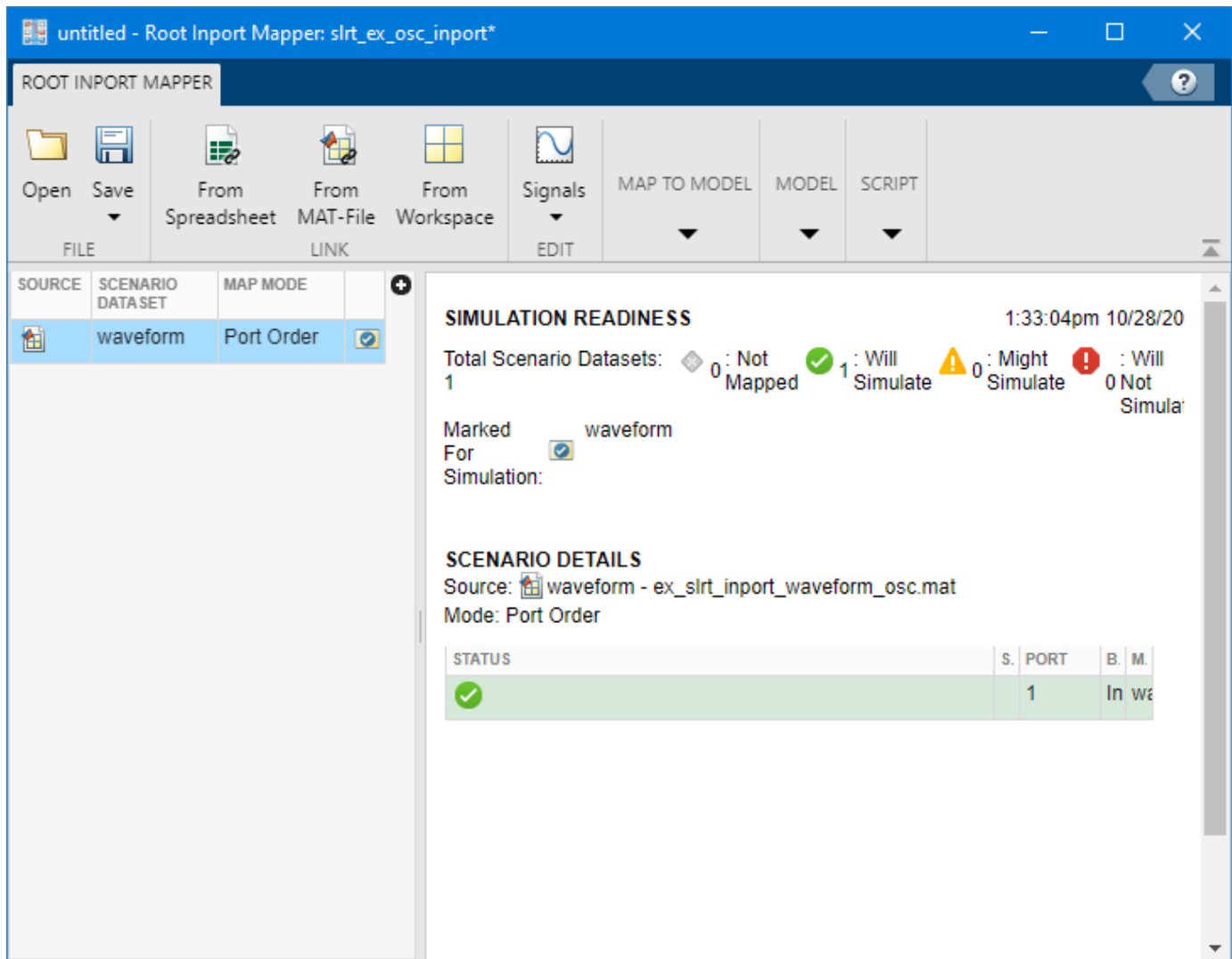
In the Root Inport Mapper, click **From Workspace** and select variable waveform. Clear the other variables.

In the **Save to** text box, enter a name such as `ex_slrt_inport_waveform_osc.mat`, and then click **OK**.

Select the map to model option **Port order** and, from the **Options** menu, select **Update Model**.

Click **Map to Model**.

To update the model with the mapped input data, select scenario waveform, and then click **Mark for Simulation**.



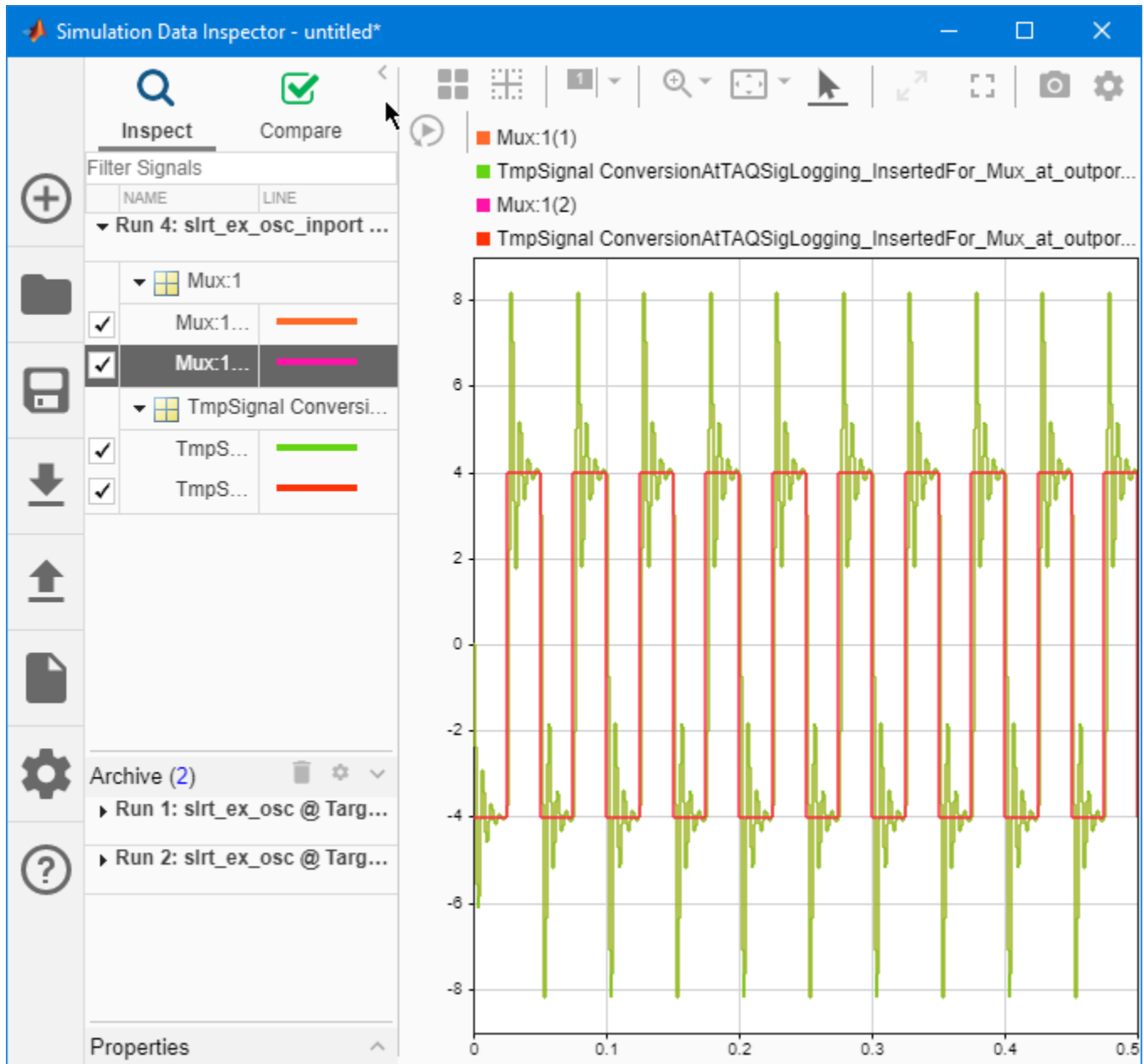
Click **Save**.

Save the scenario under a name such as `slrt_ex_inport_waveform_scenario.mldatx`.

Close the Root Inport Mapper. In the In1 block parameters dialog box, click **OK**.

To display the output of the Mux block with the Simulation Data Inspector, right-click the output signal and select **Log Selected Signals**.

You can now save, build, download, and execute the real-time application. Display the output by using the Simulation Data Inspector.



Update Inport to Use Sawtooth Wave

You can update the inport data to use a different data file without rebuilding the real-time application. The `slrt_ex_osc_inport.mldatx` file must be in the working folder.

Load `slrt_ex_inport_sawtooth.mat`, and then assign `sawtooth` to the temporary variable that you used with the Root Inport Mapper.

```
load((fullfile(matlabroot, 'toolbox', 'slrealtime', ...  
'examples', 'slrt_ex_inport_sawtooth.mat')));  
waveform = sawtooth;
```

Create an application object.

```
app_object = slrealtime.Application('slrt_ex_osc_inport');
```

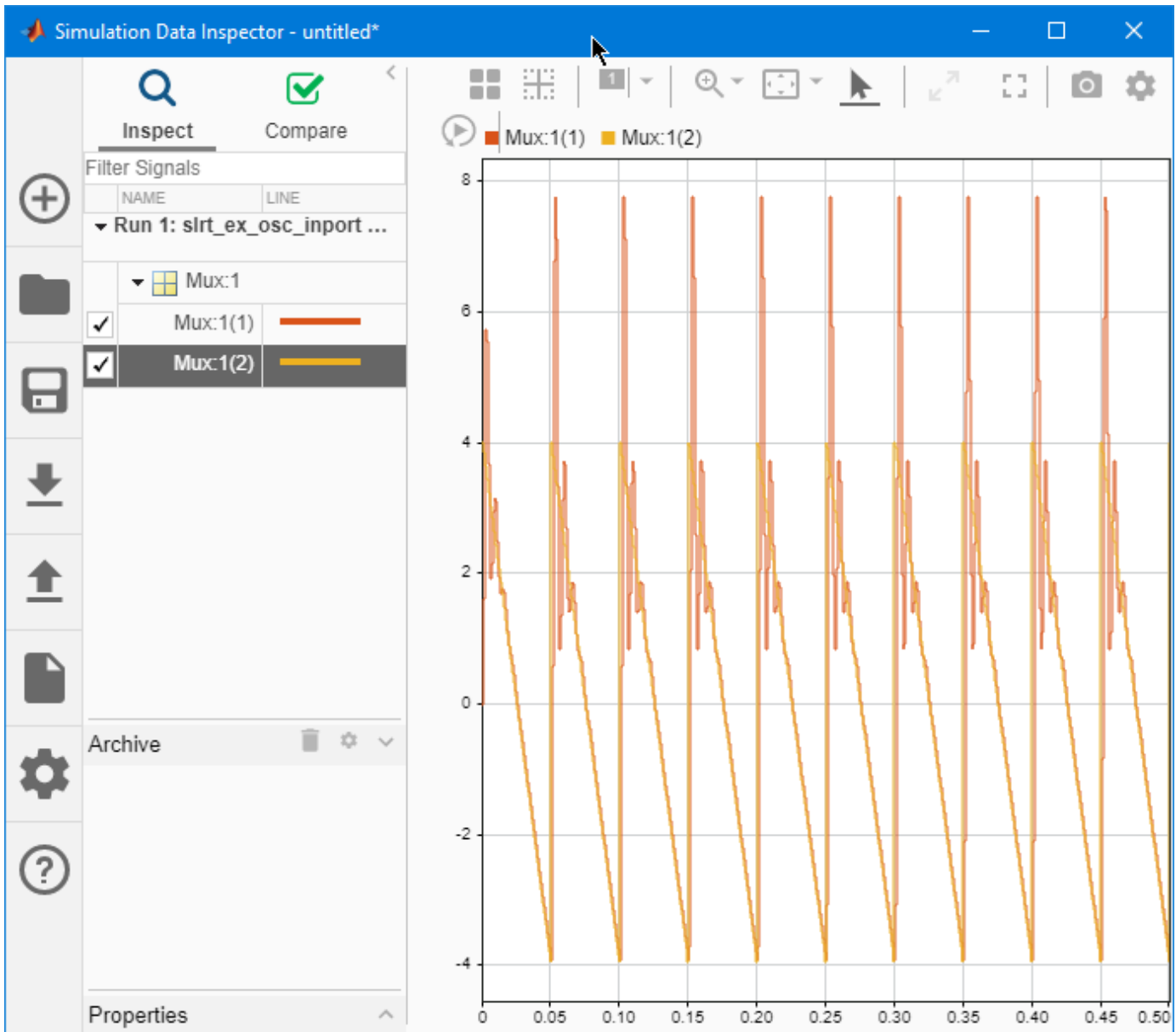
Update the application object.

```
updateRootLevelInportData(app_object);
```

Load the updated object to the target computer and execute it.

```
tg = slrealtime;  
load(tg, 'slrt_ex_osc_inport');  
start(tg);
```

Display the output by using the Simulation Data Inspector.



See Also

More About

- “Define and Update Inport Data by Using MATLAB Language” on page 7-60
- “Load Data to Root-Level Input Ports”
- “Inport Data Mapping Limitations” on page 7-65
- “Data Logging with Simulation Data Inspector (SDI)” on page 7-14

Define and Update Inport Data by Using MATLAB Language

In this section...

“Required Files” on page 7-60

“Map Inport to Use Square Wave” on page 7-60

“Update Inport to Use Sawtooth Wave” on page 7-61

You can create root-level input ports and use the MATLAB language to define input data and to update the input data without rebuilding the model.

Required Files

This procedure has these file dependencies:

- `slrt_ex_osc_inport` — Damped oscillator that takes its input data from input port `In1` and sends its multiplexed output to output port `Out1`. To open this model, in the MATLAB Command Window, type:

```
open_system(fullfile(matlabroot, 'toolbox', 'slrealtime', ...
    'examples', 'slrt_ex_osc_inport'))
```

- `slrt_ex_inport_square.mat`— One second of output from a Signal Generator block that is configured to output a square wave. To load this data, in the MATLAB Command Window, type:

```
load(fullfile(matlabroot, 'toolbox', 'slrealtime', ...
    'examples', 'slrt_ex_inport_square.mat'))
```

- `slrt_ex_inport_sawtooth.mat` — One second of output from a Signal Generator block that is configured to output a sawtooth wave. To load this data, in the MATLAB Command Window, type:

```
load(fullfile(matlabroot, 'toolbox', 'slrealtime', ...
    'examples', 'slrt_ex_inport_sawtooth.mat'))
```

Before starting this procedure, navigate to a working folder.

Map Inport to Use Square Wave

This procedure maps an inport.

Open `slrt_ex_osc_inport`.

```
model = fullfile(matlabroot, 'toolbox', 'slrealtime', ...
    'examples', 'slrt_ex_osc_inport');
open_system(model);
myFolder = fullfile(userpath, 'temp');
save_system(model, [myFolder '/slrt_ex_osc_inport.slx']);
```

Load `slrt_ex_inport_square.mat`, and then assign `square` to a temporary workspace variable.

```
load(fullfile(matlabroot, 'toolbox', 'slrealtime', ...
    'examples', 'slrt_ex_inport_square.mat'));
waveform = square;
```

Open `slrt_ex_osc_inport/In1`


```
inport = [model '/In1'];
load_system(inport);
```

Turn off inport data interpolation.

```
set_param(inport, 'Interpolate', 'off');
```

Set the external input variable.

```
set_param(model, 'ExternalInput', 'waveform');
```

Load external input data.

```
set_param(model, 'LoadExternalInput', 'on');
```

You can now build, download, and execute the real-time application.

```
slbuild(model);
tg = slrealtime('TargetPC1');
load(tg, model);
start(tg);
```

View the signals in the Simulation Data Inspector.

```
Simulink.sdi.view;
```

Update Inport to Use Sawtooth Wave

You can update the inport data to use a different data file without rebuilding the real-time application. The `slrt_ex_osc_inport.mldatx` file must be in the working folder.

Load `slrt_ex_inport_sawtooth.mat`, and then assign sawtooth to the temporary variable that you used with the Root Inport Mapper.

```
load(fullfile(matlabroot, 'toolbox', 'slrealtime', ...
    'examples', 'slrt_ex_inport_sawtooth.mat'));
waveform = sawtooth;
```

Create an application object.

```
app_object = SimulinkRealTime.Application('slrt_ex_osc_inport');
```

Update the application object.

```
updateRootLevelInportData(app_object);
```

Download the updated object to the target computer and execute it.

```
tg = slrealtime;
load(tg, 'slrt_ex_osc_inport');
start(tg);
```

View the signals in the Simulation Data Inspector.

`Simulink.sdi.view;`

See Also

More About

- “Define and Update Inport Data” on page 7-55
- “Load Data to Root-Level Input Ports”
- “Inport Data Mapping Limitations” on page 7-65
- “Data Logging with Simulation Data Inspector (SDI)” on page 7-14

Stimulate Root Inport by Using MATLAB Language

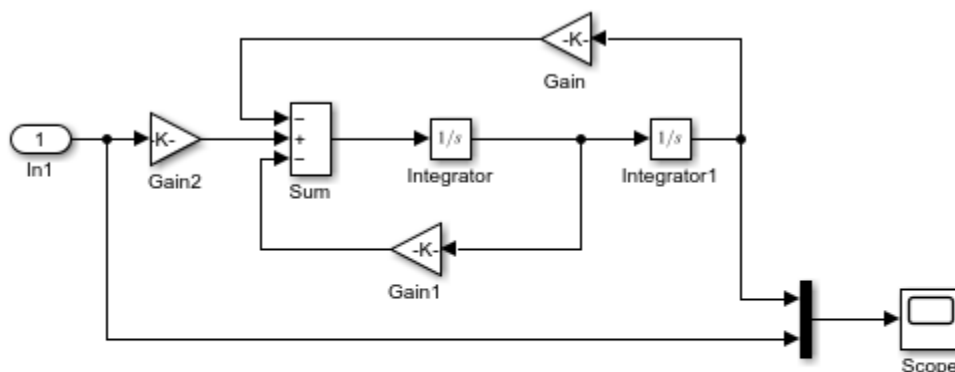
This example shows how to stimulate root inports in a model by using the Stimulation object and related functions:

- start
- stop
- getStatus
- reloadData
- pause

Open Model and Map Inport to Wave Data

Open model `slrt_ex_osc_inport`. Save the model to a working folder. Map the inport to use square wave data. For inport `In1`, interpolated is off.

```
model = ('slrt_ex_osc_inport');
open_system(model);
load(fullfile(matlabroot, 'toolbox', 'slrealtime', 'examples', 'slrt_ex_inport_square.mat'));
waveform = square;
set_param(model, 'ExternalInput', 'waveform');
set_param(model, 'LoadExternalInput', 'on');
set_param(model, 'StopTime', 'Inf');
```



Model `slrt_ex_osc_inport`
 Simulink Real-Time example model
 Copyright 2020 The MathWorks, Inc.

Build Model and Download Real-Time Application

Build, download, and execute the real-time application.

```
evalc('slbuild(model)');
tg = slrealtime('TargetPC1');
load(tg,model);
```

Stimulate Root Inport Data

Start root inport stimulation of inports 1. Open Scope block and observe results.

```
start(tg.Stimulation,[1]);  
start(tg);
```

Pause root inport stimulation of inport 1.

```
pause(tg.Stimulation,[1]);
```

Stop and start the stimulation of inport 1.

```
stop(tg.Stimulation,[1]);  
start(tg.Stimulation,[1]);
```

Check the status of stimulation of the inports.

```
getStatus(tg.Stimulation,'all');
```

Create a time-series object to load data to an inport.

```
sampleTime = 0.1;  
endTime = 10;  
numberOfSamples = endTime * 1/sampleTime + 1;  
timeVector = (0:numberOfSamples) * sampleTime;  
u = timeseries(timeVector*10,timeVector);
```

Object u is created for 10 seconds. Load it to the inport 1. Stimulation of an inport should be stopped before loading data.

```
stop(tg.Stimulation,[1]);  
reloadData(tg.Stimulation,[1],u);
```

Stop real-time application and close all.

```
stop(tg);  
bdclose('all');
```

Inport Data Mapping Limitations

In Simulink Real-Time, you cannot:

- Create data at run time for each time step by using the input $u = UT(t)$ for MATLAB functions or expressions.
- Import complex values and asynchronous function-call signals into top-level input ports.
- Import signals of type `Stateflow.SimulationData.State` into top-level input ports.

See Also

More About

- “Define and Update Inport Data” on page 7-55
- “Load Data to Root-Level Input Ports”

Display and Filter Hierarchical Signals and Parameters

In this section...


“Hierarchical Display” on page 7-66

“Filtered Display” on page 7-67

“Sorted Display” on page 7-68

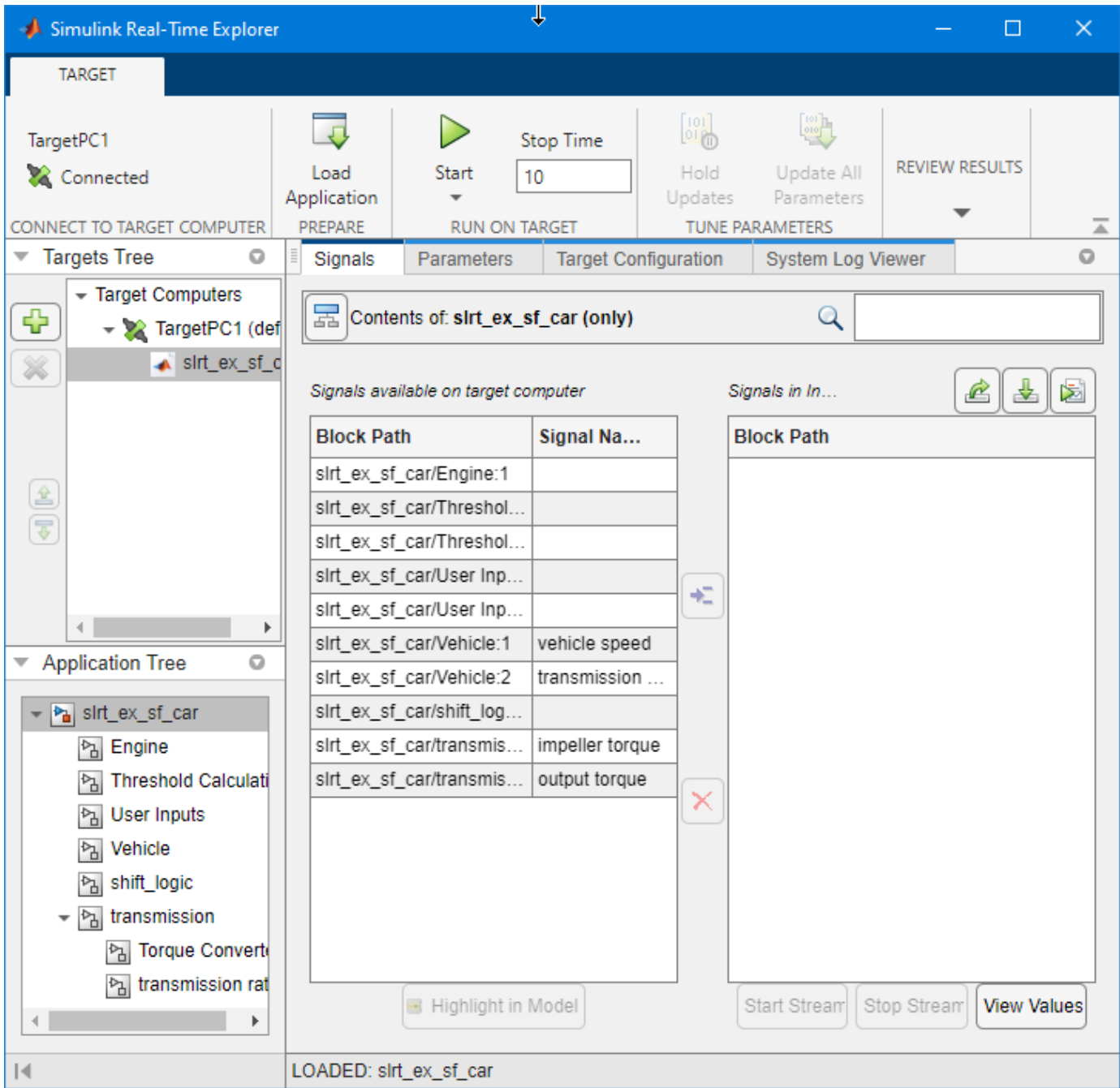
In Simulink Real-Time Explorer, the default view of the signal and parameter lists shows the signals and parameters only at the hierarchy level that you selected. You can display signals and parameters for the current level and below and filter the display to show only the items that you are interested in.

Hierarchical Display

To show signals and parameters from the current level and below, navigate to the hierarchical level that you are interested in. Click **Contents of** ( on the toolbar).

The figure shows the contents of the top level of the `slrt_ex_sf_car` real-time application. To open this model, in the MATLAB Command Window, type:

```
open_system(fullfile(matlabroot, 'toolbox', 'slrealtime', ...  
'examples', 'slrt_ex_sf_car'))
```



Filtered Display

To restrict the display to signals or parameters with a particular characteristic, use the **Filter** text box. You can restrict the scope of the filtered display by selecting a level of the application in the **Application Tree** panel.

Simulink Real-Time Explorer supports filtering by values in these columns:

- Signals — **Block Path** and **Signal Name**

- Parameters — **Block Path** and **Name**

For example, to restrict the display of signals and parameters to the `shift_logic` subsystem, select column **Signal Name**. Type `shift_logic` into the **Filter** text box.

The screenshot shows the Simulink Real-Time Explorer interface. The 'Parameters' tab is active, displaying a table of signals available on the target computer. A search filter 'shift_logic' is applied to the 'Signal Name' column. The table shows one entry: 'slrt_ex_sf_car/shift_log...'. The 'Application Tree' on the left shows the 'slrt_ex_sf_car' model with its subsystems expanded, including 'shift_logic'. The 'Target Tree' on the left shows the 'slrt_ex_sf_car' target computer. The 'System Log Viewer' tab is also visible.

Block Path	Signal Na...
slrt_ex_sf_car/shift_log...	

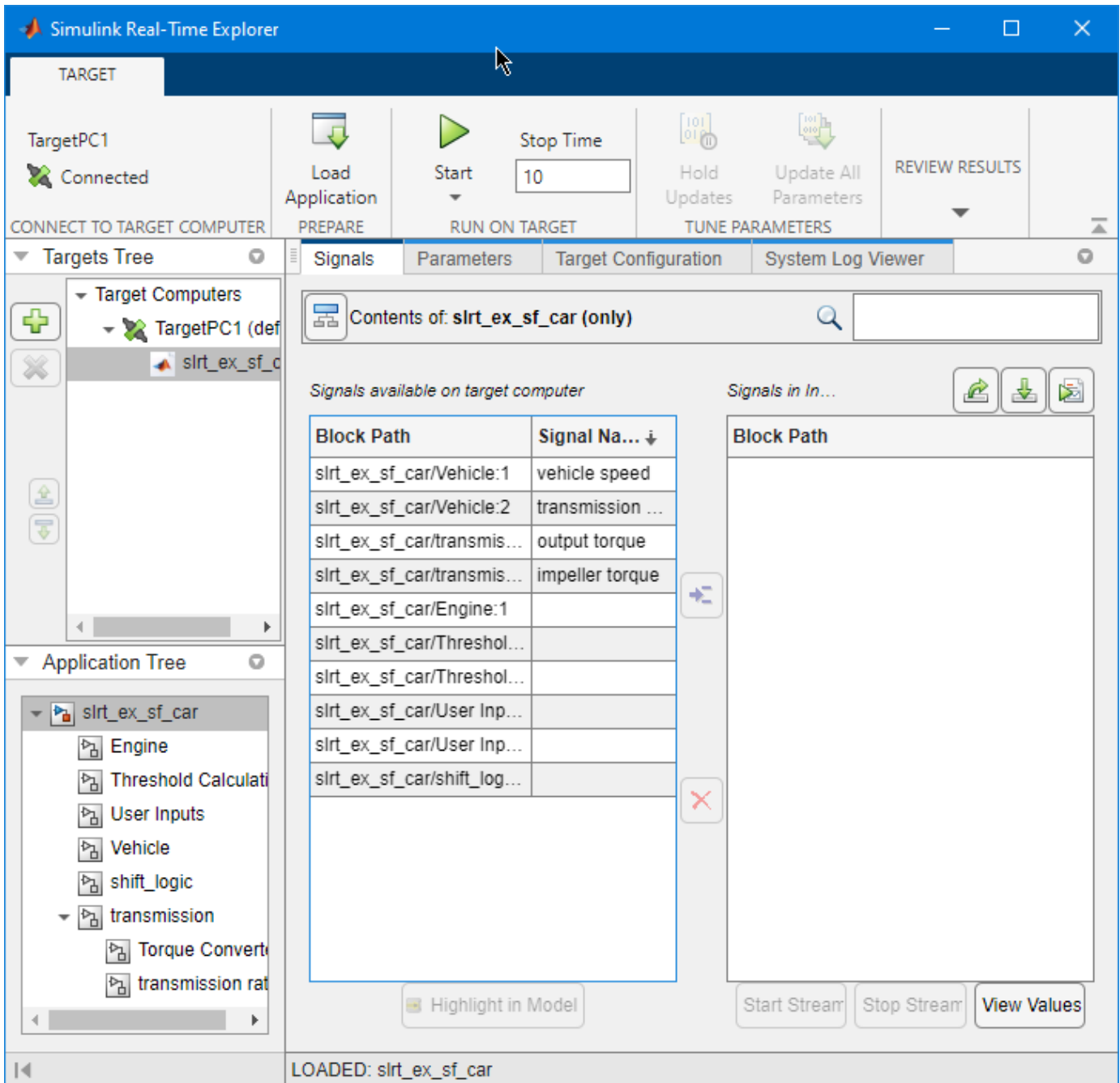
Sorted Display

To group signals and parameters by columns, select the column head, hover the cursor near the right border of the column head (displays the **Sort by** icon), and click the **Sort by** icon.

Explorer supports grouping by the following columns:

- Signals — **Block Path** and **Signal Name**
- Parameters — **Block Path, Name, Value, Type,** and **Size**

For example, to sort signals by name, right-click the **Signal Name** column and select the **Sort by** icon.



Troubleshoot Signals Not Accessible by Name

I cannot monitor, trace, or log some signal types in the real-time application.

What This Issue Means

You cannot monitor, trace, or log by name these types of signals in the real-time application:

- Virtual or bus signals (including signals from bus creator blocks and virtual blocks). For example, assume that you connect the output of a Mux block (a virtual block) to a Simulink Scope block. The Scope block displays the names of the Mux input signals rather than the names of the Mux output signals.
- Signals that Simulink optimizes away after you set the **Signal storage reuse** or **Block reduction** configuration parameters.

The output of a block that was optimized away is replaced with the corresponding input signal to the block. To access these signals, make them test points.

- Signals of complex or multiword data types.
- If a block name consists only of spaces, Simulink Real-Time Explorer does not display a node for signals from that block. To reference such a block:
 - Provide an alphanumeric name for the block.
 - Rebuild and download the model to the target computer.
 - Reconnect the MATLAB session to the target computer.

Try This Workaround

Check these signal types are not being monitored, traced, or logged by name in the real-time application::

- Virtual or bus signals (including signals from bus creator blocks and virtual blocks)
- Signals that Simulink optimizes away
- Signals of complex or multiword data types
- Blocks without alphanumeric names

See Also

Gain

More About

- “Nonvirtual and Virtual Blocks”
- “Composite Interface Guidelines”
- Signal storage reuse
- “Block reduction”
- “Troubleshoot Parameters Not Accessible by Name” on page 7-72
- “Internationalization Issues” on page 7-74

External Websites

- MathWorks Help Center website

Troubleshoot Parameters Not Accessible by Name

I cannot observe or tune some parameters in the real-time application.

What This Issue Means

Reasons that you cannot observe or tune some parameters in the real-time application are:

- Simulink Real-Time does not support parameters of multiword data types.
- During execution, you cannot tune parameters that change the model structure, for example, by adding a port. To change these parameters, you must stop the execution, change the parameter, and rebuild the real-time application.

Try This Workaround

Check the parameters for the issues described in “What This Issue Means” on page 7-72.

See Also

More About

- “Troubleshoot Signals Not Accessible by Name” on page 7-70
- “Internationalization Issues” on page 7-74

External Websites

- MathWorks Help Center website

Troubleshoot Instance-Specific Parameters Not Saved

The `saveParamSet` function does not save instance-specific parameters and parameters that have custom storage classes to a MAT file for loading by using the `loadParamSet` function. When I use the `saveParamSet` function on a model that contains only instance-specific parameters, I get an error message.

```
Error using SimulinkRealTime.target/saveparamset  
TargetPC1: Error writing file
```

What This Issue Means

The `saveParamSet` function saves parameters that appear in the `rtP` structure of the model. Instance-specific parameters and parameters with custom storage classes are global variables that are not by default represented in the `rtP` structure.

Try This Workaround

You can use the `saveParamSet` function to save parameter sets from models that include instance-specific parameters or parameters that have custom storage classes. But, these parameters do not appear in the saved parameter set.

See Also

`ParameterSet`

More About

- “Save and Reload Parameters by Using the MATLAB Language” on page 7-37

Internationalization Issues

Simulink Real-Time inherits the internationalization support of the products that it works with: Simulink, Simulink Coder, and Embedded Coder®. Signal and parameter names that include Unicode® characters are displayed as expected in Simulink Real-Time Explorer and at the MATLAB command line.

When you use the Simulation Data Inspector to observe signals, the non-ASCII signal names are displayed as expected. For example, assume that the signal with ID 1 appears in an English-language and a Japanese-language version of the same model. In the English-language version, the signal label is `input1` and the block path is `block1/block2`. In the Japanese-language version, the signal label is `入力 1` and the block path is `ブロック 1/ブロック 2`.

Third-party code (for example, parsers for vendor configuration files) sometimes does not support cross-locale, cross-platform internationalization. For such code, you must give files and folders locale-specific names. For example, when parsing a configuration file on an English-locale machine, name the file and enclosing folder with English-locale-specific names.

See Also

More About

- “Troubleshoot Signals Not Accessible by Name” on page 7-70
- “Troubleshoot Parameters Not Accessible by Name” on page 7-72

Execution Modes

Execution Modes

The Simulink Real-Time RTOS has two mutually exclusive execution modes.

- Interrupt mode — The scheduler implements real-time single-tasking and multitasking execution of single-rate or multirate systems, including asynchronous events (interrupts). You can interact with the target computer while the real-time application is executing at high sample rates. To use this real-time mode:
 - Leave the **Force polling mode** configuration parameter disabled (default).
 - Leave the `pollingThreshold` application option at the default value.
- Polling mode — The RTOS executes real-time applications at sample times close to the limit of the CPU. Using polling mode with high-speed and low-latency I/O boards and drivers enables you to achieve real-time application sample times that you cannot achieve by using interrupt mode. Because polling mode disables interrupts on the processor core where the model runs, it imposes restrictions on the model architecture and on target communication. The base rate of the real-time application is always running when executing in polling mode. To use this real-time mode, either:
 - Enable the **Force polling mode** configuration parameter.
 - Set the `pollingThreshold` application option sample time value to a rate below the base rate of the model.

For more information, see **Force polling mode** and **Application**.

See Also

Thread Trigger | “TLC Command-Line Options”

Related Examples

- “Concurrent Execution on Simulink Real-Time” on page 16-11

More About

- “Set Configuration Parameters”
- “Performance Optimization”
- “About RTOS Tasks and Priorities”
- “Troubleshoot Overloaded CPU from Executing Real-Time Application” on page 17-32

Real-Time Application Execution

Working with the Target Computer Command Line

- “Control Real-Time Application at Target Computer Command Line” on page 9-2
- “Execute Target Computer RTOS Commands at Target Computer Command Line” on page 9-3

Control Real-Time Application at Target Computer Command Line

The Simulink Real-Time software provides a set of commands that you can use to interact with the real-time application on the target computer. You can load, start, stop, and check the status of the real-time application.

These commands let you interact with real-time applications on standalone target computers that are not connected to Simulink Real-Time software on a development computer.

To enter commands, type the commands by using a keyboard attached to the target computer or by using an SSH utility (such as PuTTY) to send commands to the target computer from a development computer.

Note To run user commands, log in as user `slrt` by using password `slrt`. To run the system commands (for example, `date`, `ntdate`, `ntpd`, `rtc`, or setting the time zone), login as user `root` by using password `root`.

The target computer commands are case-sensitive. For more information, see “Target Computer Command-Line Interface”.

To read the target computer console log, open the **Simulink Real-Time Explorer** and click the **System Log Viewer** tab. You can also export the system log by using the `SystemLog` function.

See Also

Simulink Real-Time Explorer | `slrtExplorer` | `SystemLog`

Related Examples

- “Target Object Commands”
- “Target Computer RTOS System Commands”

Execute Target Computer RTOS Commands at Target Computer Command Line

To enter target computer RTOS commands, type the commands by using a keyboard attached to the target computer or by using an SSH utility (such as PuTTY) to send commands to the target computer from a development computer.

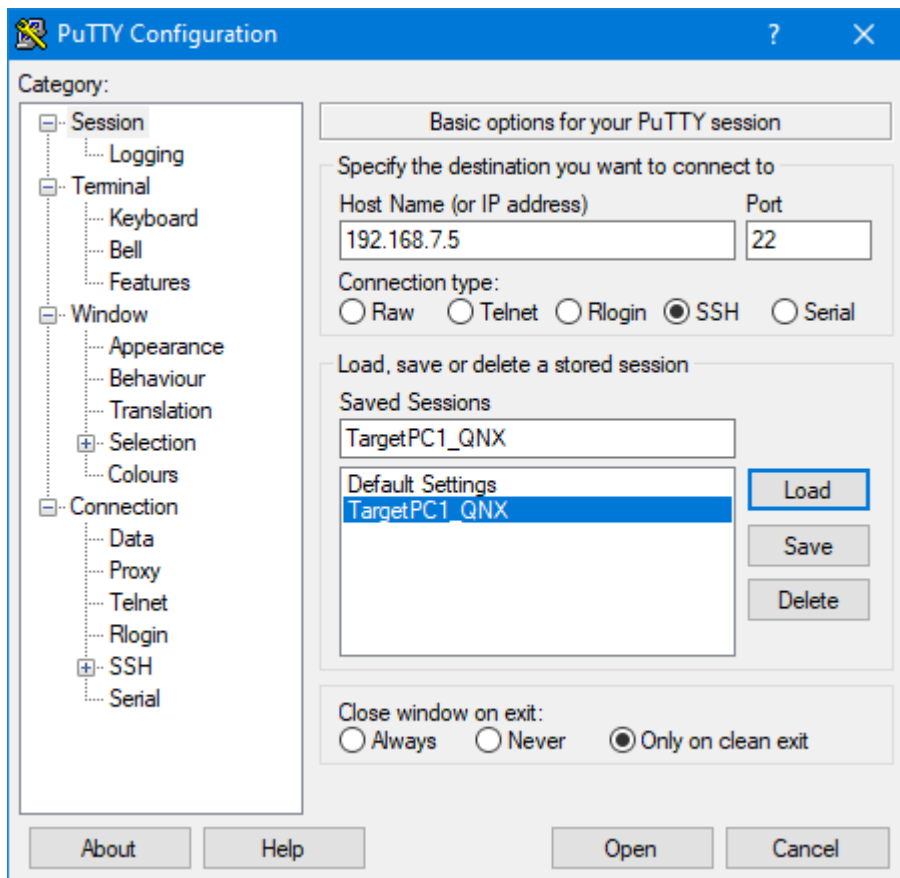
The target computer commands are case-sensitive. For more information, see “Target Computer Command-Line Interface”.

The command examples use the PuTTY SSH utility. You can download and install this utility from www.putty.org.

Note To run user commands, log in as user `slrt` by using password `slrt`. To run the system commands (for example, `date`, `ntdate`, `ntpd`, `rtc`, or setting the time zone), login as user `root` by using password `root`.

- 1 Boot the target computer.
- 2 Connect the development computer and target computer. In the MATLAB Command Window, type:

```
tg = slrealtime;  
connect(tg);
```
- 3 Start the SSH utility. This example uses PuTTY.
- 4 Load the PuTTY session for the target computer and click **Open**.



5 To configure the target computer date, log in to the PuTTY session as user `root` with password `root`.

6 Set the time zone. This example sets the time zone to Eastern Standard Time.

```
# env TZ=EST5EDT
# export TZ=EST5EDT
# setconf _CS_TIMEZONE EST5EDT
```

7 Set the date and time. This example sets the date and time to September 10, 2019 at 11:25 AM.

```
# date 091011252019
Tue Sep 10 11:25:15 EDT 2019
```

8 Set the hardware clock from the system date and time.

```
# rtc -s hw
```

See Also

Targets

Related Examples

- “Target Object Commands”
- “Target Computer RTOS System Commands”

External Websites

- QNX Momentics IDE 7.1 User's Guide
- QNX Momentics IDE 7.1 User's Guide, Utilities Reference

Tuning Performance

- “CPU Overload” on page 10-2
- “Monitor CPU Overload Rate” on page 10-3
- “Execution Profiling for Real-Time Applications” on page 10-7
- “Reduce Build Time for Simulink Real-Time Referenced Models” on page 10-13

CPU Overload

Sometimes a real-time application running on the target computer does not have enough time to complete processing before the next time step. This condition is called a CPU overload. An overload is registered every time an execution step is triggered while the previous step is running.

See Also

SLRT Overload Options

Related Examples

- “Monitor CPU Overload Rate” on page 10-3
- “Concurrent Execution on Simulink Real-Time” on page 16-11

More About

- “Troubleshoot Overloaded CPU from Executing Real-Time Application” on page 17-32

Monitor CPU Overload Rate

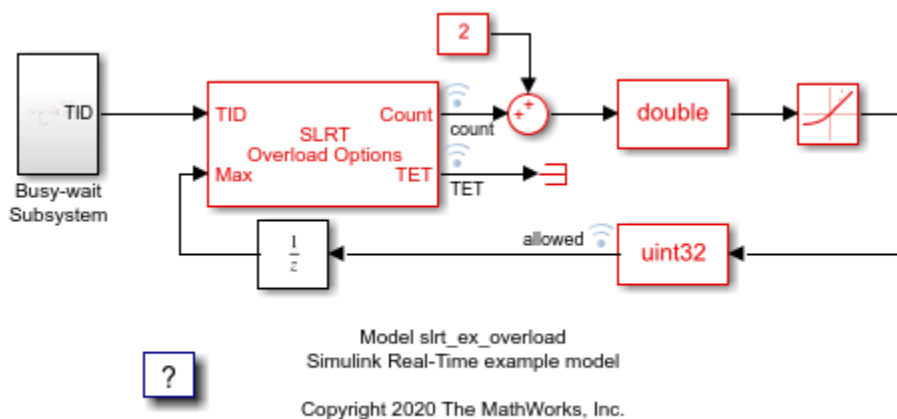
The SLRT Overload Options block outputs the current CPU overload count for the identified sample rate.

This example shows how to design a model that uses the SLRT Overload Options block to monitor the rate at which CPU overloads occur. The rate of CPU overloads information can be useful when tuning performance of a model for which a low CPU overload rate is acceptable.

Open, Build, and Run the Model

In the MATLAB Command Window, type:

```
open_system(fullfile(matlabroot, 'toolbox', 'slrealtime', 'examples', 'slrt_ex_overload'));
```



Name the signal coming out from the output of rate limiter block as Rate Limiter and log it in the Simulation Data Inspector.

```
p = get_param('slrt_ex_overload/Rate Limiter', 'PortHandles');
l = get_param(p.Outport, 'Line');
set_param(l, 'Name', 'Rate Limiter');
Simulink.sdi.markSignalForStreaming('slrt_ex_overload/Rate Limiter', 1, 'on');
```

Build the model.

```
model = 'slrt_ex_overload';
set_param(model, 'RTWVerbose', 'off');
evalc('slbuild(model)');
```

Download the application and run it on the target computer.

```
tg = slrealtime;
connect(tg);
load(tg, model);
start(tg);
pause(20);
stop(tg);
```

Open Simulation Data Inspector

To view the rate at which CPU overloads occur, open the Simulation Data Inspector.

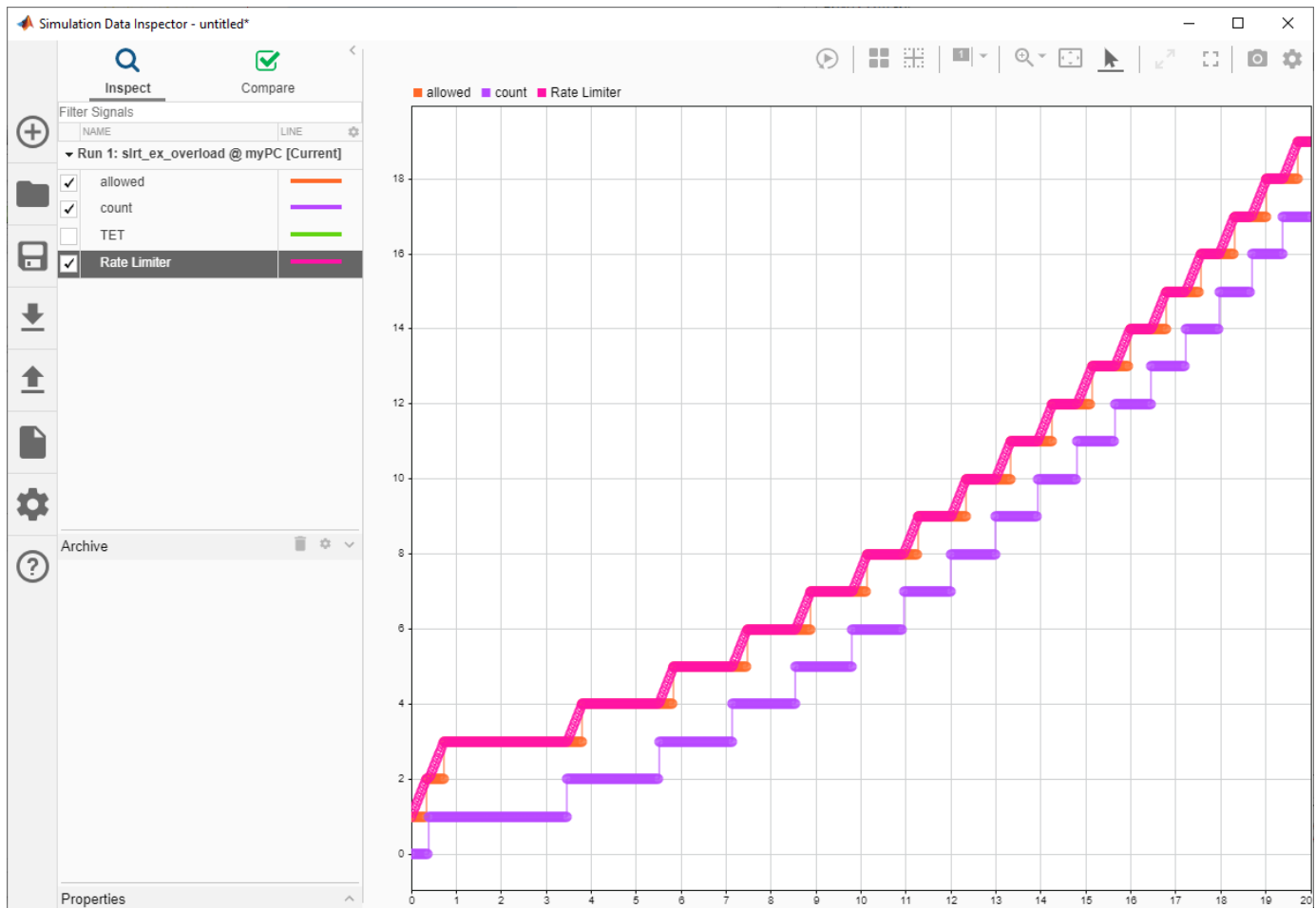
In the MATLAB Command Window, type:

```
Simulink.sdi.view;
```

Examine CPU Overload Rate Data

In the Simulation Data Inspector, the graph shows:

- Bottom rising stair step signal -- This signal indicates the number of CPU overloads that occurred.
- Top rising stair step signal -- This signal indicates the number of CPU overloads that are allowed, which is (occurred + 2).
- Rising slew rate -- This signal indicates the rate at which CPU overloads occur. When the rising slew rate becomes greater than the top rising stair step signal, the rate of CPU overloads is greater than are allowed.



Modify Rate of CPU Overloads

To modify the rate at which CPU overloads occur in the model, modify the Constant2 parameter value.

Modify Allowed Rate of CPU Overloads

To modify the rate of CPU overloads that are acceptable in the model, modify the `RisingSlowLimit` parameter value.

Build and Run Model with Changed Overload Rates

In the MATLAB Command Window, type:

```
load(tg,model);
```

To modify the rate of CPU overloads that are acceptable in the model

```
tg.setparam('slrt_ex_overload/Rate Limiter','RisingSlewLimit',0.004);
```

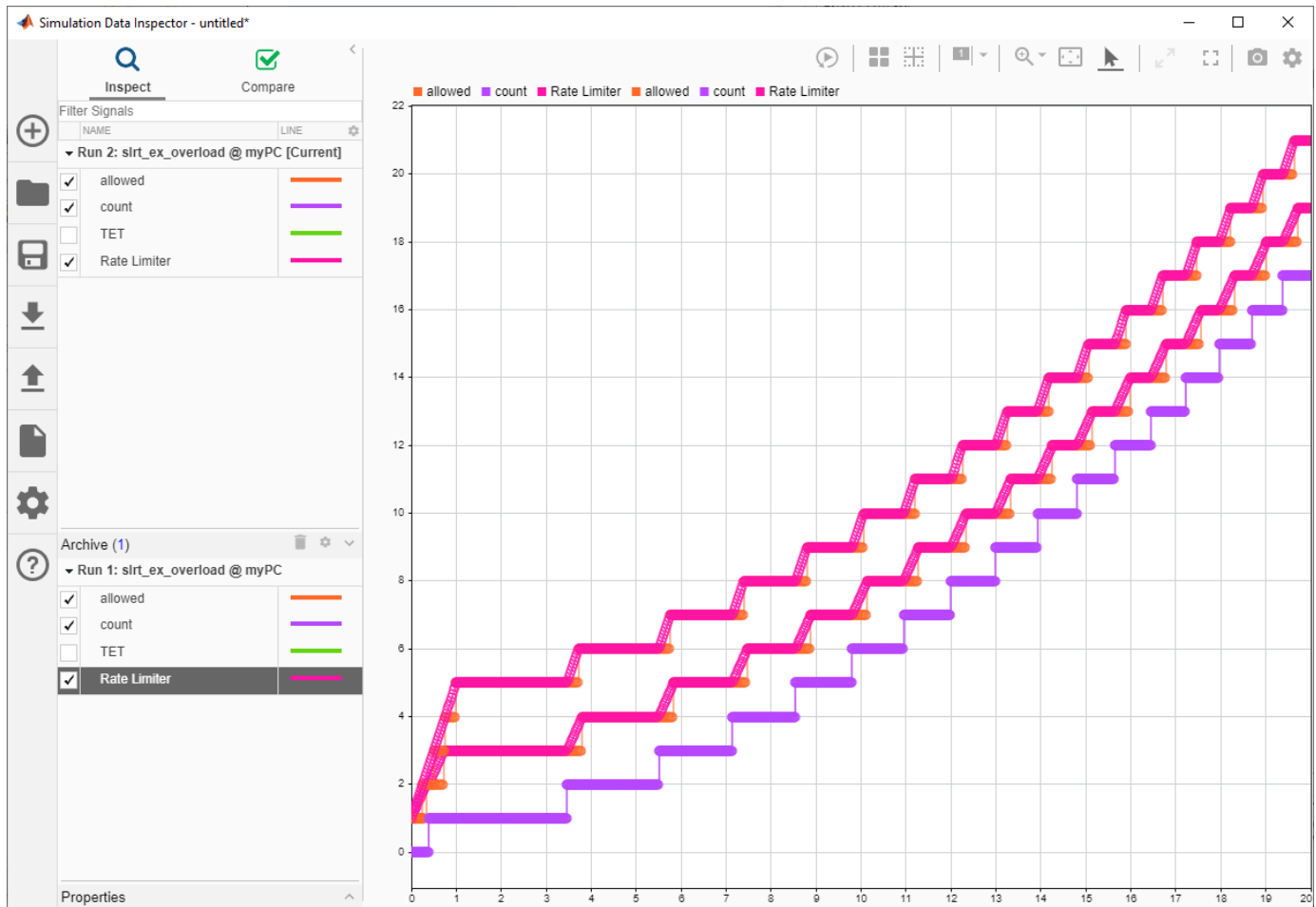
To modify the rate at which CPU overloads occur in the model

```
tg.setparam('slrt_ex_overload/Constant2','Value',4);
```

run the modified application on the target computer

```
start(tg);  
pause(20);  
stop(tg);
```

In the Simulation Data Inspector, compare the signal data from the simulation runs and observe the change to the CPU overload rate.



```
bdclose('all');
```

See Also

SLRT Overload Options

Related Examples

- “Concurrent Execution on Simulink Real-Time” on page 16-11

More About

- “CPU Overload” on page 10-2
- “Troubleshoot Overloaded CPU from Executing Real-Time Application” on page 17-32

Execution Profiling for Real-Time Applications

This example shows how you can profile the task execution time and function execution time of your real-time application that is running on the target computer. Using that information, you can then tune its performance.

Profiling is especially useful if you configure your real-time application to take advantage of multicore processors on the target computer. To profile the real-time application:

- In the Configuration Parameters dialog box for the model, enable the collection of function execution time data during execution.
- Build, download, and execute the model.
- Start and stop the profiler.
- Display the profiler data.

The Execution Profiler and SLRT Overload Options block use different mechanisms to measure TET and do not generate identical TET values.

Configure Real-Time Application for Function Execution Profiling

The model is `slrt_ex_mds_and_tasks`. To open this model, open the subsystem models first:

- `slrt_ex_mds_subsystem1`
- `slrt_ex_mds_subsystem2`
- `slrt_ex_mds_and_tasks`

1. Open model `slrt_ex_mds_and_tasks`.

2. In the top model, open the Configuration Parameters dialog box. Select **Code Generation >> Verification**.

3. For **Measure function execution times**, select **Coarse (reference models and subsystems only)**. The **Measure task execution time** check box is selected and locked. Or, in the MATLAB Command Window, type:

```
set_param('slrt_ex_mds_and_tasks','CodeProfilingInstrumentation','Coarse');
```

4. Click **OK**. Save model `slrt_ex_mds_and_tasks` in a local folder.

Generate Real-Time Application Execution Profile

Generate profile data for model `slrt_ex_mds_and_tasks` on a multicore target computer.

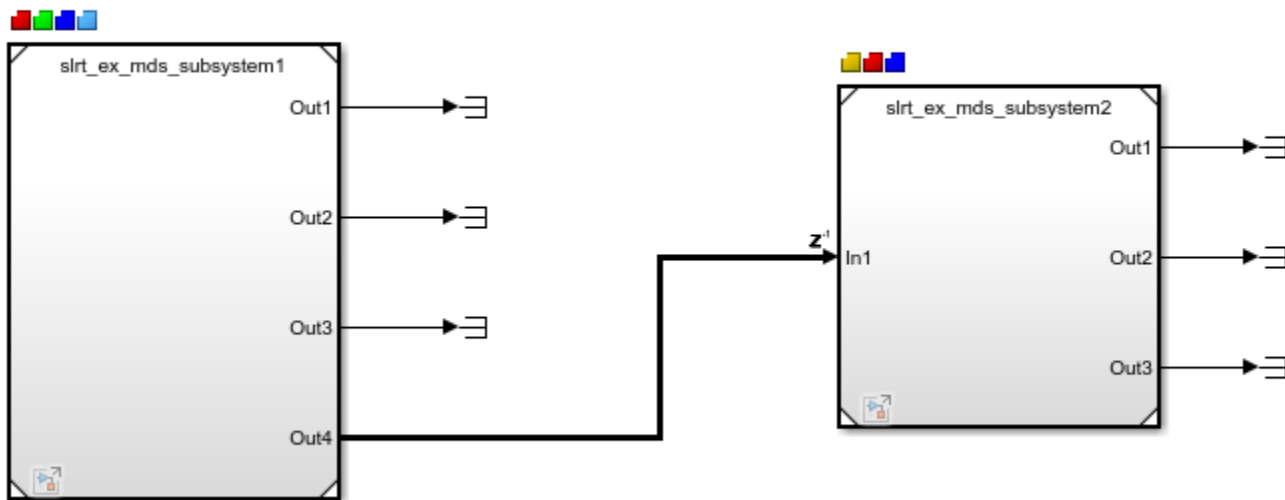
You must have previously configured the target computer to take advantage of multiple cores and configured the model for task and function execution profiling.

1. Open, build, and download the model.

```
model = 'slrt_ex_mds_and_tasks';
open_system(model);
evalc('slbuild(model)');
tg = slrealtime;
```

```
load(tg,model);  
setStopTime(tg,20);
```

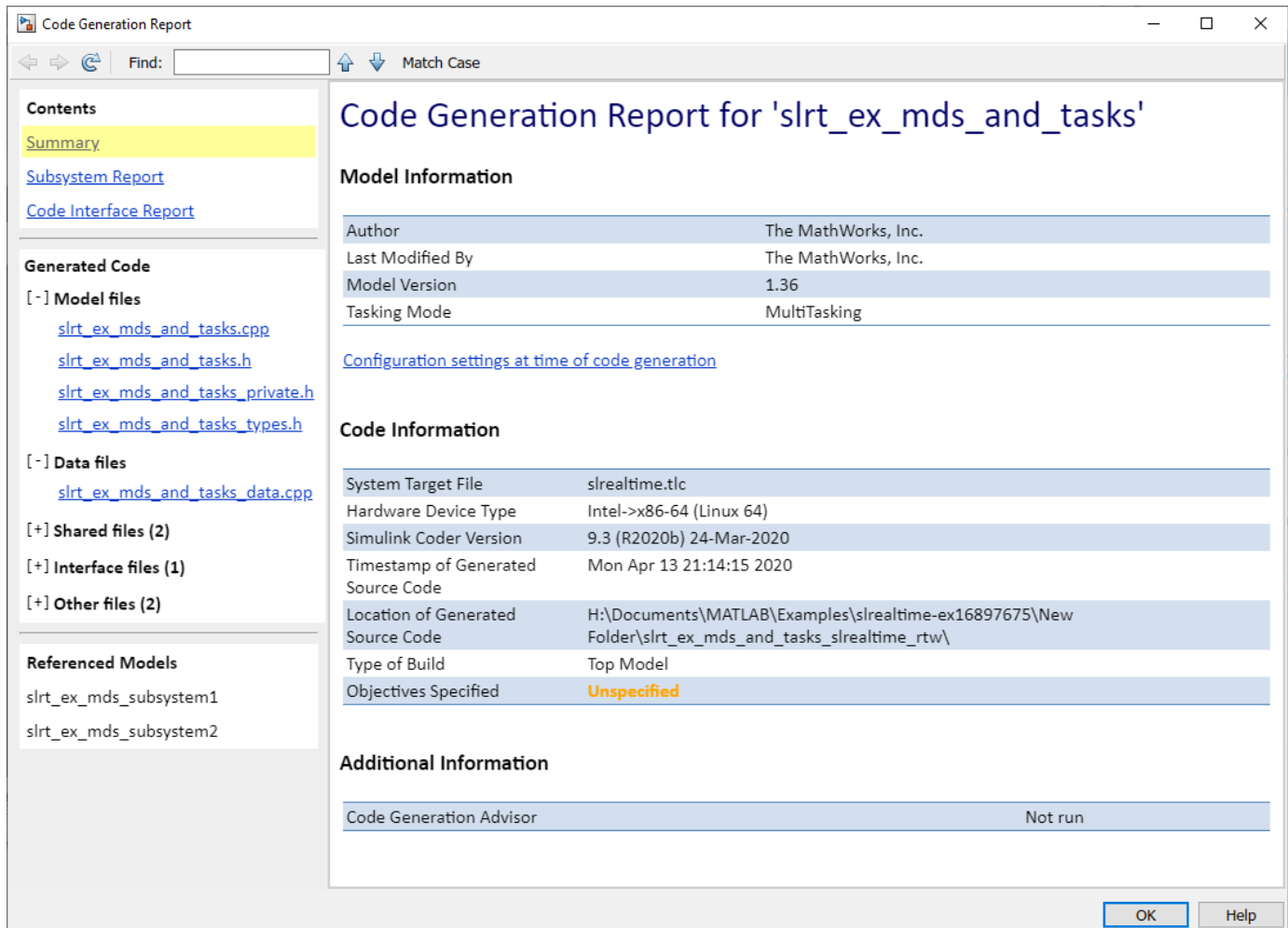
Concurrent Execution on Simulink Real-Time Illustrated by Profiling Tool



Copyright 2020-2021 The MathWorks, Inc.



When you include profiling, the Code Generation Report is generated by default. It contains links to the generated C code and include files. By clicking these links, you can examine the generated code and interpret the Code Execution Profile Report.



2. Start the real-time application, then start the profiler.

```
startProfiler(tg);
start(tg);
pause(5);
stopProfiler(tg);
stop(tg);
```

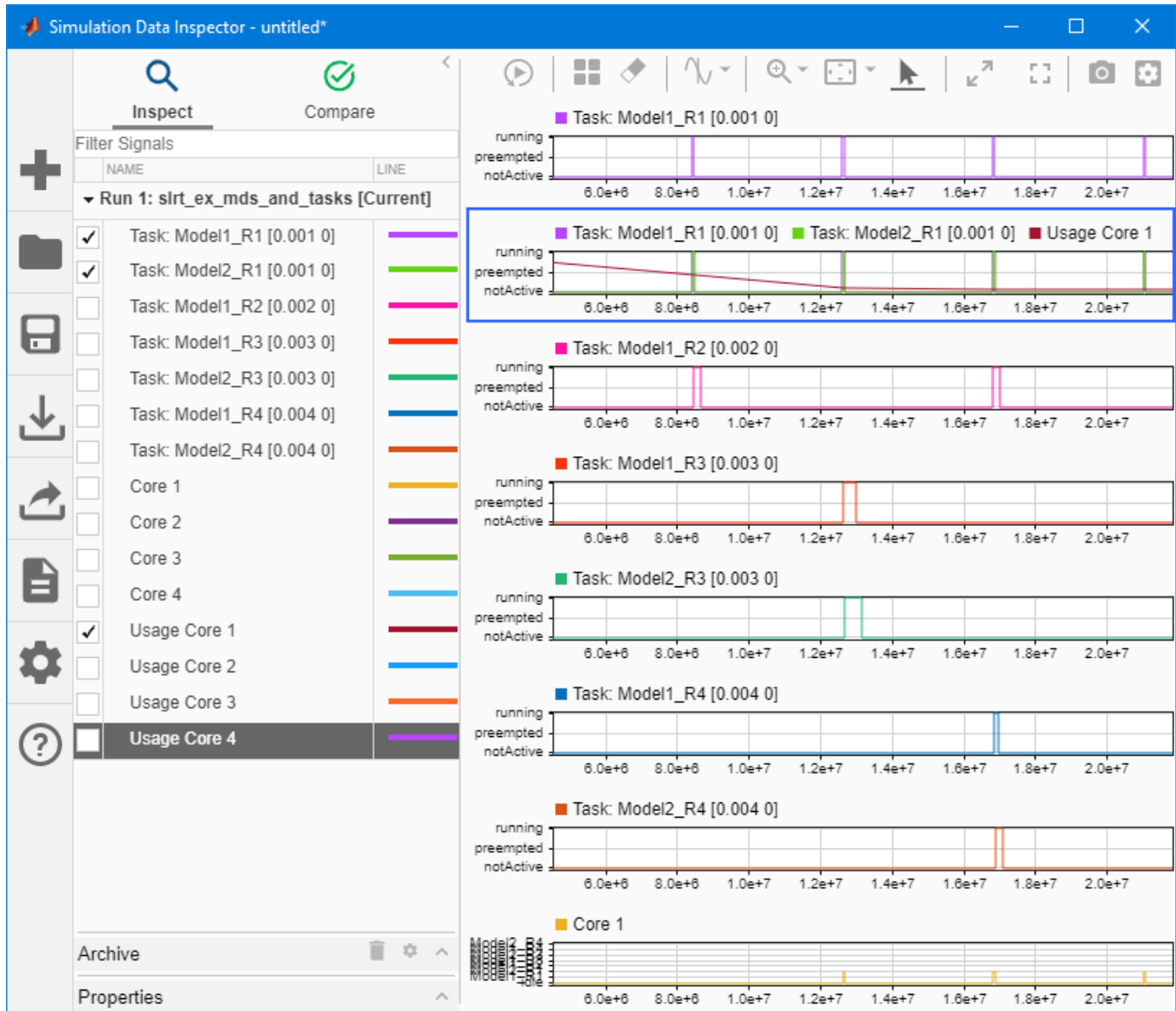
3. Display the profiler data.

```
while 1
    tmp = strcmp(tg.ProfilerStatus, 'DataAvailable');
    if tmp == true
        break
    end
end
profiler_data = getProfilerData(tg);
plot(profiler_data);
report(profiler_data);
```

```
Processing data on target computer ...
Transferring data from target computer ...
```

Processing data on host computer ...

The Execution Profile schedule display in the Simulation Data Inspector shows how scheduling is generated in real-time simulation. To open the schedule display in the Simulation Data Inspector after creating the executionProfile object, use the `executionProfile.schedule()` function.



The Code Execution Profiling Report displays model execution profile results for each task.

- To display the profile data for a section of the model, in the **Section** column, click the **Membrane** button next to the task.
- To display the TET data for the section in Simulation Data Inspector, click the **Plot time series data** button.

- To view the section in Simulink Editor, click the link next to the **Expand Tree** button.
- To view the lines of generated code corresponding to the section, click the **Expand Tree** button, and then click the **View Source** button.

Code Execution Profiling Report

























Code Execution Profiling Report for slrt_ex_mds_and_tasks

The code execution profiling report provides metrics based on data collected from real-time simulation. Execution times are calculated from data recorded by instrumentation probes added to the generated code. See [Code Execution Profiling](#) for more information.

1. Summary

Total time	249937177
Unit of time	ns
Command	report(, 'Units', 'seconds', 'ScaleFactor', '1e-09', 'NumericFormat', '%0.0f');
Timer frequency (ticks per second)	4.20001e+09
Profiling data created	13-Apr-2020 20:52:29

2. Profiled Sections of Code

Section	Maximum Turnaround Time in ns	Average Turnaround Time in ns	Maximum Execution Time in ns	Average Execution Time in ns	Calls	
[+] Model1_R1[0.001 0]	36555	12427	36555	12427	2001	  
[+] Model2_R1[0.001 0]	29099	13555	29099	13555	2003	  
[+] Model1_R2[0.002 0]	108247	38432	108247	38432	1003	  
[+] Model1_R3[0.003 0]	230666	73550	230666	73550	669	  
[-] Model2_R3[0.003 0]	237935	98862	237935	98862	669	  
profiled_section_5 [Note 1]	235639	94498	235639	94498	667	  
[+] Model1_R4[0.004 0]	87263	11897	87263	11897	509	  
[+] Model2_R4[0.004 0]	164480	75342	164480	75342	504	  

Notes:

[1] Multiple entities in the model map to a single function in the generated code, as a result of code reuse. Click the entry in the Model column to highlight all of the model entities. Browse through the model to identify all the highlighted entities.

3. Definitions

Execution Time: Time between start and end of code section, which excludes preemption time.

Turnaround Time: Time between start and end of code section, which includes preemption time.

OK Help

See Also

[schedule](#) | [report](#) | [plot](#) | [ProfilerData](#) | [stopProfiler](#) | [startProfiler](#) | [resetProfiler](#) | [getProfilerData](#) | [getAvailableProfile](#) | [deleteProfilerData](#)

Related Examples

- “Concurrent Execution on Simulink Real-Time” on page 16-11

Reduce Build Time for Simulink Real-Time Referenced Models

In a parallel computing environment, you can increase the speed of code generation and compilation for models containing large model reference hierarchies. Achieve the speed by building referenced models in parallel whenever conditions allow. For example, if you have Parallel Computing Toolbox software, you can distribute code generation and compilation for each referenced model across the cores of a multicore host computer. If you also have MATLAB Parallel Server™ software, you can distribute code generation and compilation for each referenced model across remote workers in your MATLAB Parallel Server configuration.

You can build referenced models in parallel on a compute cluster. In this way, you can more quickly build and download real-time applications to the target computer.

For this procedure, you must have a functioning Simulink Real-Time installation on your development computer.

- 1 Identify a set of worker computers, which can be separate cores on your development computer or computers in a remote cluster running under Windows®.
- 2 If you intend to use separate cores on the development computer, install Parallel Computing Toolbox on the development computer.
- 3 If you intend to use computers in a remote cluster:
 - a On each cluster computer, install:
 - MATLAB
 - Parallel Computing Toolbox
 - MATLAB Parallel Server
 - Simulink Real-Time
 - Simulink Real-Time Target Support Package
 - b Start and configure the remote cluster according to the instructions at “Get Started with MATLAB Parallel Server” (MATLAB Parallel Server).
- 4 Run MATLAB on the development computer.
- 5 In MATLAB, call the `parpool` function to open a parallel pool on the cluster.
- 6 To configure the compiler for the remote workers as a group, call the `pctRunOnAll` function.

In this configuration, the development computer and the remote workers have installed a supported version of a C++ compiler that is compatible with the code generation target. For the current list of supported compilers, see Supported and Compatible Compilers.

- 7 From the top model of the model reference hierarchy, open the Configuration Parameters dialog box. Go to the **Model Referencing** pane and select the “Enable parallel model reference builds” option. This selection enables the parameter “MATLAB worker initialization for builds”. For more information, see “Reduce Build Time for Referenced Models by Using Parallel Builds”.
- 8 Build and download your model.

See Also

`parpool` | `pctRunOnAll`

More About

- “Reduce Build Time for Referenced Models by Using Parallel Builds”

External Code Integration

External Code Integration of Libraries and C/C++ Code with Simulink Real-Time Models

In this section...

“Considerations for Integrating Third-Party Libraries and External Code into Simulink Real-Time” on page 11-2

“Value of Upgrading Your C/C++ Code for Integration into Simulink Real-Time” on page 11-2

“Approaches for C/C++ Code Integration into Simulink Real-Time” on page 11-3

“Build Libraries from Source Code for Simulink Real-Time” on page 11-3

“External Code Integration for S-Functions and Simulink Real-Time” on page 11-4

“Hello World! Example External Code Integration for Simulink Real-Time” on page 11-5

“Additional C/C++ Project for Simulink Real-Time” on page 11-7

Considerations for Integrating Third-Party Libraries and External Code into Simulink Real-Time

When integrating code into Simulink Real-Time applications, start by following the guidance in “Build Integrated Code Within the Simulink Environment”. Developers who integrate C/C++ code with Simulink Real-Time applications notice some differences when they migrate the code that they integrated with Simulink Real-Time applications from previous releases to R2020b and later releases. These differences include:

- In release R2020a and previous releases, the On-Time RTOS on the target computer shared some libraries and system calls with Windows. In release R2020b and later releases, the QNX Neutrino RTOS on the target computer does not share libraries or system calls specific to Windows.
- In release R2020a and previous releases, developers could use Microsoft® Visual Studio® to compile libraries to integrate with Simulink Real-Time applications. In release R2020b and later releases, you cannot use the Microsoft Visual Studio compiler for this purpose. You can configure Microsoft Visual Studio to use the QNX Neutrino compiler from the Simulink Real-Time target support package.
- In R2020b and later releases, developers use cross-compiling to produce libraries on their development computer for deployment to their target computer.

Value of Upgrading Your C/C++ Code for Integration into Simulink Real-Time

By updating your C/C++ code for integration into your Simulink Real-Time application, you gain these benefits:

- Leverage the QNX Neutrino 64-bit and POSIX® compatible RTOS.
- Code directly in C++ or wrap your legacy C code.
- Use the code editor of your choice.

For instance, customizing Visual Studio Code with the source files and shipped QCC compiler from the Simulink Real-Time Target Support Package provides a similar experience to a full IDE.

- Leverage the precompiled QNX Neutrino libraries and headers that are included in Simulink Real-Time to extend the functionality of your real-time application.
- Integrate any C/C++ application based on modern build and package software such as CMake.

Approaches for C/C++ Code Integration into Simulink Real-Time

There are advantages and disadvantages to each of these external code integration approaches.

Approach 1: Directly Call C/C++ Code. In this approach, you use C Caller or C Function blocks in the model. For more information, see “Integrate External C/C++ Code Using C Function Blocks”.

- Advantages: There is no need to compile source code before building the model.
- Disadvantages: This approach is hard to use for complex projects that have many files and dependencies. Also, in this approach you need to write a C wrapper in Simulink for C++ code. For more information, see “Call C++ Class Methods Using a C-style Wrapper Function From a C Function Block”.

Approach 2: Build, link, and use static libraries (.a files)

- Advantages: All required files are packed in the real-time application MLDATX file. In this approach, there is no need to install libraries on the target. And, this approach lets you protect your intellectual property.
- Disadvantages: This approach is non-modular. A change in the library requires rebuilding the whole real-time application. Also, this approach tends to produce larger real-time application MLDATX files.

Approach 3: Build, deploy and use shared objects (.so files)

- Advantages: This approach is modular. You can build the real-time application and shared object independently. Also, this approach tends to produce smaller real-time application MLDATX files. And this approach lets you protect your intellectual property.
- Disadvantages: In this approach, you need to access the target computer file system before running the real-time application and install (copy) the shared objects to any of the common lib paths on the target computers.

Build Libraries from Source Code for Simulink Real-Time

To integrate external code in a real-time application, the most flexible approaches are to build static libraries or shared objects from source code.

- The library build workflow is similar to the workflow used by most developers for release R2020a and previous releases. In those releases, the library build workflow for the target computer On-Time RTOS produced static libraries built with Microsoft Visual Studio and produced .lib files.
- You achieve better usability when working with complex C++ projects that have many dependencies and source code files.
- S-functions offer better granularity when handling third-party libraries in Simulink. S-functions enable the flexibility to use the same S-function source code with different platforms, including simulation on the desktop in different operating systems. The S-functions are deployed and function in real-time on a target computer.

Cross-compiling is compiling a library for a target operating system (for example, QNX Neutrino RTOS) on a development operating system (for example, Windows). Some cross-compiling considerations for Simulink Real-Time are:

- Choice of development environment. Many modern C++ projects use the CMake build environment. For more information, see the CMake website.
- Extensibility of development environment. For example, it is a common practice to extend most common CMake support for the QNX Neutrino RTOS by leveraging similarities with the UNIX® OS and its POSIX compatibility.
- In your libraries, save cross-compiling libraries, including dependencies that might be already included in the Simulink Real-Time Target Support Package. These libraries can be linked to other C++ projects.

The suggested workflow for integration of complex C++ applications into Simulink Real-Time is:

- 1 Start from a C++ project with CMake as the build environment.
- 2 Set the dependencies, such as headers and libraries, in your Simulink model.
- 3 On the development computer, cross-compile libraries for the QNX Neutrino RTOS on the target computer.
- 4 Create an S-function, for instance using the S-function Builder block or a handwritten C-MEX S-function, as the main function that calls the C++ functions defined in the header files and implemented in the compiled libraries for the QNX Neutrino RTOS.
- 5 Build the real-time application.
- 6 By using SSH or FTP, copy your cross-compiled libraries to a location on the target computer where they can be found and loaded at run time. The recommended locations are `/lib`, `/usr/lib`, or `/usr/local/lib`.
- 7 Load and run the real-time application.

External Code Integration for S-Functions and Simulink Real-Time

When you include static libraries or shared objects in S-functions for external code integration with a real-time application, there are some tips for your development.

When building from Simulink:

- Use `rtwmakecfg.m` and `makeInfo` object to map libraries and header files. For more information, see “Use `rtwmakecfg.m` API to Customize Generated Makefiles”.

```
function makeInfo = rtwmakecfg
proj = currentProject;
rootPath = proj.RootFolder;
makeInfo.linkLibsObjs = {};
sysTarget = get_param(bdroot, 'RTWSystemTargetFile');
switch sysTarget
    case 'slrealtime.tlc'
        makeInfo.includePath = '<includePath>';
        makeInfo.linkLibsObjs{end+1} = '<libraryPath>';
    otherwise
        error('No rtwmakecfg found for %s target file', sysTarget);
end
end
```

- Enable linking for different target files.
- Use macros, such as `SIMULINK_REAL_TIME`, in your source code to add lines at compile time for real-time simulation. `SIMULINK_REAL_TIME` is useful to wrap the LOG function calls.

When cross-compiling, use macros such as `__unix__` and `__QNXNTO__` in your source code to add lines at compile time.

Hello World! Example External Code Integration for Simulink Real-Time

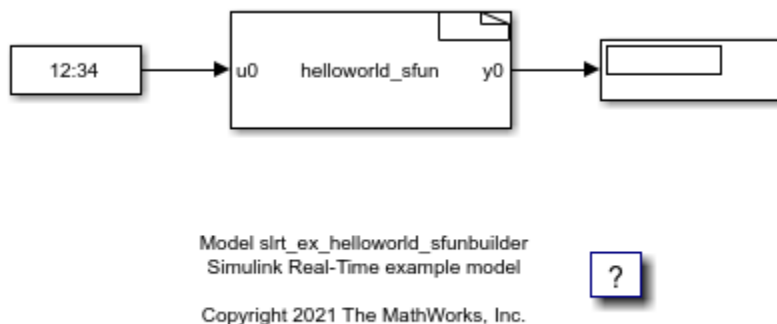
This example shows how to use an S-Function Builder block for external code integration. The example adds a hello message to the system log.

Before running this example, install the Simulink Real-Time Target Support Package. The support package includes the tools that compile the code that runs on the target computer.

Open the Model

Use the **Open Model** button to open the `slrt_ex_helloworld_sfunbuilder` model.

```
open_system(fullfile(matlabroot, 'toolbox', 'slrealtime', 'examples', 'slrt_ex_helloworld_sfunbuilder
```



Open the S-Function Block

Double-click the `helloworld_sfun` S-Function block. The S-Function Builder opens and displays the S-function code.

```
/* Includes_BEGIN */
#ifdef SIMULINK_REAL_TIME
#include "slrt_log.hpp"
#endif
/* Includes_END */

/* Externs_BEGIN */
/* extern double func(double a); */
/* Externs_END */

void helloworld_sfun_Start_wrapper(SimStruct *S)
{
/* Start_BEGIN */
```

```
/* Start_END */
}

void helloworld_sfun_Outputs_wrapper(const real_T *u0,
                                     real_T *y0,
                                     SimStruct *S)
{
/* Output_BEGIN */
// Create custom message
static char hellormsg[100];
sprintf(hellormsg, "Hello World! t=%f \n", *u0);
// Use macros for platform dependent code
#ifdef SIMULINK_REAL_TIME
slrealtime::log_info(hellormsg);
#else
ssPrintf(hellormsg);
#endif

// Generic platform independent code
*y0 = *u0;
/* Output_END */
}

void helloworld_sfun_Terminate_wrapper(SimStruct *S)
{
/* Terminate_BEGIN */
/*
 * Custom Terminate code goes here.
 */
/* Terminate_END */
}
```

Build Model and Run Real-Time Application

Before building the model, you can run the model on your desktop and view the output message in the Simulink Real-Time system log viewer.

When you are ready to build the model, on the Simulink Editor **Real-Time** tab, connect to the target computer and click **Run on Target**. Or, in the MATLAB Command Window, type:

```
tg = slrealtime;
connect(tg);
model = 'slrt_ex_helloworld_sfunbuilder';
evalc('slbuild(model)');
load(tg,model);
start(tg);
pause(20);
stop(tg);
```

View Message in Status Log

Open the target computer status log and view the Hello World! message. On the Simulink Editor **Real-Time** tab, select **Prepare > SLRT Explorer**. Then, select the **System Log Viewer** tab. Or, in the MATLAB Command Window, type:

```
slrtLogViewer;
```

The viewer shows the Hello World! messages in the system log.

Timestamp	Message	Se...	Categ...
25-06-2021 22:22:15...	Loading model lamp	info	0
25-06-2021 22:22:15...	Ready to start	info	0
25-06-2021 22:22:17...	Starting model lamp	info	0
25-06-2021 22:22:34...	TET 0 avg: 1.467e-06 min: 1.301e-06 max: 2.209e-06	info	0
25-06-2021 22:22:34...	Stopping model lamp at 17s	info	0
27-06-2021 21:22:33...	Loading model helloworld	info	0
27-06-2021 21:22:33...	Ready to start	info	0
27-06-2021 21:22:40...	Starting model helloworld	info	0
27-06-2021 21:22:41...	Hello World! t=0.000000	info	100
27-06-2021 21:22:42...	Hello World! t=1.000000	info	100
27-06-2021 21:22:43...	Hello World! t=2.000000	info	100
27-06-2021 21:22:44...	Hello World! t=3.000000	info	100
27-06-2021 21:22:45...	Hello World! t=4.000000	info	100
27-06-2021 21:22:46...	Hello World! t=5.000000	info	100
27-06-2021 21:22:47...	Hello World! t=6.000000	info	100
27-06-2021 21:22:48...	Hello World! t=7.000000	info	100
27-06-2021 21:22:49...	Hello World! t=8.000000	info	100
27-06-2021 21:22:50...	Hello World! t=9.000000	info	100
27-06-2021 21:22:51...	Hello World! t=10.000000	info	100
27-06-2021 21:22:51...	Stopping model helloworld at 10s	info	0
27-06-2021 21:22:51...	TET 0 avg: 2.8876e-05 min: 2.7718e-05 max: 5.3294e-05	info	0

Close All Files

```
bdclose('all');
```

Additional C/C++ Project for Simulink Real-Time

The eCAL Toolbox for Simulink project on github.com/mathworks/ecal-toolbox shows complete external code integration with Simulink Real-Time, including S-function wrappers, `rtwmakecfg` customization, and shared object compilation. You also can simulate this example on your development computer.

See Also

More About

- “Build Support for S-Functions”
- “Compile Source Code for Functional Mock-up Units” on page 3-3
- “Troubleshoot Model Links to Static Libraries or Shared Objects” on page 17-24

- “Troubleshoot Cannot Load Shared Object on Target Computer” on page 17-13

External Websites

- MathWorks Help Center website
- CMake website

See Also

Simulation Data Inspector

- “View Data in the Simulation Data Inspector” on page 12-2
- “Import Data from a CSV File into the Simulation Data Inspector” on page 12-11
- “Microsoft Excel Import, Export, and Logging Format” on page 12-16
- “Configure the Simulation Data Inspector” on page 12-24
- “How the Simulation Data Inspector Compares Data” on page 12-32
- “Save and Share Simulation Data Inspector Data and Views” on page 12-37
- “Inspect and Compare Data Programmatically” on page 12-43
- “Limit the Size of Logged Data” on page 12-49

View Data in the Simulation Data Inspector

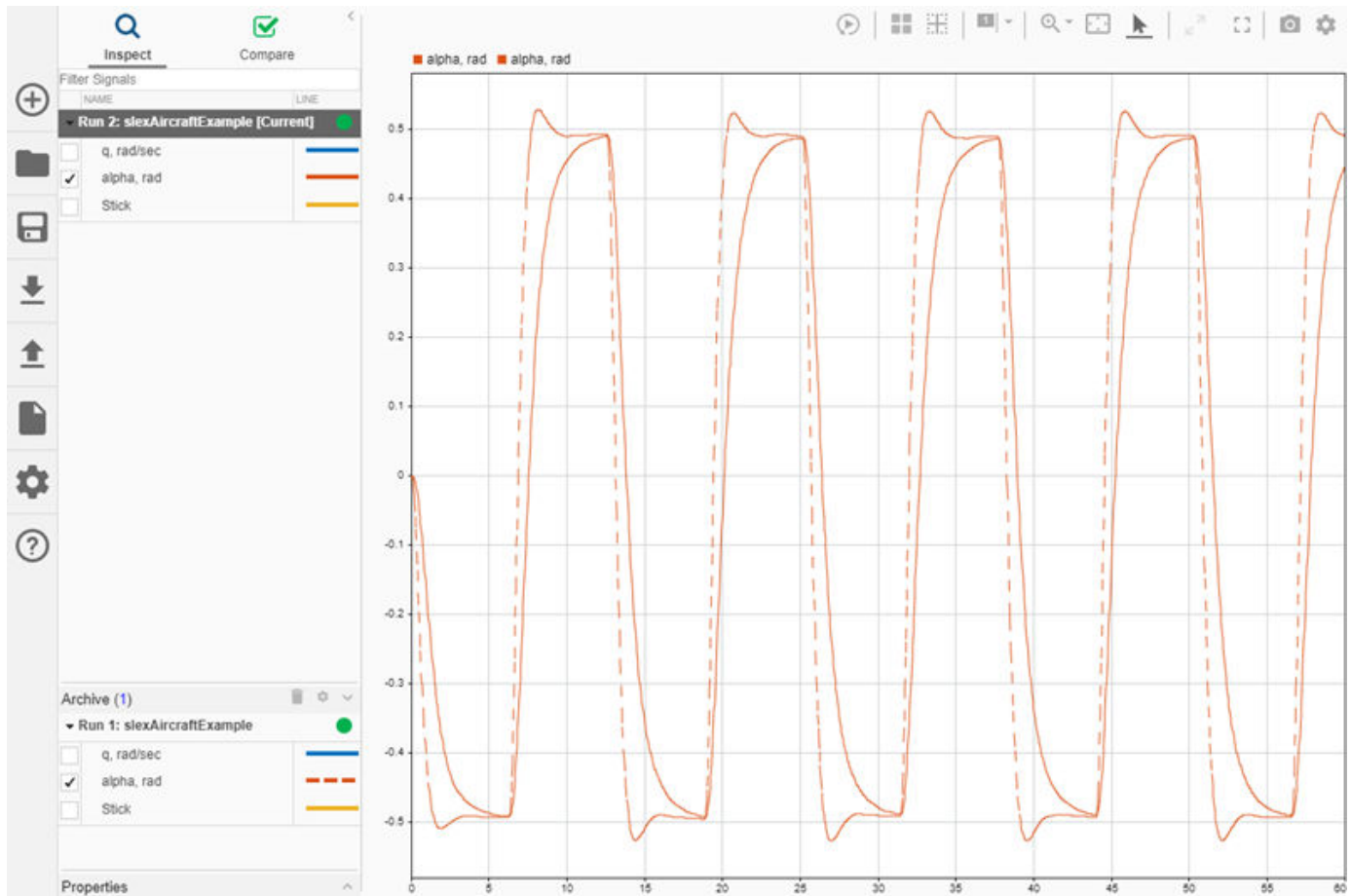
You can use the Simulation Data Inspector to visualize the data you generate throughout the design process. Simulation data that you log in a Simulink model logs to the Simulation Data Inspector. You can also import test data and other recorded data into the Simulation Data Inspector to inspect and analyze it alongside the logged simulation data. The Simulation Data Inspector offers several types of plots, which allow you to easily create complex visualizations of your data.

View Logged Data

Logged signals as well as outputs and states logged using the `Dataset` format automatically log to the Simulation Data Inspector when you simulate a model. You can also record other kinds of simulation data so the data appears in the Simulation Data Inspector at the end of the simulation. To see states and output data logged using a format other than `Dataset` in the Simulation Data Inspector, open the Configuration Parameters dialog box and, in the **Data Import/Export** pane, select the **Record logged workspace data in Simulation Data Inspector** parameter.

Note When you log states and outputs using the `Structure` or `Array` format, you must also log time for the data to record to the Simulation Data Inspector.

The Simulation Data Inspector displays available data in the table in the **Inspect** pane. To plot a signal, select the check box next to the signal. You can modify the layout and add different visualizations to analyze the simulation data. For more information, see “Create Plots Using the Simulation Data Inspector”.



The Simulation Data Inspector manages incoming simulation data using the archive. By default, the previous run moves to the archive when you start a new simulation. You can plot signals from the archive, or you can drag runs of interest back into the work area.

Import Data from the Workspace or a File

You can import data from the base workspace or from a file to view on its own or alongside simulation data. The Simulation Data Inspector supports all built-in data types and many data formats for importing data from the workspace. In general, whatever the format, sample values must be paired with sample times. The Simulation Data Inspector allows up to 8000 channels per signal in a run created from imported workspace data.

You can also import data from these types of files:

- MAT file
- CSV file — Format data as shown in “Import Data from a CSV File into the Simulation Data Inspector”.
- Microsoft Excel® file — Format data as described in “Microsoft Excel Import, Export, and Logging Format”.

- **MDF file** — MDF file import is supported for Linux® and Windows operating systems. The MDF file must have a `.mdf`, `.mf4`, `.mf3`, `.data`, or `.dat` file extension and contain data with only integer and floating data types.
- **ULG file** — Flight log data import requires a UAV Toolbox license.

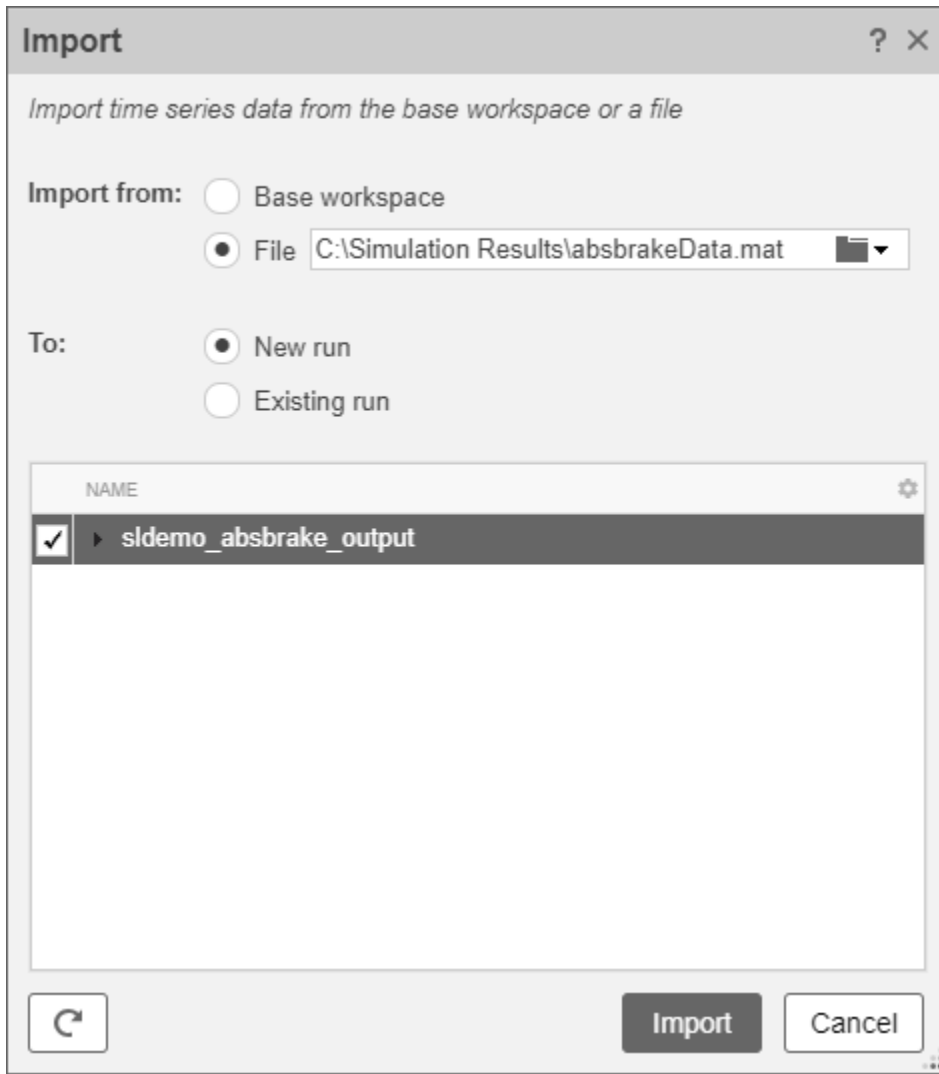
To import data from the workspace or from a file that is saved in a data or file format that the Simulation Data Inspector does not support, you can write your own workspace data or file reader to import the data using the `io.reader` class. You can also write a custom reader to use instead of the built-in reader for supported file types. For examples, see:

- “Import Data Using a Custom File Reader”
- “Import Workspace Variables Using a Custom Data Reader”



To import data, select the **Import** button in the Simulation Data Inspector.

In the Import dialog, you can choose to import data from the workspace or from a file. The table below the options shows data available for import. If you do not see your workspace variable or file contents in the table, that means the Simulation Data Inspector does not have a built-in or registered reader that supports that data. You can select which data to import using the check boxes, and you can choose whether to import that data into an existing run or a new run.



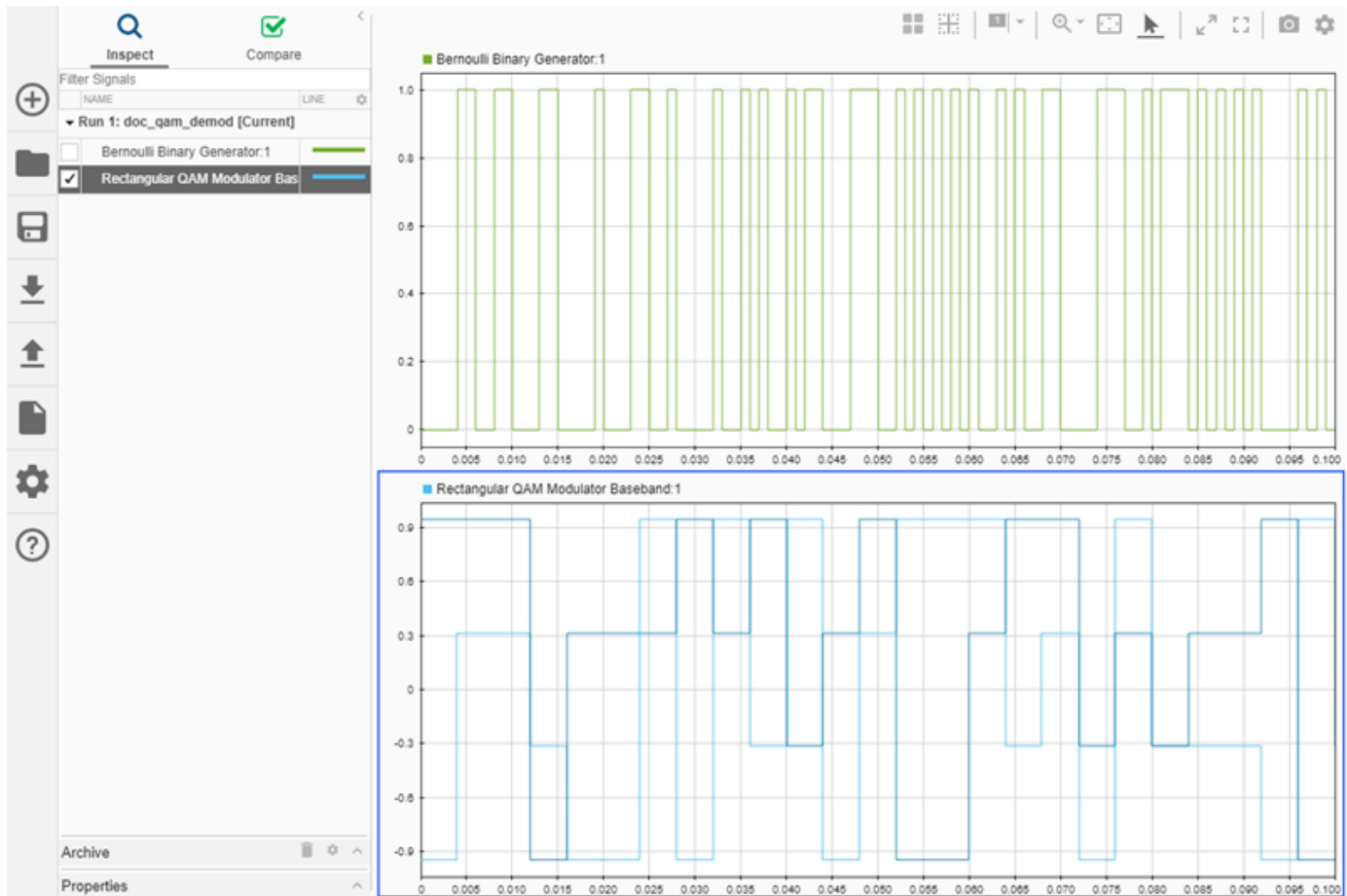
When you import data into a new run, the run always appears in the work area. You can manually move imported runs to the archive.

View Complex Data

To view complex data in the Simulation Data Inspector, import the data or log the signals to the Simulation Data Inspector. You can control how to visualize the complex signal using the **Properties** pane in the Simulation Data Inspector and in the **Instrumentation Properties** for the signal in the model. To access the **Instrumentation Properties** for a signal, right-click the logging badge for the signal and select **Properties**.

You can specify the **Complex Format** as Magnitude, Magnitude-Phase, Phase, or Real-Imaginary. If you select Magnitude-Phase or Real-Imaginary for the **Complex Format**, the Simulation Data Inspector plots both components of the signal when you select the check box for the signal. For signals in Real-Imaginary format, the **Line Color** specifies the color of the real component of the signal, and the imaginary component is a different shade of the **Line Color**. For example, the Rectangular QAM Modular Baseband signal on the lower graph displays the real component of

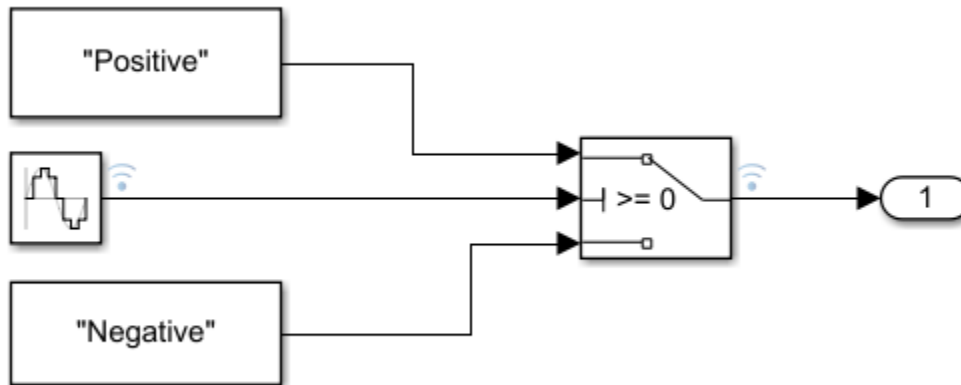
the signal in light blue, matching the **Line Color** parameter, and the imaginary component is shown in a darker shade of blue.



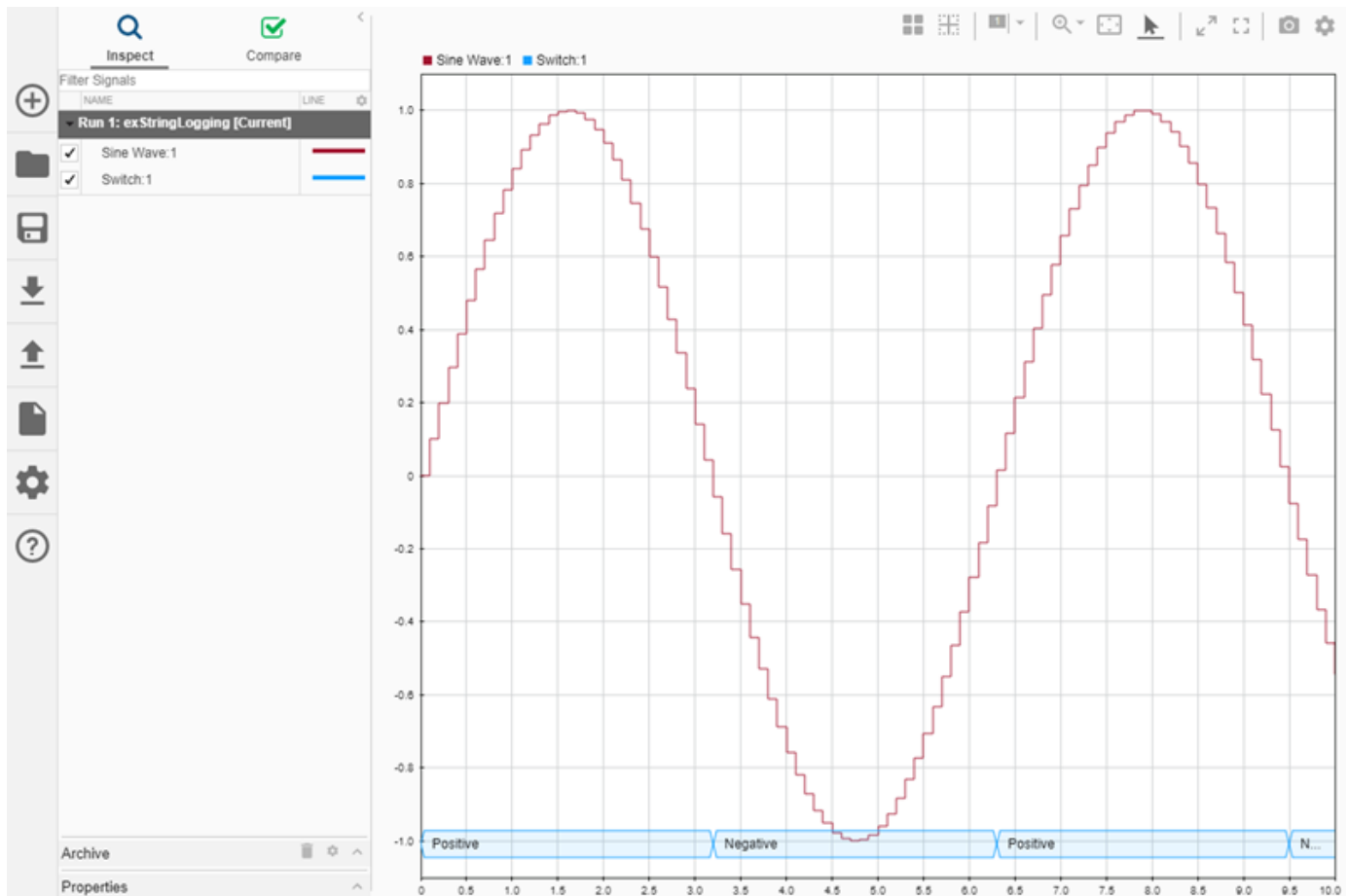
For signals in Magnitude-Phase format, the **Line Color** specifies the color of the magnitude component, and the phase is displayed in a different shade of the **Line Color**.

View String Data

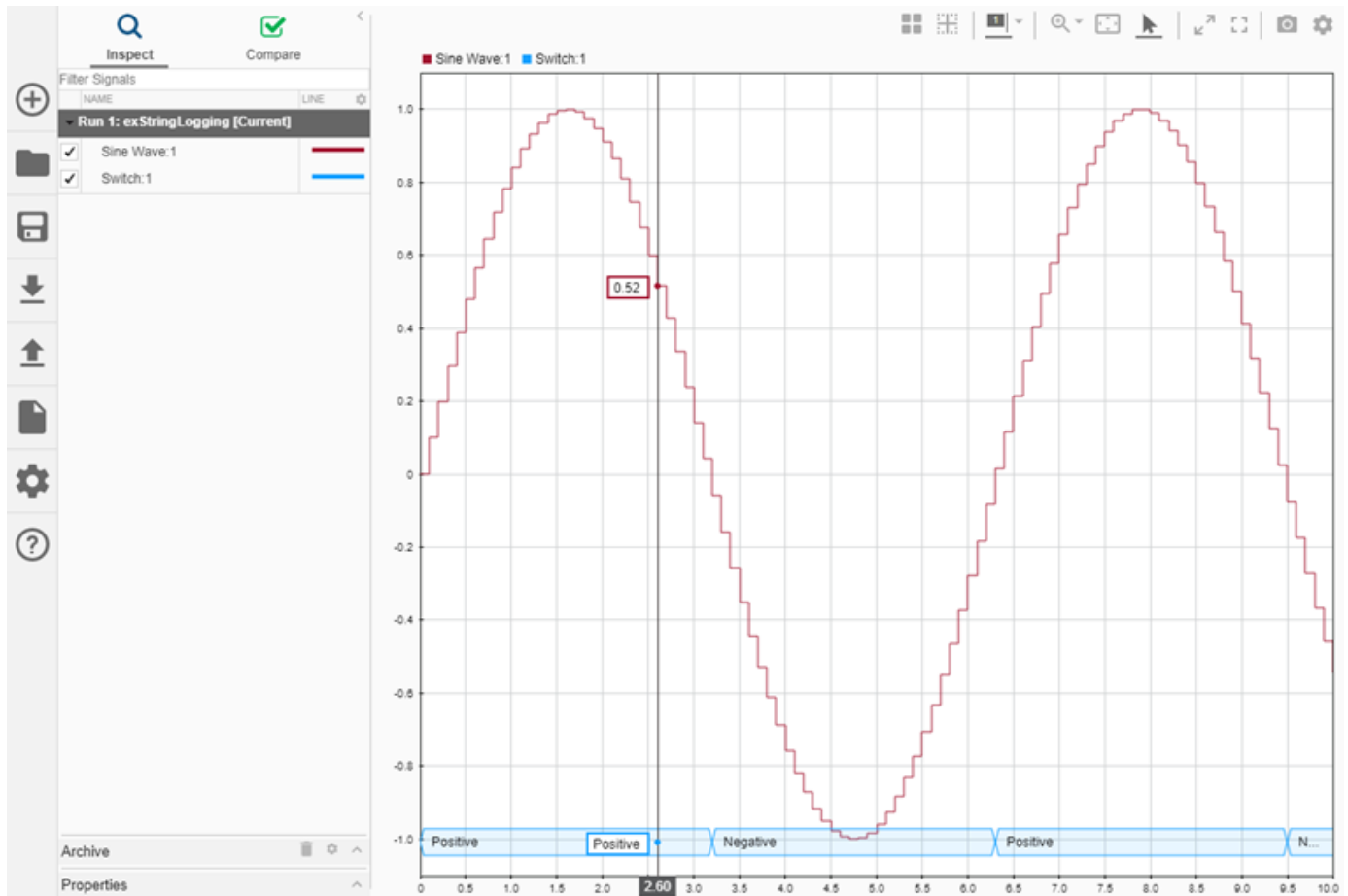
You can log and view string data with your signal data in the Simulation Data Inspector. For example, consider this simple model. The value of the sine wave block controls whether the switch sends a string reading Positive or Negative to the output.



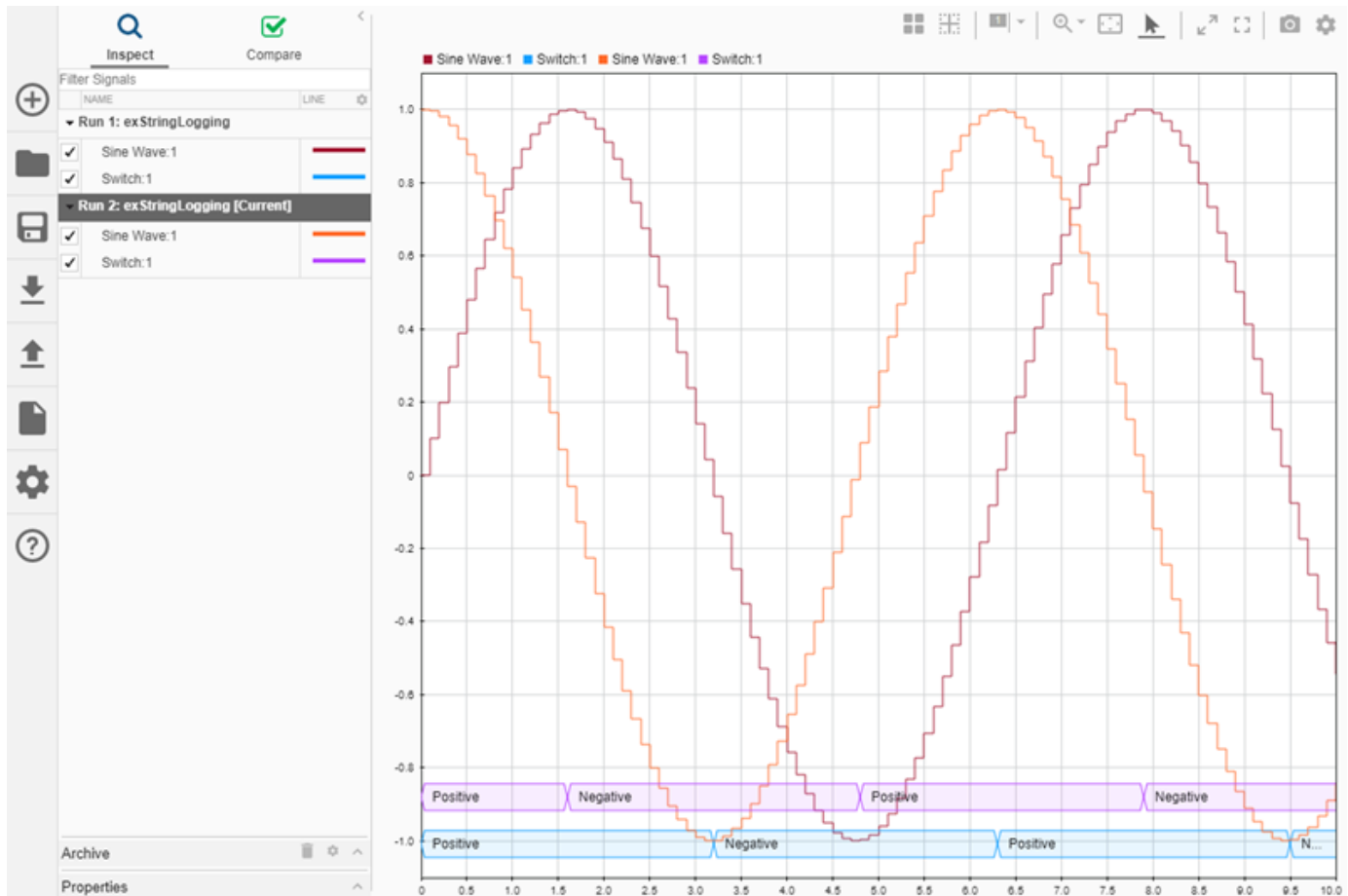
The plot shows the results of simulating the model. The string signal is shown at the bottom of the graphical viewing area. The value of the signal is displayed inside a band, and transitions in the string signal's value are marked with criss-crossed lines.



You can use cursors to inspect how the string signal values correspond with the sine signal's values.



When you plot multiple string signals on a plot, the signals stack in the order they were simulated or imported, with the most recent signal positioned at the top. For example, you might consider the effect of changing the phase of the sine wave controlling the switch.

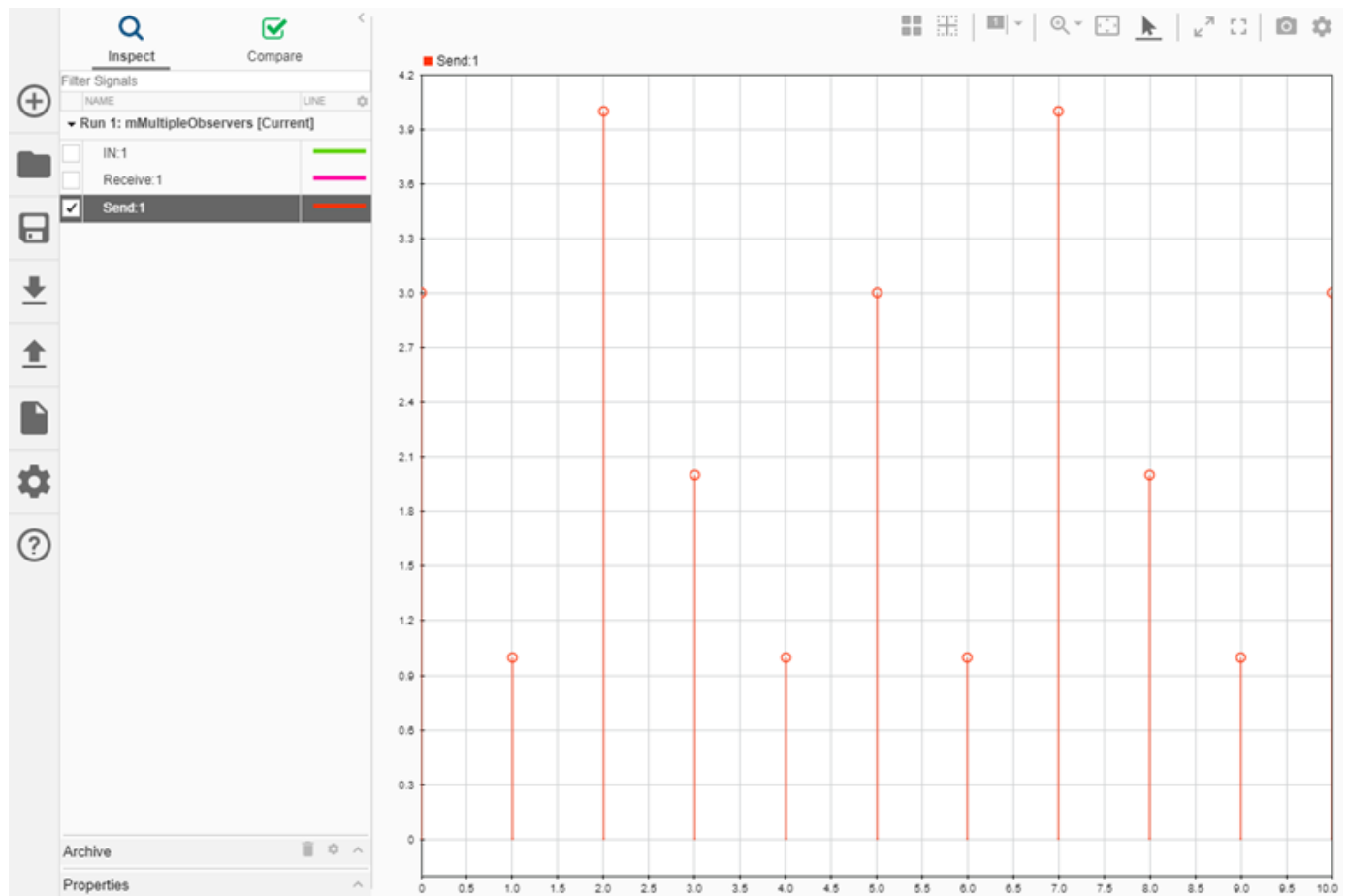


View Frame-Based Data

Processing data in frames rather than point by point provides a performance boost needed in some applications. To view frame-based data in the Simulation Data Inspector, you have to specify that the signal is frame-based in the **Instrumentation Properties** for the signal. To access the **Instrumentation Properties** dialog for a signal, right-click the signal's logging badge and select **Properties**. To specify a signal as frame-based, select **Columns as channels (frame based)** for **Input processing**.

View Event-Based Data

You can log or import event data to the Simulation Data Inspector. To view the logged event-based data, select the check box next to **Send: 1**. The Simulation Data Inspector displays the data as a stem plot, with each stem representing the number of events that occurred for a given sample time.



See Also

More About

- Inspect Simulation Data
- Compare Simulation Data
- Share Simulation Data Inspector Data and Views on page 12-37
- Decide How to Visualize Data
- Dataset Conversion for Logged Data

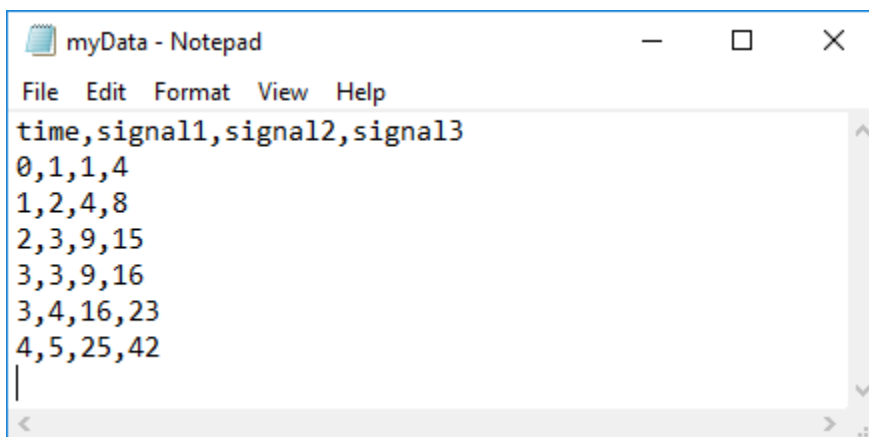
Import Data from a CSV File into the Simulation Data Inspector

To import data into the Simulation Data Inspector from a CSV file, format the data in the CSV file. Then, you can import the data using the Simulation Data Inspector UI or the `Simulink.sdi.createRun` function.

Tip When you want to import data from a CSV file where the data is formatted differently from the specification in this topic, you can write your own file reader for the Simulation Data Inspector using the `io.reader` class.

Basic File Format

In the simplest format, the first row in the CSV file is a header that lists the names of the signals in the file. The first column is time. The name for the time column must be `time`, and the time values must increase monotonically. The rows below the signal names list the signal values that correspond to each time step.



```
myData - Notepad
File Edit Format View Help
time,signal1,signal2,signal3
0,1,1,4
1,2,4,8
2,3,9,15
3,3,9,16
3,4,16,23
4,5,25,42
```

The import operation does not support time data that includes `Inf` or `NaN` values or signal data that includes `Inf` values. Empty or `NaN` signal values render as missing data. All built-in data types are supported.

Multiple Time Vectors

When your data includes signals with different time vectors, the file can include more than one time vector. Every time column must be named `time`. Time columns specify the sample times for signals to the right, up to the next time vector. For example, the first time column defines the time for `signal1` and `signal2`, and the second time column defines the time steps for `signal3`.

```

myData - Notepad
File Edit Format View Help
time,signal1,signal2,time,signal3
0,1,1,0,4
1,2,4,2,8
2,3,9,3,15
3,3,9,5,16
3,4,16
4,5,25

```

Signal columns must have the same number of data points as the associated time vector.

Signal Metadata

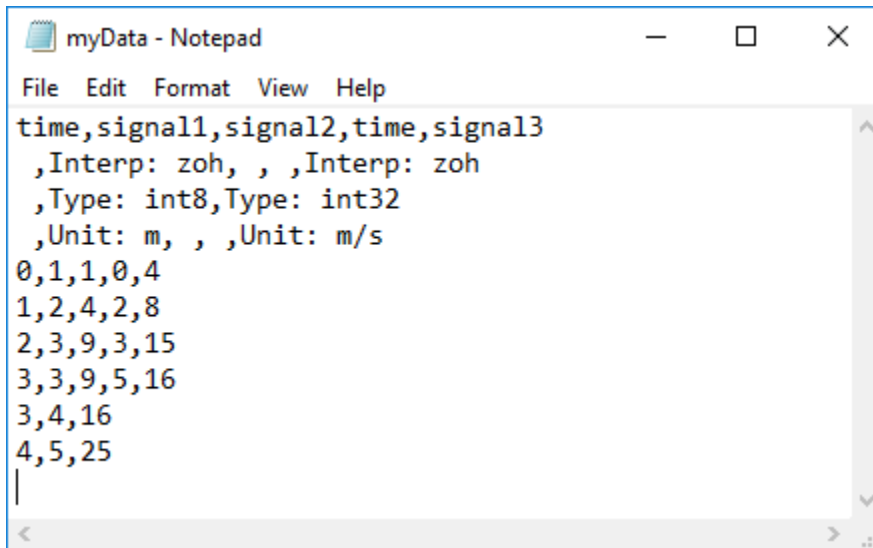
You can specify signal metadata in the CSV file to indicate the signal data type, units, interpolation method, block path, and port index. List metadata for each signal in rows between the signal name and the signal data. Label metadata according to this table.

Signal Property	Label	Value
Data type	Type:	Built-in data type.
Units	Unit:	Supported unit. For example, Unit: m/s specifies units of meters per second. For a list of supported units, enter <code>showunitslist</code> in the MATLAB Command Window.
Interpolation method	Interp:	linear, zoh for zero order hold, or none.
Block Path	BlockPath:	Path to the block that generated the signal.
Port Index	PortIndex:	Integer.

You can also import a signal with a data type defined by an enumeration class. Instead of using the Type: label, use the Enum: label and specify the value as the name of the enumeration class. The definition for the enumeration class must be saved on the MATLAB path.

When an imported file does not specify signal metadata, the Simulation Data Inspector assumes double data type and linear interpolation. You can specify the interpolation method as linear, zoh (zero-order hold), or none. If you do not specify units for the signals in your file, you can assign units to the signals in the Simulation Data Inspector after you import the file.

You can specify any combination of metadata for each signal. Leave a blank cell for signals with less specified metadata.



```

myData - Notepad
File Edit Format View Help
time,signal1,signal2,time,signal3
,Interp: zoh, , ,Interp: zoh
,Type: int8,Type: int32
,Unit: m, , ,Unit: m/s
0,1,1,0,4
1,2,4,2,8
2,3,9,3,15
3,3,9,5,16
3,4,16
4,5,25
|

```

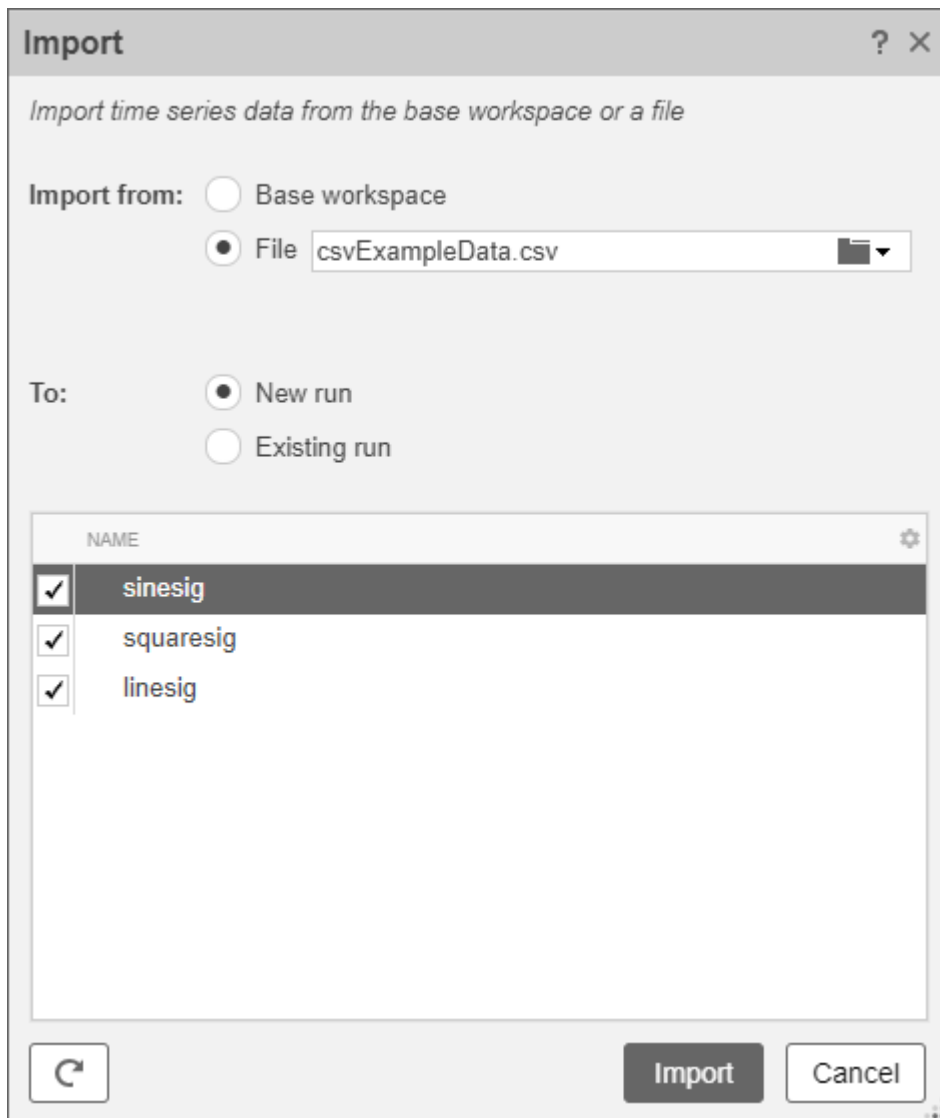
Import Data from a CSV File

You can import data from a CSV file using the Simulation Data Inspector UI or using the `Simulink.sdi.createRun` function.

To import data using the UI, open the Simulation Data Inspector using the `Simulink.sdi.view` function or the **Data Inspector** button in the Simulink™ toolstrip. Then, click the **Import** button.



In the Import dialog, select the option to import data from a file and navigate in the file system to select the file. After you select the file, data available for import shows in the table. You can choose which signals to import and whether to import them to a new or existing run. This example imports all available signals to a new run. After selecting the options, click the **Import** button.



When you import data into a new run using the UI, the new run name includes the run number followed by `Imported_Data`.

When you import data programmatically, you can specify the name of the imported run.

```
csvRunID = Simulink.sdi.createRun('CSV File Run', 'file', 'csvExampleData.csv');
```

See Also

Functions

`Simulink.sdi.createRun`

More About

- “View Data in the Simulation Data Inspector”

- “Microsoft Excel Import, Export, and Logging Format”
- “Import Data Using a Custom File Reader”

Microsoft Excel Import, Export, and Logging Format

Using the Simulation Data Inspector or Simulink Test, you can import data from a Microsoft Excel file or export data to a Microsoft Excel file. You can also log data to an Excel file using the Record block. The Simulation Data Inspector, Simulink Test, and the Record block all use the same file format, so you can use the same Microsoft Excel file with multiple applications.

Tip When the format of the data in your Excel file does not match the specification in this topic, you can write your own file reader to import the data using the `io.reader` class.

Basic File Format

In the simplest format, the first row in the Excel file is a header that lists the names of the signals in the file. The first column is time. The name for the time column must be `time`, and the time values must increase monotonically. The rows below the signal names list the signal values that correspond to each time step.

	A	B	C	D
1	<code>time</code>	<code>signal1</code>	<code>signal2</code>	<code>signal3</code>
2	0	1	1	4
3	1	2	4	8
4	2	3	9	15
5	3	3	9	16
6	3	4	16	23
7	4	5	25	42

The import operation does not support time data that includes `Inf` or `NaN` values or signal data that includes `Inf` values. Empty or `NaN` signal values imported from the Excel file render as missing data in the Simulation Data Inspector. All built-in data types are supported.

Multiple Time Vectors

When your data includes signals with different time vectors, the file can include more than one time vector. Every time column must be named `time`. Time columns specify the sample times for signals to the right, up to the next time vector. For example, the first time column defines the time for `signal1` and `signal2`, and the second time column defines the time steps for `signal3`.

	A	B	C	D	E
1	<code>time</code>	<code>signal1</code>	<code>signal2</code>	<code>time</code>	<code>signal3</code>
2	0	1	1	0	4
3	1	2	4	2	8
4	2	3	9	3	15
5	3	3	9	5	16
6	3	4	16		
7	4	5	25		

Signal columns must have the same number of data points as the associated time vector.

Signal Metadata

The file can include metadata for signals such as data type, units, and interpolation method. The metadata is used to determine how to plot the data, how to apply unit and data conversions, and how to compute comparison results. For more information about how metadata is used in comparisons, see “How the Simulation Data Inspector Compares Data”.

Metadata for each signal is listed in rows between the signal names and the signal data. You can specify any combination of metadata for each signal. Leave a blank cell for signals with less specified metadata.

	A	B	C	D	E
1	time	signal1	signal2	time	signal3
2		Interp: zoh			Interp: zoh
3		Type: int8	Type: int32		
4		Unit: m			Unit: m/s
5	0	1	1	0	4
6	1	2	4	2	8
7	2	3	9	3	15
8	3	3	9	5	16
9	3	4	16		
10	4	5	25		

Label each piece of metadata according to this table. The table also indicates which tools and operations support each piece of metadata. When an imported file does not specify signal metadata, double data type, linear interpolation, and union synchronization are used.

Property Descriptions

Signal Property	Label	Values	Simulation Data Inspector Import	Record Block Logging and Simulation Data Inspector Export	Simulink Test Import and Export
Data type	Type:	Built-in data type.	Supported	Supported	Supported
Units	Unit:	Supported unit. For example, Unit: m/s specifies units of meters per second. For a list of supported units, enter showunitslist in the MATLAB Command Window.	Supported	Supported	Supported
Interpolation method	Interp:	linear, zoh for zero order hold, or none.	Supported	Supported	Supported
Synchronization method	Sync:	union or intersection.	Supported	Not Supported <i>Metadata not included in exported file.</i>	Supported
Relative tolerance	RelTol:	Percentage, represented as a decimal. For example, RelTol: 0.1 specifies a 10% relative tolerance.	Supported	Not Supported <i>Metadata not included in exported file.</i>	Supported
Absolute tolerance	AbsTol:	Numeric value.	Supported	Not Supported <i>Metadata not included in exported file.</i>	Supported
Time tolerance	TimeTol:	Numeric value, in seconds.	Supported	Not Supported <i>Metadata not included in exported file.</i>	Supported

Signal Property	Label	Values	Simulation Data Inspector Import	Record Block Logging and Simulation Data Inspector Export	Simulink Test Import and Export
Leading tolerance	LeadingTol :	Numeric value, in seconds.	Supported <i>Only visible in Simulink Test.</i>	Not Supported <i>Metadata not included in exported file.</i>	Supported
Lagging tolerance	LaggingTol :	Numeric Value, in seconds.	Supported <i>Only visible in Simulink Test.</i>	Not Supported <i>Metadata not included in exported file.</i>	Supported
Block Path	BlockPath :	Path to the block that generated the signal.	Supported	Supported	Supported
Port Index	PortIndex :	Integer.	Supported	Supported	Supported
Name	Name :	Signal name	Supported	Not Supported <i>Metadata not included in exported file.</i>	Supported

User-Defined Data Types

In addition to built-in data types, you can use other labels in place of the `DataType: label` to specify fixed-point, enumerated, alias, and bus data types.

Property Descriptions

Data Type	Label	Values	Simulation Data Inspector Import	Record Block Logging and Simulation Data Inspector Export	Simulink Test Import and Export
Enumeration	Enum:	Name of the enumeration class.	Supported <i>Enumeration class definition must be saved on the MATLAB path.</i>	Supported <i>Enumeration class definition must be saved on the MATLAB path.</i>	Supported <i>Enumeration class definition must be saved on the MATLAB path.</i>
Alias	Alias:	Name of a Simulink.AliasType object in the MATLAB workspace.	Supported <i>For matrix and complex signals, specify the alias data type on the first channel.</i>	Not Supported	Supported <i>For matrix and complex signals, specify the alias data type on the first channel.</i>
Fixed-point	Fixdt:	<ul style="list-style-type: none"> fixdt constructor. Name of a Simulink.NumericType object in the MATLAB workspace. Name of a fixed-point data type as described in “Fixed-Point Numbers in Simulink” (Fixed-Point Designer). 	Supported	Not Supported	Supported
Bus	Bus:	Name of a Simulink.Bus object in the MATLAB workspace.	Supported	Not Supported	Supported

When you specify the type using the name of a Simulink.Bus object and the object is not in the MATLAB workspace, the data still imports from the file. However, individual signals in the bus use data types described in the file rather than data types defined in the Simulink.Bus object.

Complex, Multidimensional, and Bus Signals

You can import and export complex, multidimensional, and bus signals using an Excel file. The signal name for a column of data indicates whether that data is part of a complex, multidimensional, or bus signal. Excel file import and export do not support array of bus signals.

Multidimensional signal names include index information in parentheses. For example, the signal name for a column might be `signal1(2,3)`. When you import data from a file that includes multidimensional signal data, elements in the data not included in the file take zero sample values with the same data type and complexity as the other elements.

Complex signal data is always in real-imaginary format. Signal names for columns containing complex signal data include `(real)` and `(imag)` to indicate which data each column contains. When you import data from a file that includes imaginary signal data without specifying values for the real component of that signal, the signal values for the real component default to zero.

Multidimensional signals can contain complex data. The signal name includes the indication for the index within the multidimensional signal and the real or imaginary tag. For example, `signal1(1,3) (real)`.

Dots in signal names specify the hierarchy for bus signals. For example:

- `bus.y.a`
- `bus.y.b`
- `bus.x`

	A	B	C	D	E
1	time	bus.y.a	bus.y.b	time	bus.x
2		Interp: zoh			Interp: zoh
3		Type: int8	Type: int32		
4		Unit: m			Unit: m/s
5	0	1	1	0	4
6	1	2	4	2	8
7	2	3	9	3	15
8	3	3	9	5	16
9	3	4	16		
10	4	5	25		

Tip When the name of your signal includes characters that could make it appear as though it were part of a matrix, complex signal, or bus, use the **Name** metadata option to specify the name you want the imported signal to use in the Simulation Data Inspector and Simulink Test.

Function-Call Signals

Signal data specified in columns before the first time column is imported as one or more function-call signals. The data in the column specifies the times at which the function-call signal was enabled. The imported signals have a value of 1 for the times specified in the column. The time values for function-call signals must be double, scalar, and real, and must increase monotonically.

When you export data from the Simulation Data Inspector, function-call signals are formatted the same as other signals, with a time column and a column for signal values.

Simulation Parameters

You can import data for parameter values used in simulation. In the Simulation Data Inspector, the parameter values are shown as signals. Simulink Test uses imported parameter values to specify values for those parameters in the tests it runs based on imported data.

Parameter data is specified using two or three columns. The first column specifies the parameter names, with the cell in the header row for that column labeled **Parameter:**. The second column specifies the value used for each parameter, with the cell in the header row labeled **Value:**. Parameter data may also include a third column that contains the block path associated with each parameter, with the cell in the header row labeled **BlockPath:**. Specify names, values, and block paths for parameters starting in the first row that contains signal data, below rows used to specify signal metadata. For example, this file specifies values for two parameters, X and Y.

	A	B	C	D	E	F	G
1	time	signal1	signal2	time	signal3	Parameter: Value:	
2		Interp: zoh			Interp: zoh		
3		Type: int8	Type: int32				
4		Unit: m			Unit: m/s		
5	0	1	1	0	4 X		2
6	1	2	4	2	8 Y		1.2
7	2	3	9	3	15		
8	3	3	9	5	16		
9	3	4	16				
10	4	5	25				

Multiple Runs

You can include data for multiple runs in a single file. Within a sheet, you can divide data into runs by labeling data with a simulation number and a source type, such as **Input** or **Output**. Specify the simulation number and source type as additional signal metadata, using the label **Simulation:** for the simulation number and the label **Source:** for the source type. The Simulation Data Inspector uses the simulation number and source type only to determine which signals belong in each run. Simulink Test uses the information to define inputs, parameters, and acceptance criteria for tests to run based on imported data.

You do not need to specify the simulation number and output type for every signal. Signals to the right of a signal with a simulation number and source use the same simulation number and source until the next signal with a different source or simulation number. For example, this file defines data for two simulations and imports into four runs in the Simulation Data Inspector:

- **Run 1** contains signal1 and signal2.
- **Run 2** contains signal3, X, and Y.
- **Run 3** contains signal4.

- **Run 4** contains signal5.

	A	B	C	D	E	F	G	H	I	J
1	time	signal1	signal2	time	signal3	Parameter:	Values:	time	signal4	signal5
2		Interp: zoh			Interp: zoh					
3		Type: int8	Type: int32							
4		Unit: m			Unit: m/s					
5		Simulation: 1							Simulation: 2	
6		Source: Input			Source: Output				Source: Input	Source: Output
7	0	1	1	0	4 X		2	1	2	1
8	1	2	4	2	8 Y		1.2	2	6	3
9	2	3	9	3	15			3	4	5
10	3	3	9	5	16			4	8	7
11	3	4	16					5	10	2
12	4	5	25							

You can also use sheets within the Microsoft Excel file to divide the data into runs and tests. When you do not specify simulation number and source information, the data on each sheet is imported into a separate run in the Simulation Data Inspector. When you export multiple runs from the Simulation Data Inspector, the data for each run is saved on a separate sheet. When you import a Microsoft Excel file that contains data on multiple sheets into Simulink Test, you are prompted to specify how to import the data.

See Also

`Simulink.sdi.createRun` | `Simulink.sdi.exportRun`

More About

- “View Data in the Simulation Data Inspector”
- “Import Data from a CSV File into the Simulation Data Inspector”
- “Import Data Using a Custom File Reader”

Configure the Simulation Data Inspector

The Simulation Data Inspector supports a wide range of use cases for analyzing and visualizing data. You can modify preferences in the Simulation Data Inspector to match your visualization and analysis requirements. The preferences that you specify persist between MATLAB sessions.

By specifying preferences in the Simulation Data Inspector, you can configure options such as:

- How signals and metadata are displayed.
- Which data automatically imports from parallel simulations.
- Where prior run data is retained and how much prior data to store.
- How much memory is used during save operations.
- The system of units used to display signals.



To open the Simulation Data Inspector preferences, click Preferences.

Note You can restore all preferences in the Simulation Data Inspector to default values by clicking **Restore Defaults** in the Preferences menu or by using the `Simulink.sdi.clearPreferences` function.

Logged Data Size and Location

By default, simulation data logs to disk with data loaded into memory on demand, and the maximum size of logged data is constrained only by available disk space. You can use the **Disk Management** settings in the Simulation Data Inspector to directly control the size and location of logged data.

The **Record mode** setting specifies whether logged data is retained after simulation. When you change the **Record mode** setting to **View during simulation only**, no logged data is available in the Simulation Data Inspector or the workspace after the simulation completes. Only use this mode when you do not want to save logged data. The **Record mode** setting reverts to **View and record data** each time you start MATLAB. Changing the **Record mode** setting can affect other applications, such as visualization tools. For details, see “View Data Only During Simulation”.

To directly limit the size of logged data, you can specify a minimum amount of free disk space or a maximum size for the logged data. By default, logged data must leave at least 100 MB of free disk space with no maximum size limit. Specify the required disk space and maximum size in GB, and specify 0 to apply no disk space requirement or no maximum size limit.

When you specify a minimum disk space requirement or a maximum size for logged data, you can also specify whether to prioritize retaining data from the current simulation or data from prior simulations when approaching the limit. By default, the Simulation Data Inspector prioritizes retaining data for the current run by deleting data for prior runs. To prioritize retaining prior data, change the **When low on disk space** setting to **Keep prior runs and stop recording**. You see a warning message when prior runs are deleted and when recording is disabled. If recording is disabled due to the size of logged data, you need to change the **Record Mode** back to **View and**

record data to continue logging data, after you have freed up disk space. For more information, see “Specify a Minimum Disk Space Requirement or Maximum Size for Logged Data”.

The **Storage Mode** setting specifies whether to log data to disk or to memory. By default, data logs to disk. When you configure a parallel worker to log data to memory, data transfer back to the host is not supported. Logging data to memory is not supported for rapid accelerator simulations or models deployed using Simulink Compiler™.

You can also specify the location of the temporary file that stores logged data. By default, data logs to the temporary files directory on your computer. You may change the file location when you need to log large amounts of data and a secondary drive provides more storage capacity. Logging data to a network location can degrade performance.

Programmatic Use

You can programmatically configure and check each preference value.

Preference	Functions
Record mode	Simulink.sdi.setRecordData Simulink.sdi.getRecordData
Required Free Space	Simulink.sdi.setRequiredFreeSpace Simulink.sdi.getRequiredFreeSpace
Max Disk Usage	Simulink.sdi.setMaxDiskUsage Simulink.sdi.getMaxDiskUsage
When low on disk space	Simulink.sdi.setDeleteRunsOnLowSpace Simulink.sdi.getDeleteRunsOnLowSpace
Storage Mode	Simulink.sdi.setStorageMode Simulink.sdi.getStorageMode
Storage Location	Simulink.sdi.setStorageLocation Simulink.sdi.getStorageLocation

Archive Behavior and Run Limit

When you run multiple simulations in a single MATLAB session, the Simulation Data Inspector retains results from each simulation so you can analyze the results together. Use the Simulation Data Inspector archive to manage runs in the user interface and control the number of runs the Simulation Data Inspector retains.


You can configure a limit for the number of runs to retain in the archive and whether the Simulation Data Inspector automatically moves prior runs into the archive.

Manage Runs Using the Archive

By default, the Simulation Data Inspector automatically archives simulation runs. When you simulate a model, the prior simulation run moves to the archive, and the Simulation Data Inspector updates the view to show data for aligned signals in the current run.

The archive does not impose functional limitations on the runs and signals it contains. You can plot signals from the archive, and you can use runs and signals in the archive in comparisons. You can drag runs of interest from the archive to the work area and vice versa whether **Automatically Archive** is selected or disabled.

To prevent the Simulation Data Inspector from automatically moving prior simulations runs to the archive, clear the **Automatically archive** setting. With automatic archiving disabled, the Simulation Data Inspector does not move prior runs into the **Archive** pane or automatically update plots to display data from the current simulation.

Tip To manually delete the contents of the archive, click Delete archived runs .

Control Number of Runs Retained in Simulation Data Inspector

You can specify a limit for the number of runs to retain in the archive. When the number of runs in the archive reaches the limit, the Simulation Data Inspector deletes runs in the archive on a first-in, first-out basis.

The run limit applies only to runs in the archive. For the Simulation Data Inspector to automatically limit the data it retains by deleting old runs, select **Automatically archive** and specify a size limit.

By default, the Simulation Data Inspector retains the last 20 runs moved to the archive. To remove the limit, select **No limit**. To specify the maximum number of runs to store in the archive, select **Last n runs** and enter the limit. A warning occurs if you specify a limit that would delete runs already in the archive.

Programmatic Use

You can programmatically configure and check the archive behavior and run limit.

Preference	Functions
Automatically archive	<code>Simulink.sdi.setAutoArchiveMode</code> <code>Simulink.sdi.getAutoArchiveMode</code>
Size	<code>Simulink.sdi.setArchiveRunLimit</code> <code>Simulink.sdi.getArchiveRunLimit</code>

Incoming Run Names and Location

You can configure how the Simulation Data Inspector handles incoming runs from import or simulation. You can choose whether new runs are added at the top of the work area or the bottom and specify a naming rule to use for runs created from simulation.

By default, the Simulation Data Inspector adds new runs below prior runs in the work area. The **Archive** settings also affect the location of runs. By default, prior runs are moved to the archive when a new simulation run is created.

The run naming rule is used to name runs created from simulation. You can create the run naming rule using a mix of literal text that is used in the run name as-is and one or more tokens that represent metadata about the run. By default, the Simulation Data Inspector names runs using the run index and model name: Run <run_index>: <model_name>.

Tip To rename an existing run, double-click the name in the work area and enter the new name, or modify the run name in the **Properties** pane.

Programmatic Use

You can programmatically configure and check incoming run names and locations.

Preference	Functions
Add New Runs	Simulink.sdi.appendRunToTop Simulink.sdi.getAppendRunToTop
Naming Rule	Simulink.sdi.setRunNamingRule Simulink.sdi.getRunNamingRule Simulink.sdi.resetRunNamingRule

Signal Metadata to Display

You can control which signal metadata is displayed in the work area of the **Inspect** pane and in the results section on the **Compare** pane in the Simulation Data Inspector. You specify the metadata to display separately for each pane using the **Table Columns** preferences in the **Inspect** and **Compare** sections of the Preferences dialog, respectively.

Inspect Pane

By default, the signal name and the line style and color used to plot the signal are displayed on the **Inspect** pane. To display different or additional metadata in the work area on the **Inspect** pane, select the check box next to each piece of metadata you want to display in the **Table Columns** preference in the **Inspect** section. You can always view complete metadata for the selected signal in the **Inspect** pane using the **Properties** pane.

Note Metadata displayed in the work area on **Inspect** pane is included when you generate a report of plotted signals. You can also specify metadata to include in the report regardless of what is displayed in the work area when you create the report programmatically using the `Simulink.sdi.report` function.

Compare Pane

By default, the **Compare** pane shows the signal name, the absolute and relative tolerances used in the signal comparison, and the maximum difference from the comparison result. To display different or additional metadata in the results on the **Compare** pane, select the check box next to each piece of metadata you want to display in the **Table Columns** preference in the **Compare** section. You can always view complete metadata for the signals compared for a selected signal result using the **Properties** pane, where metadata that differs between the compared signals is highlighted. Signal metadata displayed on the **Compare** pane does not affect the contents of comparison reports.

Signal Selection on the Inspect Pane

You can configure how you select signals to plot on the selected subplot in the Simulation Data Inspector. By default, you use check boxes next to each signal to plot. You can also choose to plot signals based on selection in the work area. Use **Check Mode** when creating views and visualizations that represent findings and analysis of a data set. Use **Browse Mode** to quickly view and analyze data sets with a large number of signals.

For more information about creating visualizations using **Check Mode**, see “Create Plots Using the Simulation Data Inspector”.

For more information about using **Browse Mode**, see “Visualize Many Logged Signals”.

Note To use **Browse Mode**, your layout must include only **Time Plot** visualizations.

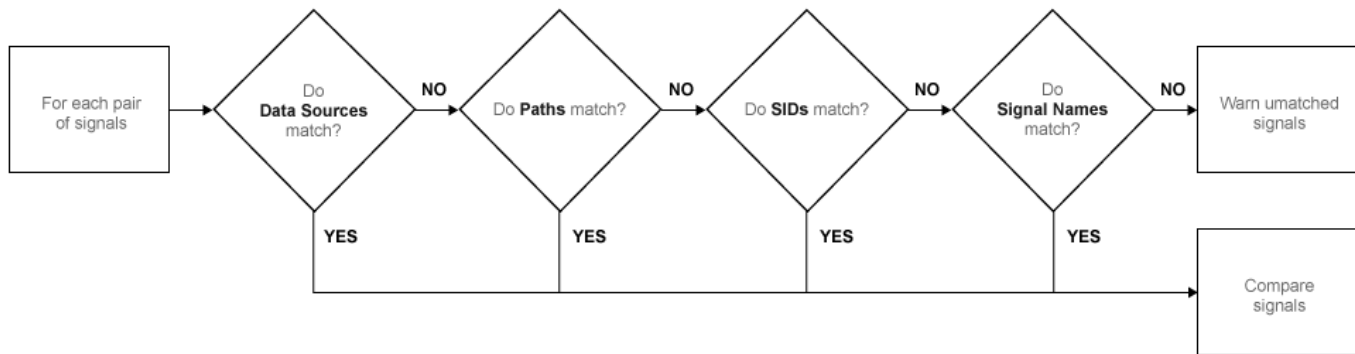
How Signals Are Aligned for Comparison

When you compare runs using the Simulation Data Inspector, the comparison algorithm pairs signals for signal comparison through a process called alignment. You can align signals between the compared runs using one or more of the signal properties shown in the table.

Property	Description
Data Source	Path of the variable in the MATLAB workspace for data imported from the workspace
Path	Block path for the source of the data in its model
SID	Automatically assigned Simulink identifier
Signal Name	Name of the signal

You can specify the priority for each piece of metadata used for alignment. The **Align By** field specifies the highest priority property used to align signals. The priority drops with each subsequent **Then By** field. You must specify a primary alignment property in the **Align By** field, but you can leave any number of **Then By** fields blank.

By default, the Simulation Data Inspector aligns signals between runs according to this flow chart.



For more information about configuring comparisons in the Simulation Data Inspector, see “How the Simulation Data Inspector Compares Data”.

Colors Used to Display Comparison Results

You can configure the colors used to display comparison results using the Simulation Data Inspector preferences. You can specify whether to use the signal color from the **Inspect** pane or a fixed color for the baseline and compared signals. You can also choose colors for the tolerance and the difference signal.

By default, the Simulation Data Inspector displays comparison results using fixed colors for the baseline and compared signals. Using a fixed color allows you to avoid the baseline signal color and compared signal color being either the same or too similar to distinguish.

Signal Grouping

You can specify how to group signals within a run in the Simulation Data Inspector. The preferences apply to both the **Inspect** and **Compare** panes and comparison reports. You can group signals by:

- **Domain** — Signal type. For example, signals created by signal logging have a domain of **Signal**, while signals created from logging model outputs have a domain of **Outputs**.
- **Physical System Hierarchy** — Signal Simscape™ physical system hierarchy. The option to group by physical system hierarchy is available when you have a Simscape license.
- **Data Hierarchy** — Signal location within structured data. For example, data hierarchy grouping reflects the hierarchy of a bus.
- **Model Hierarchy** — Signal location within model hierarchy. Grouping by model hierarchy can be helpful when you log data from a model that includes model or subsystem references.

Grouping signals adds rows for the hierarchical nodes, which you can expand to show the signals within that node. By default, the Simulation Data Inspector groups signals by domain, then by physical system hierarchy (if you have a Simscape license), and then by data hierarchy.

To remove grouping and display a flat list of signals in each run, select **None** for all grouping options.

Programmatic Use

To specify how to group signals programmatically, use the `Simulink.sdi.setTableGrouping` function.

Data to Stream from Parallel Simulations

When you run parallel simulations using the `parsim` function, you can stream logged simulation data to the Simulation Data Inspector. A dot next to the run name in the **Inspect** pane indicates the status of the simulation that corresponds to the run, so you can monitor simulation progress while visualizing the streamed data. You can control whether data streams from a parallel simulation based on the type of worker the data comes from.

By default, the Simulation Data Inspector is configured for manual import of data from parallel workers. You can use the Simulation Data Inspector programmatic interface to inspect the data on the worker and decide whether to send it to the client Simulation Data Inspector for further analysis. To manually move data from a parallel worker to the Simulation Data Inspector, use the `Simulink.sdi.sendWorkerRunToClient` function.

You may want to automatically stream data from parallel simulations that run on local workers or on local and remote workers. Streaming data from both local and remote workers may affect simulation performance, depending on how many simulations you run and how much data you log. When you choose to stream data from local workers or all parallel workers, all logged simulation data automatically shows in the Simulation Data Inspector.

Programmatic Use

You can configure Simulation Data Inspector support for parallel worker data programmatically using the `Simulink.sdi.enablePCTSupport` function.

Options for Saving and Loading Session Files

You can specify a maximum amount of memory to use while loading or saving a session file. By default, the Simulation Data Inspector uses a maximum of 100 MB of memory when you load or save a session file. You can specify a memory use limit as low as 50 MB.

To reduce the size of the saved session file, you can specify a compression option.

- **None** — Do not compress saved data.
- **Normal** — Compress the saved file as much as possible.
- **Fastest** — Compress the saved file less than **Normal** compression for faster save time.

Signal Display Units

Signals in the Simulation Data Inspector have two units properties: stored units and display units. The stored units represent the units of the data saved to disk. The display units specify how the Simulation Data Inspector displays the data. You can configure the Simulation Data Inspector to use a system of units to define the display units for all signals. You can choose either the **SI** or **US Customary** system of units, or you can display data using its stored units.

When you use a system of units to define display units for signals in the Simulation Data Inspector, the display units update for any signal with display units that are not valid for that unit system. For example, if you select **SI** units, the display units for a signal may update from ft to m.

Note The system of units you choose to use in the Simulation Data Inspector does not affect the stored units for any signal. You can convert the stored units for a signal using the `convertUnits` function. Conversion may result in loss of precision.

In addition to selecting a system of units, you can specify override units so that all signals of a given measurement type are displayed using consistent units. For example, if you want to visualize all signals that represent weight using units of kg, specify kg as an override unit.

Tip For a list of units supported by Simulink, enter `showunitslist` in the MATLAB Command Window.

You can also modify the display units for a specific signal using the **Properties** pane. For more information, see “Modify Signal Properties in the Simulation Data Inspector”.

Programmatic Use

Configure the unit system and override units using the `Simulink.sdi.setUnitSystem` function. You can check the current units preferences using the `Simulink.sdi.getUnitSystem` function.

See Also

Functions

`Simulink.sdi.clearPreferences` | `Simulink.sdi.setRunNamingRule` |
`Simulink.sdi.setTableGrouping` | `Simulink.sdi.enablePCTSupport` |
`Simulink.sdi.setArchiveRunLimit` | `Simulink.sdi.setAutoArchiveMode`

More About

- “Iterate Model Design Using the Simulation Data Inspector”
- “How the Simulation Data Inspector Compares Data”
- “Compare Simulation Data”
- “Create Plots Using the Simulation Data Inspector”
- “Modify Signal Properties in the Simulation Data Inspector”

How the Simulation Data Inspector Compares Data

You can tailor the Simulation Data Inspector comparison process to fit your requirements in multiple ways. When comparing runs, the Simulation Data Inspector:

- 1 Aligns signal pairs in the **Baseline** and **Compare To** runs based on the **Alignment** settings.




The Simulation Data Inspector does not compare signals that it cannot align.

- 2 Synchronizes aligned signal pairs according to the specified **Sync Method**.

Values for time points added in synchronization are interpolated according to the specified **Interpolation Method**.

- 3 Computes the difference of the signal pairs.
- 4 Compares the difference result against specified tolerances.

When the comparison run completes, the results of the comparison are displayed in the navigation pane.

Status	Comparison Result
	Difference falls within the specified tolerance.
	Difference violates specified tolerance.
	The signal does not align with a signal from the Compare To run.

When you compare signals with differing time intervals, the Simulation Data Inspector compares the signals on their overlapping interval.

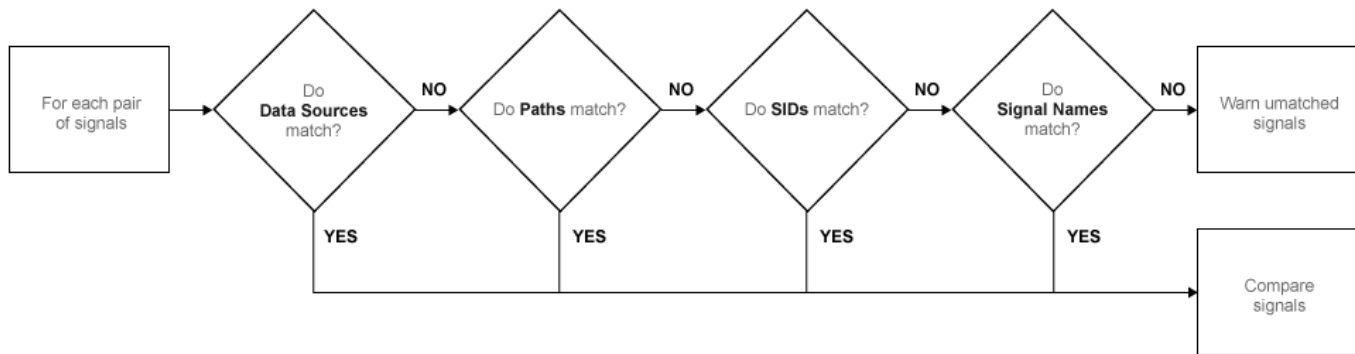
Signal Alignment

In the alignment step, the Simulation Data Inspector decides which signal from the **Compare To** run pairs with a given signal in the **Baseline** run. When you compare signals with the Simulation Data Inspector, you complete the alignment step by selecting the **Baseline** and **Compare To** signals.

The Simulation Data Inspector aligns signals using a combination of their Data Source, Path, SID, and Signal Name properties.

Property	Description
Data Source	Path of the variable in the MATLAB workspace for data imported from the workspace
Path	Block path for the source of the data in its model
SID	Automatically assigned Simulink identifier
Signal Name	Name of the signal in the model

With the default alignment settings, the Simulation Data Inspector aligns signals between runs according to this flow chart.

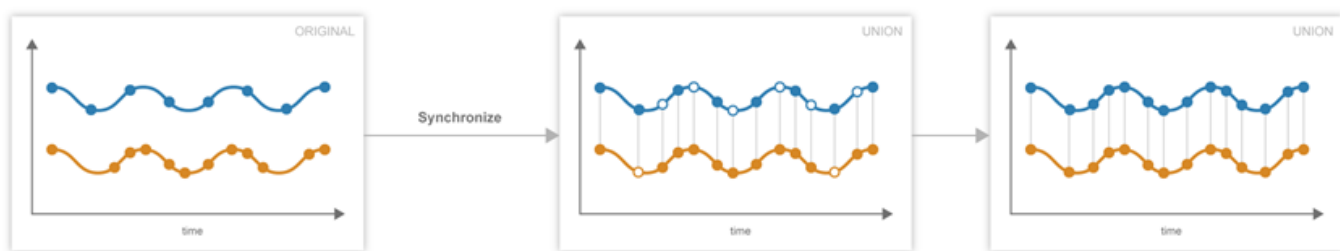


You can specify the priority for each of the signal properties used for alignment in the Simulation Data Inspector **Preferences**. The **Align By** field specifies the highest priority property used to align signals. The priority drops with each subsequent **Then By** field. You must specify a primary alignment property in the **Align By** field, but you can leave any number of the **Then By** fields blank.

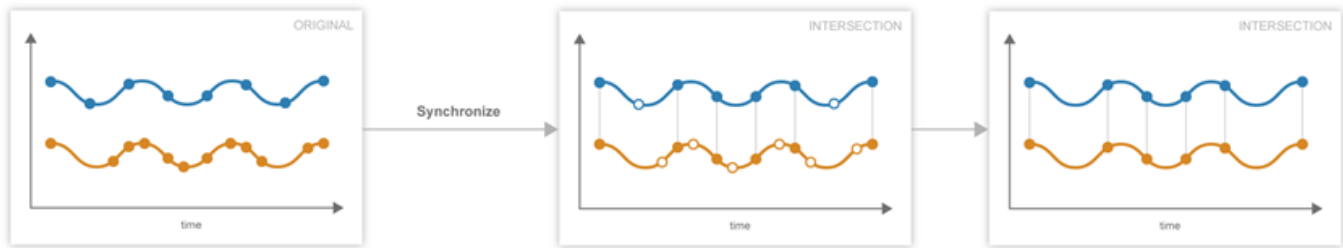
Synchronization

Often, signals that you want to compare don't contain the exact same set of time points. The synchronization step in Simulation Data Inspector comparisons resolves discrepancies in signals' time vectors. You can choose union or intersection as the synchronization method.

When you specify union synchronization, the Simulation Data Inspector builds a time vector that includes every sample time between the two signals. For each sample time not originally present in either signal, the Simulation Data Inspector interpolates the value. The second graph in the illustration shows the union synchronization process, where the Simulation Data Inspector identifies samples to add in each signal, represented by the unfilled circles. The final plot shows the signals after the Simulation Data Inspector has interpolated values for the added time points. The Simulation Data Inspector computes the difference using the signals in the final graph, so that the computed difference signal contains all the data points between the signals.



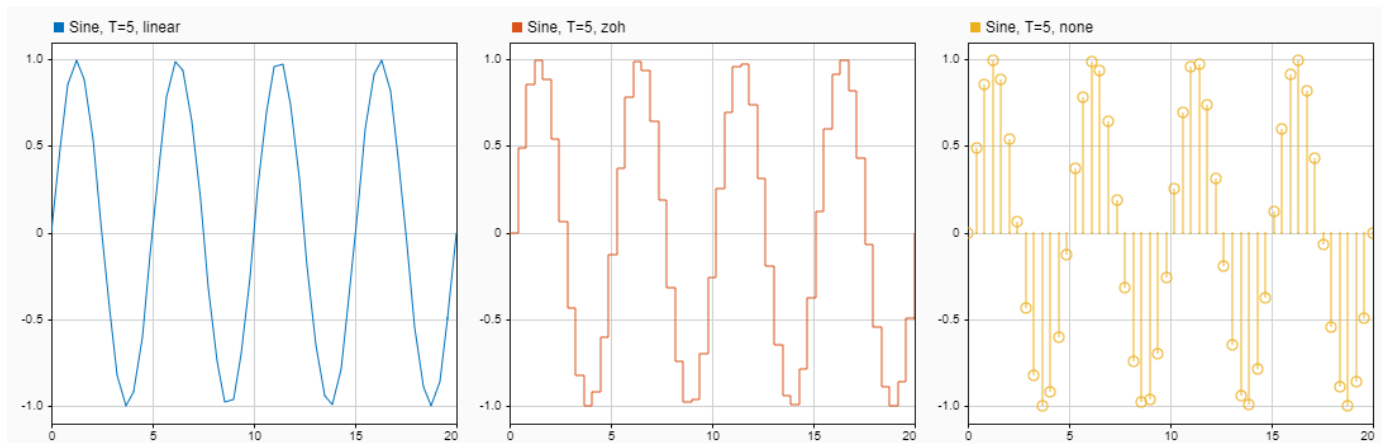
When you specify intersection synchronization, the Simulation Data Inspector uses only the sample times present in both signals in the comparison. In the second graph, the Simulation Data Inspector identifies samples that do not have a corresponding sample for comparison, shown as unfilled circles. The final graph shows the signals used for the comparison, without the samples identified in the second graph.



The choice between the synchronization options involves a trade off between speed and accuracy. The interpolation required by `union` synchronization takes time, but provides a more precise result. When you use `intersection` synchronization, the comparison finishes quickly because the Simulation Data Inspector computes the difference for fewer data points and does not interpolate. However, some data is discarded and precision is lost with `intersection` synchronization.

Interpolation

The interpolation property of a signal determines how the Simulation Data Inspector displays the signal and how additional data values are computed in synchronization. You can choose to interpolate your data with a zero-order hold (`zoh`) or a linear approximation. You can also specify no interpolation.



When you specify `zoh` or `none` for the **Interpolation Method**, the Simulation Data Inspector replicates the data of the previous sample for interpolated sample times. When you specify `linear` interpolation, the Simulation Data Inspector uses samples on either side of the interpolated point to linearly approximate the interpolated value. Typically, discrete signals use `zoh` interpolation and continuous signals use `linear` interpolation. You can specify the **Interpolation Method** for your signals in the signal properties.

Tolerance Specification

The Simulation Data Inspector allows you to specify the scope and value of the tolerance for your signal. You can define a tolerance band using any combination of absolute, relative, and time tolerance values, and you can specify whether the specified tolerance applies to an individual signal or to all the signals in a run.

Tolerance Scope

In the Simulation Data Inspector, you can specify the tolerance for your data globally or for an individual signal. Global tolerance values apply to all signals in a run that do not have **Override Global Tol** set to yes. You can specify global tolerance values for your data at the top of the graphical viewing area in the **Compare** view. To specify signal specific tolerance values, edit the signal properties and ensure the **Override Global Tol** property is set to yes.

Tolerance Computation

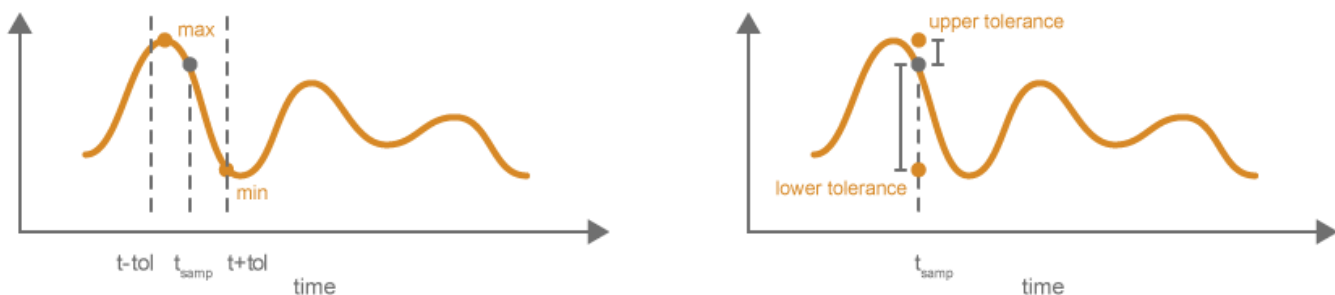
In the Simulation Data Inspector, you can specify a tolerance band for your run or signal using a combination of absolute, relative, and time tolerance values. When you specify the tolerance for your run or signal using multiple types of tolerances, each tolerance can yield a different answer for the tolerance at each point. The Simulation Data Inspector computes the overall tolerance band by selecting the most lenient tolerance result for each data point.

When you define your tolerance using only the absolute and relative tolerance properties, the Simulation Data Inspector computes the tolerance for each point as a simple maximum.

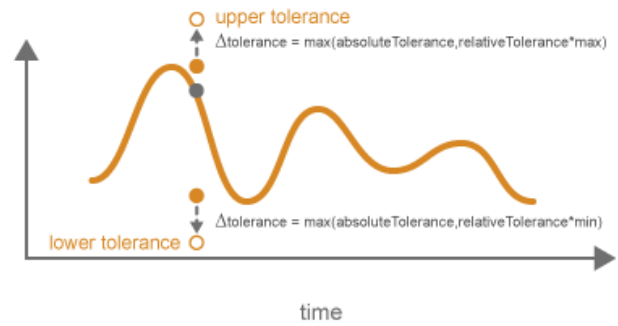
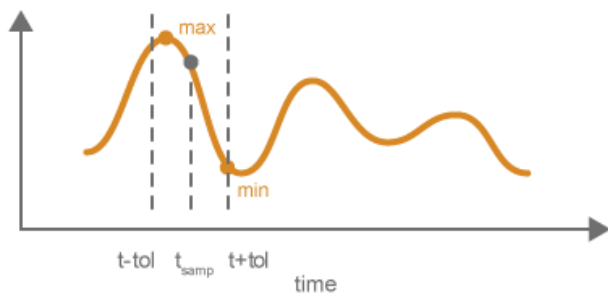
```
tolerance = max(absoluteTolerance, relativeTolerance*abs(baselineData));
```

The upper boundary of the tolerance band is formed by adding tolerance to the **Baseline** signal. Similarly, the Simulation Data Inspector computes the lower boundary of the tolerance band by subtracting tolerance from the **Baseline** signal.

When you specify a time tolerance, the Simulation Data Inspector evaluates the time tolerance first, over a time interval defined as $[(t_{\text{samp}} - \text{tol}), (t_{\text{samp}} + \text{tol})]$ for each sample. The Simulation Data Inspector builds the lower tolerance band by selecting the minimum point on the interval for each sample. Similarly, the maximum point on the interval defines the upper tolerance for each sample.



If you specify a tolerance band using an absolute or relative tolerance in addition to a time tolerance, the Simulation Data Inspector applies the time tolerance first, and then applies the absolute and relative tolerances to the maximum and minimum points selected with the time tolerance.



`upperTolerance = max + max(absoluteTolerance, relativeTolerance*max)`

`lowerTolerance = min - max(absoluteTolerance, relativeTolerance*min)`

Limitations

The Simulation Data Inspector does not support comparing:

- Signals of data types `int64` or `uint64`.
- Variable-size signals.

See Also

Related Examples

- “Compare Simulation Data”

Save and Share Simulation Data Inspector Data and Views

After you inspect, analyze, or compare your data in the Simulation Data Inspector, you can share your results with others. The Simulation Data Inspector provides several options for sharing and saving your data and results, depending on your needs. With the Simulation Data Inspector, you can:

- Save your data and layout modifications in a Simulation Data Inspector session.
- Share your layout modifications in a Simulation Data Inspector view.
- Share images and figures of plots you create in the Simulation Data Inspector.
- Create a Simulation Data Inspector report.
- Export data to the workspace.
- Export data to a file.

Save and Load Simulation Data Inspector Sessions

If you want to save or share data along with a configured view in the Simulation Data Inspector, save your data and settings in a Simulation Data Inspector session. You can save sessions as MAT- or MLDATX-files. The default format is MLDATX. When you save a Simulation Data Inspector session, the session file contains:

- All runs, data, and properties from the **Inspect** pane, including which run is the current run and which runs are in the archive.
- Plot display selection for signals in the **Inspect** pane.
- Subplot layout and line style and color selections.

Note Comparison results and global tolerances are not saved in Simulation Data Inspector sessions.

To save a Simulation Data Inspector session:

- 1 Hover over the save icon on the left side bar. Then, click **Save As**.



- 2 Name the file.
- 3 Browse to the location where you want to save the session, and click **Save**.


For large datasets, a status overlay in the bottom right of the graphical viewing area displays information about the progress of the save operation and allows you to cancel the save operation.

The **Save** tab of the Simulation Data Inspector preferences menu on the left side bar allows you to configure options related to save operations for MLDATX-files. You can set a limit as low as 50MB on the amount of memory used for the save operation. You can also select one of three **Compression** options:

- **None**, the default, applies no compression during the save operation.

- **Normal** creates the smallest file size.
- **Fastest** creates a smaller file size than you would get by selecting **None**, but provides a faster save time than **Normal**.



To load a Simulation Data Inspector session, click the open icon  on the left side bar. Then, browse to select the MLDATX-file you want to open, and click **Open**.

Alternatively, you can double-click the MLDATX-file. MATLAB and the Simulation Data Inspector open if they are not already open.

When the Simulation Data Inspector already contains runs and you open a session, all of the runs in the session move to the archive. The view updates to reflect show plotted signals from the session file. You can drag runs between the work area and archive as desired.


When the Simulation Data Inspector does not contain runs and you open a session, the Simulation Data Inspector puts runs in the work area and archive as specified in the file.

Share Simulation Data Inspector Views


When you have different sets of data that you want to visualize the same way, you can save a view. A view saves the layout and appearance characteristics of the Simulation Data Inspector without saving the data. Specifically, a view saves:

- Plot visualization type, layout, axis ranges, linking characteristics, and normalized axes
- Location of signals in the plots, including plotted signals in the archive
- Signal grouping and columns on display in the **Inspect** pane
- Signal color and line styling

To save a view:

- 1 Click Visualizations and layouts .
- 2 In **Saved Views**, click **Save current view**.
- 3 In the dialog box, specify a name for the view and browse to the location where you want to save the MLDATX-file.
- 4 Click **Save**.


To load a view:

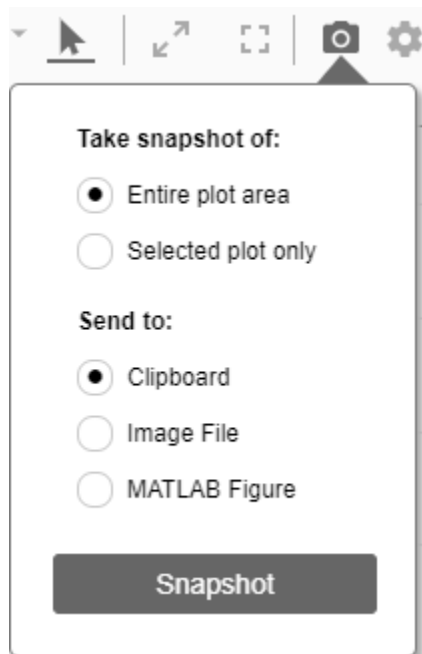
- 1 Click Visualizations and layouts .
- 2 In **Saved Views**, click **Open saved view**.
- 3 Browse to the view you would like to load, and click **Open**.

Share Simulation Data Inspector Plots

Use the snapshot feature to share the plots you generate in the Simulation Data Inspector. You can export your plots to the clipboard to paste into a document, as an image file, or to a MATLAB figure.

You can choose to capture the entire plot area, including all subplots in the plot area, or to capture only the selected subplot.

Click the camera icon  on the toolbar to access the snapshot menu. Use the radio buttons to select the area you want to share and how you want to share the plot. After you make your selections, click **Snapshot** to export the plot.



If you create an image, select where you would like to save the image in the file browser.

You can create snapshots of your plots in the Simulation Data Inspector programmatically using `Simulink.sdi.snapshot`.

Create Simulation Data Inspector Report

To generate documentation of your results quickly, create a Simulation Data Inspector report. You can create a report of your data in either the **Inspect** or the **Compare** pane. The report is an HTML file that includes information about all the signals and plots in the active pane. The report includes all signal information displayed in the signal table in the navigation pane. For more information about configuring the table, see “Inspect Metadata”.

To generate a Simulation Data Inspector Report:

1

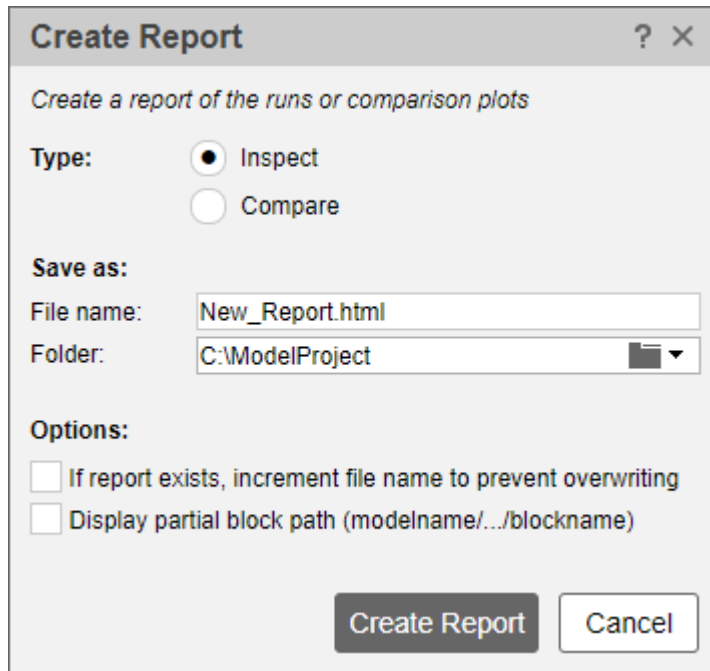


Click the create report icon on the left bar.

2 Specify the type of report you want to create.

- Select **Inspect** to include the plots and signals from the **Inspect** pane.

- Select **Compare** to include the data and plots from the **Compare** pane. When you generate a **Compare Runs** report, you can choose to **Report only mismatched signals** or to **Report all signals**. If you select **Report only mismatched signals**, the report shows only signal comparisons that are not within the specified tolerances.



- 3 Specify a **File name** for the report, and navigate to the **Folder** where you want to save the report.
- 4 Click **Create Report**.

The generated report automatically opens in your default browser.

Export Data to the Workspace or a File

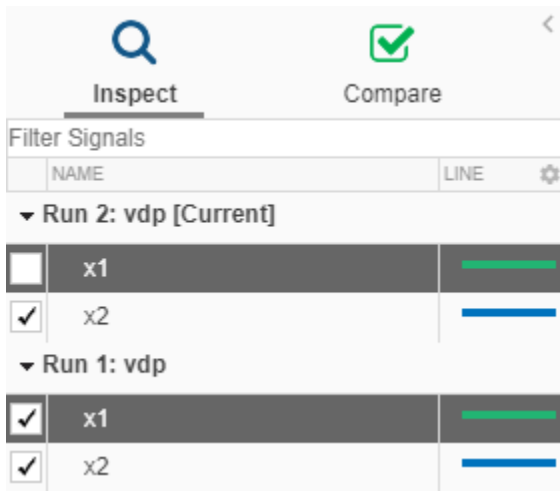
You can use the Simulation Data Inspector to export data to the base workspace, a MAT file, or a Microsoft Excel file. You can export a selection of runs and signals, runs in the work area, or all runs in the **Inspect** pane, including the **Archive**.

When you export a selection of runs and signals, make the selection of data to export before clicking



the export button .

Only the selected runs and signals are exported. In this example, only the x1 signals from Run 1 and Run 2 are exported. The check box selections for plotting data do not affect whether a signal is exported.



When you export a single signal to the workspace or a MAT file, the signal is exported to a `timeseries` object. Data exported to the workspace or a MAT file for a run or multiple signals is stored as a `Simulink.SimulationData.Dataset` object.

To export data to a file, select the **File** option in the **Export** dialog. You can specify a file name and browse to the location where you want to save the exported file. When you export data to a MAT file, a single exported signal is stored as a `timeseries` object, and runs or multiple signals are stored as a `Simulink.SimulationData.Dataset` object. When you export data to a Microsoft Excel file, the data is stored using the format described in “Microsoft Excel Import, Export, and Logging Format”.

To export to a Microsoft Excel file, select the XLSX extension from the drop-down. When you export data to a Microsoft Excel file, you can specify additional options for the format of the data in the exported file. If the file name you provided already exists, you can choose to overwrite the entire file or to only overwrite sheets containing data that corresponds to the exported data. You can also choose which metadata to include and whether signals with identical time data share a time column in the exported file.

Export Video Signal to an MP4 File

You can export a 2D or 3D signal that contains RGB or monochrome video data to an MP4 file using the Simulation Data Inspector. For example, when you log a video signal in a simulation, you can export the data to an MP4 file and view the video using a video player. To export a video signal to an MP4 file:

- 1 Select the signal you want to export.

2



Click **Export** in the toolbar on the left or right-click the signal and select **Export**.

- 3 In the Export dialog box, choose to export **Selected runs and signals** to a file.
- 4 Specify a file name and the path to the location where you want to save the file.
- 5 Select **MP4 video file** from the list and click **Export**.

For the option to export to an MP4 file to be available:

- You must export only one signal at a time.
- The selected signal must be 2D or 3D and contain RGB or monochrome video data.
- The selected signal must be represented in the Simulation Data Inspector as a single signal with multidimensional sample values.

You may need to convert the signal representation before exporting the signal data. For more information, see “Analyze Multidimensional Signal Data”.

- The data type for the signal values must be `double`, `single`, or `uint8`.

Exporting a video signal to an MP4 file is not supported for Linux operating systems.

See Also

Functions

`Simulink.sdi.saveView`

Related Examples

- “View Data in the Simulation Data Inspector”
- “Inspect Simulation Data”
- “Compare Simulation Data”

Inspect and Compare Data Programmatically

You can harness the capabilities of the Simulation Data Inspector from the MATLAB command line using the Simulation Data Inspector API.

The Simulation Data Inspector organizes data in runs and signals, assigning a unique numeric identification to each run and signal. Some Simulation Data Inspector API functions use the run and signal IDs to reference data, rather than accepting the run or signal itself as an input. To access the run IDs in the workspace, you can use `Simulink.sdi.getAllRunIDs` or `Simulink.sdi.getRunIDByIndex`. You can access signal IDs through a `Simulink.sdi.Run` object using the `getSignalIDByIndex` method.

The `Simulink.sdi.Run` and `Simulink.sdi.Signal` classes provide access to your data and allow you to view and modify run and signal metadata. You can modify the Simulation Data Inspector preferences using functions like `Simulink.sdi.setSubPlotLayout`, `Simulink.sdi.setRunNamingRule`, and `Simulink.sdi.setMarkersOn`. To restore the Simulation Data Inspector's default settings, use `Simulink.sdi.clearPreferences`.

Create a Run and View the Data

This example shows how to create a run, add data to it, and then view the data in the Simulation Data Inspector.

Create Data for the Run

Create `timeseries` objects to contain data for a sine signal and a cosine signal. Give each `timeseries` object a descriptive name.

```
time = linspace(0,20,100);

sine_vals = sin(2*pi/5*time);
sine_ts = timeseries(sine_vals,time);
sine_ts.Name = 'Sine, T=5';

cos_vals = cos(2*pi/8*time);
cos_ts = timeseries(cos_vals,time);
cos_ts.Name = 'Cosine, T=8';
```

Create a Run and Add Data

Use the `Simulink.sdi.view` function to open the Simulation Data Inspector.

```
Simulink.sdi.view
```

To import data into the Simulation Data Inspector from the workspace, create a `Simulink.sdi.Run` object using the `Simulink.sdi.Run.create` function. Add information about the run to its metadata using the `Name` and `Description` properties of the `Run` object.

```
sinusoidsRun = Simulink.sdi.Run.create;
sinusoidsRun.Name = 'Sinusoids';
sinusoidsRun.Description = 'Sine and cosine signals with different frequencies';
```

Use the `add` function to add the data you created in the workspace to the empty run.

```
add(sinusoidsRun, 'vars', sine_ts, cos_ts);
```

Plot the Data in the Simulation Data Inspector

Use the `getSignalByIndex` function to access `Simulink.sdi.Signal` objects that contain the signal data. You can use the `Simulink.sdi.Signal` object properties to specify the line style and color for the signal and plot it in the Simulation Data Inspector. Specify the `LineColor` and `LineDashed` properties for each signal.

```
sine_sig = getSignalByIndex(sinusoidsRun,1);
sine_sig.LineColor = [0 0 1];
sine_sig.LineDashed = '-.';
```

```
cos_sig = sinusoidsRun.getSignalByIndex(2);
cos_sig.LineColor = [0 1 0];
cos_sig.LineDashed = '--';
```

Use the `Simulink.sdi.setSubPlotLayout` function to configure a 2-by-1 subplot layout in the Simulation Data Inspector plotting area. Then use the `plotOnSubPlot` function to plot the sine signal on the top subplot and the cosine signal on the lower subplot.

```
Simulink.sdi.setSubPlotLayout(2,1);
```

```
plotOnSubPlot(sine_sig,1,1,true);
plotOnSubPlot(cos_sig,2,1,true);
```

Close the Simulation Data Inspector and Save Your Data

When you have finished inspecting the plotted signal data, you can close the Simulation Data Inspector and save the session to an MLDATX file.

```
Simulink.sdi.close('sinusoids.mldatx')
```

Compare Two Signals in the Same Run

You can use the Simulation Data Inspector programmatic interface to compare signals within a single run. This example compares the input and output signals of an aircraft longitudinal controller.

First, load the session that contains the data.

```
Simulink.sdi.load('AircraftExample.mldatx');
```

Use the `Simulink.sdi.Run.getLatest` function to access the latest run in the data.

```
aircraftRun = Simulink.sdi.Run.getLatest;
```

Then, you can use the `Simulink.sdi.getSignalsByName` function to access the `Stick` signal, which represents the input to the controller, and the `alpha, rad` signal that represents the output.

```
stick = getSignalsByName(aircraftRun, 'Stick');
alpha = getSignalsByName(aircraftRun, 'alpha, rad');
```

Before you compare the signals, you can specify a tolerance value to use for the comparison. Comparisons use tolerance values specified for the baseline signal in the comparison, so set an absolute tolerance value of 0.1 on the `Stick` signal.

```
stick.AbsTol = 0.1;
```

Now, compare the signals using the `Simulink.sdi.compareSignals` function. The `Stick` signal is the baseline, and the `alpha_rad` signal is the signal to compare against the baseline.

```
comparisonResults = Simulink.sdi.compareSignals(stick.ID,alpha.ID);
match = comparisonResults.Status
```

```
match =
    ComparisonSignalStatus enumeration
        OutOfTolerance
```

The comparison result is out of tolerance. You can use the `Simulink.sdi.view` function to open the Simulation Data Inspector to view and analyze the comparison results.

Compare Runs with Global Tolerance

You can specify global tolerance values to use when comparing two simulation runs. Global tolerance values are applied to all signals within the run. This example shows how to specify global tolerance values for a run comparison and how to analyze and save the comparison results.

First, load the session file that contains the data to compare. The session file contains data for four simulations of an aircraft longitudinal controller. This example compares data from two runs that use different input filter time constants.

```
Simulink.sdi.load('AircraftExample.mldatx');
```

To access the run data to compare, use the `Simulink.sdi.getAllRunIDs` function to get the run IDs that correspond to the last two simulation runs.

```
runIDs = Simulink.sdi.getAllRunIDs;
runID1 = runIDs(end - 1);
runID2 = runIDs(end);
```

Use the `Simulink.sdi.compareRuns` function to compare the runs. Specify a global relative tolerance value of `0.2` and a global time tolerance value of `0.5`.

```
runResult = Simulink.sdi.compareRuns(runID1,runID2,'reltol',0.2,'timetol',0.5);
```

Check the `Summary` property of the returned `Simulink.sdi.DiffRunResult` object to see whether signals were within the tolerance values or out of tolerance.

```
runResult.Summary
ans = struct with fields:
    OutOfTolerance: 0
    WithinTolerance: 3
    Unaligned: 0
    UnitsMismatch: 0
    Empty: 0
    Canceled: 0
    EmptySynced: 0
    DataTypeMismatch: 0
```

```
TimeMismatch: 0
StartStopMismatch: 0
Unsupported: 0
```

All three signal comparison results fell within the specified global tolerance.

You can save the comparison results to an MLDATX file using the `saveResult` function.

```
saveResult(runResult, 'InputFilterComparison');
```

Analyze Simulation Data Using Signal Tolerances

You can programmatically specify signal tolerance values to use in comparisons performed using the Simulation Data Inspector. In this example, you compare data collected by simulating a model of an aircraft longitudinal flight control system. Each simulation uses a different value for the input filter time constant and logs the input and output signals. You analyze the effect of the time constant change by comparing results using the Simulation Data Inspector and signal tolerances.

First, load the session file that contains the simulation data.

```
Simulink.sdi.load('AircraftExample.mldatx');
```

The session file contains four runs. In this example, you compare data from the first two runs in the file. Access the `Simulink.sdi.Run` objects for the first two runs loaded from the file.

```
runIDs = Simulink.sdi.getAllRunIDs;
runIDs1 = runIDs(end-3);
runIDs2 = runIDs(end-2);
```

Now, compare the two runs without specifying any tolerances.

```
noTolDiffResult = Simulink.sdi.compareRuns(runIDs1, runIDs2);
```

Use the `getResultByIndex` function to access the comparison results for the `q` and `alpha` signals.

```
qResult = getResultByIndex(noTolDiffResult, 1);
alphaResult = getResultByIndex(noTolDiffResult, 2);
```

Check the `Status` of each signal result to see whether the comparison result fell within our out of tolerance.

```
qResult.Status
```

```
ans =
    ComparisonSignalStatus enumeration
    OutOfTolerance
```

```
alphaResult.Status
```

```
ans =
    ComparisonSignalStatus enumeration
```

OutOfTolerance

The comparison used a value of 0 for all tolerances, so the `OutOfTolerance` result means the signals are not identical.

You can further analyze the effect of the time constant by specifying tolerance values for the signals. Specify the tolerances by setting the properties for the `Simulink.sdi.Signal` objects that correspond to the signals being compared. Comparisons use tolerances specified for the baseline signals. This example specifies a time tolerance and an absolute tolerance.

To specify a tolerance, first access the `Signal` objects from the baseline run.

```
runTs1 = Simulink.sdi.getRun(runIDTs1);
qSig = getSignalsByName(runTs1, 'q, rad/sec');
alphaSig = getSignalsByName(runTs1, 'alpha, rad');
```

Specify an absolute tolerance of 0.1 and a time tolerance of 0.6 for the `q` signal using the `AbsTol` and `TimeTol` properties.

```
qSig.AbsTol = 0.1;
qSig.TimeTol = 0.6;
```

Specify an absolute tolerance of 0.2 and a time tolerance of 0.8 for the `alpha` signal.

```
alphaSig.AbsTol = 0.2;
alphaSig.TimeTol = 0.8;
```

Compare the results again. Access the results from the comparison and check the `Status` property for each signal.

```
tolDiffResult = Simulink.sdi.compareRuns(runIDTs1, runIDTs2);
qResult2 = getResultByIndex(tolDiffResult, 1);
alphaResult2 = getResultByIndex(tolDiffResult, 2);
```

```
qResult2.Status
```

```
ans =
  ComparisonSignalStatus enumeration
    WithinTolerance
```

```
alphaResult2.Status
```

```
ans =
  ComparisonSignalStatus enumeration
    WithinTolerance
```

See Also
Simulation Data Inspector

Related Examples

- “Compare Simulation Data”
- “How the Simulation Data Inspector Compares Data”
- “Create Plots Using the Simulation Data Inspector”

Limit the Size of Logged Data

In this section...

“Limit the Number of Runs Retained in the Simulation Data Inspector Archive” on page 12-49

“Specify a Minimum Disk Space Requirement or Maximum Size for Logged Data” on page 12-49

“View Data Only During Simulation” on page 12-50

“Reduce the Number of Data Points Logged from Simulation” on page 12-50

Logging simulation data can produce large amounts of data that fill up disk space. Such situations include logging many signals, logging data for long simulations, and running many simulations without deleting run data from the Simulation Data Inspector. You can choose among several options to limit the size of logged simulation data. You can:

- Limit the number of runs retained in the Simulation Data Inspector archive.
- Reduce the number of data points logged in each simulation.
- Specify a minimum disk space requirement or maximum size for logged data.
- Configure logging for only viewing data during simulation.

Depending on your requirements, you can use more than one strategy to limit the size of logged data.

Limit the Number of Runs Retained in the Simulation Data Inspector Archive

When you run multiple simulations in a single MATLAB session, logged simulation data accumulates in the Simulation Data Inspector even if you overwrite the logging data in the MATLAB workspace. To reduce the amount of data the Simulation Data Inspector retains, you can configure a limit for the number of runs stored in the archive. When the number of runs in the archive reaches the size limit, the Simulation Data Inspector starts to delete runs from the archive on a first-in, first-out basis.

Configure the archive **Size** setting in the Simulation Data Inspector preferences. The size limit only applies to runs in the archive. For the Simulation Data Inspector to automatically limit data retention, select **Automatically archive** and specify the maximum number of runs to retain in the archive. By default, **Automatically archive** is enabled with an archive size limit of twenty runs. If you experience issues with logged data consuming too much disk space, consider adjusting the size limit for the archive in the Simulation Data Inspector preferences.

Specify a Minimum Disk Space Requirement or Maximum Size for Logged Data

You can use preferences in the Simulation Data Inspector to directly limit the size of logged data by specifying a minimum amount of disk space to leave free or by specifying a maximum size for logged data on disk. Each setting accounts for all kinds of logged data. By default, logged data must leave at least 100 MB of free disk space with no maximum size limit. Specify the required disk space and maximum size in GB, and specify 0 to apply no disk space requirement or no maximum size limit.

When you specify a minimum disk space requirement or a maximum size for logged data, you can also specify whether to prioritize retaining data from the current simulation or data from prior simulations when approaching the limit. By default, the Simulation Data Inspector prioritizes

retaining data for the current run. As the free disk space or logged data size approaches the limit, prior runs are deleted first to free up space for data being logged in the current run. If deleting runs does not free up enough space, recording is disabled. To prioritize retaining prior data, change the **When low on disk space** setting to **Keep prior runs and stop recording**. You see a warning message when prior runs are deleted and when recording is disabled. If recording is disabled due to the size of logged data, you need to change the **Record Mode** back to **View and record data** to continue logging data, after you have freed up disk space.

View Data Only During Simulation

In some situations, you may want to only view the data for logged signals and not save the values. For example, when using the Simulation Data Inspector to visualize data streaming from hardware, you may only want to view the data live and not record it. You can change the **Record mode** in the Simulation Data Inspector preferences to **View during simulation only** so that logged data is not saved and you can still view the data during simulation. The **Record mode** is reset to **View and record data** at the start of each MATLAB session.

When you change the **Record mode** to **View during simulation only**:

- Logged data is not available in the Simulation Data Inspector or workspace after simulation.
- You can view data using dashboard blocks, scopes, and the Simulation Data Inspector, but plots clear when you pan or zoom.
- You cannot access logged data during simulation using the Simulation Data Inspector programmatic interface.

Reduce the Number of Data Points Logged from Simulation

Model configuration parameters and signal properties allow you to limit the number of data points logged in a simulation. Be sure to carefully consider data requirements when limiting logged data points. Limiting data can skip critical time points, and can lead to aliasing, if your effective sample rate is too low.

You can reduce the number of data points using:

- Decimation — Log every n th signal value.
- Limit data points to last — Only log the last n signal values.
- Logging intervals — Specify specific time intervals in which to log data.

For details, see “Specify Signal Values to Log”.

See Also

Tools

Simulation Data Inspector

Related Examples

- “Specify Signal Values to Log”
- “Configure the Simulation Data Inspector”

Execution with MATLAB Scripts

Real-Time Application Objects and Options in the MATLAB Interface

Target and Application Objects

The Simulink Real-Time software uses a `Target` object to represent a target computer and an `Application` object to represent a real-time application. To run and control real-time applications on the target computer, use the object functions.

An understanding of the `Target` and `Application` object properties and functions helps you to control and test your real-time application on the target computer.

A `Target` object on the development computer represents the interface to a real-time application and the RTOS on the target computer. To run and control the real-time application, use `Target` objects.

When you change a `Target` object property on the development computer, information is exchanged between the target computer and the real-time application.

To create a `Target` object for the default target computer, in the MATLAB Command Window, type:

```
tg = slrealtime
```

A `Target` object has properties and functions specific to that object. The real-time application object functions enables you to control a real-time application on the target computer from the development computer. You enter real-time application object functions in the MATLAB Command Window on the development computer or you can use MATLAB code scripts. To access the help for these functions from the command line, use the syntax:

```
doc slrealtime/function_name
```

For example, to get help on the `load` function, type:

```
doc slrealtime/load
```

To get a list of all the functions for the `Target` object, use the `methods` function. For example, to get the functions for `Target` object `tg`, type:

```
methods(tg)
```

If you want to control the real-time application from the target computer, use target computer commands (see “Control Real-Time Application at Target Computer Command Line” on page 9-2).

Control Real-Time Application by Using Objects

You can create a real-time application and control it by using `Target` and `Application` objects

Open a model and build a real-time application. This example uses the `slrt_ex_osc` model.

```
open_system(fullfile(matlabroot, 'toolbox', 'slrealtime', ...  
'examples', 'slrt_ex_osc'));  
slbuild('slrt_ex_osc');
```

Create `Target` and `Application` objects to represent the target computer and the real-time application.

```
tg = slrealtime('TargetPC1');  
app = slrealtime.Application('slrt_ex_osc');
```

Load the real-time application on the target computer by using the `Target` object.

```
load(tg, 'slrt_ex_osc');
```

Set the `Target` object `stoptime` property for the real-time application.

```
setStopTime(tg, inf);
```

Get the `Application` object options property values from the real-time application.

```
app.Options.get("stoptime")
```

```
ans =
```

```
    Inf
```

Start the real-time application by using the `Target` object .

```
start(tg);
```

Stop the real-time application by using the `Target` object .

```
stop(tg);
```

Use Real-Time Application Object Functions

To run `Target` object and `Application` functions, use the `function_name(target_object, argument_list)` syntax.

Unlike properties, for which partial but unambiguous names are permitted, you must enter function names in full, in lowercase. For example, to start a real-time application on target computer `tg`, in the MATLAB Command Window, type:

```
tg = slrealtime;  
start(tg);
```

See Also

Target | Application

More About

- “Control Real-Time Application at Target Computer Command Line” on page 9-2

Simulink Real-Time Instruments and Instrument Panel Apps

- “Add Instruments to Real-Time Application from Simulink Model” on page 14-2
- “Instrumentation Apps for Real-Time Applications” on page 14-5
- “Create App Designer Instrument Panels by Using App Generator” on page 14-6
- “Create App Designer Instrument Panels by Using Simulink Real-Time Components” on page 14-9
- “Create Standalone Instrument Panel App by Using Application Compiler” on page 14-14

Add Instruments to Real-Time Application from Simulink Model

As an alternative to marking signals in your model for logging, connecting a signal to a File Log block in the model, or selecting signals in a real-time application to stream in the Simulink Real-Time explorer, you can stream signal data to the Simulation Data Inspector by using Instrument buttons in the Simulink Editor. You can select a signal for streaming by using the Instrument buttons when the signal is:


- Available in the model and in the real-time application that is built from the model.
- Uses specified (not inherited) sample time.
- Uses globally accessible memory in the real-time application.
- Not connected to a Send or Message Send block.

Unlike marking signals for streaming or connecting signals to File Log blocks, the Instrument buttons use bind mode in workflows that let you add an instrument to the model and make the instrument available in the real-time application without rebuilding the real-time application. You can export the instrument from the model or import the instrument into the model.

To select signals for streaming to the Simulation Data Inspector by using the Instrument buttons in the Simulink Editor:

- 1 Open the model.
- 2 Connect the development computer to the target computer.
- 3 To generate the real-time application, build the model.
- 4 To put the model in bind mode by using the Instrument buttons, in the Simulink Editor, select **Real-Time > Review Results > Add Instrument**.

While in bind mode, a link symbol appears with the cursor, and an exit icon appears in the upper-right corner of the model. When you are ready to exit bind mode, on the model canvas, click the






exit icon. 

- 5 To add a signals to an instrument, select a block and select the check box next to the signal.

When you run the real-time application, the signals that you have added to an instrument are streamed to the Simulation Data Inspector. After you have added an instrument to the model, the label on the **Add Instrument** button changes to **Configure Instrument**.

- 6 To re-enter bind mode by using the Instrument buttons and add or remove signals from the instrument, select **Real-Time > Review Results > Configure Instrument**.
- 7 To remove the instrument added by using **Add Instrument**, select **Real-Time > Review Results > Remove Instrument**.
- 8 To highlight all signals in the model that are in the instrument, select **Real-Time > Review Results > Highlight Instrument**.
- 9 To import or export an instrument in the model, select **Real-Time > Review Results > Import Instrument** or select **Real-Time > Review Results > Export Instrument**.
- 10 After making changes to an instrument, to stream signals from the real-time application to the Simulation Data Inspector, deploy the real-time application to the target computer and start the application. For example, select **Real-Time > Run on Target**.

The Instrument buttons on the **Real-Time** tab of the Simulink Editor provide additional workflow options.

Commands	Instrument Button Operations
	<p>To enter bind mode to add an instrument or enter bind mode to add or remove signals from an instrument, click the Add Instrument button or Configure Instrument button on the Real-Time tab in the Simulink Editor.</p> <p>The Add Instrument button creates an Instrument object, similar to the operation of the <code>slrealtime.Instrument</code> function. The Add Instrument button puts the model in bind mode to create an Instrument object. You can select any number of signals from the model to include in the Instrument object.</p> <p>After creating the Instrument, the Add Instrument button changes to the Configure Instrument button. The Configure Instrument button puts the model in bind mode and lets you add or remove signals from an Instrument, similar to the operation of the <code>addSignal</code> function and <code>removeSignal</code> function.</p>
	<p>The Remove Instrument button removes the instrument created by Add Instrument or Configure Instrument, similar to the way that the <code>removeInstrument</code> function removes an instrument from the selected target object.</p>
	<p>Use the Highlight Instrument button to indicate signals that are included in an instrument in the model.</p>
	<p>Use the Import Instrument button to import an instrument (previously saved to a MAT file) into the model.</p>
	<p>Use the Export Instrument button to export an instrument (as a MAT file) from the model.</p>

An instrument that you add to a model is retained in the model, unless you remove the instrument with the **Remove Instrument** button. To remove an instrument that was added in a previous editing session, use the `removeAllInstruments` function.

To save and restore an instrument in a model, use the **Export Instrument** button and **Import Instrument** button. A suggested workflow for saving and restoring an instrument in a model is:

- 1 Add an instrument to the model. Use the instrument to stream signals from the real-time application.
- 2 Export an instrument from the model for streaming in future real-time application runs.
- 3 Remove an instrument from the model before exiting the Simulink Editor.
- 4 Import an instrument to the model when needed to stream signals from the real-time application.

See Also

[Simulink Real-Time Explorer | Instrument](#)

Related Examples

- “Add App Designer App to Inverted Pendulum Model” on page 16-18
- “Control Color of Lamp on Instrument Panel” on page 16-112

More About

- “Display and Filter Hierarchical Signals and Parameters” on page 7-66

Instrumentation Apps for Real-Time Applications

To visualize the behavior of a real-time application running on a target computer, you can create instrument panel apps. An instrument panel app is a user-interface application into which you can insert one or more instruments. To create an instrument panel app, use App Designer or an m-script.

- When you create an instrument panel app in the App Designer **Design View**, you add instrument components from the App Designer **Component Library** to the app. You configure each instrument by using fields in the **Component Browser**. In the App Designer **Code View**, you add callback code to handle component events, such as new streaming data or interaction with the app. For more information, see “App Building Components” and “Manage Code in App Designer Code View”.
- When you create an instrument panel app by using an m-script, you use a programmatic approach to add each instrument to the panel as UI component. For more information, see “Create Callbacks for Apps Created Programmatically”.

To stream signal and parameter data to the instrument panel app from the real-time application, you use the `Instrument` object. After you create an instrument object for a real-time application, you can use instrument object functions to connect signals and parameters from the real-time application to instrument panel app callbacks.

When identifying parameters and output signals to stream signal to the instrument panel app from the real-time application, it can be helpful to use the hierarchical display of signals and parameters. See **Simulink Real-Time Explorer**. For more information, see “Display and Filter Hierarchical Signals and Parameters” on page 7-66.

See Also

Simulink Real-Time Explorer | Instrument

Related Examples

- “Add App Designer App to Inverted Pendulum Model” on page 16-18
- “Create App Designer Instrument Panels by Using Simulink Real-Time Components” on page 14-9
- “Create Standalone Instrument Panel App by Using Application Compiler” on page 14-14

More About

- “App Building Components”
- “Manage Code in App Designer Code View”
- “Display and Filter Hierarchical Signals and Parameters” on page 7-66

Create App Designer Instrument Panels by Using App Generator

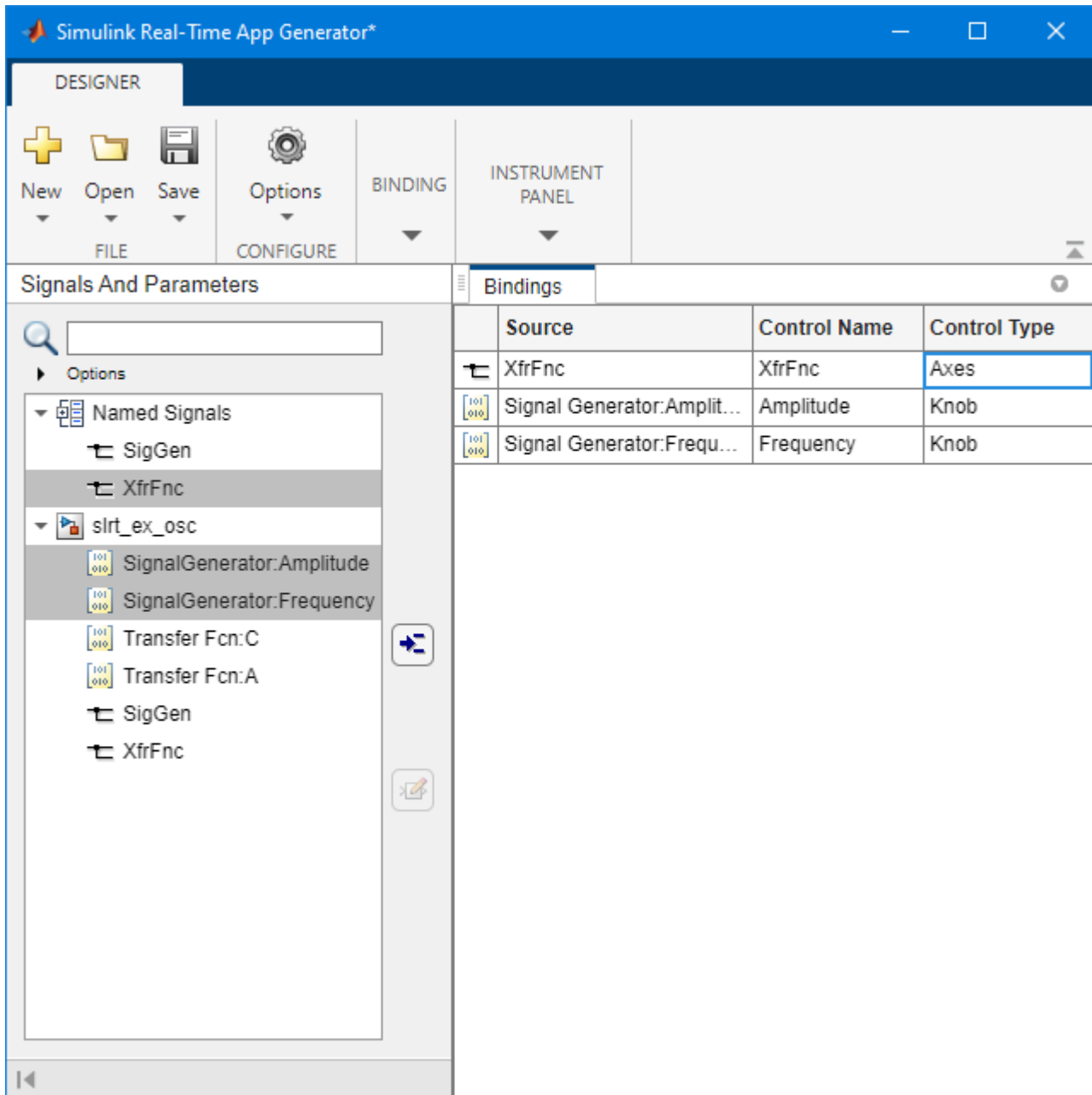
You can create App Designer instrument panels to interface with real-time applications by using the **App Generator** button on the **Real-Time** tab in the Simulink Editor. By using the **App Generator**, you can generate an instrument panel for selected signals and parameters in your model. You can open the generated app in App Designer to customize the instrument panel.

To create an instrument panel for your model by using the **App Generator** button:

- 1 Open the Simulink Real-Time model.
- 2 In the Simulink Editor, on the **Real-Time** tab, click **Review Results > App Generator**.
- 3 To create an instrument panel from the real-time application file MLDATX, select **New > New**, click **No** to remove the current session, and select the MLDATX file. For information about the difference between developing an instrument panel from a model SLX file or a real-time application MLDATX file, see the “Tip About MLDATX and SLX Files” on page 14-8.
- 4 In the App Generator, select Signals and Parameters in the model to add as components on the instrument panel. Click the **Add to panel** button.



- 5 After you add each signal or parameter, configure the **Control Name** and **Control Type**. The figure shows some App Generator selections for the `slrt_ex_osc` model.



- 6 After configuring the **Control Name** and **Control Type** for the signals and parameters, click the **Generate App** button.
- 7 To customize the generated application, click the **Open in App Designer** button.

The App Generator adds controls to your instrument panel that enable the panel to interface with the real-time application. These controls include the target computer selector, connect button, load application button, start/stop button, stop time field, and system log. Any instrumented signals from the model are added in an axis component. For more information, see “Create App Designer Instrument Panels by Using Simulink Real-Time Components” on page 14-9.

Tip About MLDATX and SLX Files

You can develop an instrument panel app in the App Generator from a model SLX file (if you start the App Generator from the Real-Time tab in the Simulink Editor) or from a real-time application MLDATX file. It is recommended that you develop the instrument panel based on the MLDATX file, because—when developing from the MLDATX file—the App Generator only lists the signals and parameters that are present in the generated code. If you develop the instrument panel based on the SLX file, the App Generator can list more signals than are present in the generated code. These signals include virtual signals and signals to Scope blocks.

See Also

Simulink Real-Time App Generator | Instrument

Related Examples

- “Add App Designer App to Inverted Pendulum Model” on page 16-18

More About

- “Display and Filter Hierarchical Signals and Parameters” on page 7-66

Create App Designer Instrument Panels by Using Simulink Real-Time Components

The Simulink® Real-Time™ components in App Designer ease creation of App Designer instrument panels for real-time applications. By using the Simulink Real-Time components, you can add frequently used operations, such as select target and load real-time application, as controls on your instrument panel with minimal programming of callback functions.

This example shows how the Simulink Real-Time components in App Designer help you develop instrument panels that are reusable. When you examine the callback code in the example, see that only code that connects instruments from the real-time application to controls in the instrument panel uses block path information that is specific to the real-time application. This approach makes it easier to reuse instrument panels as the interface for other real-time applications.

For more information, see *Develop Apps Using App Designer*. You also can create instrument panel applications without using App Designer. For more information, see “Add UI Components to App Designer Programmatically”.

Create Blank App

To create an App Designer instrument panel by using Simulink Real-Time components:

Open the App Designer. In MATLAB®, select **Home > New > App**.

Or, in the MATLAB Command Window, type `appdesigner`. Then, select **New > Blank App**

Add Components to App

From the **Simulink Real-Time** group in the **Component Library**, add real-time application components to the instrument panel. For this instrument panel, add:

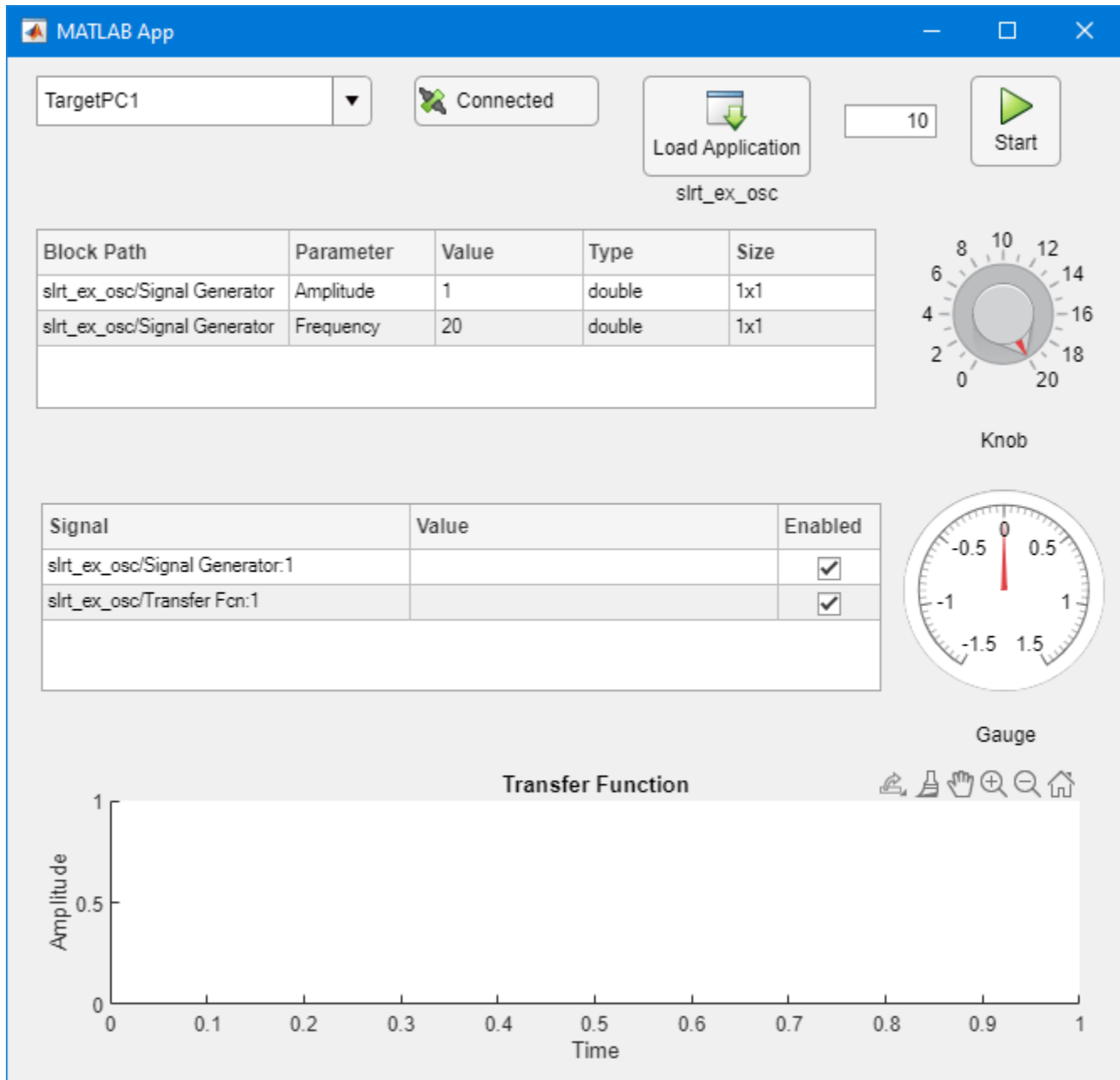
- **Target Selector**
- **Connect**
- **Load**
- **Start/Stop**
- **Stop Time**
- **Parameter Table**
- **Signal Table**

From the **Instrumentation** group in the **Component Library**, add Instrumentation components to the instrument panel. For this instrument panel, add **Knob** and **Gauge**.

From the **Common** group in the **Component Library**, add common components to the instrument panel. For this instrument panel, add **Axes**.

Arrange Components in App

Arrange the instrument panel to appear as shown in this instrument panel.



Configure Options for Components

Select the **Knob** instrument and change the Limits to 0, 20 in the **Inspector** tab of the **Component Browser**.

Select the **Gauge** instrument and change the Limits to -1.5, 1.5 in the **Inspector** tab of the **Component Browser**.

Save the instrument panel as `myInstPanel_slrt_ex_osc.mlapp`.

Add Callback Code

To add callback code to your App Designer instrument panel and test the instrument panel as a UI to a real-time application:

Change to the **Code View** tab.

In the **Component Browser**, select node **myInstPanel_slrt_ex_osc**. Select **Callbacks**. From the **StartupFcn** drop-down list, select <add StartupFcn callback>.

Add callback code to connect the instruments to the real-time application. Paste this callback code into the startup function callback of the instrument panel application. This code connects the instruments to the real-time application `slrt_ex_osc`.

```
% Define the real-time application file to load.
app.LoadButton.Application = 'slrt_ex_osc';

% Define parameters parameters to display in the
% Parameter Table component. The parameters are
% defined in a structure. The block path is the
% first element the parameter name.
app.ParameterTable.Parameters = struct( ...
    'BlockPath', {'slrt_ex_osc/Signal Generator', ...
                  'slrt_ex_osc/Signal Generator'}, ...
    'ParameterName', {'Amplitude', ...
                      'Frequency'});

% Create a ParameterTuner object and bind
% to the knob component.
myParamFreq = slrealtime.ui.tool.ParameterTuner(app.UIFigure);
myParamFreq.Component = app.Knob;
myParamFreq.BlockPath = 'slrt_ex_osc/Signal Generator';
myParamFreq.ParameterName = 'Frequency';

% Define the signals to display in the Signal Table.
% The structure requires the block path of all parameters,
% and the port index of the port connected to the signal.
app.SignalTable.Signals = struct( ...
    'BlockPath', {'slrt_ex_osc/Signal Generator', ...
                  'slrt_ex_osc/Transfer Fcn'}, ...
    'PortIndex', {1,1});

% Create an Instrument object and connect the gauge
% component.
instMyGauge = slrealtime.Instrument();
instMyGauge.connectScalar(app.Gauge, 'slrt_ex_osc/Transfer Fcn', 1);

% Create another Instrument object and connect to the
% axes component.
%
% An Instrument object is needed for each component, but
% you can add more signals to the same axes by using connectLine.
instMyAxes = slrealtime.Instrument();
instMyAxes.connectLine(app.UIAxes, 'XfrFnc');
instMyAxes.AxesTimeSpan = 10;
instMyAxes.AxesTimeSpanOverrun = 'scroll';
```

```
% Create an InstrumentManager and connect the previously created
% Instrument object.
instMgr = slrealtime.ui.tool.InstrumentManager(app.UIFigure);
instMgr.Instruments = [instMyGauge, instMyAxes];
```

Save the instrument panel.

Tuning a Workspace Variable

The code shows how to connect the `ParameterTuner` component to a value on a block. Instead, you can use slightly different code to connect the `ParameterTuner` component to a variable or parameter in the workspace. In the code for the component, the `BlockPath` is empty, and the `ParameterName` is the parameter name instead of the property of the block to tune. The syntax for the callback could be:

```
% Create Parameter Tuning object
hParamTuner = slrealtime.ui.tool.ParameterTuner(app.UIFigure);
hParamTuner.Component = app.Knob;
hParamTuner.BlockPath = '';
hParamTuner.ParameterName = 'myParameter';
```

Recommendations for Callback Code

It is recommended that you make the bindings between instrument panel controls and their related signals or parameters robust. Robust bindings do not break due to minor changes in a model and are more easily re-used with a new model.

For signals, a technique that helps produce robust bindings is to specify signals by using signal names instead of by using a full block path and port index. This technique applies to the `SignalTable` component and functions such as `connectLine` or `connectScalar`.

For parameters, a technique that helps produce robust bindings is to use workspace variables instead of block parameters. This technique applies to the `ParameterTable` component.

For more information, see the examples for `SignalTable`, `ParameterTable`, `connectLine`, and `connectScalar`.

Open Model, Build Real-Time Application, Run Instrument Panel

In MATLAB, open the `slrt_ex_osc` model. In the Command Window, type:

```
open_system(fullfile(matlabroot, 'toolbox', ...
    'slrealtime', 'examples', 'slrt_ex_osc'))
```

Build the real-time application. In the Simulink Editor, select **Real-Time > Run on Target**.

After the model builds and the real-time application runs, in the App Designer, run the instrument panel application.

From the instrument panel application, connect to the target computer, load the real-time application `slrt_ex_osc`, set the stop time at 10 seconds, and start the real-time application.

The instrument panel controls indicate signal and parameter values for the real-time application. Start the real-time application, use the knob to change the parameter value, and see the affect on the output.

When done observing the operation of the instrument panel, close the app and close the App Designer.

See Also

Simulink Real-Time Explorer | Instrument | ConnectButton | InstrumentManager | LoadButton | Menu | ParameterTable | ParameterTuner | SignalTable | SimulationTimeEditField | StartStopButton | StatusBar | StopTimeEditField | SystemLog | TETMonitor | TargetSelector | UpdateButton | slrealtime.ui.control Properties

Related Examples

- “Add App Designer App to Inverted Pendulum Model” on page 16-18

More About

- “App Building Components”
- “Manage Code in App Designer Code View”
- “Display and Filter Hierarchical Signals and Parameters” on page 7-66

Create Standalone Instrument Panel App by Using Application Compiler

After creating an instrument panel app to serve as an interface to your real-time application, you can share a standalone executable instrument panel and installer with others, such as test engineers. For more information about developing an App Designer instrument panel for your real-time application, see “Create App Designer Instrument Panels by Using Simulink Real-Time Components” on page 14-9.

When you share a standalone executable instrument panel and real-time application, the people with whom you share it with must be using a target computer with the same version of RTOS software and configuration as you used to compile the instrument panel. This workflow uses the Application Compiler tool to package the instrument panel app.

The standalone executable application is not cross platform, and the executable type depends on the platform (for example, Windows®) on which it was generated.

Package App by Using Application Compiler

After developing a real-time application and an App Designer instrument panel app that provides and interface to the real-time application, you can use the Application Compiler to package the app.

Open MATLAB and set the current folder to the folder in which you are creating the standalone executable instrument panel.

Select **Apps > Application Compiler**. For more information, see Application Compiler (MATLAB Compiler).

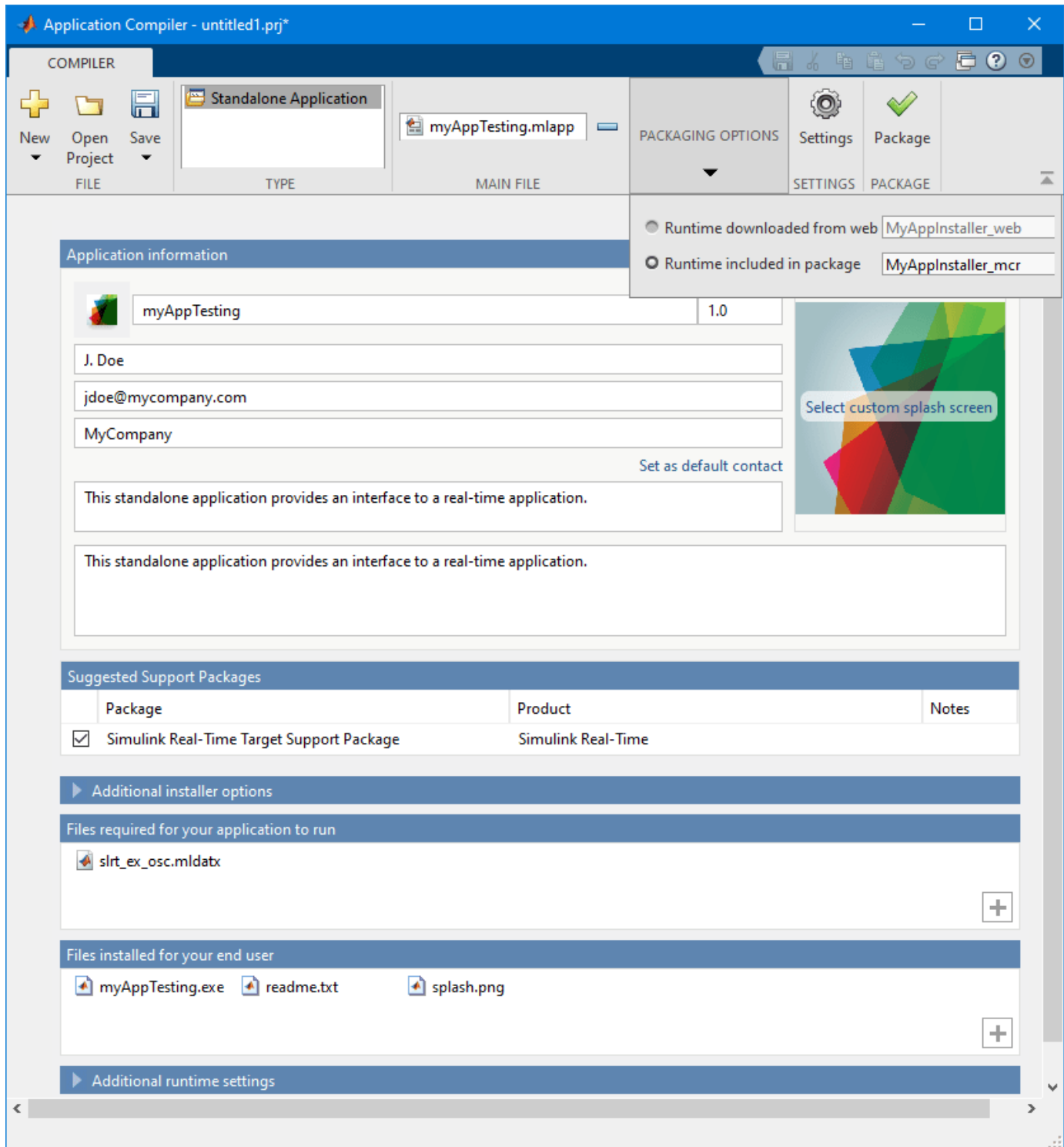
Populate the Application Compiler dialog box with information for the application compiler project. The image shows the example selections for project MyInstPanel_slrt_ex_osc.prj. The compiler settings are:

- **Main file:** myInstPanel_slrt_ex_osc.mlapp
- **Application Name:** myInstPanel
- **Author Name:** J. Doe
- **Email:** jdoe@mycompany.com
- **Company:** MyCompany
- **Summary:** This standalone application provides an interface to a real-time application.
- **Description:** This standalone application provides an interface to a real-time application.
- **Default installation folder:** %ProgramFiles%\MyCompany\myAppTesting\
- **Do not display the Windows Command Shell (console) for execution:** Yes

Your choice of whether to select **Runtime downloaded** or **Runtime included** packaging options for the project influences the length of time for packaging and for installing the application. If the development computer that will be running the standalone executable instrument panel has internet access, select **downloaded**. If not, select **included**.

For **Suggested Support Packages**, select the Simulink Real-Time Target Support Package.

For **Files required for your application to run**, select the real-time application MLDATX file.



Save the Application Compiler project as `myInstPanel.prj`.

In the Application Compiler dialog box, select **Package**. The Package status indicates completion of the packaging stages. When completed, click **Close**.

The packaging process outputs folders `for_redistribution`, `for_redistribution_files_only`, and `for_testing`.

Install Instrument Panel Application

To install the instrument panel application, run the executable file from the `for_redistribution` folder.

On Windows

- If you selected Runtime downloaded from web for the project, run installer `MyAppInstaller_web.exe`.
- If you selected Runtime included in package for the project, run installer `MyAppInstaller_mcr.exe`.

On Linux

- If you selected Runtime downloaded from web for the project, run installer `MyAppInstaller_web.install`.
- If you selected Runtime included in package for the project, run installer `MyAppInstaller_mcr.install`.

For this example, installer executable file is `MyAppInstaller_mcr.exe`. When run, this file installs the MATLAB runtime and installs the instrument panel executable file:

```
C:\Program Files\MyCompany\MyInstPanel_slrt_ex_osc\application\MyInstPanel_slrt_ex_osc.exe
```

Tip: Make a note of the MATLAB run time path in this step. The path can be used to run the standalone application on Linux system.

To test the standalone executable instrument panel, close MATLAB and run the `MyInstPanel_slrt_ex_osc EXE` file.

On Windows

- run the `MyInstPanel_slrt_ex_osc EXE` file

On Linux

- run the command

```
./run_MyInstPanel_slrt_ex_osc.sh /usr/local/MATLAB/MATLAB_Runtime/v911
```

Use the instrument panel to connect to the target computer by inserting the target computer IP address (for example, 192.168.7.5) in place of the target computer name (for example, TargetPC1). Load the real-time application, and start the application. Observe that the instrument panel provides an interface to control the real-time application.

If you modify your real-time application or instrument panel app and repackage these, you do not need to send the installer to your end-users. Instead, you can send them the updated EXE file from the `for_redistribution_files_only` folder to replace the EXE file in their application folder.

See Also

Simulink Real-Time Explorer | Instrument

Related Examples

- “Add App Designer App to Inverted Pendulum Model” on page 16-18

More About

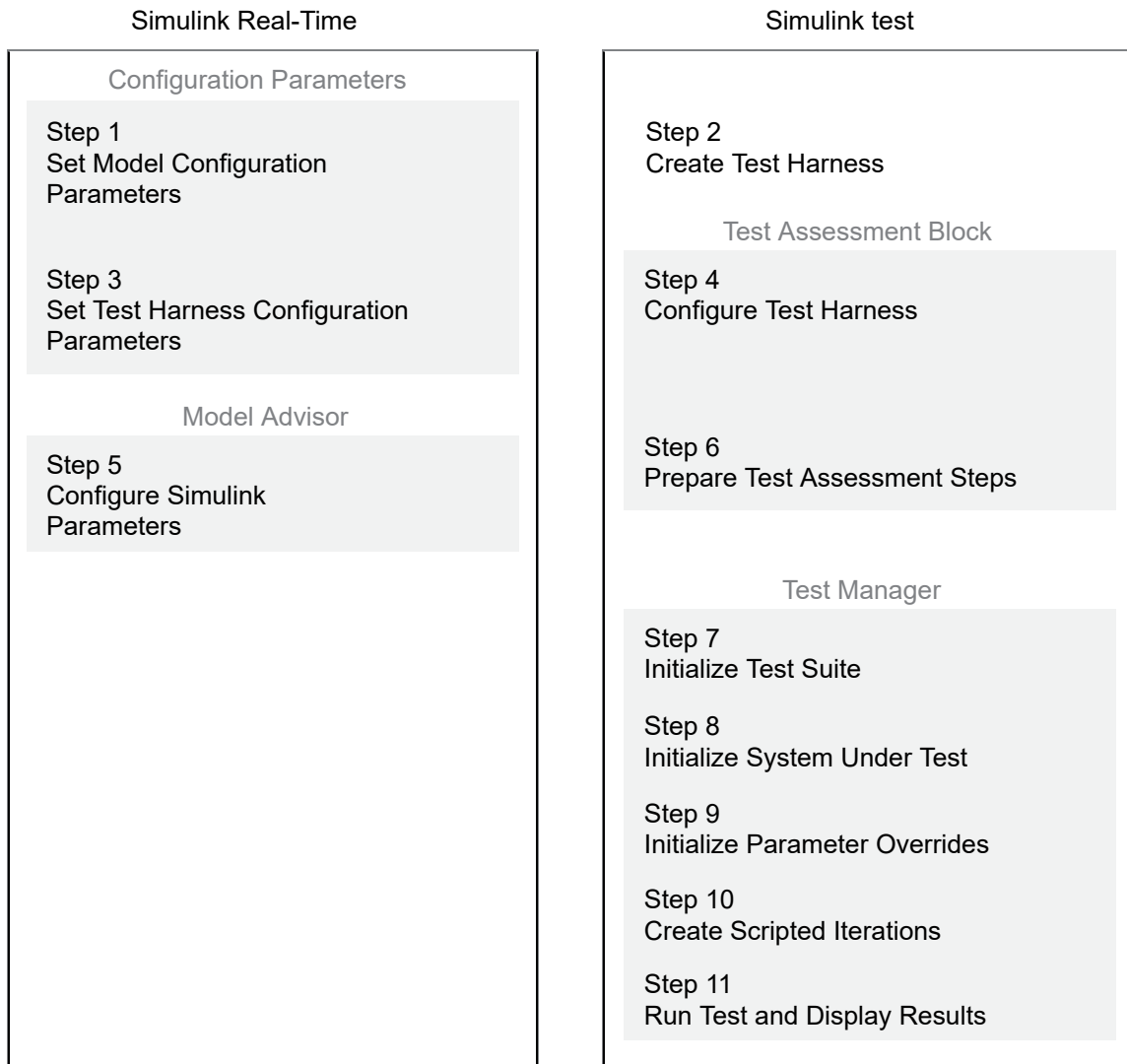
- “App Building Components”
- “Manage Code in App Designer Code View”
- “Display and Filter Hierarchical Signals and Parameters” on page 7-66
- Application Compiler (MATLAB Compiler)

Automated Test with Simulink Test

Test Real-Time Application in Simulink Test

This example shows how to perform a frequency-response test of the model `slrt_ex_osc_sltest`.

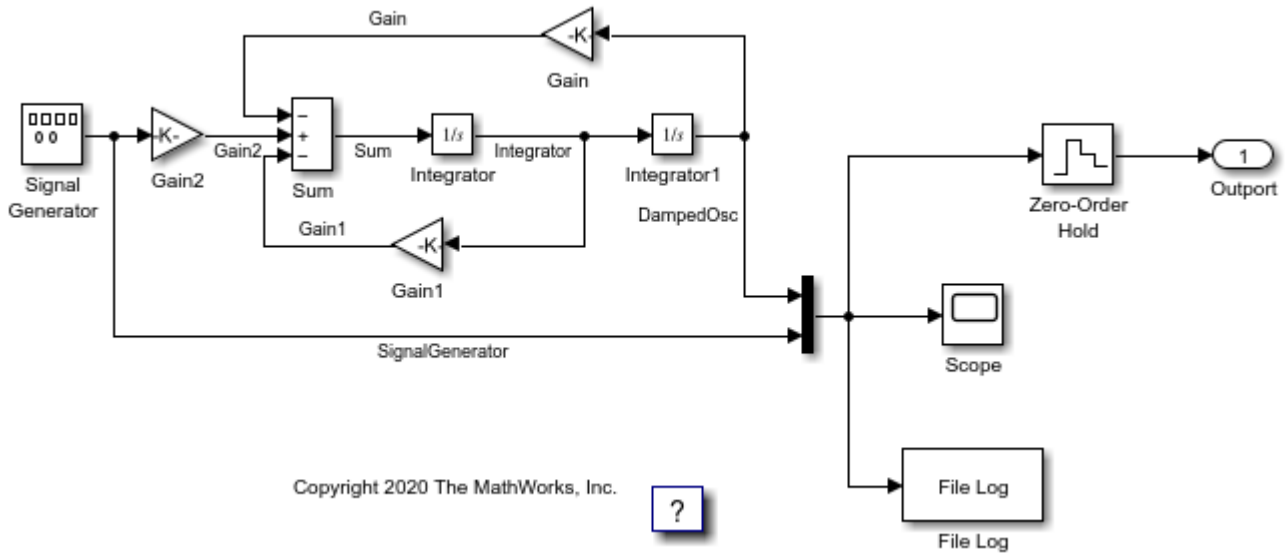
Using this information, in the design phase, you can modify the internal parameters of the model to meet your frequency requirements. In the production phase, you can bin manufactured parts based on frequency response.



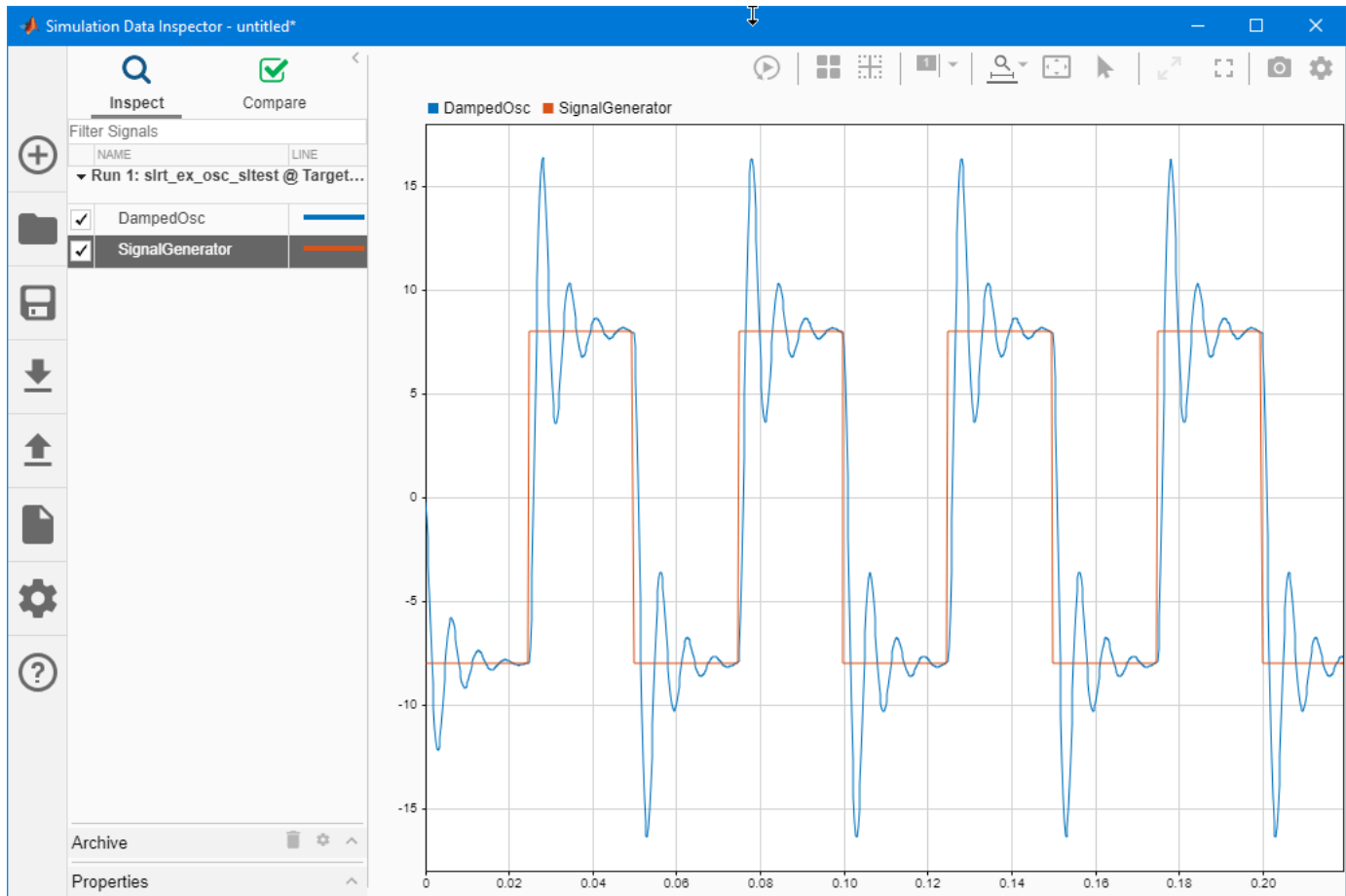
Open the Model

To open the model, in the MATLAB® Command Window, type:

```
open_system(fullfile(matlabroot, 'toolbox', 'srealtime', 'examples', 'slrt_ex_osc_sltest'));
```



The figure shows representative output from a real-time application running on a target computer. At low frequencies, the output of the Integrator1 block settles to the same value as the output of the Signal Generator block. At high frequencies, the output of the Integrator1 block is still ringing at the end of each pulse.



The test determines the highest frequency at which the output values of the Integrator and Signal Generator blocks are within a specified criterion of each other. The test uses the model itself as a signal source and uses a test harness to compare the outputs of the Integrator and Signal Generator blocks.

Step 1. Set Model Configuration Parameters

- 1 Open model `slrt_ex_osc_sltest` in a writable folder.
- 2 Open the Configuration Parameters. On the **Real-Time** tab, click **Hardware Settings**.
- 3 Select **Model Referencing** > **Total number of instances allowed per top model** > **One**.
- 4 Select **Data Import/Export** > **Format** > **Structure with time**.
- 5 Select **Data Import/Export** > **Time**.
- 6 Select **Data Import/Export** > **Output**.
- 7 De-select **Data Import/Export** > **States**.
- 8 De-select **Data Import/Export** > **Final states**.
- 9 De-select **Data Import/Export** > **Signal logging**.
- 10 De-select **Data Import/Export** > **Data stores**.
- 11 De-select **Data Import/Export** > **Log Dataset data to file**.

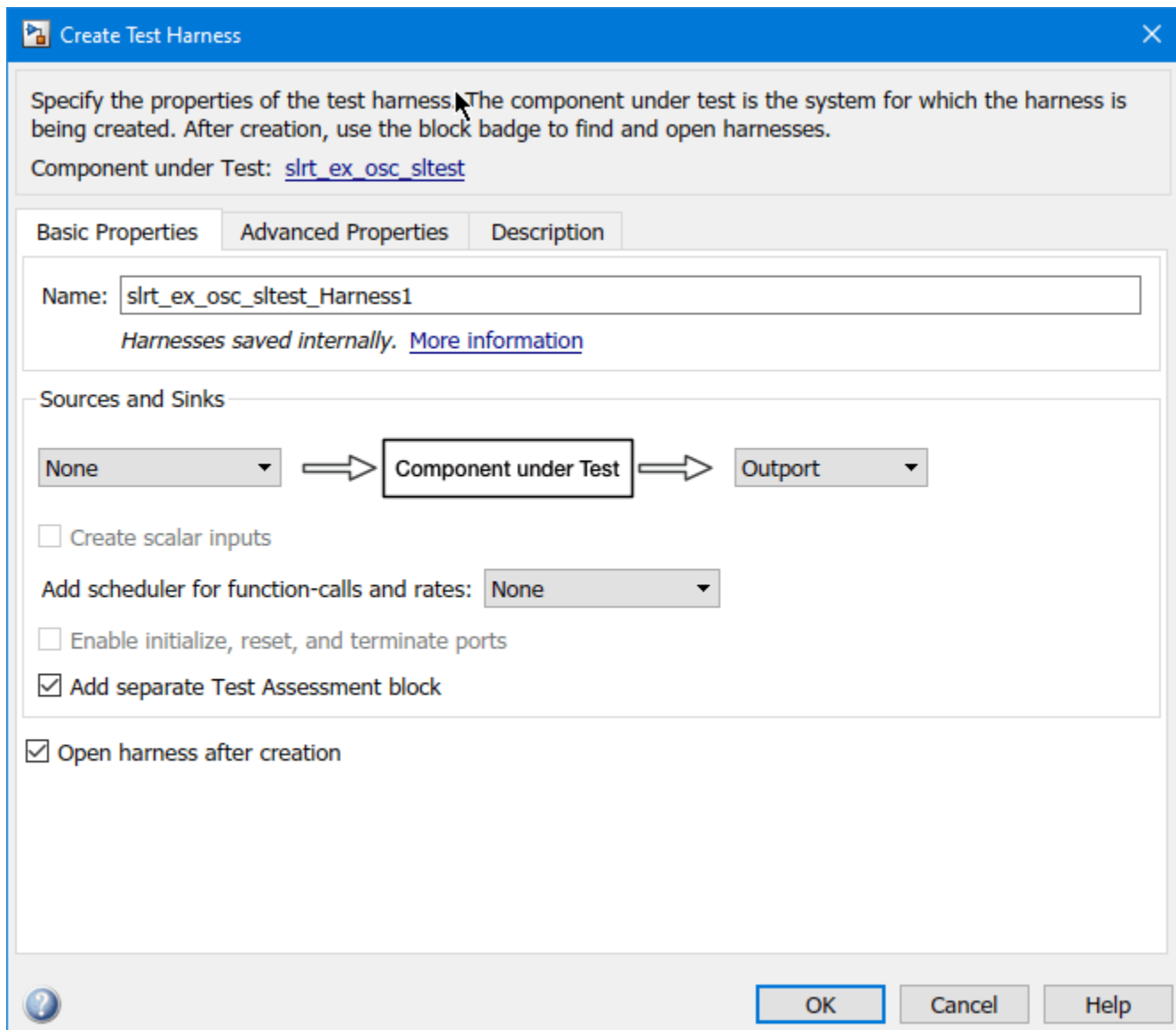
Step 2. Create Frequency Parameter

Create the parameter Frequency that is tuned at the end of this example.

- 1 Open model explorer.
- 2 Create a Simulink® parameter Frequency in model workspace for model `slrt_ex_osc_sltest`.
- 3 Mark the parameter as a model argument.

Step 3. Create Test Harness

- 1 On the Simulink **Apps** tab, click **Simulink Test**.
- 2 On the **Test** tab, click **Add Test Harness**. The software creates a test harness with the default name `slrt_ex_osc_sltest_Harness1`.
- 3 In the **Basic Properties** tab, for the Input to **Component under Test**, select **None**.
- 4 For the Output from **Component under Test**, select **Outport**.
- 5 Select the **Add separate assessment block** check box.
- 6 Select the **Open harness after creation** check box.
- 7 Take the defaults in the remaining tabs.



8. Click **OK**.

The example model `slrt_ex_osc_sltest` stores the test harness within the model. To access the test harness from the example model:

- 1 In Simulink Editor, on the **Test** tab, click **Manage Test Harnesses**.
- 2 Click `slrt_ex_osc_sltest_Harness1`.
- 3 To return to the example model, select it in the perspectives view in the lower right corner of the test harness.

Step 4. Set Test Harness Configuration Parameters

- 1 Open test harness `slrt_ex_osc_sltest_Harness1`.
- 2 Open the Configuration Parameters. On the **Real-Time** tab, click **Hardware Settings**.

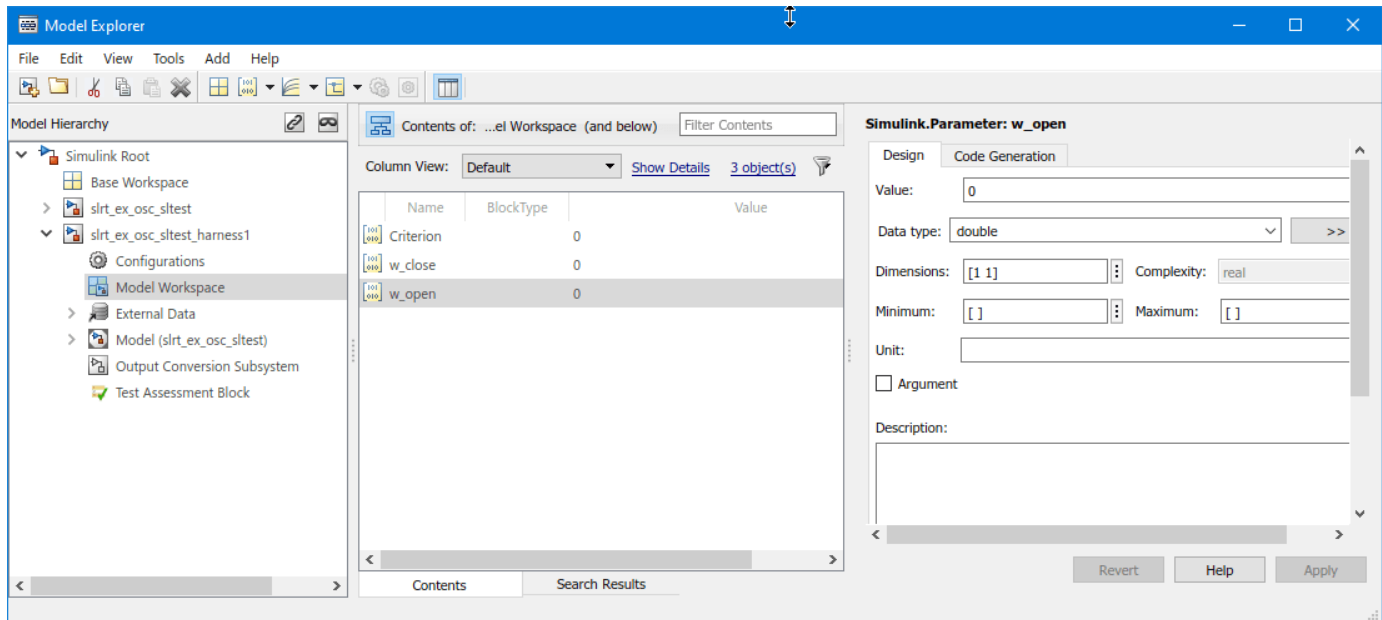
- 3 Select **Model Referencing > Total number of instances allowed per top model > One.**
- 4 Select **Data Import/Export > Format > Structure with time.**
- 5 Select **Data Import/Export > Time.**
- 6 Select **Data Import/Export > Output.**
- 7 De-select **Data Import/Export > States.**
- 8 De-select **Data Import/Export > Final states.**
- 9 De-select **Data Import/Export > Signal logging.**
- 10 De-select **Data Import/Export > Data stores.**
- 11 De-select **Data Import/Export > Log Dataset data to file.**

Step 5. Configure Test Harness

- 1 Open the Test Assessment block.
- 2 To simplify the test assessment configuration, in the **Input** symbol list, replace input Outputport with inputs Int1 and SigGen.
- 3 In slrt_ex_osc_sltest_Harness1, connect a Demux block to slrt_ex_osc_sltest/Outputport.
- 4 In the Demux block dialog box, set **Number of outputs** to 2.
- 5 To make the Demux outputs visible to the Test Assessment block, connect unitary Gain blocks to each of the Demux block outputs.
- 6 Connect the top Demux block output to Test Assessment/Int1 and the bottom output to Test Assessment/SigGen.

Step 6. Configure Simulink Parameters

- 1 Open the Model Explorer. On the **Modeling** tab, pull down the **Design** section and click **Model Explorer**.
- 2 Click node **slrt_ex_osc_sltest_Harness1 > Model Workspace**.
- 3 In the toolbar, click the **Add Simulink Parameter** button.
- 4 Add the following data object:
 - Name — Criterion
 - Value — 0
 - DataType — double
 - Storage Class — ExportedGlobal
5. In a similar manner, add Simulink parameters w_open and w_close. Because these parameters are in the slrt_ex_osc_sltest_Harness1 model workspace as model parameters, you access them by name directly, without model hierarchy.



6. Save the model.

Step 7. Setup Frequency Argument

- 1 Right click on slrt_ex_osc_sltest_Harness1/slrt_ex_osc_sltest.
- 2 Select Block Parameter(Model reference).
- 3 Select Instance parameters on pop out window.
- 4 Mark Frequency as an argument.

Step 8. Prepare Test Assessment Steps

1. Open the Test Assessment block
2. Add these parameters to the Parameter symbol list:
 - Criterion
 - w_open
 - w_close
3. To add a step, in the **Step** column, move the cursor to the top row, click **Add step after**, and type:

CheckSetting
4. Right-click step CheckSetting and set the **When decomposition** check box.
5. To add a substep to CheckSetting, click **Add sub-step**, and type:

Hi when (SigGen > 0)

The when expression selects one half of the waveform.
6. Right-click substep Hi when and set the **When decomposition** check box.

7. To substep Hi when, add substep:

```
HiCheck when ((et >= w_open) && (et <= w_close))
verify((abs(Int1) >= abs(SigGen) * (1.0 - Criterion)) && ...
(abs(Int1) <= abs(SigGen) * (1.0 + Criterion)));
```

The when expression selects the time window for testing the acceptance criterion. The verify command tests the acceptance criterion.

8. In a similar manner, to step CheckSetting, add substep:

```
Lo when (SigGen < 0)
```

9. To substep Lo when, add substep:

```
LoCheck when ((et >= w_open) && (et <= w_close))
verify((abs(Int1) >= abs(SigGen) * (1.0 - Criterion)) && ...
(abs(Int1) <= abs(SigGen) * (1.0 + Criterion)));
```

10. Right-click substep Lo when and set the **When decomposition** check box.

11. To satisfy the requirements of **When decomposition**, remove the default Run step and insert DefaultStep substeps after steps CheckSetting, Hi when, and Lo when. **When decomposition** requires at least two steps at each level of nesting, and one nondecomposed step at the end of each list of steps.

Step	Transition	Next Step	Description
CheckSetting			
Hi when (SigGen > 0)			Selects half of waveform
HiCheck when ((et >= w_open) && (et <= w_close)) verify((abs(Int1) >= abs(SigGen) * (1.0 - Criterion)) && .. (abs(Int1) <= abs(SigGen) * (1.0 + Criterion)));			Selects time window Tests acceptance criterion
DefaultStep_1			Required for 'when decomposition'
Lo when (SigGen < 0)			
LoCheck when ((et >= w_open) && (et <= w_close)) verify((abs(Int1) >= abs(SigGen) * (1.0 - Criterion)) && .. (abs(Int1) <= abs(SigGen) * (1.0 + Criterion)));			
DefaultStep_2			
DefaultStep			

Step 9. Initialize Test Suite

- 1 Click on the slrt_ex_osc_sltest subsystem.

- 2 On the **Apps** tab, click **Simulink Test**.
- 3 On the **Test** tab, click **Test Manager**.
- 4 Select **New > Test File**.
- 5 Name the test file `realtimetest`.
- 6 Right-click the test file and select **New > Real-Time Test**.
- 7 In the new real-time test dialog box, enter `Simulation` in the **Test Type** field.
- 8 Click **Create**.
- 9 Rename the new test suite to `realtimesuite`.
- 10 Rename the new test case to `frequencysweep`.

Step 10. Initialize System Under Test

- 1 In Test Manager, select node `frequencysweep`.
- 2 Select tab **System Under Test**.
- 3 Set **Model** to `slrt_ex_osc_sltest`.
- 4 In tab **Test Harness**, set **Harness** to `slrt_ex_osc_sltest_Harness1`.
- 5 In tab **Simulation Settings and Release Overrides**, select the **Stop Time** check box.
- 6 Take the defaults for the other fields.

Step 11. Initialize Parameter Overrides

1. In Test Manager, select tab **Parameter Overrides**.
2. Click the **Add** button. A dialog box opens containing a list of parameters. If parameters are not visible, click the **Refresh** line at the top of the dialog box.



The refresh builds the model and uploads the model and block parameters from `slrt_ex_osc_sltest_Harness1` and `slrt_ex_osc_sltest`.

3. Open **Parameter Set 1** and select the **Criterion**, **Frequency**, **w_close**, and **w_open** check boxes. Leave the other check boxes cleared.

Step 12. Create Scripted Iterations

To configure and control iterated runs of the test harness, a number of constants and variables provide input.

Test harness constants include:

- `cStartFreq = 15.0` Start frequency of parameter sweep.
- `cStopFreq = 25.0` End frequency of parameter sweep.
- `cFreqIncr = 1.0` Frequency increment.
- `cWOpen = 0.90` Start of time window for evaluating criterion.
- `cWClose = 0.99` End of time window for evaluating criterion.
- `cCriterion = 0.025` Maximum normalized amplitude difference between Signal Generator and Integrator1 within the time window.

Test harness variables include:

- `vfreq` Frequency at each iteration.
- `vw_open` Window opens once in each half-period.
- `vw_close` Window closes once in each half-period.

- 1 In Test Manager, select tab **Iterations > Scripted Iterations**.
- 2 In the text box, enter the following code. To resize the **Scripted Iterations** text box, click and drag the lower-right corner of the box.

```
% Initialize constants
cStartFreq = 15.0;
cStopFreq = 25.0;
cFreqIncr = 5.0;
cWOpen = 0.90;
cWClose = 0.99;
cCriterion = 0.025;
% Loop through test frequencies
for vfreq = cStartFreq:cFreqIncr:cStopFreq
    % Create a new iteration
    testItr = sltest.testmanager.TestIteration();
    % Calculate the time window
    half_period = 0.5 * (1.0/vfreq);
    vw_open = half_period * cWOpen;
    vw_close = half_period * cWClose;
    % Set the parameters for the iteration
    testItr.setVariable('Name','Frequency','Source', ...
        'slrt_ex_osc_sltest','Value',vfreq);
    testItr.setVariable('Name','w_open','Source', ...
        '', 'Value', vw_open);
    testItr.setVariable('Name','w_close','Source', ...
        '', 'Value', vw_close);
    testItr.setVariable('Name','Criterion','Source', ...
        '', 'Value', cCriterion);
    % Name and add the iteration to the testcase
    str = sprintf('%.0f Hz', vfreq);
    addIteration(sltest_testCase, testItr, str);
end
```

Step 13. Run Test

- 1 Build and download `slrt_ex_osc_sltest` to the target computer.
- 2 In Test Manager, click the **Run** button.
- 3 To view test results, in the left column, click **Results and Artifacts**. In this case, the test failed at iteration **23 Hz**.
- 4 To view the failing results, open nodes **23 Hz > Verify Statements** and **23 Hz > Sim Output (slrt_ex_osc_sltest)**.

Step 14. Display Results

- 1 In the Simulation Data Inspector pane, select the **Layout** button.
- 2 Select two horizontal displays.
- 3 In the Simulation Data Inspector top display, select the two Out check boxes and the top **Test Assessment** check box. This assessment corresponds to the HiCheck substep.

- 4 In the bottom display, select the two Out check boxes and the bottom **Test Assessment** check box. This assessment corresponds to the LoCheck substep.
- 5 Click the **Zoom in Time** button and select the range 4.00-4.1.

In the top display, the vertical red line near 4.04 followed by a horizontal green line shows that the HiCheck test failed briefly before succeeding. In the bottom display, the vertical red spike near 4.02 followed by a horizontal green line shows that the LoCheck test failed briefly before succeeding.

See Also

Test Assessment | Test Sequence

More About

- “Test Models in Real Time” (Simulink Test)
- “Reuse Desktop Test Cases for Real-Time Testing” (Simulink Test)

Examples

Simulink Real-Time Examples

Parameter Tuning and Data Logging

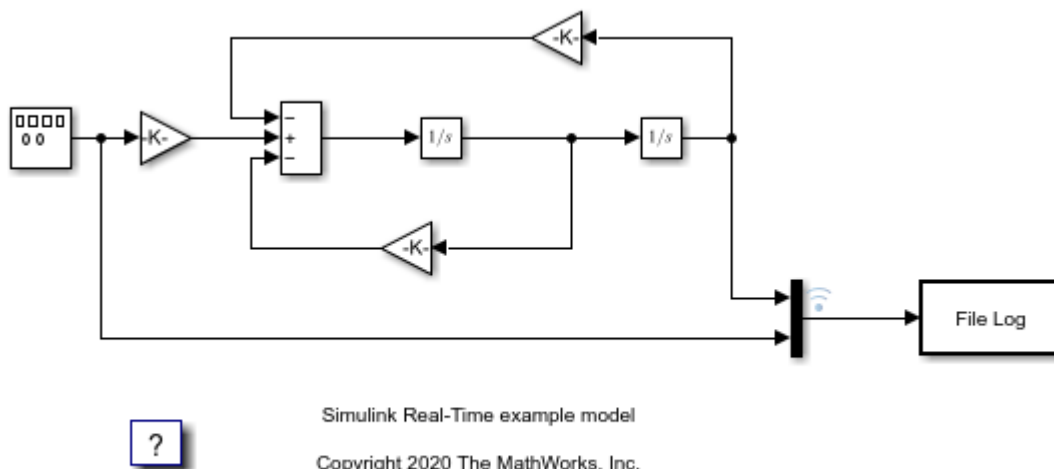
This example shows how to use real-time parameter tuning and data logging with Simulink® Real-Time™. After the example builds the model and downloads the real-time application, `slrt_ex_param_tuning`, to the target computer, the example executes multiple runs with the gain 'Gain1/Gain' changed (tuned) before each run. The gain sweeps from 0.1 to 0.7 in steps of 0.05.

The example uses the data logging capabilities of Simulink Real-Time to capture signals of interest during each run. The logged signals are uploaded to the development computer and plotted. A 3-D plot of the oscillator output versus time versus gain is displayed.

Open, Build, and Download Model to the Target Computer

Open the model, `slrt_ex_param_tuning`. The model configuration parameters select the `slrealtime.tlc` system target file as the code generation target. Building the model creates a real-time application, `slrt_ex_param_tuning.mldatx`, that runs on the target computer.

```
model = 'slrt_ex_param_tuning';
open_system(fullfile(matlabroot, 'toolbox', 'slrealtime', 'examples', model));
```



Build the model and download the real-time application, `slrt_ex_param_tuning.mldatx`, to the target computer.

- Configure for a non-Verbose build.
- Build and download application.

```
set_param(model, 'RTWVerbose', 'off');
set_param(model, 'StopTime', '0.2');
evalc('slbuild(model)');
tg = slrealtime;
load(tg, model);
```

Run Model, Sweep 'Gain' Parameter, Plot Logged Data

This code accomplishes several tasks.

Task 1: Create Target Object

Create the MATLAB® variable, `tg`, that contains the Simulink Real-Time target object. This object lets you communicate with and control the target computer.

- Create a Simulink Real-Time target object.
- Set stop time to 0.2s.

Task 2: Run the Model and Plot Results

Run the model, sweeping through and changing the gain (damping parameter) before each run. Plot the results for each run.

- If no plot figure exist, create the figure.
- If the plot figure exist, make it the current figure.

Task 3: Loop over damping factor z

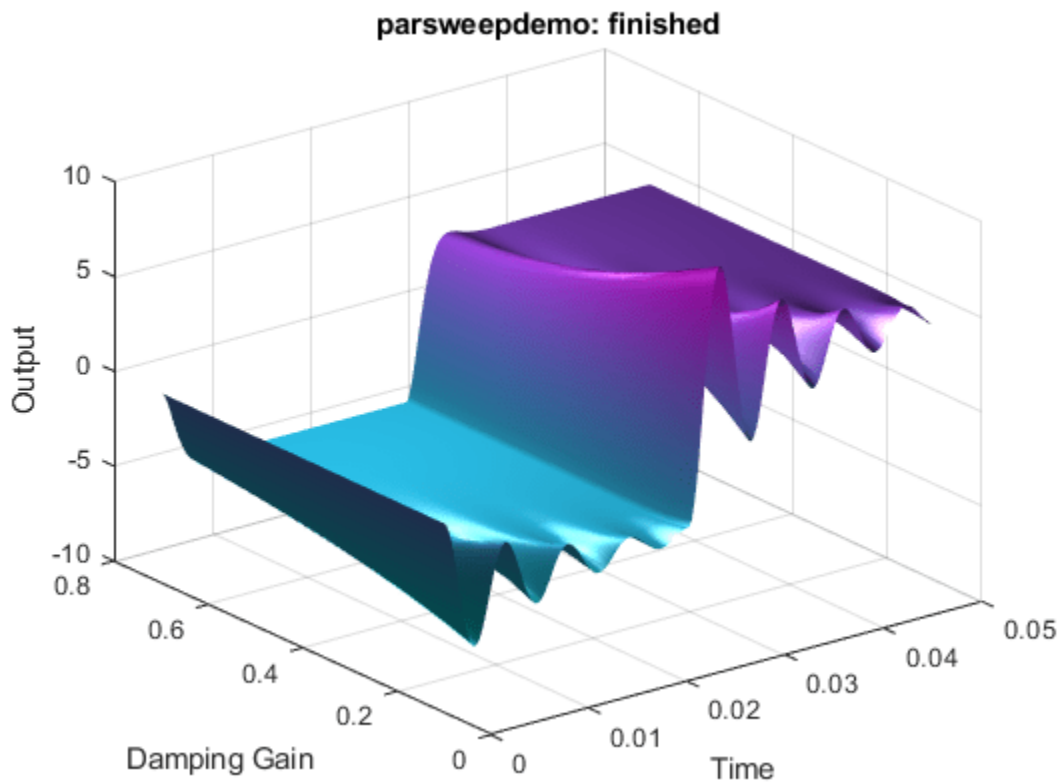
- Set damping factor (Gain1/Gain).
- Start run of the real-time application.
- Store output data in `outp`, `y`, and `t` variables.
- Plot data for current run.

Task 4: Create 3-D Plot (Oscillator Output vs. Time vs. Gain)

- Loop over damping factor.
- Create a plot of oscillator output versus time versus gain.
- Create 3-D plot.

```
figh = findobj('Name', 'parsweepdemo');
if isempty(figh)
    figh = figure;
    set(figh, 'Name', 'parsweepdemo', 'NumberTitle', 'off');
else
    figure(figh);
end
y = []; flag = 0;
for z = 0.1 : 0.05 : 0.7
    if isempty(find(get(0, 'Children') == figh, 1))
        flag = 1;
        break;
    end
    load(tg,model);
    tg.setparam([model '/Gain1'],'Gain',2 * 1000 * z);
    tg.start('AutoImportFileLog',true, 'ExportToBaseWorkspace', true);
    pause(0.4);
    outp = logsOut{1}.Values;
    y = [y,outp.Data(:,1)];
    t = outp.Time;
    plot(t,y);
    set(gca, 'XLim', [t(1), t(end)], 'YLim', [-10, 10]);
    title(['parsweepdemo: Damping Gain = ', num2str(z)]);
    xlabel('Time'); ylabel('Output');
    drawnow;
end
```

```
if ~flag
    delete(gca);
    surf(t(1 : 200), 0.1 : 0.05 : 0.7, y(1 : 200, :));
    colormap cool
    shading interp
    h = light;
    set(h, 'Position', [0.0125, 0.6, 10], 'Style', 'local');
    lighting gouraud
    title('parsweepdemo: finished');
    xlabel('Time'); ylabel('Damping Gain'); zlabel('Output');
end
```



Close Model

When done, close the model.

```
close_system(model,0);
```

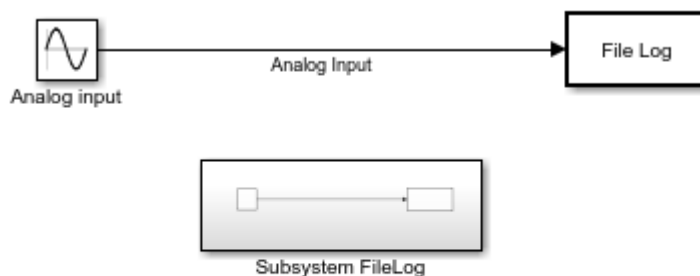
Tune Decimation for File Log Data Without Model Rebuild

This example shows how to tune the decimation parameter on the File Log blocks in a real-time application without rebuilding the model. The Application object methods `getAllFileLogBlocks`, `getFileLogDecimation`, and `setFileLogDecimation` are used to change the decimation value for the application MLDATX file. The application can be run again on the target computer to observe the updated decimation of the signals connected to File Log blocks in the model.

Open, Build, and Download Model

Open the model `slrt_ex_filelogtunabledecimation>`. This model uses File Log blocks to log data on the target computer. The default setting for decimation is set to 1 for the File Log blocks. In the MATLAB Command Window, type:

```
model = 'slrt_ex_filelogtunabledecimation';
open_system(fullfile(matlabroot, 'toolbox', 'slrealtime', 'examples', model));
```



Copyright 2022 The MathWorks, Inc.

Build the top model and download to the target computer.

- Configure for a non-Verbose build.
- Build and download application.

```
set_param(model, 'RTWVerbose', 'off');
set_param(model, 'StopTime', '5');
evalc('slbuild(model)');
tg = slrealtime;
load(tg, model);
```

Close the model

```
bdclose(model);
```

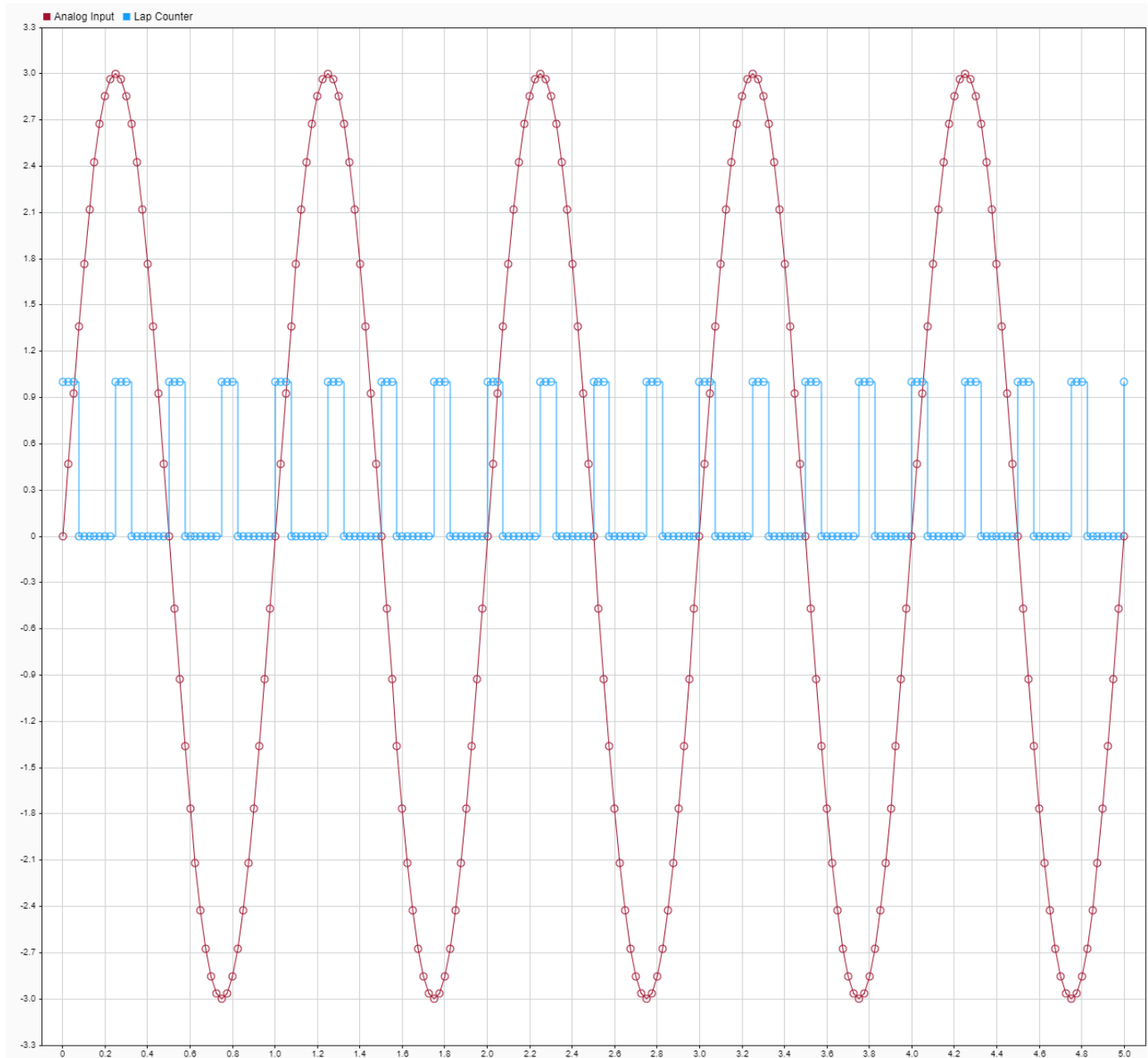
Run Real-Time Application

Run the real-time application on the target computer. Wait for the application to stop.

```
start(tg);
pause(7);
```

View Signals in the Simulation Data Inspector

```
Simulink.sdi.view;
```



Set File Log Block Decimation

Set the File Log block decimation for the blocks to the same value by using the Application object. Create the Application object.

```
appObj = slrealtime.Application(model);
```

Get the File Log blocks in the application by using the `getAllFileLogBlocks` function.

```
fileLogBlocks = appObj.getAllFileLogBlocks;
```

Get the File Log decimation setting for the blocks by using the `getFileLogDecimation` function. The setting is 1 for both blocks.

```
oldDecimation = appObj.getFileLogDecimation(fileLogBlocks)
```

```
oldDecimation =
```

```
    1
```

Change the decimation for both blocks to 5 by using the `setFileLogDecimation` function.

```
appObj.setFileLogDecimation(fileLogBlocks, 5);
```

Confirm the new decimation value is set to 5.

```
newDecimation = appObj.getFileLogDecimation(fileLogBlocks)
```

```
newDecimation =
```

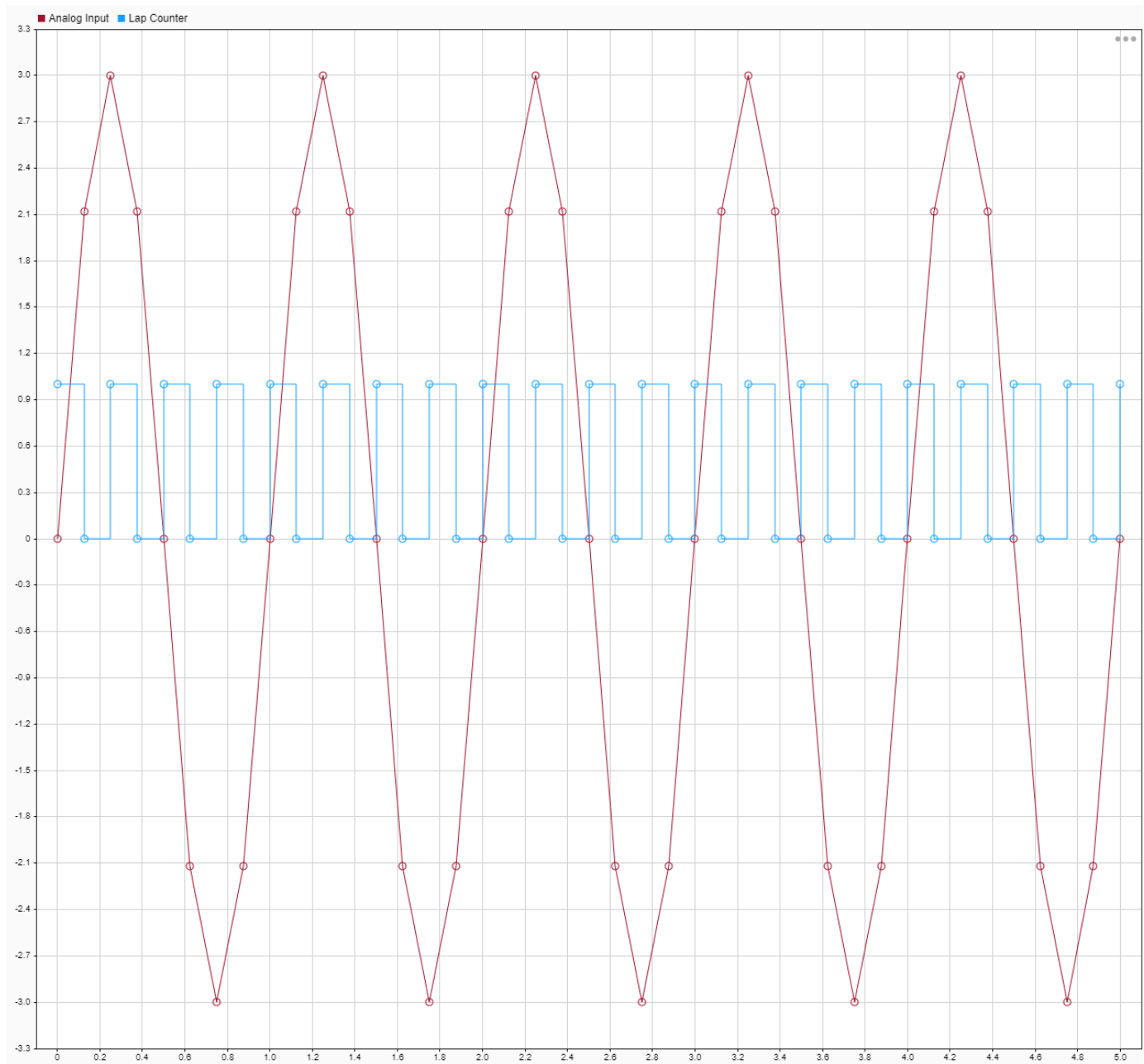
```
    5
```

Reload Application on Target Computer and Rerun

```
load(tg,model);  
start(tg);  
pause(7);
```

View New Signals in the Simulation Data Inspector

```
Simulink.sdi.view;
```



Set File Log Block Decimation

Set the File Log block decimation for blocks to different values by using the `Application` object. Create the `Application` object

```
appObj = slrealtime.Application(model);
```

Get the File Log blocks in the application by using the `getAllFileLogBlocks` function.

```
fileLogBlocks = appObj.getAllFileLogBlocks;
```

Get the File Log decimation setting for the blocks by using the `getFileLogDecimation` function. The setting is 1 for both blocks.

```
oldDecimation = appObj.getFileLogDecimation(fileLogBlocks)
```

```
oldDecimation =
```

```
    5
```

Change the decimation for both blocks to 5 by using the `setFileLogDecimation` function.

```
appObj.setFileLogDecimation(fileLogBlocks, [1 2]);
```

Confirm the new decimation value is set to 5.

```
newDecimation = appObj.getFileLogDecimation(fileLogBlocks)
```

```
newDecimation =
```

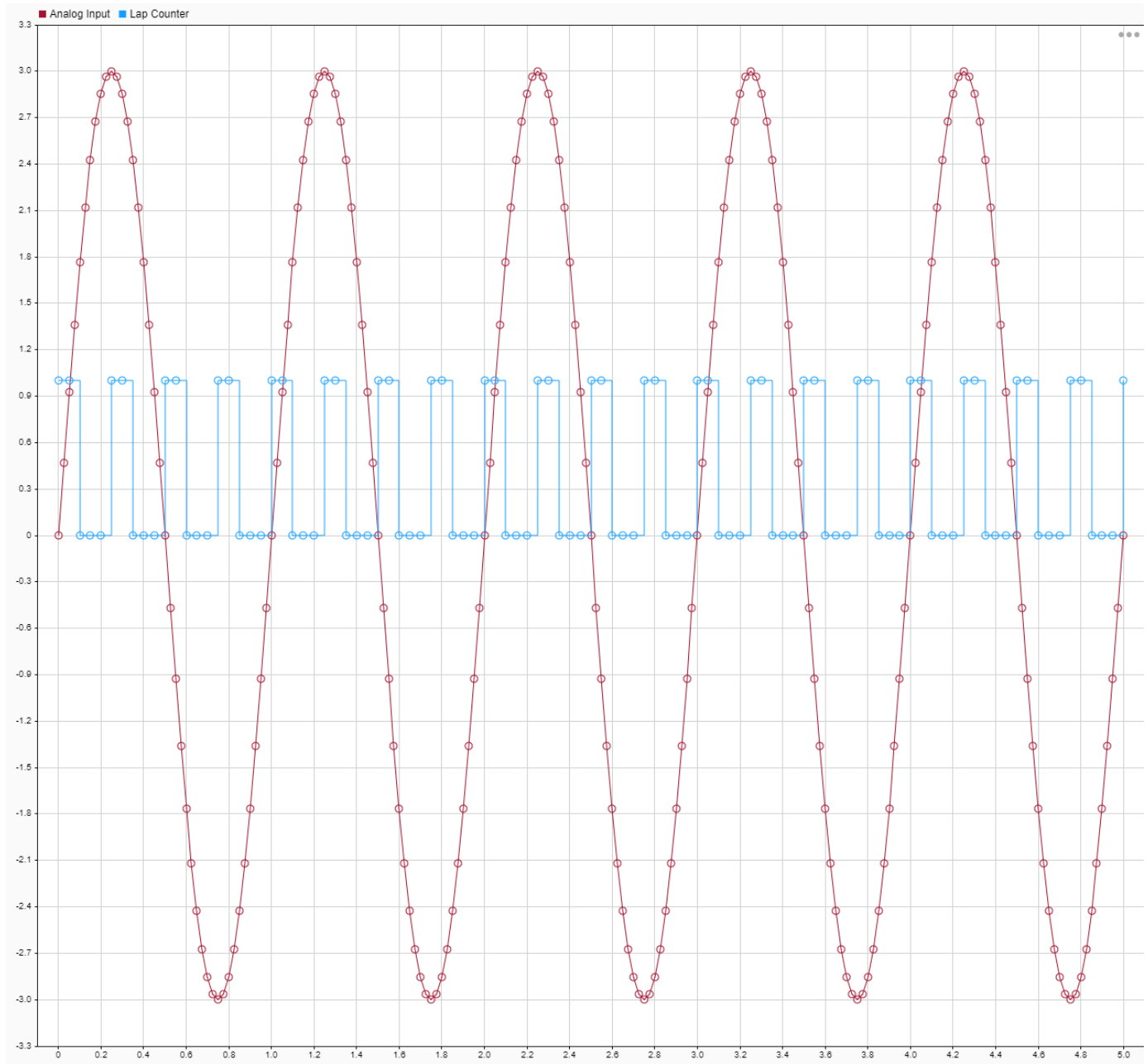
```
    1  
    2
```

Reload Real-Time Application and Rerun

```
load(tg,model);  
start(tg);  
pause(7);
```

View Signals in the Simulation Data Inspector

```
Simulink.sdi.view;
```



Concurrent Execution on Simulink Real-Time

This example shows how to apply explicit partitioning to enhance concurrent execution of a real-time application that you generate by using Simulink® Real-Time™.

Simulink Real-Time supports concurrent execution by using implicit partitioning or explicit partitioning of models. This example shows the relationship between the explicit partitioning of the tasks in the model subsystems and the execution of tasks by using the Simulink Real-Time profiling tool.

The example model `slrt_ex_mds_and_tasks` runs at sample rate of 0.001 second.

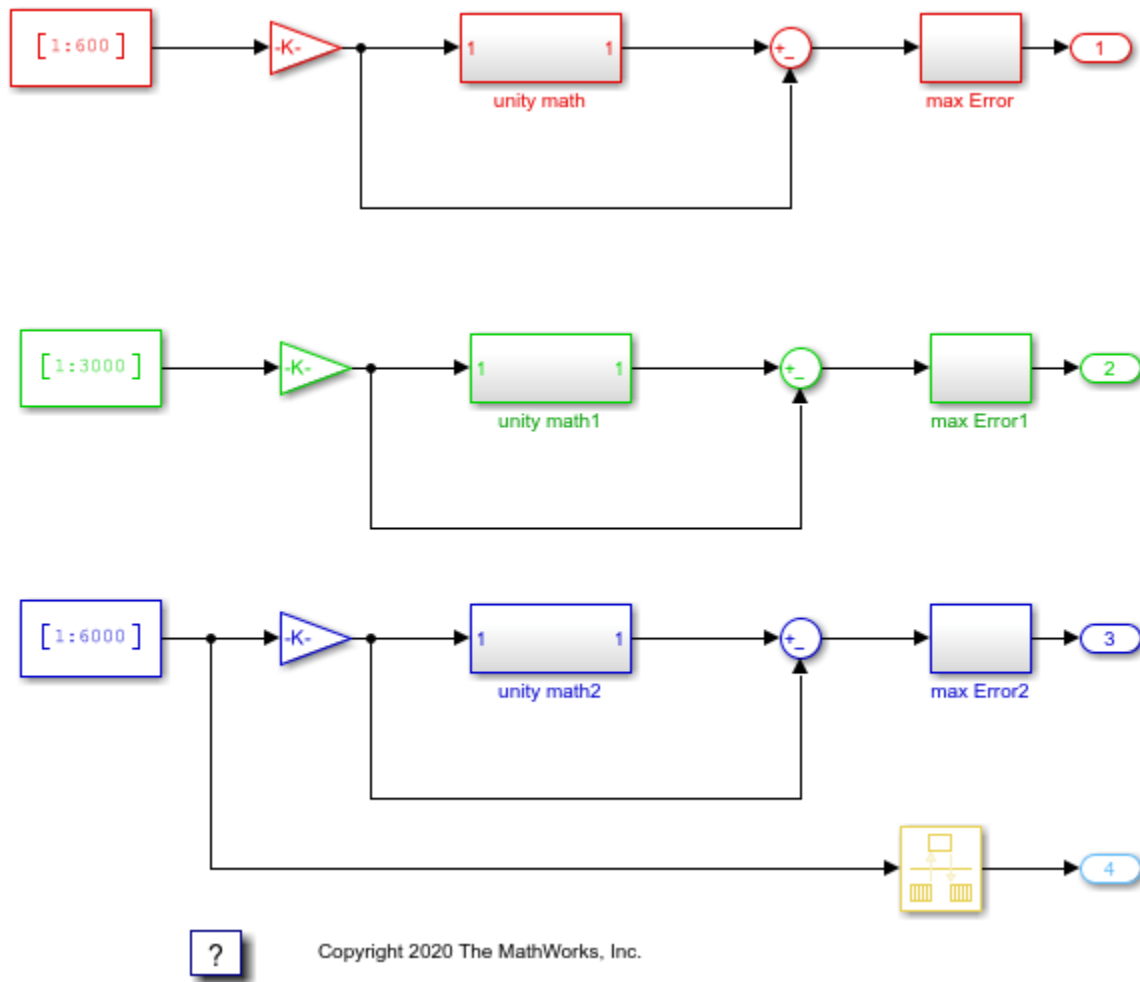
To run the model with adjusted sample rate of 0.01 second, change the sample rate before running the example. In the MATLAB Command Window, type:

```
Ts = 0.01;
```

Open, Build, and Download the Model

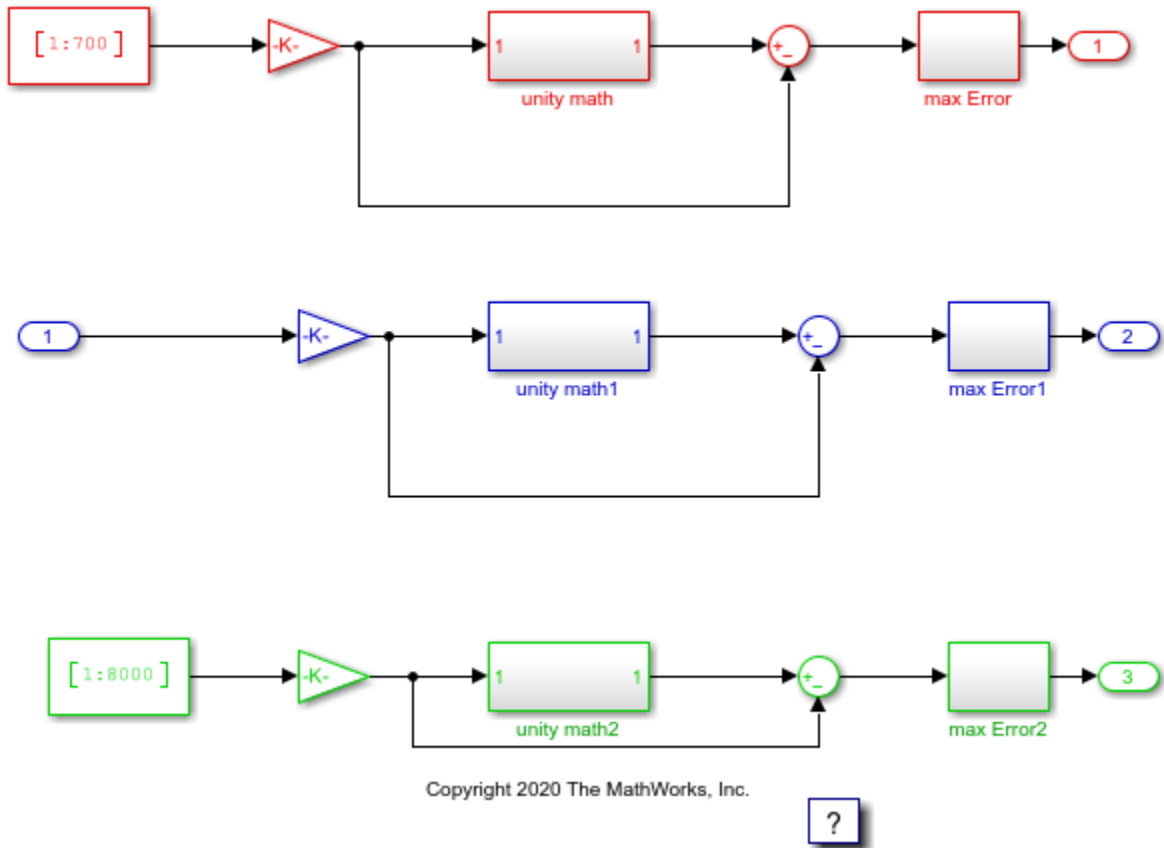
The explicit partitioning in the top-level model occurs in `subsystem1`.

```
open_system(fullfile(matlabroot, 'toolbox', 'slrealtime', 'examples', 'slrt_ex_mds_subsystem1'));
```



The explicit partitioning in the top-level model occurs in subsystem2.

```
open_system(fullfile(matlabroot, 'toolbox', 'slrealtime', 'examples', 'slrt_ex_mds_subsystem2'));
```

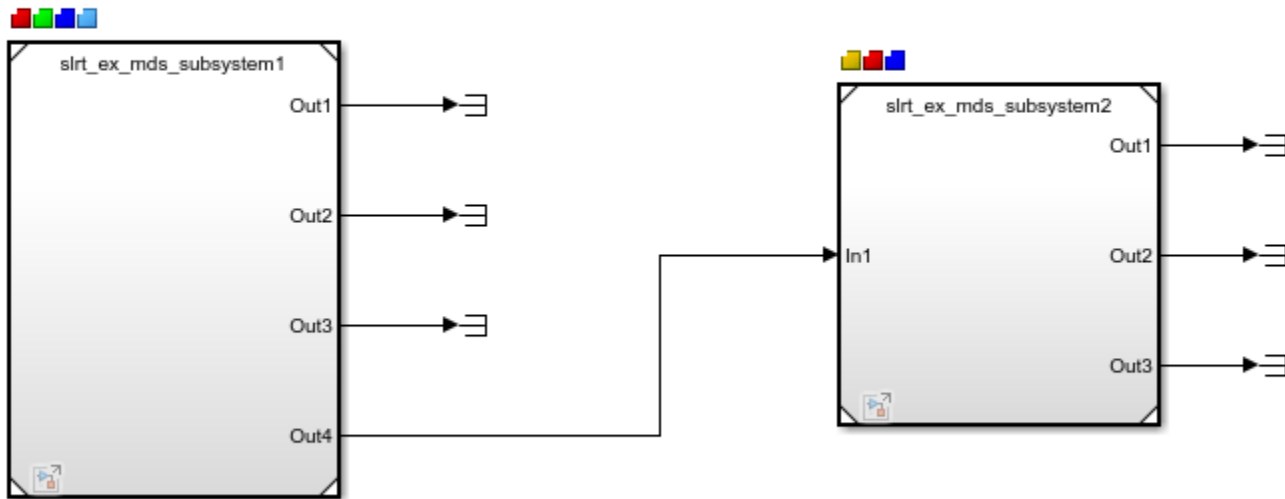


Open the model `slrt_ex_mds_and_tasks`. The model is mapped to seven threads: `Model1_R1`, `Model1_R2`, `Model1_R3`, `Model1_R4`, `Model2_R1`, `Model2_R3`, and `Model2_R4`.

These threads run at sample rates of T_s , $2 \cdot T_s$, $3 \cdot T_s$, $4 \cdot T_s$, T_s , $3 \cdot T_s$, and $4 \cdot T_s$.

```
model='slrt_ex_mds_and_tasks';
open_system(fullfile(matlabroot,'toolbox','slrealtime','examples',model));
```

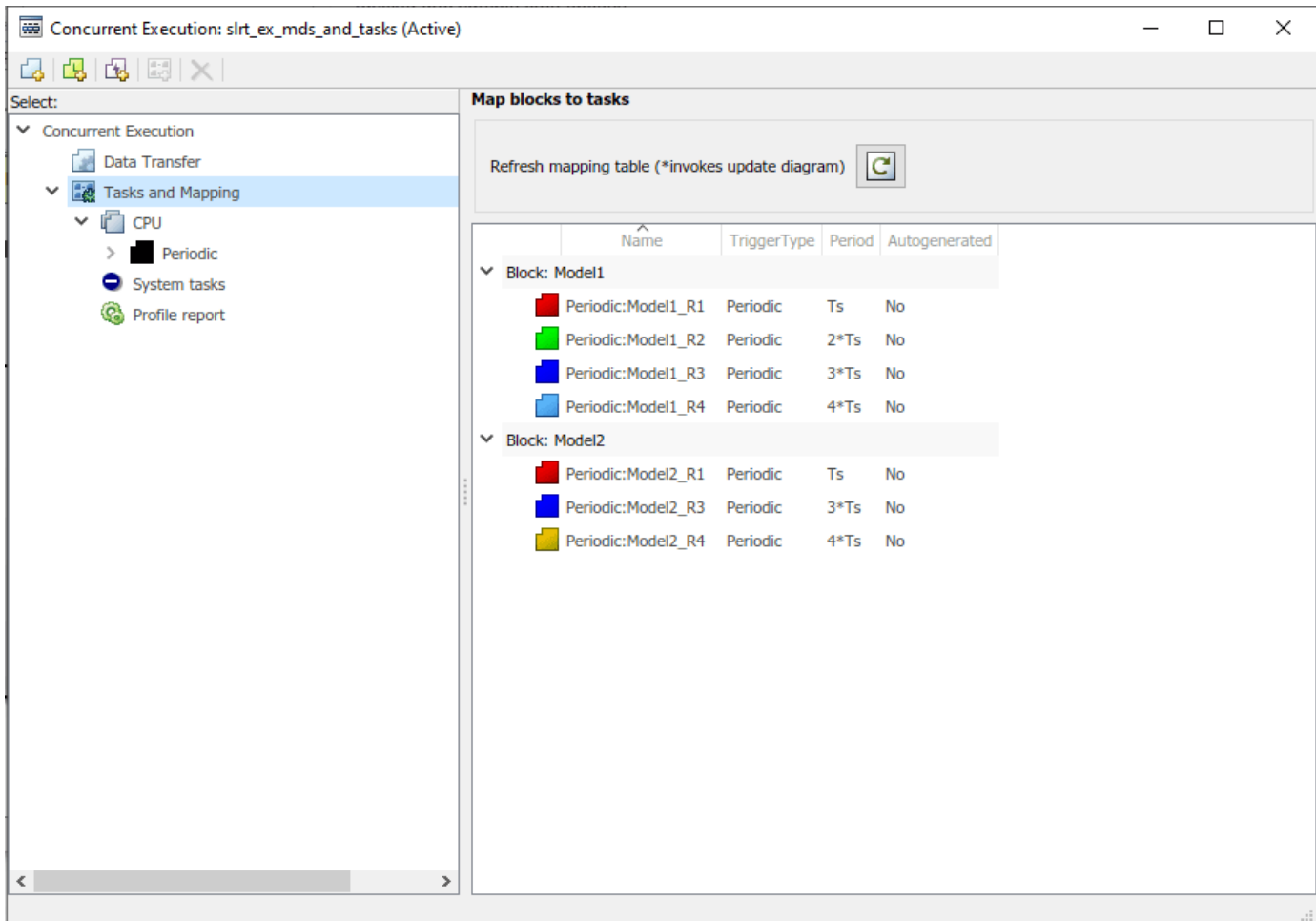
Concurrent Execution on Simulink Real-Time Illustrated by Profiling Tool



Copyright 2020-2021 The MathWorks, Inc.



To apply explicit partitioning, in the Simulink Editor, on the **Real-Time** tab, click **Hardware Settings**, and then select **Solver > Configure Tasks**. Select the Tasks and Mapping node.



Build, download, and run the model.

```
set_param(model, 'RTWVerbose', 'off');
evalc('slbuild(model)');
tg = srealttime;
load(tg, model);
% Open TET Monitor
slrtTETMonitor;
% Start profiler on the target computer
startProfiler(tg);
start(tg);
pause(2);
stop(tg);
```

Display Profiling Data

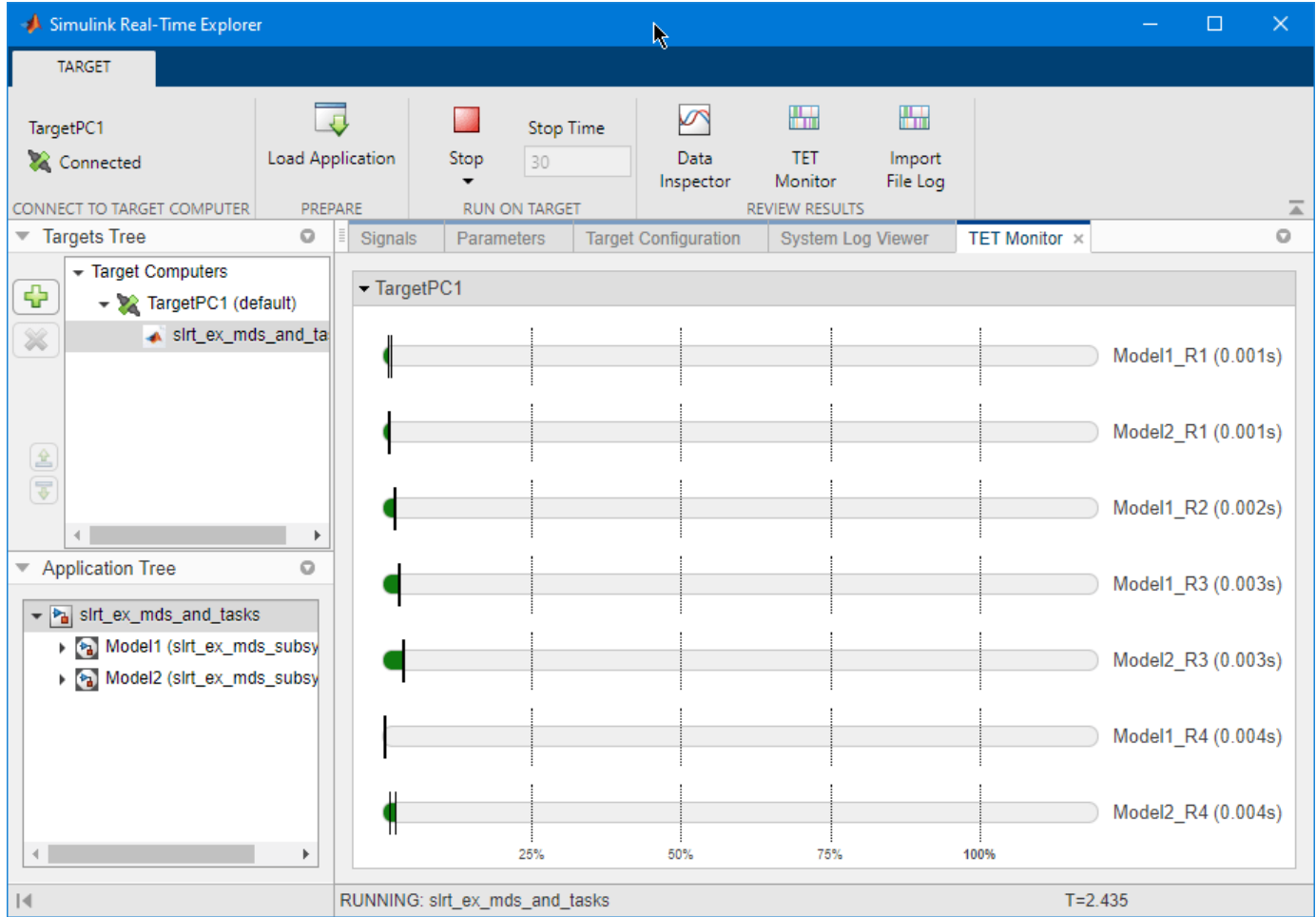
The profiling data shows the execution time of each thread on a multi-core target computer.

```
profData = tg.getProfilerData;
profData.plot;
```

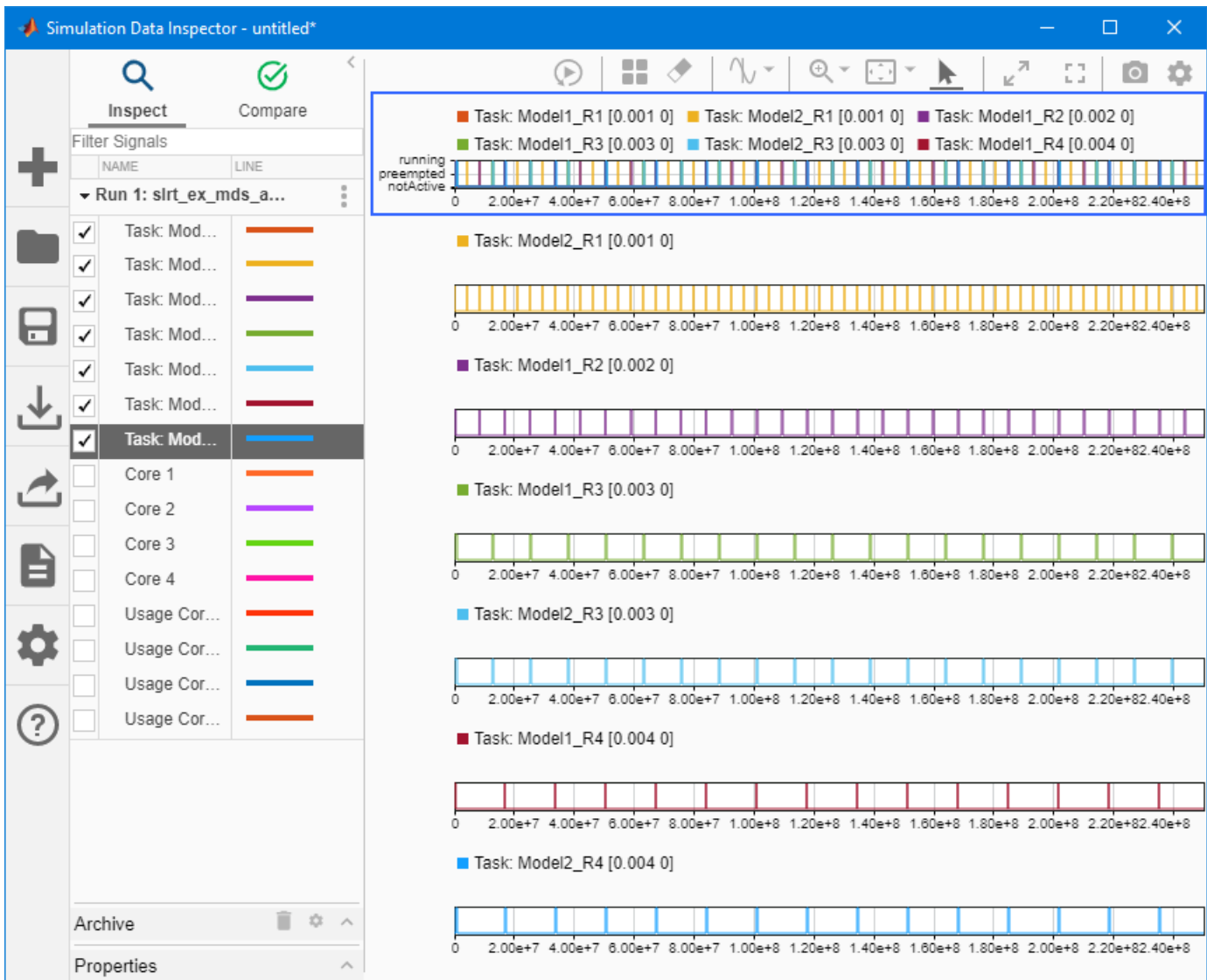
```
Processing data on target computer ...
Transferring data from target computer ...
```

Processing data on host computer ...

View TET Information in TET monitor



View TET Information in the Simulation Data Inspector



See Also

For more information, see:

- “Generate Subsystem Code as Separate Function and Files”
- “Generate Code and Executables for Individual Subsystems”
- “Generate Inlined Subsystem Code”
- “Generate Subsystem Code as Separate Function and Files”
- “Generate Reusable Code from Library Subsystems Shared Across Models”

Close the Model

```
bdclose('all');
```

Add App Designer App to Inverted Pendulum Model

This example shows how to stream signal signals to an App Designer instrument panel app from a Simulink® Real-Time™ application. The example builds the real-time application from the model `slrt_ex_pendulum_100Hz`. The instrument panel contains these App Designer components:

- Target selector dropdown list — To show all the available target computers.
- Connect/disconnect button — To connect or disconnect the target computer chosen in the drop down window.
- Load button — To load the application to the target computer.
- Start/stop button — To start or stop the application on the target computer.
- Stop time edit field — To display and set the stop time of the application loaded on the target computer.
- Status message box — To display target computer status information.
- Axes — To display an animation for the two inverted pendulum and cart system.
- Axes — To display signal output for responses to disrupting the pendulum.
- Nudge cart button — To apply input (nudge) to the cart that hold the pendulum.
- Reference position knob — To change the reference position of the pendulum and cart system.
- Reference variation pattern knob — To add a variation pattern to the reference position of the pendulum and cart system.
- Amplitude slider — To adjust the amplitude of the chosen reference variation pattern.
- Frequency slider — To modify the frequency of the chosen reference variation pattern.

To stream signal and parameter data between the real-time application and the instrument panel app, the app uses the instrumentation object.

Open Example and Load Model

```
openExample('SlrtAddAppDesignerAppToInvertedPendulumModelExample');
load_system(fullfile(matlabroot, 'toolbox', 'slrealtime', 'examples', 'slrt_ex_pendulum_100Hz'));
```

Start Target Computer and Build Real-Time Application

These tasks generate the real-time application that streams data to the App Designer instrument panel app.

- 1 Start the target computer.
- 2 Open the model `slrt_ex_pendulum_100Hz`.
- 3 Connect the development computer to the target computer. Build the `slrt_ex_pendulum_100Hz` model.
- 4 Deploy the real-time application to the target computer.

In the MATLAB® Command Window, type:

```
model = 'slrt_ex_pendulum_100Hz';
set_param(model, 'RTWVerbose', 'off');
tg = slrealtime;
```



```
evalc('slbuild(model)');  
load(tg,model);
```

Run App Designer Instrument Panel App

The App Designer instrument panel app `slrt_ex_pendulumApp` provides controls to start and interact with the real-time application `slrt_ex_pendulum_100Hz`.

1. Run the app. To start the App Designer app `slrt_ex_pendulumApp.mlapp` and create the handle `app`, in the MATLAB Command Window, type:

```
app = slrt_ex_pendulumApp;
```

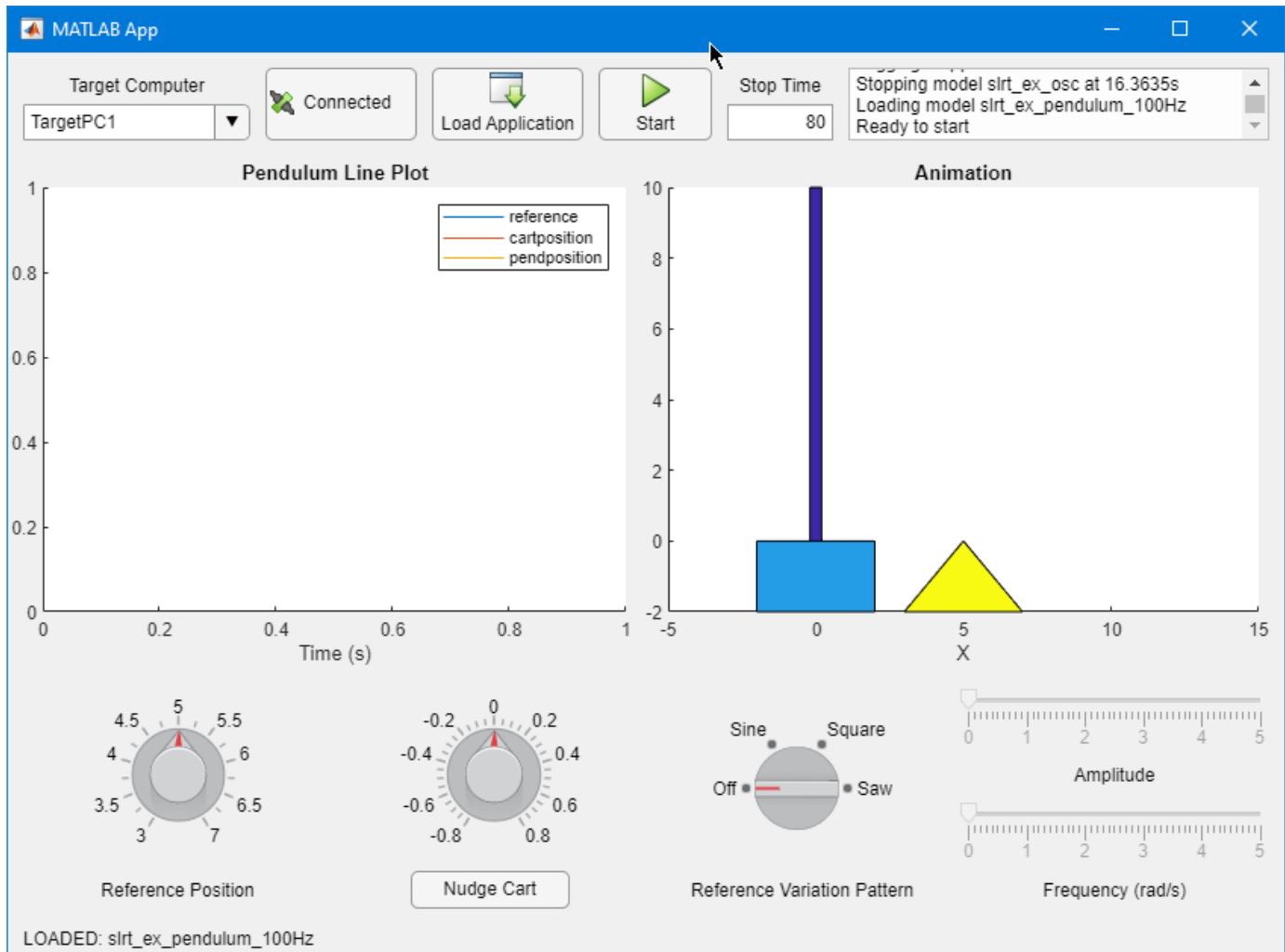
2. To connect with an available target computer, click the **connect** button. The text on the button will switch to 'disconnect' and the **load** button will be enabled.

3. To load the application to the target computer, click the **load** button. After the application is loaded on the target computer, the **start** button and **stop time** edit field will be enabled.

4. To set the stop time of the application, type your preferred stop time in the edit field and hit **enter** button.

5. To start running the application, click the **start** button.

6. To disrupt the equilibrium of the pendulum on each cart, click the **Nudge** button. You can adjust the nudge magnitude by using the value selection next to the button, change the reference position by adjusting the value of reference position spinner, or choose a variation pattern for the reference position.



App Callback Code

The instrument panel app functionality is provided by callback code.

Comments in the callback code in the instrument panel app `slrt_ex_pendulumApp.mlapp` describe the callback operations and programming suggestions. To view the callback code, open `slrt_ex_pendulumApp.mlapp` in the App Designer, and then click the **Code View** tab. In the Command Window, type:

```
edit slrt_ex_pendulumApp
```

Specify Block Paths for Signals

To stream data from signals in the model, see the use of `connectLine` functions in the `setupInstrumentation(app)` function in the app.

updateAnimationCallback Function

For each `AcquireGroup`, this function checks whether there is fresh data since the last time the callback was called. If there is data, the function updates the animation objects.

Signals are placed in Acquire Groups based on sample rate and decimation such that all signals in an Acquire Group have the same time vector.

Update Axes and Animation by Using Acquire Groups

In the callback code, this processing is visible as `AcquireGroupData` signal groups in the `updateAnimationCallback` function.

Close the App and Models

The instrument panel app handle `app` provides access to close the app.

Close the app. In the MATLAB Command Window, type:

```
close(app.UIFigure)
```

Close the open models. In the Command Window, type:

```
bdclose ('all');
```

Basic App Designer App for Real-Time Application Interface

This example shows a basic App Designer app that provides an interface to a real-time application.

Open Model and Build Real-Time Application

Open the model `slrt_ex_waves` and build the real-time application.

```
open_system(fullfile(matlabroot, 'toolbox', 'slrealtime', 'examples', 'slrt_ex_sine_waves'));  
model='slrt_ex_sine_waves';  
evalc('slbuild(model)');
```

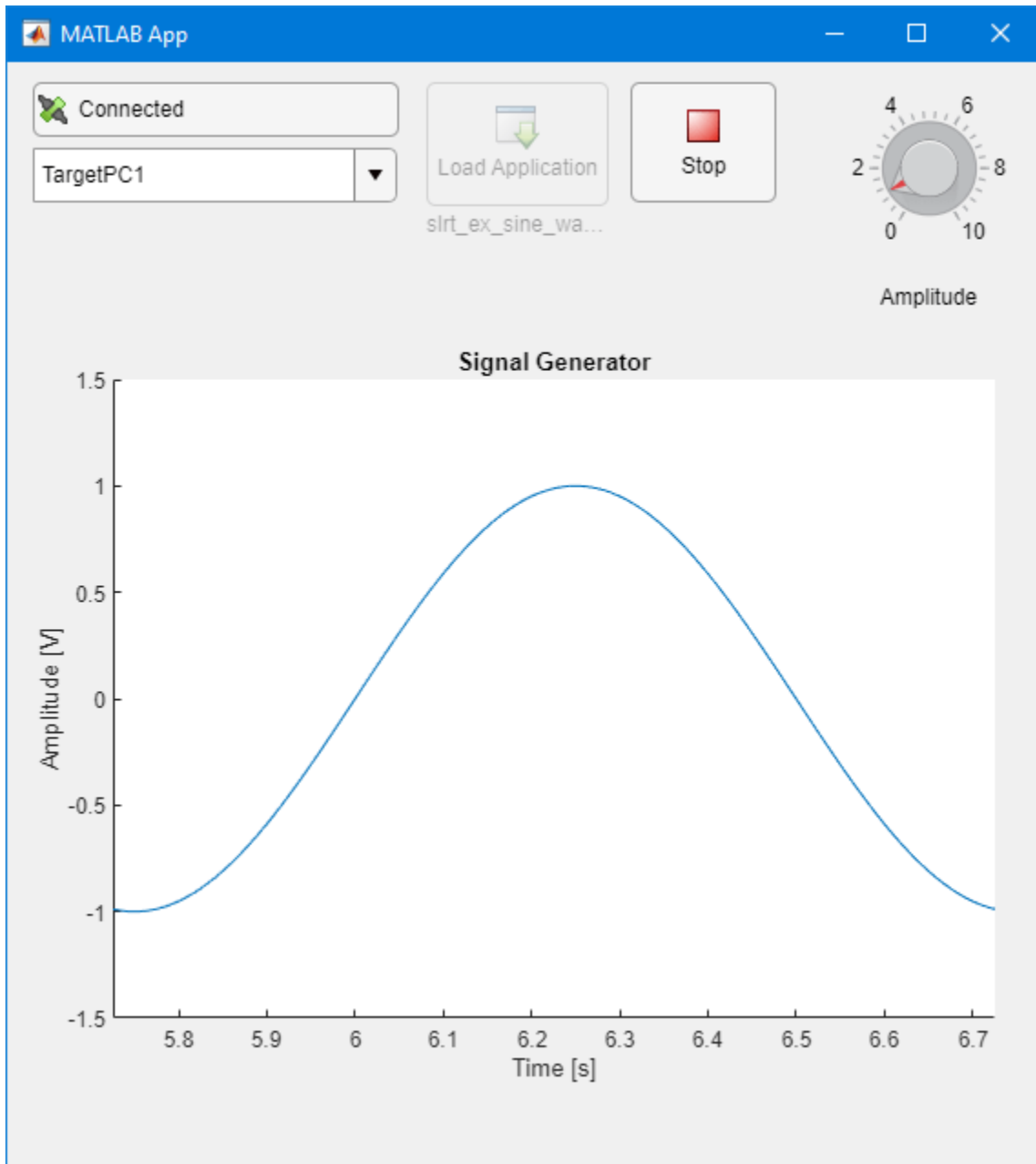


Model `slrt_ex_sine_waves`
Simulink Real-Time example model
Copyright 2021 The MathWorks, Inc.



Open Real-Time Application Interface App

Open the real-time application interface app `slrt_sine_waves_App`.



Control Run Real-Time Application

Use the App Designer app controls to:

- Select the target computer by using the target selector list.
- Toggle from **Disconnected** to **Connected** by using the connect button.
- Load the real-time application by using the load button.
- Start the real-time application by using the start button.

When done, stop the application by using the stop button.

Examining Code View for Interface App

In App Designer, click the **Code View** tab to view the code that connects the real-time application to the App Designer app. This code that executes after the app creates components uses a small number of Simulink® Real-Time™ functions to:

- Create and configure the tuner component
- Create and configure the instrument component
- Start the instrument manager for the app

```
function startupFcn(app)
    % Add Parameter Tuner Component
    Tuner = slrealtime.ui.tool.ParameterTuner(app.UIFigure);
    Tuner.Component = app.AmplitudeKnob;
    Tuner.BlockPath = 'slrt_ex_sine_waves/Sine Wave';
    Tuner.ParameterName = 'Amplitude';

    % Add Instrument Component
    Instrument = slrealtime.Instrument;
    Instrument.connectLine(app.UIAxes,'slrt_ex_sine_waves/Sine Wave',1);
    Instrument.AxesTimeSpan = 1;

    InstrumentManager = slrealtime.ui.tool.InstrumentManager(app.UIFigure);
    InstrumentManager.Instruments = Instrument;
end
```

Close All Open Files

```
bdclose('all');
```

Create Instrument Panel for Testing Highway Lane Following Controller

This example shows how to use the Simulink® Real-Time™ App Generator to create an App Designer instrument panel for the test system described in “Automate Testing for Highway Lane Following Controller” (Automated Driving Toolbox). You can more easily operate apply the automated testing and control the real-time application on the target computer by using an instrument panel.

Open Simulink Project and Models

Open the lane following with PID controller Simulink project `LaneFollowingPIDController.prj`.

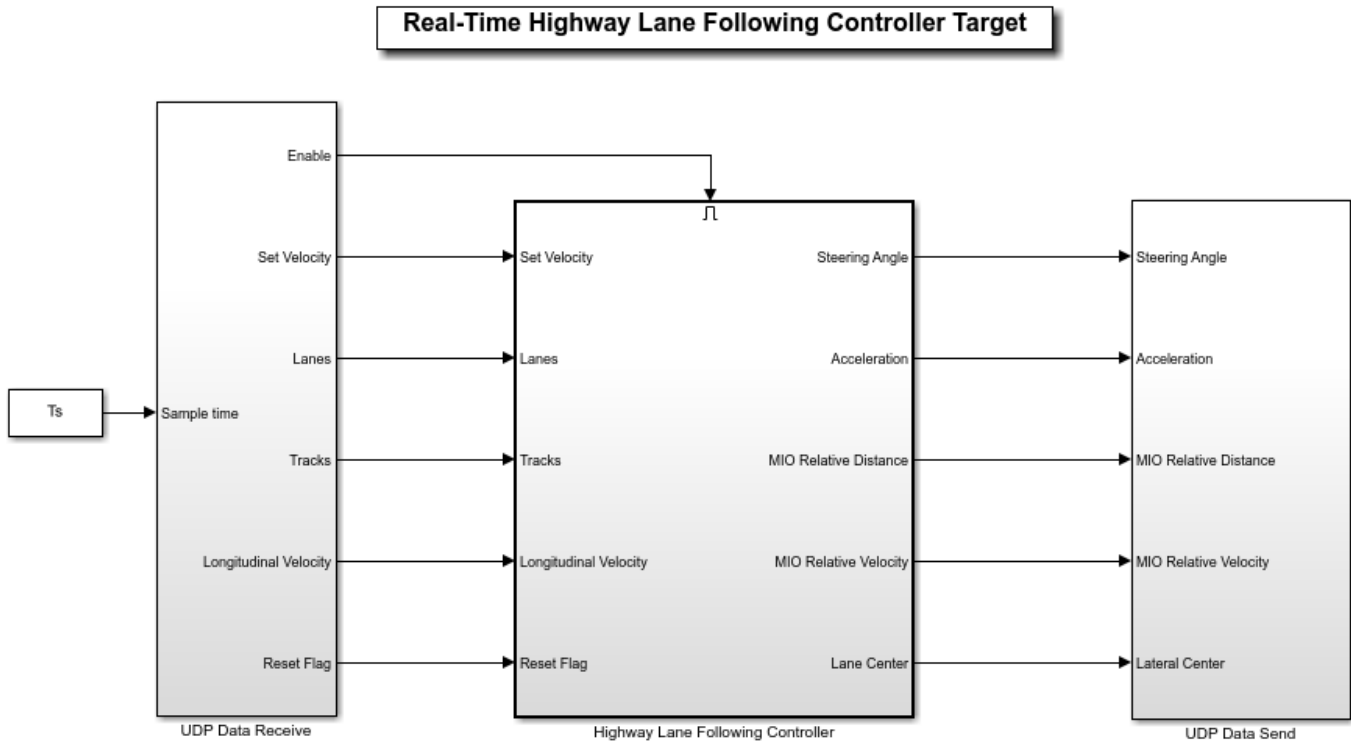
```
unzip(fullfile('LaneFollowingPIDController.zip'))  
proj = openProject(pwd);
```

Open Models and Build Real-Time Application

Open the target computer model and the development computer model. Configure the IP addresses of UDP blocks to match your system configuration.

These commands configure block parameters in target computer model `RTHLFControllerTarget` for development computer IP address: `192.168.7.2`

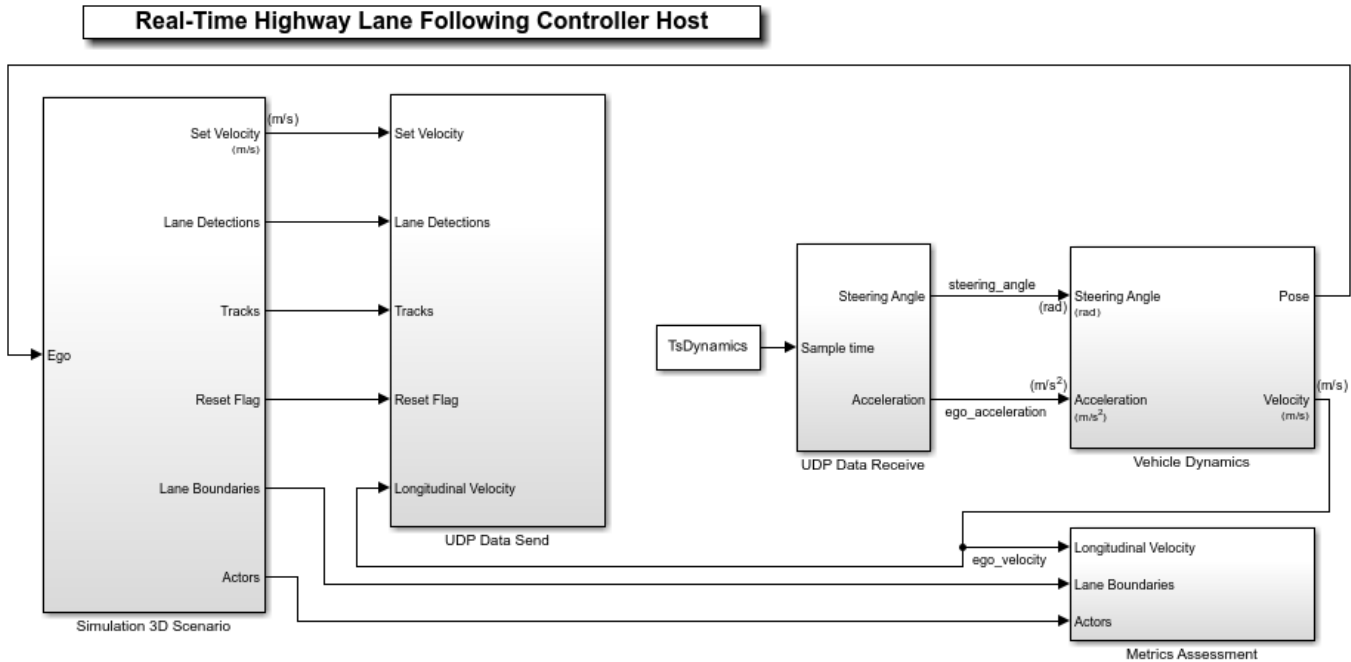
```
open_system('RTHLFControllerTarget');  
set_param('RTHLFControllerTarget/UDP Data Receive/UDP Receive: Lanes /UDP Receive','fmAddress',  
set_param('RTHLFControllerTarget/UDP Data Receive/UDP Receive: Longitudinal Velocity /UDP Receive',  
set_param('RTHLFControllerTarget/UDP Data Receive/UDP Receive: Reset Flag/UDP Receive2','fmAddress',  
set_param('RTHLFControllerTarget/UDP Data Receive/UDP Receive: Set Velocity/UDP Receive2','fmAddress',  
set_param('RTHLFControllerTarget/UDP Data Receive/UDP Receive: Tracks/UDP Receive','fmAddress',  
set_param('RTHLFControllerTarget/UDP Data Send/UDP Send: Controller Intermediate Signals','toAddress',  
set_param('RTHLFControllerTarget/UDP Data Send/UDP Send: Controller Output','toAddress','192.168
```



Copyright 2021 The MathWorks, Inc.

These commands configure block parameters in development computer model RTHLFControllerHost for target computer IP address: 192.168.7.5

```
open_system('RTHLFControllerHost');
set_param('RTHLFControllerHost/UDP Data Receive/Receive Controller Output/UDP Receive: Controller', 'toAddress', '192.168.7.5');
set_param('RTHLFControllerHost/UDP Data Receive/Receive Intermediate Signals/UDP Receive: Controller', 'toAddress', '192.168.7.5');
set_param('RTHLFControllerHost/UDP Data Send/UDP Send: Lane Detections', 'toAddress', '192.168.7.5');
set_param('RTHLFControllerHost/UDP Data Send/UDP Send: Longitudinal Velocity', 'toAddress', '192.168.7.5');
set_param('RTHLFControllerHost/UDP Data Send/UDP Send: Reset Flag', 'toAddress', '192.168.7.5');
set_param('RTHLFControllerHost/UDP Data Send/UDP Send: Set Velocity', 'toAddress', '192.168.7.5');
set_param('RTHLFControllerHost/UDP Data Send/UDP Send: Tracks', 'toAddress', '192.168.7.5');
```

Copyright 2022 The MathWorks, Inc.

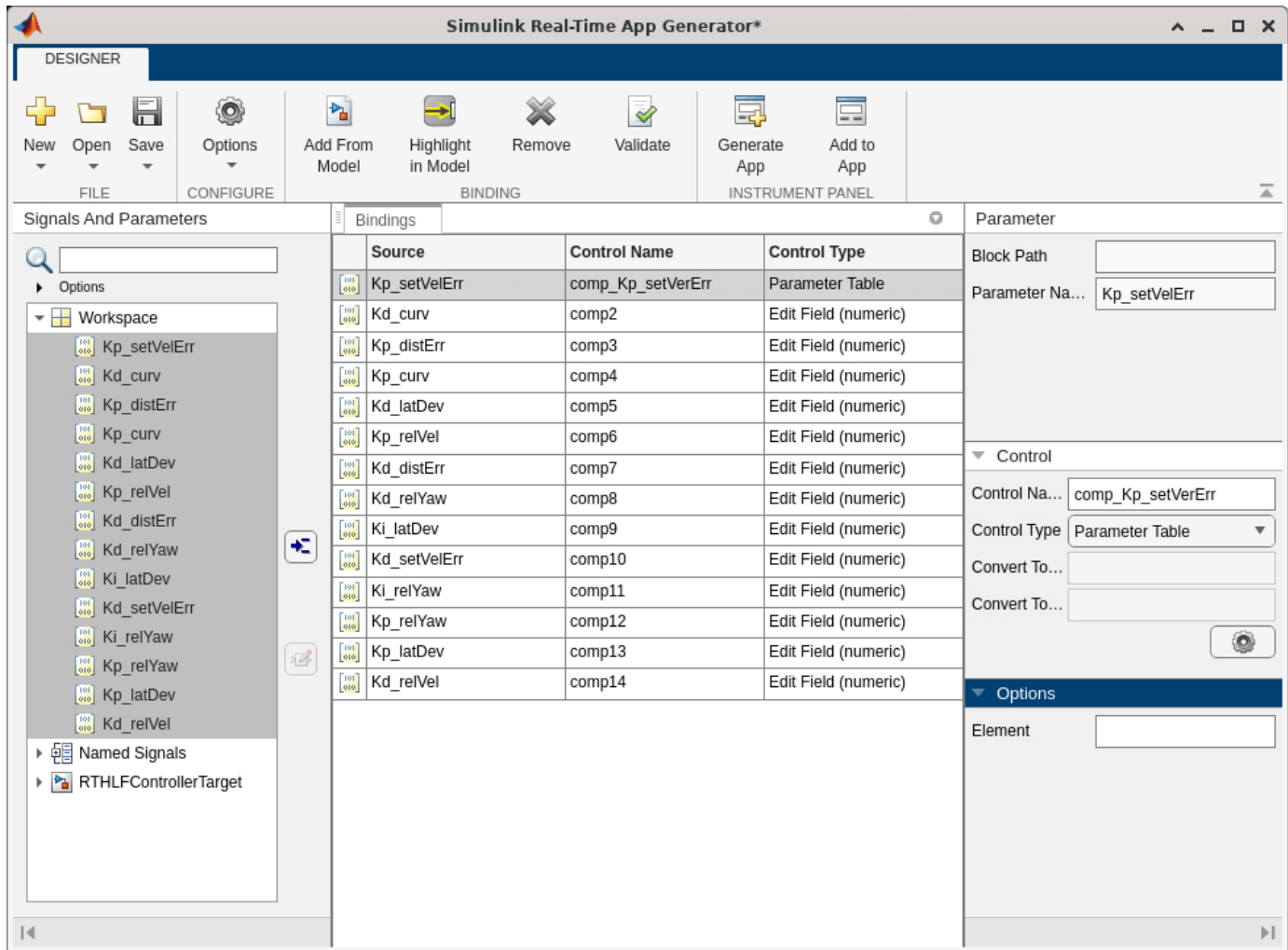
Build the real-time application from the PID controller model `RTHLFControllerTarget` by using the **Run on Target** button on the **Real-Time** tab in the Simulink editor. Or, you can use the project shortcut helper `helperBuildRealTimeApplication`. In the Command Window, type:

```
run 'helperBuildRealTimeApplication';
```

Open App Generator and Bind Signals and Parameters to Instrument in App

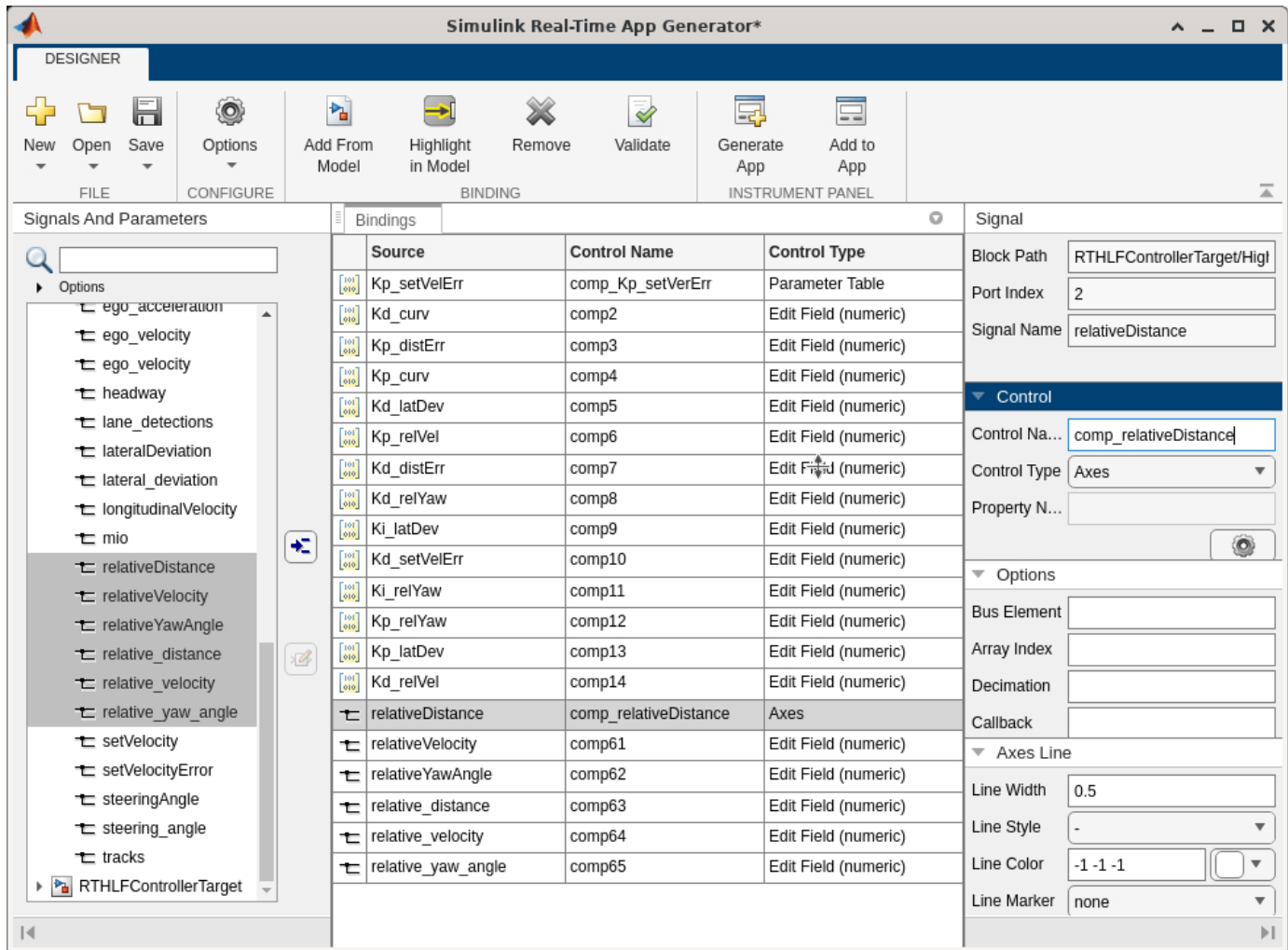
As described in Simulink Real-Time App Generator, you can start creating an App Designer instrument panel from the real-time application MLDATX file.

1. Open the Simulink Real-Time App Generator by using the command `slrtAppGenerator` in the Command Window.
2. Select **New > New**. Then, select the real-time application MLDATX file `LaneFollowingPIDController/Work/RTHLFControllerTarget.mldatx`.
3. In the **Signals And Parameters** pane, select all of the parameters in the **Workspace** group. Click the add button to add these parameters to the **Bindings** tab.



4. For each parameter, set the **Control Type** to **Parameter Table** and select a unique name for the control. For example, use the name `comp_Kp_setVerErr` for the `Kp_setVerErr` parameter.

5. In the **Signals And Parameters** pane, select all of the relative signals (for example, `relativeDistance`) from the **Named Signals** group. Click the add button to add these signals to the **Bindings** tab.



6. For each signal, set the **Control Type** to **Axes** and select a unique name for the control. For example use the name `comp_relativeDistance` for the `relativeDistance` signal.

7. To save the app settings for the App Generator session and return later to continue working with it, you can save the settings to a MAT file by using the **Save** button. After adding the parameters and signals, selecting their controls, and providing control names, click the **Generate App** button to create the App Designer instrument panel app MLAPP file for the real-time application.

8. To continue developing the instrument panel, open the MLAPP file in App Designer.

Open App Designer Instrument Panel

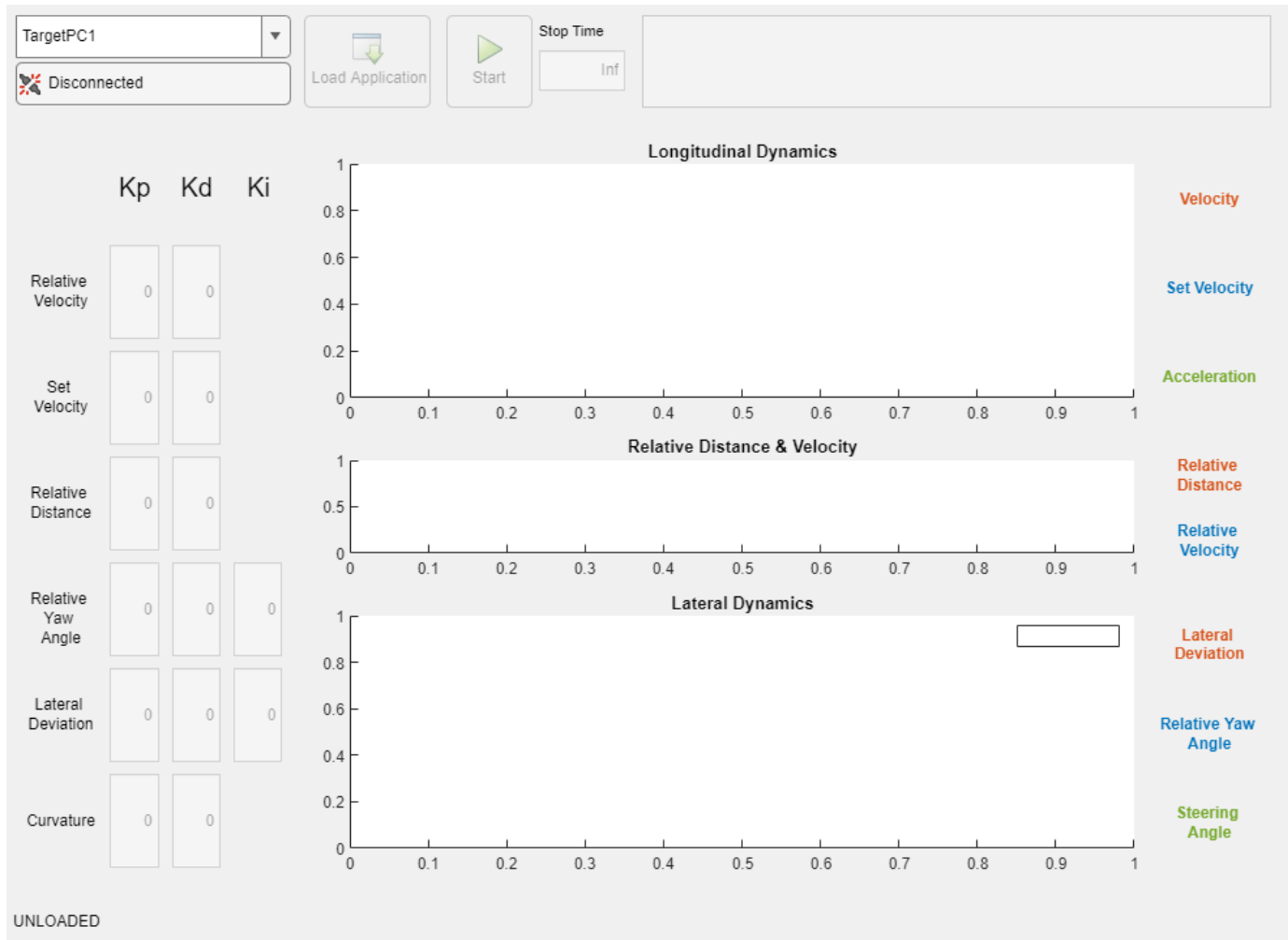
The instrument panel that you create by using the App Generator provides a starting point for instrument panel development and makes it easier to work with instrumented signals and parameters by binding them to controls in the instrument panel app.

This example includes an App Designer instrument panel app that was created by using the Simulink Real-Time App Generator. The generator was used for selecting the real-time application MLDATX file and binding signals and parameter from the application to controls in the instrument panel.

For more information about creating instrument panel apps by using the App Generator, see Simulink Real-Time App Generator.

To open the app, in the project **App** folder, double-click on the app RTHLFControllerTarget_SLRTInstrumentPanel MLAPP file. Or, you can run the project shortcut helperOpenAppDesignerInstrumentPanel. In the Command Window, type:

```
proj = openProject(pwd);
run 'helperOpenAppDesignerInstPanel';
```



Use App Designer Instrument Panel to Control Real-Time Application

To run the simulation, run the development computer model simulation, then immediately start the target computer real-time application.

1. Start the instrument panel app.
2. By using the app, select the target computer and load the real-time application.
3. Start the simulation of the development computer model RTHLFControllerHost.
4. Start the run of the target computer real-time application RTHLFControllerTarget.

5. Use the instrument panel app to control the real-time application.

Deploy Standalone Executable App Designer Instrument Panel

From your App Designer instrument panel, you can compile a standalone executable instrument panel. For more information, see “Create Standalone Instrument Panel App by Using Application Compiler” on page 14-14.

Close Models, App, and Project

After exploring the App Designer instrument panel app and the Simulink project, close the project in MATLAB®. Closing the project closes all open files related to the project. If you prefer, you can close the models and project from the Command Window by typing:

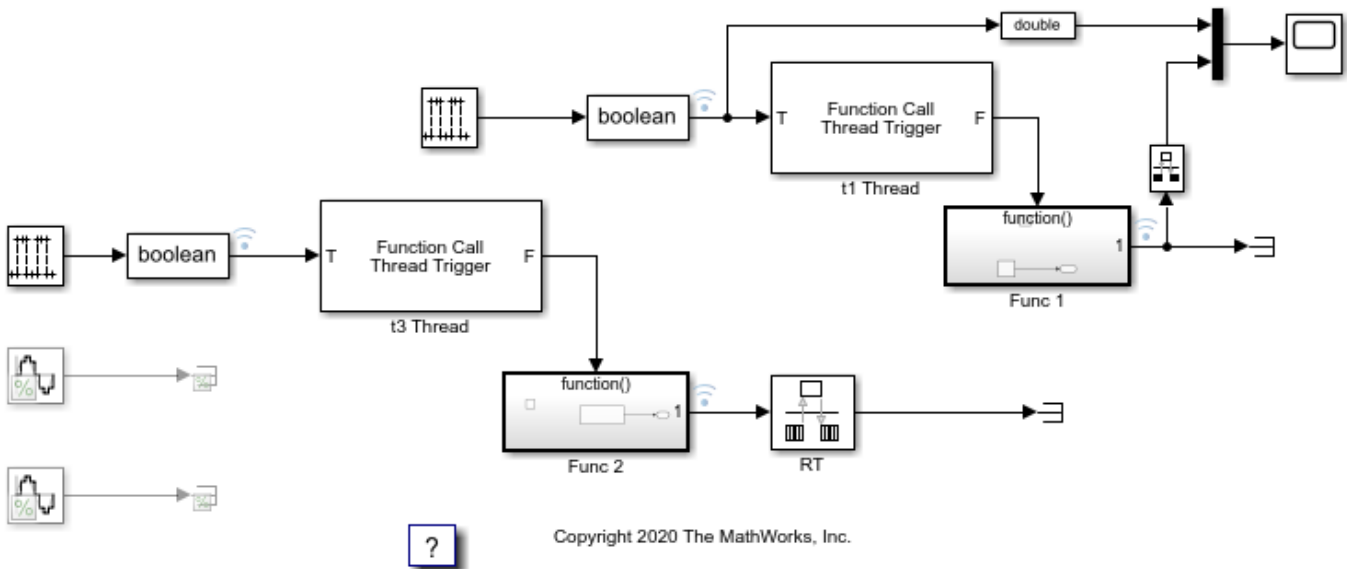
```
bdclose('all');
```

Connect Triggered Subsystem by Using Thread Trigger

This example shows how to connect the Thread Trigger block and create a triggered subsystem. This not-often-used approach lets you use conditions in the model to trigger tasks instead of by using the much more typical approach of using a hardware interrupt from an I/O device in the target computer to trigger tasks.

To open this model, in the MATLAB Command Window, type:

```
open_system(fullfile(matlabroot, 'toolbox', 'slrealtime', 'examples', 'slrt_ex_thread_trigger_fc_sub
```



See Also

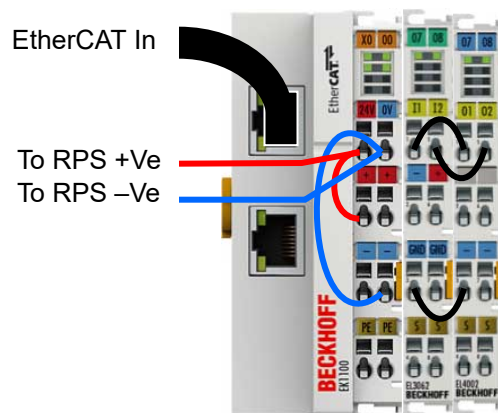
- Thread Trigger
- Function-Call Subsystem
- Triggered Subsystem
- “About RTOS Tasks and Priorities”
- “Execution Modes” on page 8-2

EtherCAT Protocol with Beckhoff Analog IO Slave Devices EL3062 and EL4002

This example shows how to communicate with EtherCAT® devices using the Beckhoff® analog I/O terminals EL3062 and EL4002.

Requirements

To run this example, you need an EtherCAT network that consists of the target computer as EtherCAT Master device and two analog input/output terminals EL3062 and EL4002 as EtherCAT Slave devices. This example requires a dedicated network port that is reserved for EtherCAT using the Ethernet Configuration tool on the target computer. Use the reserved port for EtherCAT communication. This port is in addition to the port used for the Ethernet link between the development and target computers.



To test this model:

- 1 Connect the reserved network port in the target computer to the network IN port of the Beckhoff EK1100 coupler.
- 2 Assemble Terminals EL3062 and EL4002 with Coupler EK1100.
- 3 Loop back the I/O ports: Connect each output port of Terminal EL4002 to a corresponding input port of Terminal EL3062.
- 4 Make sure that the terminals are supplied with the required 24-volt power supply.
- 5 Build and download the model onto the target.

For a complete example that configures the EtherCAT network, configures the EtherCAT master node model, and builds then runs the real-time application, see “Modeling EtherCAT Networks”.

Open the Model

This model creates two sine wave signals and sends the signals to the EL4002 terminal. The model receives input signal values from the EL3062 terminal.

The EtherCAT initialization block requires that the configuration ENI file is present in the current folder. Copy the example configuration file from the example folder to the current folder. To open the model, in the MATLAB® Command Window, type:

```
open_system(fullfile(matlabroot, 'toolbox', 'slrealtime', 'examples', 'slrt_ex_ethercat_beckhoff_aino
```

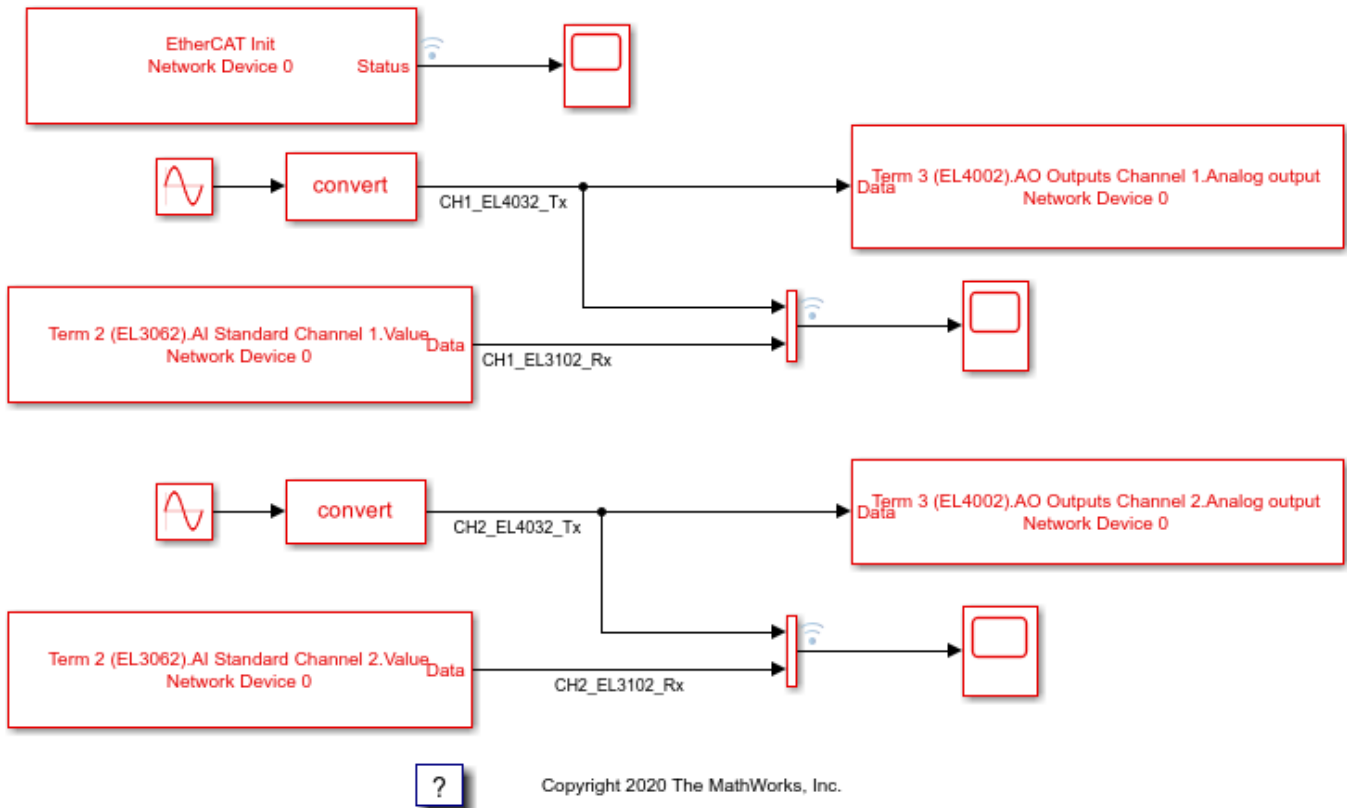


Figure 1: EtherCAT model using Beckhoff analog I/O slave devices EL3062 and EL4002.

Configure the Model

Open the mask for the **EtherCAT Init** block and observe the pre-configured values. The EtherCAT slave devices that are daisy chained together with Ethernet cable is a Device, also referred to as an EtherCAT network. The Device Index selects one such chained EtherCAT network. The Ethernet Port Number identifies which Ethernet port to use to access that Device. The EtherCAT Init block connects these two so that other EtherCAT blocks use the Device Index to communicate with the slave devices on that EtherCAT network.

If you only have one connected network of EtherCAT slaves, and you have only reserved one Ethernet port with the Ethernet configuration tool, use Device Index = 0 and Ethernet Port Number = 1.

Create an ENI File for Different A/D D/A Slave Devices, If Needed

If you need to create a new ENI file you need to use a third-party EtherCAT configurator such as TwinCAT 3 from Beckhoff that you install on a development computer. The EtherCAT configuration (ENI) file preconfigured for this model is **BeckhoffAIOconfig.xml**.

The ENI (EtherCAT Network Information) file that is provided with this example has an EK1100 with EL3062 and EL4002 slaves attached, in that order. If you have different analog IO modules, you need to create a new ENI file for that collection.

For an overview of the process for creating an ENI file, see “Configure EtherCAT Network by Using TwinCAT 3”.

Each EtherCAT configuration file (ENI file) is specific to the exact network setup for which it has been created (for example, the network discovered in step 1 of the configuration file creation process). The configuration file provided for this example is valid if and only if the EtherCAT network consists of terminals EK1100, EL3062, and EL4002.

The ENI file defines a set of transmit and receive variables. For this example, a set of receive variables are defined for each input channel of terminal EL3062. Make sure the variables for channel 1 and channel 2 of terminal EL3102 are selected respectively in the two **EtherCAT PDO Receive** blocks. These two variables are Term 2 (EL3062).AI Standard Channel 1.Value and Term 2 (EL3062).AI Standard Channel 2.Value.

A set of transmit variables are defined for the two output channels of terminal EL4002. Make sure the variables for channel 1 and channel 2 of terminal EL4002 are selected in the two **EtherCAT PDO Transmit** blocks. These two variables are Term 3 (EL4002).AO Outputs Channel 1.Analog Output and Term 3 (EL4002).AO Outputs Channel 2.Analog Output.

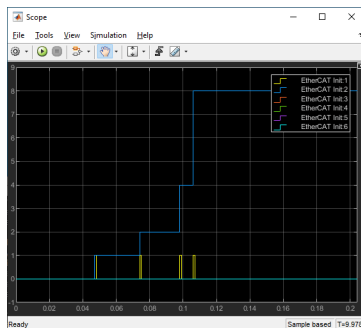
Build, Download, and Run the Model

To build, download, and run the model:

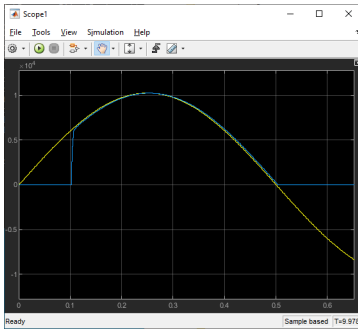
- 1 In the **Simulink Editor**, from the targets list on the **Real-Time** tab, select the target computer on which to run the real-time application.
- 2 Click **Run on Target**.

If you open the three host side scopes by double clicking each, data is relayed from the target back to the development computer and displayed there.

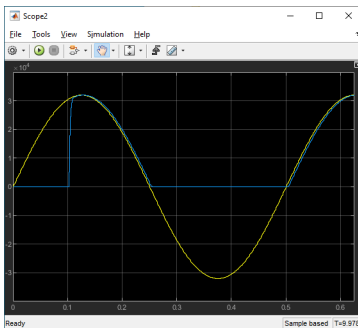
Zooming into the first quarter second of execution for this model, on all three of the scopes shows:



Scope shows the notifications in yellow and the state in blue. The only notifications have the value of 1 which has meaning that the state has changed. Each of those is aligned with a step in the state output. Because this ENI file does not use distributed clock synchronization, the progression to Op state is very fast, just over 0.1 second. Also, because this ENI file does not use distributed clocks, the last 4 elements of the vector out of the init block are all 0.



Scope1 shows the 1Hz sinewave output in yellow and the value read back by the A/D in blue. Notice that there is no input until the EtherCAT state has progressed to Op state just after .1 seconds. If you zoom in tighter, you notice that the A/D signal is delayed by several clock cycles from the D/A output. This is because the A/D is read before the D/A is commanded to a new value and the A/D value is not available until the next sample time. This D/A slave takes a signed int as input, but can only output in the range of [0,+10] volts so the input values only show positive values, even though this A/D can read inputs from [-10,+10].



Scope2 shows the 2Hz sinewave sent to the second D/A channel, with the same delayed start on input and delayed response to a change.

The second way is to build the model (slbuild() or ^B), download from the MATLAB command line and run from the command line. In that case, the scope blocks do not display data, but the Simulation Data Inspector can be used.

The model is preconfigured to run for 10 seconds. If you want to run the model longer, use the MODELING tab on the model editor toolstrip to change the Stop Time and rebuild.

Display the Target Computer data

After running the model, you can also use the Simulation Data Inspector to view any signal that has been marked for signal logging. Signals marked for signal logging have a dot with two arcs above it in the model editor.

Stop and Close the Model

When the example completes its run, stop and close the model.

```
close_system('slrt_ex_ethercat_beckhoff_aio');
```

See Also

- “EtherCAT Protocol with Beckhoff Digital IO Slave Devices EL1004 and EL2004” on page 16-38
- “Modeling EtherCAT Networks”
- “Configure EtherCAT Network by Using TwinCAT 3”

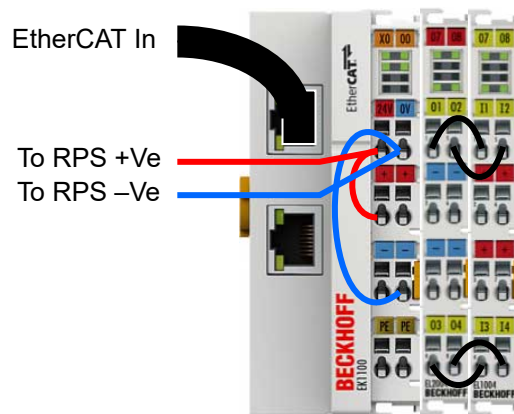
EtherCAT Protocol with Beckhoff Digital IO Slave Devices EL1004 and EL2004

This example shows how to communicate with EtherCAT® devices using the Beckhoff® digital I/O terminals EL1004 and EL2004.

Requirements

To run this example, you need an EtherCAT network that consists of the target computer as EtherCAT Master device and two analog input/output terminals EL1004 and EL2004 as EtherCAT Slave devices attached to an EK1100 coupler.

EtherCAT in Simulink® Real-Time™ requires a dedicated network port on the target computer that is reserved for EtherCAT use by using the Ethernet configuration tool. Configure the dedicated port for EtherCAT communication, not with an IP address. The dedicated port must be distinct from the port used for the Ethernet link between the development and target computers.



To test this model:

- 1 Connect the port that is reserved for EtherCAT in the target computer to the network IN port of the Beckhoff EK1100 coupler.
- 2 Assemble Terminals EL1004 and EL2004 with Coupler EK1100.
- 3 Loop back the first two I/O ports: Connect ports numbered O1 and O2 of Terminal EL2004 to ports numbered I1 and I2 of Terminal EL1004. Ports O3, O4, I3 and I4 are not used by this example.
- 4 Make sure that the terminals are supplied with the required 24-volt power supply.
- 5 Build and download the model onto the target.

For a complete example that configures the EtherCAT network, configures the EtherCAT master node model, and builds then runs the real-time application, see the Simulink Real-Time EtherCAT documentation.

Open the Model

This model drives a pulse wave signal and transmits the signal and its inverse as Boolean values to the EL2004 terminal, and receives the input signal transmitted by the EL1004 terminal.

The EtherCAT initialization block can be configured with either the full path to the ENI file or with a relative path that can be found with the MATLAB which command. Copy the example configuration file from the example folder to the current folder. To open the model, in the MATLAB® Command Window, type:

```
open_system(fullfile(matlabroot, 'toolbox', 'slrealtime', 'examples', 'slrt_ex_ethercat_beckhoff_dio
```

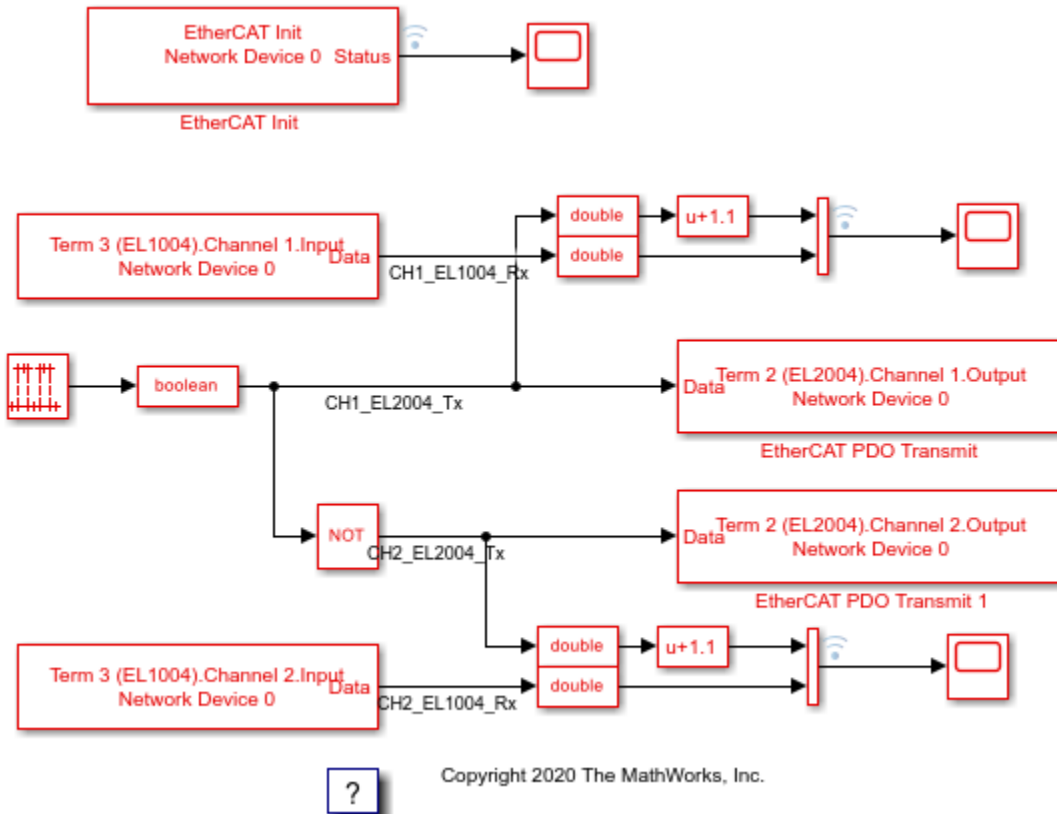


Figure 1: EtherCAT model using Beckhoff digital I/O terminals EL1004 and EL2004.

Configure the Model

Open the parameter dialog for the **EtherCAT Init** block and observe the pre-configured values. The EtherCAT slave devices that are daisy chained together with Ethernet cable is a Device, also referred to as an EtherCAT network. The Device Index selects one such chained EtherCAT network. The Ethernet Port Number identifies which Ethernet port to use to access that Device. The EtherCAT Init block connects these two so that other EtherCAT blocks use the Device Index to communicate with the slave devices on that EtherCAT network.

If you only have one connected network of EtherCAT slaves, and you have only reserved one Ethernet port with the Ethernet configuration tool, use Device Index = 0 and Ethernet Port Number = 1.

Describe Network with Configurator

Using a third-party EtherCAT configuration program that you install on a development computer, generate an EtherCAT configuration (ENI) file. The ENI file for this example is `BeckhoffDIOconfig.xml`.

The ENI (EtherCAT Network Information) file that is provided with this example has an EK1100 with EL2004 and EL1004 slaves attached, in that order. If you have different digital IO modules, you need to create a new ENI file for that collection.

For an overview of the process for creating an ENI file, see “Configure EtherCAT Network by Using TwinCAT 3”.

Each EtherCAT configuration file (ENI file) is specific to the exact network setup from which it was created (for example, the network discovered in step 1 of the configuration file creation process). The configuration file provided for this example is valid if and only if the EtherCAT network consists of Terminals EK1100, EL1004, and EL2004 from Beckhoff.

The ENI file defines a set of transmit and receive variables. For this example, four receive variables are defined for the four input channels of Terminal EL1004. Only the first two channels of Terminal EL1004 are used in this example. Make sure the receive variables for channel 1 and channel 2 of terminal EL1004 are selected respectively in the two **EtherCAT PDO Receive** blocks. These two variables are Term 3 (EL1004).Channel 1.Input and Term 3 (EL1004).Channel 2.Input. In the same way, four transmit variables are defined for the four output channels of terminal EL2004, but only the first two channels are tested in this example. Make sure the transmit variables for channel 1 and channel 2 of terminal EL2004 are selected respectively in the two **EtherCAT PDO Transmit** blocks. These two variables are Term 2 (EL2004).Channel 1.Output and Term 2 (EL2004).Channel 2.Output.

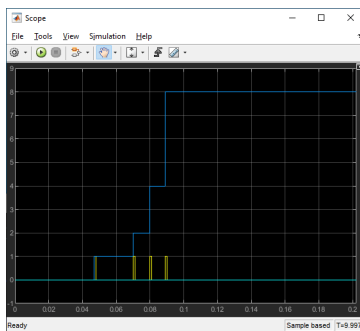
Build, Download, and Run the Model

To build, download, and run the model:

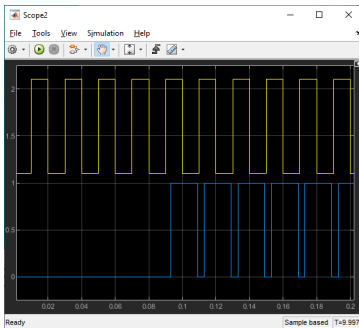
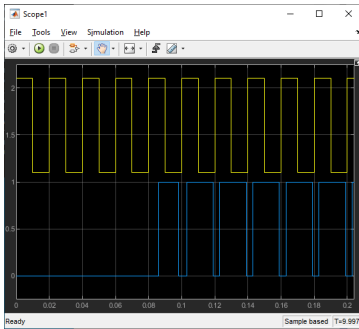
- 1 In the **Simulink Editor**, from the targets list on the **Real-Time** tab, select the target computer on which to run the real-time application.
- 2 Click **Run on Target**.

If you open the three host side scopes by double clicking each, data is relayed from the target back to the development computer and displayed.

The three scopes are Scope, Scope1 and Scope2.



Both notifications from the EtherCAT state machine and the current state are displayed in Scope. Since there are no errors, the only notifications visible are the value 1 which means a state change at that execution time step. The current state indicates the state that resulted from that state change. Notice that Op (=8) state is reached very fast since this ENI file does not include distributed clock synchronization. This view is zoomed in to the first 0.2 seconds of execution to show the transition to Op state clearly.



Scope1 and Scope2 show almost the same thing, but for two different channels. The signal is inverted between the two of them as can be seen if you compare the time when there is a rising edge in the yellow trace. The time step when physical IO starts is when the state goes to Op state. Before that, there is no input or output and the blue traces stay at 0. There is a time delay between the signal being sent to the output blocks and the signal that comes back from the input blocks for two reasons.

There is a 2 time step delay due to EtherCAT communication which is followed by an additional delay due to the speed of the hardware IO. The return signal shows a definite asymmetry between the delay after sending a rising edge and the delay after sending a falling edge. If you inspect the actual output signal with an oscilloscope, you see that the output is actually symmetric, but it is the input that has additional hardware delay in it. Other DIO slaves show different delay characteristics.

The model is preconfigured to run for 10 seconds. If you want to run the model longer, pull down the **Run on Target** menu and change the number on the bottom line. Press the green arrow to configure, build, and run.

Display the Target Computer Data

After running the model, you can use the Simulation Data Inspector to view any signal that has been marked for signal logging. Signals marked for signal logging have a dot with two arcs above it in the model editor.

Observations to Notice

Because data is both received from and sent to the slaves as the final action during execution and received data on one time step is only available during the following time step, you should see a delay between the data being sent and the return value. In addition with digital IO, writing a new value to an output takes a few microseconds to appear as a change in voltage which is after the input was captured, there is a 2 time step delay from an output edge until the input shows the edge in the data.

Close the Model

When the example completes its run, stop and close the model.

```
close_system('slrt_ex_ethercat_beckhoff_dio');
```

See Also

- “EtherCAT Protocol with Beckhoff Analog IO Slave Devices EL3062 and EL4002” on page 16-33
- “Modeling EtherCAT Networks”
- “Configure EtherCAT Network by Using TwinCAT 3”

EtherCAT Protocol Motor Velocity Control with Accelnet Drive

This example shows how to control the velocity of a motor by using EtherCAT® communication. The example motor drive is from Copley Instruments. This drive uses the CIA-402 (Can In Automation 402) device profile common to many drives. The example can work with other CIA-402 EtherCAT drives if you generate an appropriate ENI file.

Requirements

This example is preconfigured to use an EtherCAT network that consists of the target computer as EtherCAT Master device and an Accelnet™ AEP 180-18 drive from Copley Controls as EtherCAT Slave device. Connect a supported brushless or brush motor to the drive. An example motor that works with this example is the SM231BE-NFLN from PARKER.

EtherCAT in Simulink® Real-Time™ requires a dedicated network port on the target computer that is reserved for EtherCAT use by using the Ethernet configuration tool. Configure the dedicated port for EtherCAT communication, not with an IP address. The dedicated port must be distinct from the port used for the Ethernet link between the development and target computers.

To test this model:

- 1 Connect the dedicated network port in the target computer to the EtherCAT IN port of the Accelnet drive.
- 2 Connect a motor to the Accelnet drive.
- 3 Make sure that the Accelnet drive is supplied with a 24-volt power supply.
- 4 Build and download the model onto the target.

For a complete example that configures the EtherCAT network, configures the EtherCAT master node model, and builds then runs the real-time application, see “EtherCAT Protocol Sequenced Writing CoE Slave Configuration Variables” on page 16-69.

Open the Model

This model sends a varying velocity command to the drive.

The EtherCAT initialization block requires that the configuration ENI file is present in the current folder. Copy the example configuration file from the example folder to the current folder. To open the model, in the MATLAB® Command Window, type:

```
open_system(fullfile(matlabroot, 'toolbox', 'slrealtime', 'examples', 'slrt_ex_ethercatVelocityControl'))
```

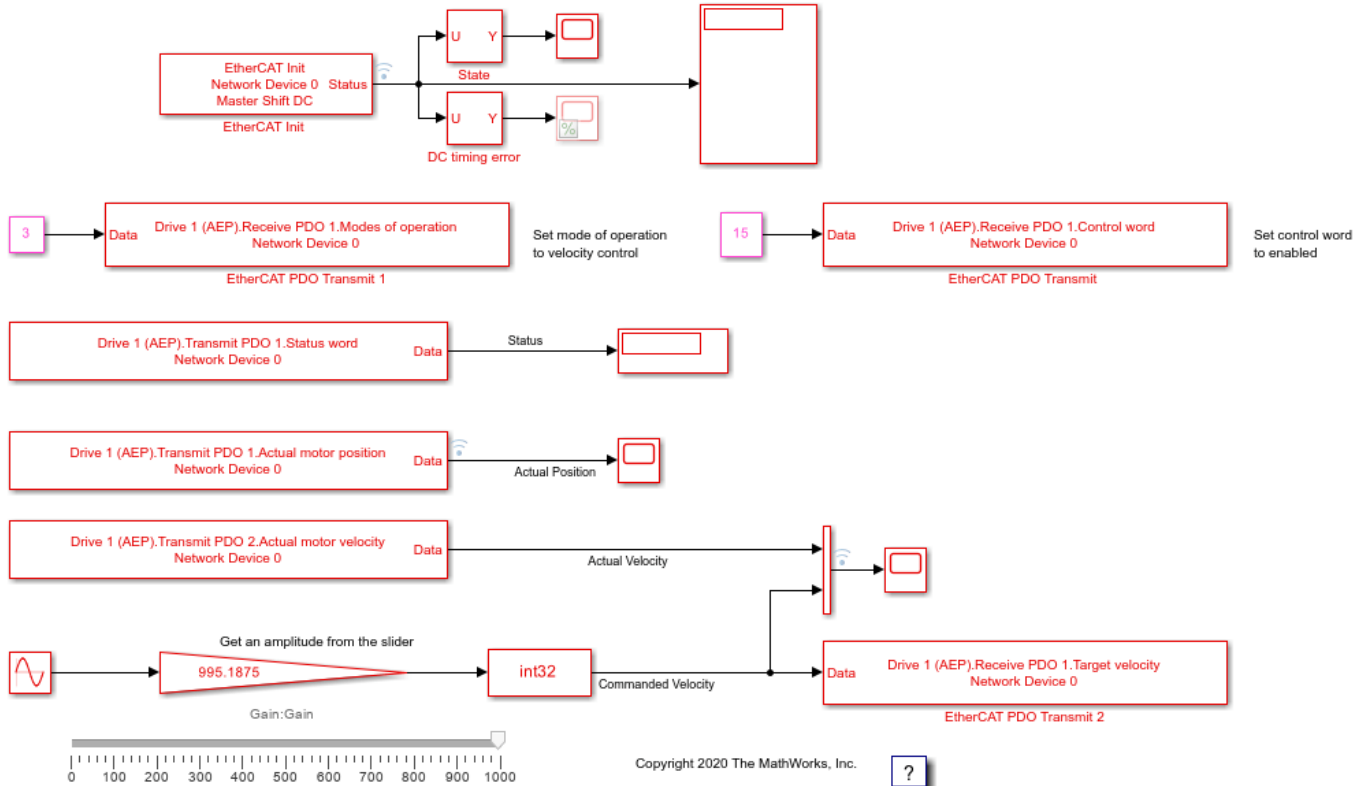


Figure 1: EtherCAT model for motor velocity control.

Configure the Model

Open the parameter dialog for the **EtherCAT Init** block and observe the pre-configured values. The EtherCAT slave devices that are daisy chained together with Ethernet cable is a Device, also referred to as an EtherCAT network. The Device Index selects one such chained EtherCAT network. The Ethernet Port Number identifies which Ethernet port to use to access that Device. The EtherCAT Init block connects these two so that other EtherCAT blocks use the Device Index to communicate with the slave devices on that EtherCAT network.

If you only have one connected network of EtherCAT slaves, and you have only reserved one Ethernet port with the Ethernet configuration tool, use Device Index = 0 and Ethernet Port Number = 1.

Create an ENI file for a Different CIA-402 Drive

If you need to create a new ENI file, you need to use a third-party EtherCAT configurator such as TwinCAT 3 from Beckhoff that you install on a development computer. The EtherCAT configuration (ENI) file preconfigured for this model is `CopleyMotorVelocityConfig.xml`.

Each EtherCAT configuration file (ENI file) is specific to the exact network setup from which it was created (for example, the network discovered in step 1 of the configuration file creation process). The configuration file provided for this example is valid if and only if the EtherCAT network consists of one Accelnet drive from Copley Controls. If you have a different EtherCAT drive that uses the CIA-402 command set, this example still works, but you need to create a new ENI file that uses your drive.

For an overview of the process for creating an ENI file, see “Configure EtherCAT Network by Using TwinCAT 3”.

For this example, four receive PDO variables are defined in the configuration file and three are used in the three **EtherCAT PDO Transmit** blocks: Control Word, Modes of Operation, and Target Velocity. The fourth variable: Profile Target Position is used in example “EtherCAT Protocol Motor Position Control with Accelnet Drive” on page 16-48.

- The Control Word PDO variable serves to control the state of the drive. The constant value 15 is given as input to the block to set the first 4 bits to 1 to enable the drive. Refer to the EtherCAT User Guide from Copley Controls for details on the bits mapping of this variable. This variable and bit mapping is in the CIA-402 standard set.
- The Modes of Operation PDO variable serves to set the drive operating mode. The constant value 3 is given as input to the block to set the mode of the drive to **Profile Velocity mode**. For details on supported modes of operation, see the Refer to the Copley Controls EtherCAT User Guide. This variable and bit mapping is in the CIA-402 standard set.
- The Target Velocity PDO variable serves to set the desired velocity. In this example, the velocity command at the input of the block can be tuned using the slider connected to the gain block parameter.

Three transmit PDO variables are also defined in the configuration file and used in the three **EtherCAT PDO Receive** blocks: Status Word, Actual Motor Velocity, and Actual Motor Position. Note that EtherCAT refers to variables that the slave sets as transmit variables which are received by the target model.

- The Status Word PDO variable indicates the current state of the drive.
- The Actual Motor Velocity and Actual Motor Position PDO variables indicate the current values of the motor velocity and position as read in the drive.

Make sure that the required transmit and receive PDO variables are selected in the blocks as illustrated in Figure 1 before running the example. You could need to refresh these variables by opening the dialogs and selecting the current variable again.

Build, Download, and Run the Model

To build, download, and run the model:

- 1** In the **Simulink Editor**, from the targets list on the **Real-Time** tab, select the target computer on which to run the real-time application.
- 2** Click **Run on Target**.

If you open the host side scopes by double clicking each, data is relayed from the target back to the development computer and displayed.

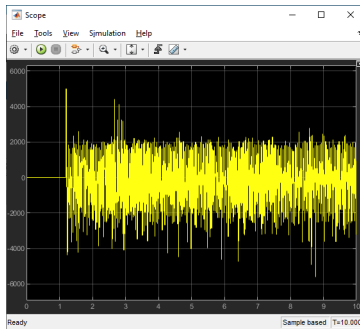
Included in the model is the ability to control the peak amplitude of the velocity. With the Run on Target button, the slider is active and connected to the Amplitude constant block.

The model is preconfigured to run for 10 seconds. If you want to run the model longer, pull down the **Run on Target** menu and change the number on the bottom line. Press the green arrow to configure, build, and run.

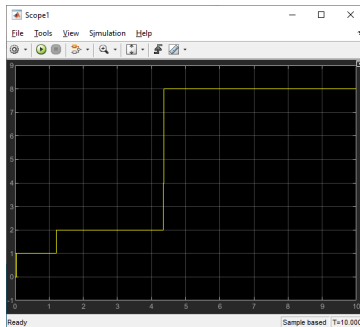
Display the Target Computer Scopes

If you run the model using the **Run on Target** button, external mode is connected and you can double click the scope blocks and see the data on the host. Also, the slider is active in external mode.

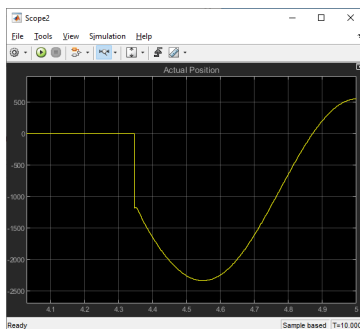
The Scope output images are referred to by the name in the title bar for each image. Discussion follows each image.



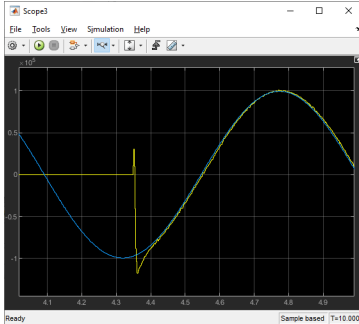
Scope shows the target to slave timing error as synchronization takes place using the bus shift method. The slave is adjusted to match the target timing resulting in a damped wave showing good phase lock around 4.5 to 5 seconds. The hash is a manifestation of the QNX® execution scheduler and is what is expected. On this graph, 5000 is in nanoseconds, so this shows synchronization between 0 and -2 microseconds with residual random errors.



Scope1 shows the progression of states as the drive is initialized. Most of the time is taken to achieve time synchronization between target and EtherCAT slaves. The SafeOp (=4) to Op (=8) state transition occurs after a short settling time once the timing error is below the allowed error.



Scope2 shows the position of the motor which is a phase shifted version of the sine wave velocity that is sent to the motor. Note that the motor position does not change until the drive goes to Op state around 4.3 seconds.



Scope3 shows the velocity that is sent to the drive and the velocity read back from the drive. The velocity does not change until the drive goes into Op state.

After running the model, you can also use the Simulation Data Inspector to view any signal that has been marked for signal logging. Signals marked for signal logging have a dot with two arcs above it in the model editor.

Observations to Notice

The velocity command for the motor is a low frequency sine wave. The actual velocity read back from the controller is delayed by several sample times and the actual position is out of phase by 90 degrees from the actual velocity, as expected for sinewave variation.

Stop and Close the Model

When the example completes its run, stop and close the model.

```
close_system('slrt_ex_ethercatVelocityControl');
```

See Also

- “EtherCAT Protocol Motor Position Control with Accelnet Drive” on page 16-48
- “Modeling EtherCAT Networks”
- “Configure EtherCAT Network by Using TwinCAT 3”

EtherCAT Protocol Motor Position Control with Accelnet Drive

This example shows how to control the position of a motor by using EtherCAT® communication. The example motor drive is from Copley Instruments. This drive uses the CIA-402 (Can In Automation 402) device profile common to many drives. The example can work with other CIA-402 EtherCAT drives if you generate an appropriate ENI file.

Requirements

This example is preconfigured to use an EtherCAT network that consists of the target computer as EtherCAT Master device and an Accelnet™ AEP 180-18 drive from Copley Controls as EtherCAT Slave device. Connect a supported brushless or brush motor to the drive. An example motor that works with this example is the SM231BE-NFLN from PARKER.

EtherCAT in Simulink® Real-Time™ requires a dedicated network port on the target computer that is reserved for EtherCAT use by using the Ethernet configuration tool. Configure the dedicated port for EtherCAT communication, not with an IP address. The dedicated port must be distinct from the port used for the Ethernet link between the development and target computers.

To test this model:

- 1 Connect the port that is reserved for EtherCAT in the target computer to the EtherCAT IN port of the Accelnet drive.
- 2 Connect a motor to the Accelnet Drive.
- 3 Make sure the Accelnet drive is supplied with a 24-volt power source.
- 4 Build and download the model onto the target.

For a complete example that configures the EtherCAT network, configures the EtherCAT master node model, and builds then runs the real-time application, see “Modeling EtherCAT Networks”.

Open the Model

This model creates a sine wave, and modulates it by multiplying by the value of the slider control. The modulated signal is sent as motor position command to the drive.

The EtherCAT initialization block requires that the configuration ENI file is present in the current folder. Copy the example configuration file from the example folder to the current folder. To open the model, in the MATLAB® Command Window, type:

```
open_system(fullfile(matlabroot, 'toolbox', 'slrealtime', 'examples', 'slrt_ex_ethercatPositionContr
```

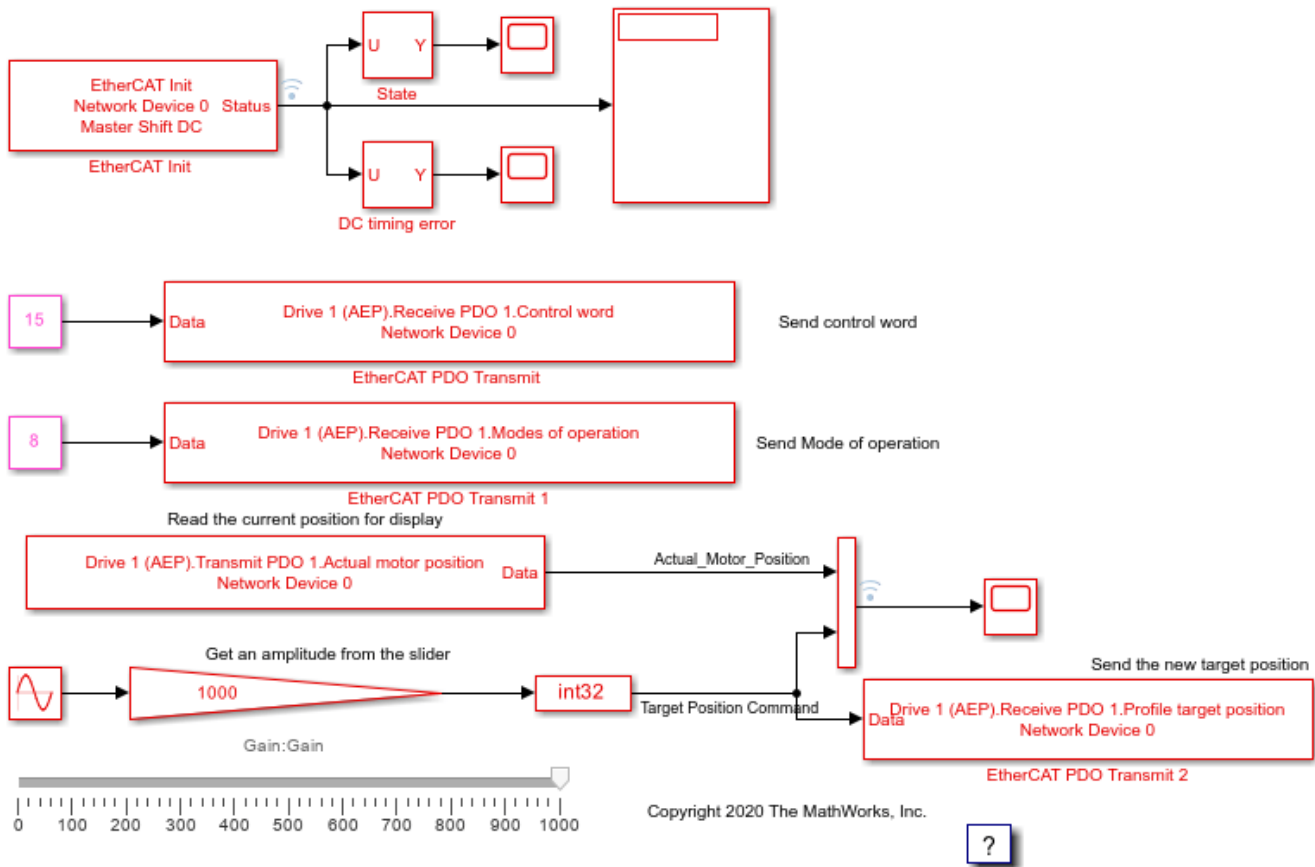


Figure 1: EtherCAT model for controlling the position of a motor.

Configure the Model

Open the parameter dialog for the **EtherCAT Init** block and observe the pre-configured values. The EtherCAT slave devices that are daisy chained together with Ethernet cable is a Device, also referred to as an EtherCAT network. The Device Index selects one such chained EtherCAT network. The Ethernet Port Number identifies which Ethernet port to use to access that Device. The EtherCAT Init block connects these two so that other EtherCAT blocks use the Device Index to communicate with the slave devices on that EtherCAT network.

If you only have one connected network of EtherCAT slaves, and you have only reserved one Ethernet port with the Ethernet configuration tool, use Device Index = 0 and Ethernet Port Number = 1.

Create an ENI File for a Different CIA-402 Drive

If you need to create a new ENI file, you need to use a third-party EtherCAT configurator such as TwinCAT 3 from Beckhoff® that you install on a development computer. The EtherCAT configuration (ENI) file preconfigured for this model is `CopleyMotorPositionConfig.xml`.

Each ENI file is specific to the exact network setup from which it was created (for example, the network discovered in step 1 of the configuration file creation process). The configuration file provided for this example is valid if and only if the EtherCAT network consists of one Accelnet drive from Copley Controls. If you have a different EtherCAT drive that uses the CIA-402 CanOpen profile,

this example still works, but you need to create a new ENI file that uses your drive. Refer to Can In Automation web site at www.can-cia.org for details. EtherCAT CoE embeds CanOpen addressing for process variables using EtherCAT as the transport layer instead of CAN.

An overview of the process for creating an ENI file is at “Configure EtherCAT Network by Using TwinCAT 3”

For this example, four receive PDO variables are defined in the configuration file and three are used in the three **EtherCAT PDO Transmit** blocks: Control Word, Modes of Operation, and Profile Target Position. The fourth variable: Target Velocity is used in example “EtherCAT Protocol Motor Velocity Control with Accelnet Drive” on page 16-43.

- The Control Word PDO variable serves to control the state of the drive. The constant value 15 is given as input to the block to set the first 4 bits to 1 to enable the drive. For details on the bit mapping of this variable, refer to the Can In Automation web site. This variable and bit mapping is in the CIA-402 device profile.
- The Modes of Operation PDO variable serves to set the operating mode of the drive. The constant value 8 is given as input to the block to set the mode of the drive to **Cyclic Synchronous Position** mode. For detailed documentation, refer to the Can In Automation web site. This variable is in the CIA-402 device profile.
- The Profile Target Position PDO variable serves to set the desired position. In this example, the position command given as input to the block is a sine wave modulated by the constant Amplitude value linked to the slider control in the model.

Transmit PDO variables (transmitted by the slave) are also defined in the configuration file and one is used in the **EtherCAT PDO Receive** block: **Actual Motor Position** for the drive. The Actual Motor Position PDO variable indicates the current value of the motor position as read in the drive. Make sure the required transmit and receive PDO variables are selected in the blocks before running the example. You could need to refresh these variables. Note that EtherCAT refers to variables that the slave sets as transmit variables which are received by the target model.

Make sure that the required transmit and receive PDO variables are selected in the blocks as illustrated in Figure 1 before running the example. You could need to refresh these variables by opening the dialogs and selecting the current variable again.

Build, Download, and Run the Model

To build, download, and run the model:

- 1 In the **Simulink Editor**, from the targets list on the **Real-Time** tab, select the target computer on which to run the real-time application.
- 2 Click **Run on Target**.

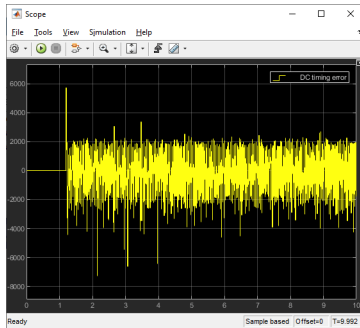
If you open the two host side scopes by double clicking each, data is relayed from the target back to the development computer and displayed.

Included in the model is the ability to control the amplitude of the cycling motion. With the Run on Target button, the slider is active and connected to the Amplitude constant block.

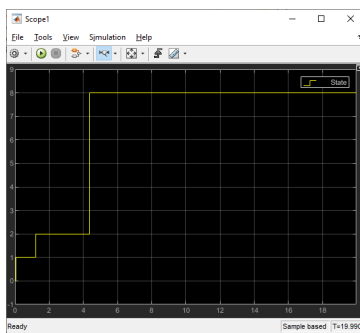
The model is preconfigured to run for 10 seconds. If you want to run the model longer, pull down the **Run on Target** menu and change the number on the bottom line. Press the green arrow to configure, build and run.

Display the Target Computer data

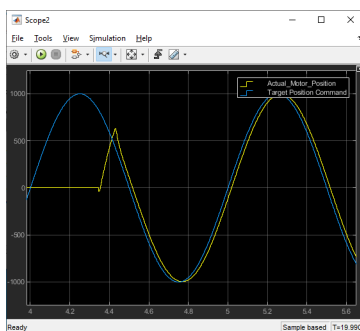
If you run the model using the **Run on Target** button, external mode is connected and you can double click the scope blocks and see the data on the host. Also, the slider is active in external mode.



Scope shows the Distributed Clocks timing difference between the master stack running on the target computer and the timing on the drive. This ENI file is configured to use Master Shift mode for DC. The clock on the target computer is adjusted to match the timing on the EtherCAT reference clock on the first DC enabled slave.



Scope1 shows the state progression from Idle to Init (=1) to PreOp (=2) to SafeOp (=4) for a very short time visible if you zoom in, to Op (=8) at around 4.3 seconds.



Scope2 shows both the sine wave being sent to the drive (blue) and the actual position (yellow). This is zoomed into the few seconds right when the drive went to Op state and external control starts. Since the motor hardware cannot respond instantaneously, and the commanded position is not 0, you see the actual position ramp up and overshoot slightly before settling down to follow the commanded position. The time delay between command and actual is roughly 18 sample time steps with this

drive. The controller inside the drive and motor inertia are responsible for this longer time delay. Other drives may have different delay characteristics.

After running the model, you can use the Simulation Data Inspector to view any signal that has been marked for signal logging. Signals marked for signal logging have a dot with two arcs above it in the model editor.

Observations to Notice

This is a simple motor control example. The numerous tunable parameters inside the drive are not adjusted in this model. Adjusting those needs a more advanced model using the CoE/SDO blocks.

Close the Model

When the example completes its run, stop and close the model.

```
close_system('slrt_ex_ethercatPositionControl');
```

See Also

- “EtherCAT Protocol Motor Velocity Control with Accelnet Drive” on page 16-43
- “Modeling EtherCAT Networks”
- “Configure EtherCAT Network by Using TwinCAT 3”

Generate ENI Files for EtherCAT Devices

This example shows how to generate EtherCAT® network information (ENI) files to use in Simulink® Real-Time™ with EtherCAT devices.

The example shows the generation process steps in EtherCAT Configurator and the process steps in the TwinCAT® XAE plugin for Microsoft® Visual Studio®.

The hardware connections are:

- EK1100 -- EtherCAT coupler
- EL3062 -- EtherCAT terminal
- EL4002 -- EtherCAT terminal
- EL9011 -- Bus End terminal

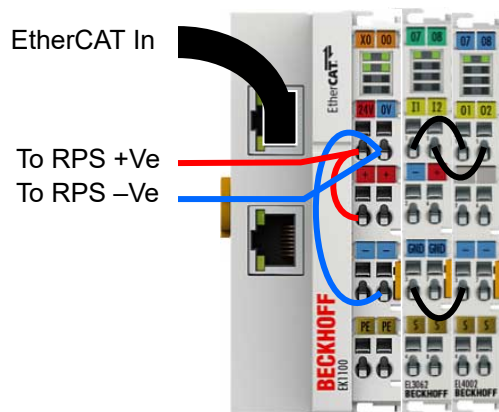
The EK1100 coupler connects EtherCAT with the EtherCAT terminals (ELxxxx). One station consists of an EK1100 coupler, any number of EtherCAT terminals, and a bus end terminal.

To provide power connections, connect the 24 V and 0 V terminals of the EK1100 to a 24 V regulated power supply (RPS) +Ve and -Ve terminals.

The EL3062 analog input terminal processes signals in the range of [-10, 10] V.

The EL4002 analog output terminal generates signals in the range of [0, 10] V.

To configure the EtherCAT network, connect the EtherCAT devices to the development computer on which the EtherCAT configurator is running. This connection permits scanning and discovery of the EtherCAT devices. After the configurator generates the XML file, you can reconnect the EtherCAT devices to the target computer. This diagram shows the suggested connections.



Install TwinCAT 3.1 XAE and Run Microsoft Visual Studio with TwinCAT

The latest version of TwinCAT is the 3.x version and that is the preferred configuration tool.

The XAE sub version does not contain the full run time engine that runs on Windows®. This is available free of charge from the Beckhoff® web site. For use with Simulink Real-Time, you do not

need the run time engine because you are using the run time implementation on the target. The full version with run time engine requires the purchase of a license from Beckhoff.

The TwinCAT 3.1 software requires a supported version of Microsoft Visual Studio to be installed. TwinCAT 3.1 uses the MSVC GUI integration and does not have a GUI by itself. The versions of MSVC with which a given version of TwinCAT works are discussed in the TwinCAT documentation. Installation finds supported MSVC versions on your machine and installs to them.

To install the TwinCAT 3.1 XAE:

- 1 Go to www.beckhoff.com and select **Download**.
- 2 Select TwinCAT 3 and download the setup.
- 3 Install TwinCAT 3.
- 4 Start Microsoft Visual Studio.
- 5 From the **TwinCAT** menu, select **Show Realtime Ethernet Compatible Devices**.
- 6 Select the Ethernet adapter for your EtherCAT device, then select **Install**.

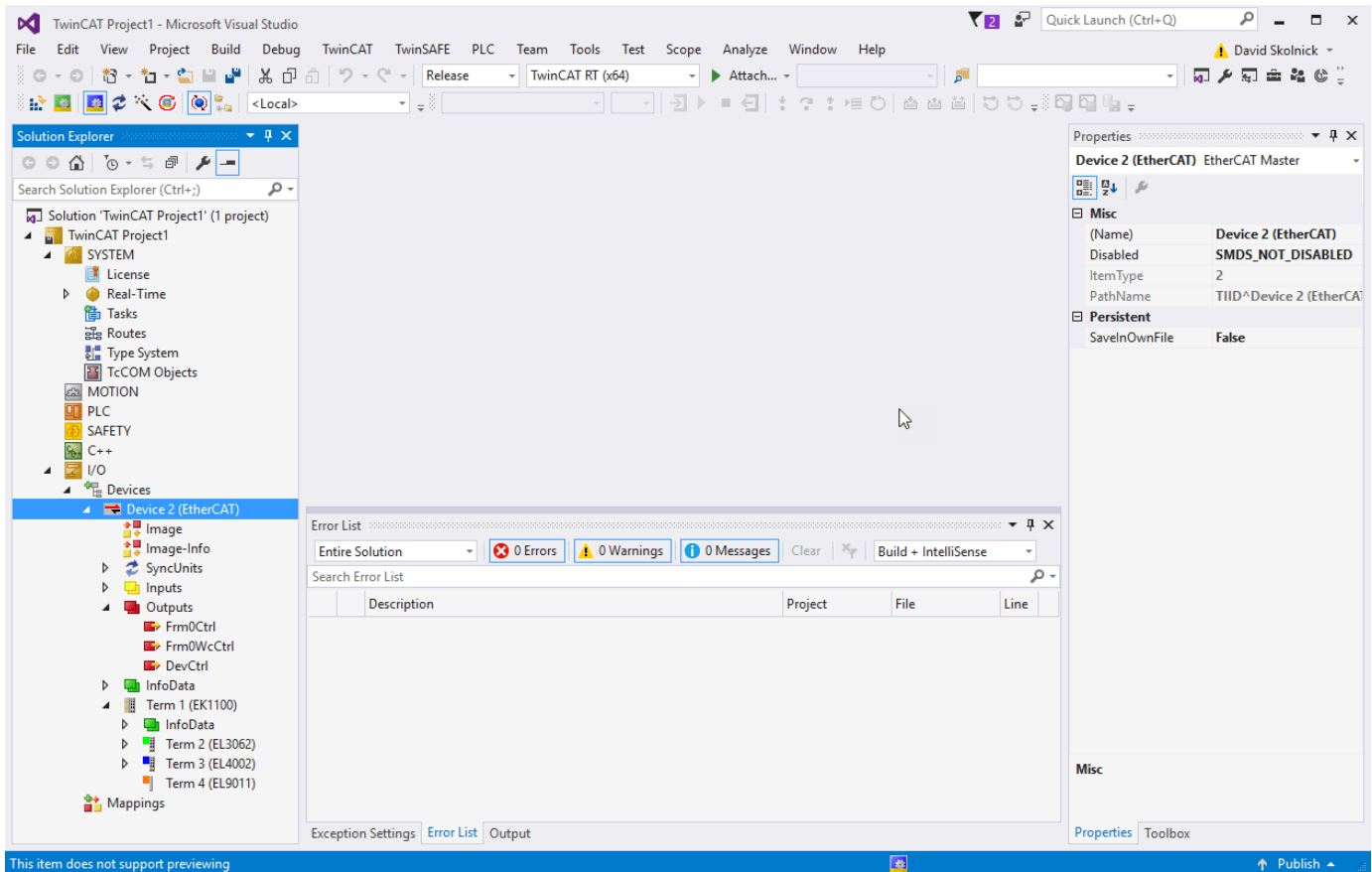
Because TwinCAT installs an Ethernet filter inline with the Ethernet port you have selected, it is good practice to add an extra Ethernet port to use exclusively with EtherCAT to avoid any possible problem the filter can cause when sharing the Simulink Real-Time host-target communication port with TwinCAT.

All EtherCAT configuration programs use EtherCAT Slave Information (ESI) files to describe the slaves that are found on the network. These Beckhoff configuration programs come prepopulated with mostly Beckhoff devices. To correctly configure an EtherCAT network with devices from other manufacturers, you may need to get the correct ESI file from the device manufacturer web site. If you do not have an ESI file for a slave on your network, the scan process does not populate the Solution Explorer with the correct name of the device and the read and write variables are not correct.

To create a new TwinCAT project in Visual Studio:

- 1 Start Visual Studio. Go to **File > New > Project**.
- 2 Under **Installed**, select **TwinCAT Projects** and click **OK**.
- 3 Verify whether the project has been created successfully in the status bar of Microsoft Visual Studio.
- 4 Enter your license if this instance is the first time that you are using TwinCAT and you installed the full version. If you are using TwinCAT in evaluation mode, fill in the Captcha.
- 5 Observe the **Solution Explorer** pane the left side of Visual Studio.
- 6 Go to **TWINCAT** in the menu and select **Scan**. You can also right click **Solution Explorer > your TwinCAT project > I/O > Devices > Scan**.
- 7 A dialog box opens with the message **All devices may not automatically be found**. Click **OK** and wait for the scan to complete. You now see a dialog box saying **New I/O devices have been found**.
- 8 Ensure that the check box is selected, then click **OK**. A dialog box appears with a **Scan for boxes?** message. Click **Yes**. The EtherCAT devices in your network are scanned, and the devices appear.
- 9 You see a dialog box that asks whether to activate free run mode. Select **No**.
- 10 Observe the Solution Explorer and verify that the devices were scanned correctly.

When you first start TwinCAT the right information panel is not displayed. You need to double-click any item in the tree view the first time. After that the information dialog for any item is displayed by a single click on that item in the **Solution Explorer** tree view.



Configure EtherCAT Master Node Data with TwinCAT

To configure the EtherCAT master node, create and configure a task, then add the inputs and outputs to the task.

To create an EtherCAT Task:

- 1 In the **Solution Explorer**, right-click the **Tasks** node and select **Add New Item**.
- 2 In the **Insert Task** dialog box, select **TwinCAT Task With Image**, provide a name for the task, and click **OK**.
- 3 Select the task that you created. The value **Cycle Ticks** determines the cycle time as a multiple of the **Base Time** determined on the **Real-Time** item. The default task time is set to 10 ms. If you are using Distributed Clock synchronization, a task time of 1-2 ms is the slowest that works with Master Shift DC mode.
- 4 Create at least one cyclic input/output task. Link this task to at least one input variable and one output variable on each slave device.
- 5 If you want to run faster than 1ms time, you need to change the base time on the **Real-Time** item above **Tasks**. On the **Settings** tab, you need to change the **Base Time** selection to a faster one.

By using distributed clocks (DC), the EtherCAT protocol can synchronize the time in all local bus devices within a narrow tolerance range. Only some EtherCAT devices support DC. It is important that if a device supports DC, you configure it accordingly. For example, in the example configuration, the EL4002 supports DC. Most motion controllers (motor drives) support DC and some require it to get to Op state.

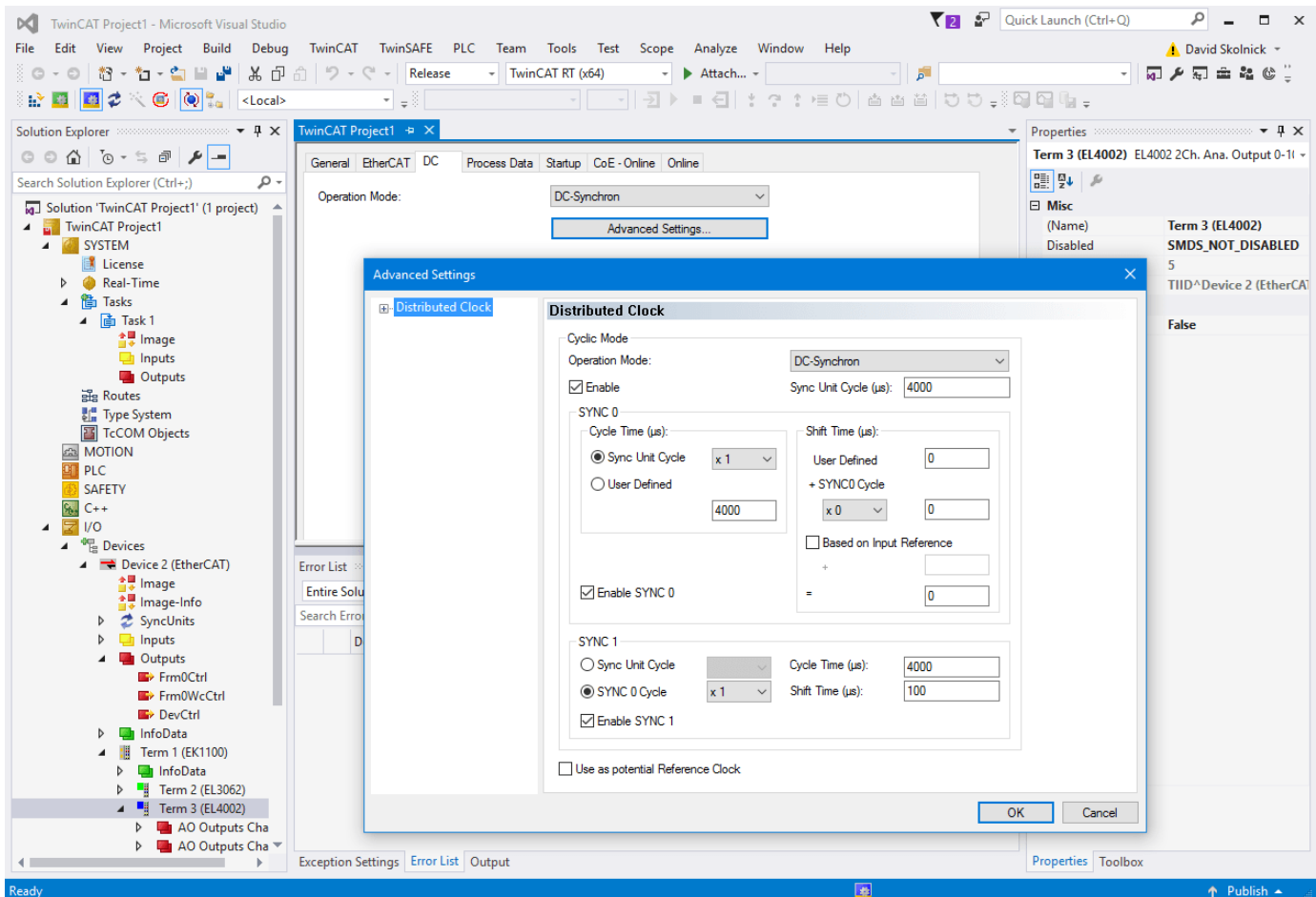
To configure EtherCAT DC:

Enable DC and choose **Bus shift** or **Master shift** DC mode.

- 1 Click on the **Device n (EtherCAT)** node
- 2 Select the **EtherCAT** tab in the information panel.
- 3 Click on **Advanced Settings** which opens a new dialog.
- 4 In the **Advanced Settings** dialog, select the **Distributed Clocks** page.
- 5 By default the **Automatic DC Mode Selection** box is checked, which generally gets you Master Shift DC mode. The first DC enabled slave is the reference clock and the Simulink Real-Time execution time is shifted slightly to align execution with the reference clock.
- 6 Deselect the **Automatic** mode and you have control over which DC mode to use or to turn it off. Next items are with **Automatic** deselected.
- 7 With **DC in use** deselected, no Distributed clock synchronization takes place. This results in much quicker initialization time to get to Op state, but there is no synchronization between slaves.
- 8 With **DC in use** selected, you have two different synchronization methods between the Speedgoat® target machine and the EtherCAT slaves.
- 9 **Independent DC Time** uses the first DC enabled slave as the reference clock and the target machine clock is adjusted slightly to phase lock model execution to the first DC enabled slave.
- 10 **DC Time controlled by TwinCAT Time** should be read as **controlled by target machine time** in Simulink Real-Time. This is bus shift mode where the target machine is the reference clock and the slave execution times are shifted slightly to phase lock to the target machine.
- 11 Select both **Continuous Run-Time Measuring** and **Sync Window Monitoring**.

Now that you have chosen the DC mode to use, you can visit all of the DC enabled slaves in your network and set them to the correct mode. This example ENI file only supports one DC enabled slave, the EL4002.

- 1 Click the node **Term 3 (EL4002)** and select the **DC** tab.
- 2 By default, the **Operation Mode** is set to **SM-Synchron** which does not synchronize output to DC time. Change the **Operation Mode** to **DC-Synchron**. Different slaves have different names for the operating mode.
- 3 Click **Advanced Settings** and set the **Distributed Clock** options as shown.



To export and save the EtherCAT configuration, generate the ENI file:

- 1 Click the node for your EtherCAT device and click the **EtherCAT** tab.
- 2 Click **Export Configuration File**.
- 3 In the **Save As** dialog box, enter an XML file name, such as `simple_adda_eni.xml`, then click **Save**. This XML file is the ENI file. The ENI file and the Simulink Real-Time model that uses the ENI file cannot have the same name. They must have different names.
- 4 When you close the TwinCAT project, the editable version of this configuration is saved in the project file. You can modify the configuration by opening this project and by exporting to XML again.

Import a Device with the Configurator

Device import is often part of the workflow for third-party (different manufacturer) devices. Use this process to configure a device that is not present in the Beckhoff system. Numerous motors and their drives fall under this category. Sometimes, you must configure a device that is not present in the Beckhoff system. The TwinCAT EtherCAT master or System Manager uses the device description files for the devices to generate the configuration in online or offline mode.

The device descriptions are contained in ESI files (EtherCAT Slave Information) in XML format. These files can be requested from the respective manufacturer and are made available for download. An XML file can contain several device descriptions.

The ESI files for Beckhoff EtherCAT devices are available on the Beckhoff website and are stored in the TwinCAT installation folder. The default for TwinCAT2 is C:\TwinCAT\IO\EtherCAT. The files are read (once) when you open a new System Manager window and if they have changed since the last time that you opened the System Manager window.

If using a TwinCAT configurator, the TwinCAT installation includes the set of Beckhoff ESI files which were current at the time when the TwinCAT build was created. For TwinCAT 2.11, TwinCAT 3, and later, you can update the ESI folder from the System Manager if the programming PC is connected to the Internet (**Option > Update EtherCAT Device Descriptions**).

See Also

- “Modeling EtherCAT Networks”
- “Configure EtherCAT Network by Using TwinCAT 3”
- “EtherCAT Configurator Component Mapping”

EtherCAT Protocol Detect Network Failure and Reset

This example shows how to use the EtherCAT® Notifications block to detect a failure in the connected network and to restart the network when the failure is corrected.

Only a disconnected Ethernet cable into the first slave is detected by this example. More complicated failure situations can be detected if you study the pattern of notifications that result and write the embedded MATLAB® block to account for those.

Requirements

To run this example as presented, you need a Beckhoff® EK1100 with EL1202, EL2202-0100, EL3102 and EL4032 slave modules. The model does not write to any process objects. Replacing the ENI file with one appropriate to your network works as well.

EtherCAT in Simulink® Real-Time™ requires a dedicated network port on the target computer that is reserved for EtherCAT use by using the Ethernet configuration tool. Configure the dedicated port for EtherCAT communication, not with an IP address. The dedicated port must be distinct from the port used for the Ethernet link between the development and target computers.

To test this model:

- 1 Connect the port that is reserved for EtherCAT in the target computer to the EtherCAT IN port of the EK1100 interface module.
- 2 Make sure the EK1100 is supplied with a 24-volt power source.
- 3 Build and download the model onto the target.

For a complete example that configures the EtherCAT network, configures the EtherCAT master node model, and builds then runs the real-time application, see “Modeling EtherCAT Networks”.

Open the Model

This model is a beginning of a full implementation to catch network failures and reinitialize the network once the failure is fixed. The simple state machine in the embedded MATLAB block can be replaced with a State Flow implementation, which may be necessary for more complicated failure detection and recovery.

The EtherCAT initialization block requires that the configuration ENI file is present in the current folder or on the MATLAB path because the file name is present without directory information.

If you want to modify this model to experiment with it, copy the example configuration file and the model file from the example folder to the current folder. To open the model, in the MATLAB Command Window, type:

```
open_system(fullfile(matlabroot, 'toolbox', 'slrealtime', 'examples', 'slrt_ex_ethercat_notifyreset')
```

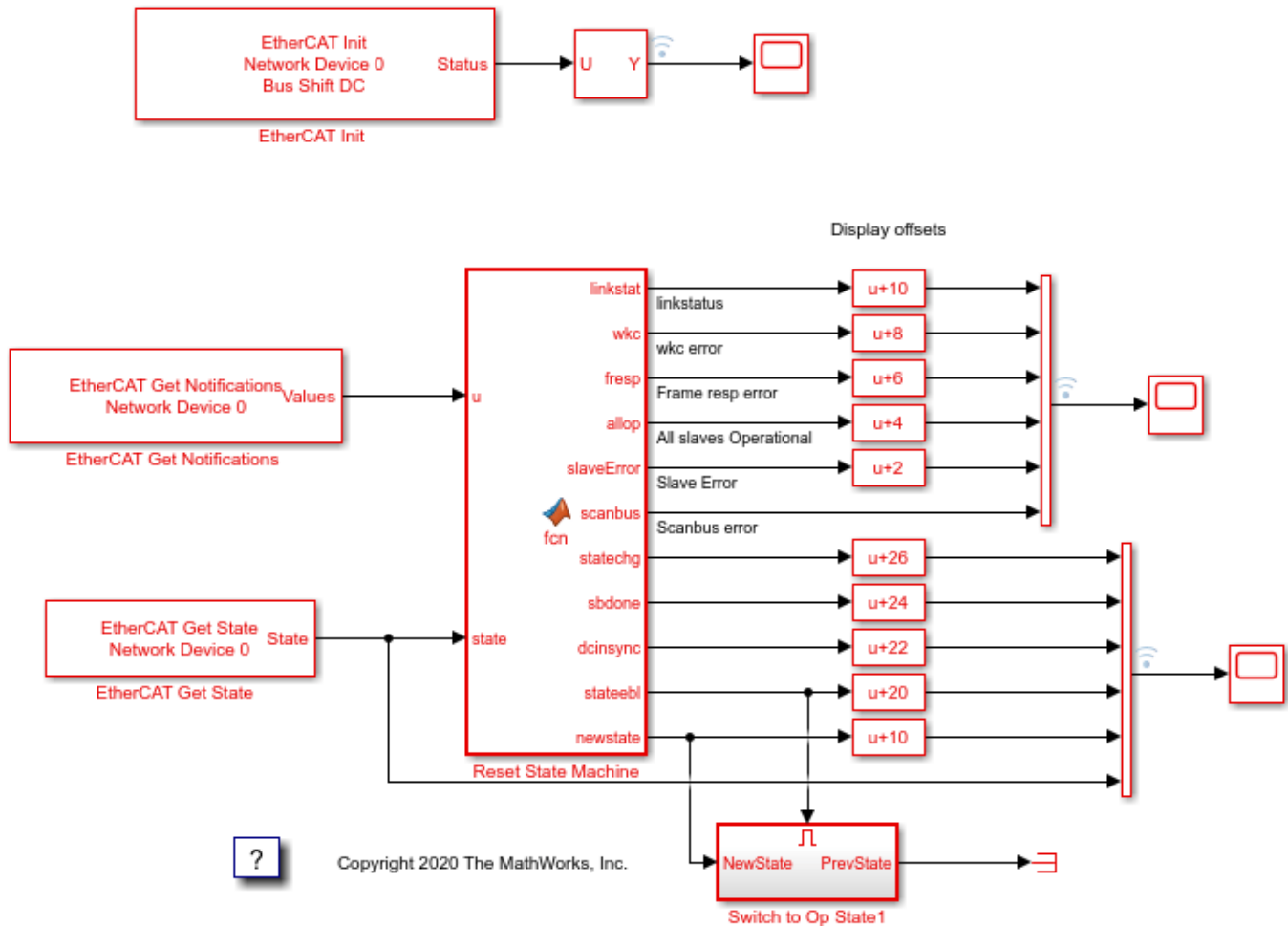


Figure 1: EtherCAT model for detecting a disconnected Ethernet cable at the first slave and reinitializing the network once the cable is reconnected.

Configure the Model

Open the parameter dialog for the **EtherCAT Init** block and observe the pre-configured values. The EtherCAT slave devices that are daisy chained together with Ethernet cable is a Device, also referred to as an EtherCAT network. The Device Index selects one such chained EtherCAT network. The Ethernet Port Number identifies which Ethernet port to use to access that Device. The EtherCAT Init block connects these two so that other EtherCAT blocks use the Device Index to communicate with the slave devices on that EtherCAT network.

If you only have one connected network of EtherCAT slaves, and you have only reserved one Ethernet port with the Ethernet configuration tool, use Device Index = 0 and Ethernet Port Number = 1.

Create an ENI File for a Different Slave Network

If you need to create a new ENI file you need to use a third-party EtherCAT configurator such as TwinCAT 3 from Beckhoff that you install on a development computer. The EtherCAT configuration (ENI) file preconfigured for this model is `Stack4_BS_1ms.xml`.

Each ENI file is specific to the exact network setup for which it was created (for example, the network discovered in step 1 of the configuration file creation process). The configuration file provided for this example is valid if and only if the EtherCAT network consists of a Beckhoff EK1100 with EL1202, EL2202-0100, EL3102 and EL4032 slave modules. If you have a different EtherCAT drive, this example still works, but you need to create a new ENI file that uses your slave devices.

For an overview of the process for creating an ENI file, see “Configure EtherCAT Network by Using TwinCAT 3”.

Build, Download, and Run the Model

To build, download, and run the model:

- 1 In the **Simulink Editor**, from the targets list on the **Real-Time** tab, select the target computer on which to run the real-time application.
- 2 Click **Run on Target**.

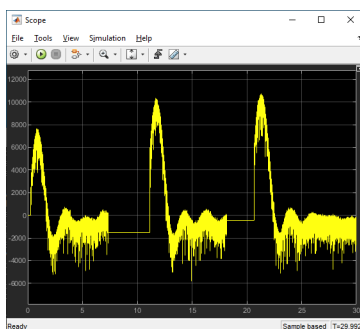
If you open the two scopes by double clicking each, the data is relayed from the target back to the development computer and displayed there.

The model is preconfigured to run for 15 seconds. If you want to run the model longer, pull down the **Run on Target** menu and change the number on the bottom line. Press the green arrow to configure, build, and run.

Display the Target Computer Data

If you run the model using the **Run on Target** button, the external mode is connected and you can double click the scope blocks and see the data on the development computer. The **Display** blocks also work.

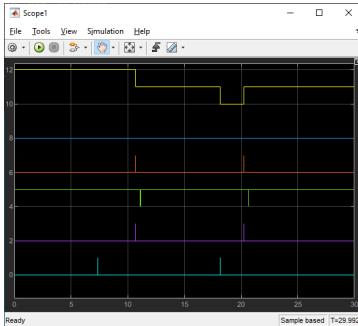
When running this model, to demonstrate the reinitialization stages, you need to disconnect and reconnect the Ethernet cable between the target machine and the EtherCAT slave network. When you reconnect the cable, you see the DC timing perform the same resynchronization that occurs during the initial period.



When using **Run on Target**, **Scope** shows the DC timing error between the master code on the target and the first DC enabled slave. Because the error is returned as nanoseconds, this graph shows that the timing difference settles down to the order of 3-5 microseconds (3000 to 5000 nanoseconds) difference between the DC enabled slaves and the target machine running the code. The residual scatter just reflects task scheduling variability in the target computer RTOS.

In this experimental run, the Ethernet cable was disconnected twice during the 30 second run. Disconnection occurred at about 7 seconds, reconnection at about 12 seconds. This process repeats

at about 18 seconds and 21 seconds. Each time the cable is reconnected, the timing error shows a pulse that shows drift between target and EtherCAT network during the time the cable was disconnected and is the expected resynchronization behavior.

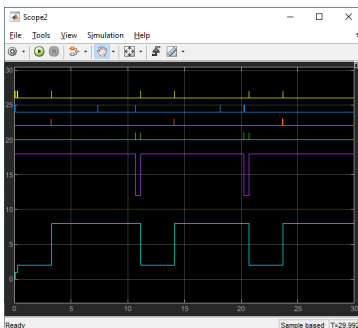


Scope1 shows several logical signals with vertical offsets to show a logic analyzer like display. From the top of the image these are:

- 1 Link status (yellow)
- 2 Working count error (blue)
- 3 Frame response error (red)
- 4 All slaves Operational (green)
- 5 Slave Error (purple)
- 6 Scanbus error (light blue)

Disconnecting the cable caused a scanbus error as seen on the light blue trace. Nothing happens until the cable is reconnected at about 12 seconds. The link status reflects the single time step notifications that indicate the link going away and the link coming back. On the first disconnection, you do not see the link going away notification, but you do see the link coming back. The embedded MATLAB block keeps a persistent variable with the link status with an initial value of 2 and changes it depending on the notifications.

After the link comes back, there is both a slave error and frame response error before All Slaves Operational goes down for a sample time. At that point timing resynchronization starts and you see the damped wave showing the timing error falling to within a few microseconds of error.



Scope2 shows more status outputs with:

- 1 statechange (yellow)

- 2 sbdone (blue)
- 3 dcinsync (red)
- 4 statechange request (green)
- 5 newstate (purple)
- 6 current state (light blue)

When the link goes down, the stack notices that and performs a scan of devices on the bus. That is the sbdone mark at about 7 seconds that also resulted in the sbscan error shown in Scope1. Next when the link is restored at 12 seconds, another bus scan is performed, shown at 12 seconds in the blue trace. The embedded MATLAB block requests a state change to PreOp (=2) shown in the green and purple traces. Once Preop is reached, you see another state change request to go to Op (=8) state which is the second change in green and purple. That starts resynchronization of the clocks between the development computer and the target computer, which takes a few seconds until you see dcinsync at about 14 seconds (red trace) with the transition to Op state right after.

Disconnect the cable again to repeat the whole sequence again starting at about 18 seconds.

While this example needs manual intervention to disconnect and reconnect the Ethernet cable, the same restart can be invoked by just requesting PreOp state followed by a request for Op state, skipping the interaction with the link status if triggered by some other condition in the model.

If you run the model from the command line, you can use the Simulation Data Inspector to view any signal that is marked for signal logging. Signals marked for logging appear with the dot with two arcs above it in the model editor.

See Also

- “Modeling EtherCAT Networks”
- “EtherCAT Protocol Sequenced Writing CoE Slave Configuration Variables” on page 16-69
- “EtherCAT Protocol Sequenced Writing SoE Slave Configuration Variables” on page 16-64

```
close_system( 'slrt_ex_ethercat_notifyreset' );
```

EtherCAT Protocol Sequenced Writing SoE Slave Configuration Variables

This example shows how to use SoE blocks and a simple state machine to write configuration values to variables that can only be written before going to EtherCAT® Op state. For code needed to use the CoE blocks for slaves that understand CoE protocol, see “EtherCAT Protocol Sequenced Writing CoE Slave Configuration Variables” on page 16-69.

For slaves that understand CoE addressing, restrictions on when a specific object can be written is somewhat rare. For slaves that understand SoE addressing, this restriction is much more common.

Changing configuration objects in slave devices before starting IO to the external connections is useful, even if modifying the values is not restricted.

This example also shows distributed clocks synchronization using the bus shift DC mode where the slaves are shifted in time to match the execution time of the master.

Requirements

To run this example, you need an EtherCAT network that consists of the target computer as EtherCAT Master device and at least one slave that has SoE addressed objects. The supplied ENI file is for a Beckhoff® AX5103 drive.

EtherCAT in Simulink® Real-Time™ requires a dedicated network port on the target computer that is reserved for EtherCAT use by using the Ethernet configuration tool. Configure the dedicated port for EtherCAT communication, not with an IP address. The dedicated port must be distinct from the port used for the Ethernet link between the development and target computers.

To test this model:

- 1 Connect the port that is reserved for EtherCAT in the target computer to the EtherCAT IN port of the AX5103 drive.
- 2 Make sure the AX5103 is supplied with a 24-volt power source.
- 3 Build and download the model onto the target.

For a complete example that configures the EtherCAT network, configures the EtherCAT master node model, and builds then runs the real-time application, see “Modeling EtherCAT Networks”.

Open the Model

This model illustrates how you can read or write to SoE/SSC objects if they are only writable in EtherCAT PreOp state. You can move the SoE/SSC transfers to EtherCAT SafeOp state by changing the **Initialization end state** in the EtherCAT Init block and by also changing the constant in the **Wait for this state** constant block. These settings direct the state machine to start sending SoE messages when it reaches the initialization end state.

- 1 Init = 1
- 2 PreOp = 2
- 3 SafeOp = 4
- 4 Op = 8

The EtherCAT initialization block requires that the configuration ENI file is present in the current folder or on the MATLAB® path because the file name is present without directory information.

If you want to modify this model to experiment with it, copy the example configuration file and the model file from the example folder to the current folder. To open the model, in the MATLAB Command Window, type:

```
open_system(fullfile(matlabroot, 'toolbox', 'slrealtime', 'examples', 'slrt_ex_ethercat_asyncSoE_SSC'))
```

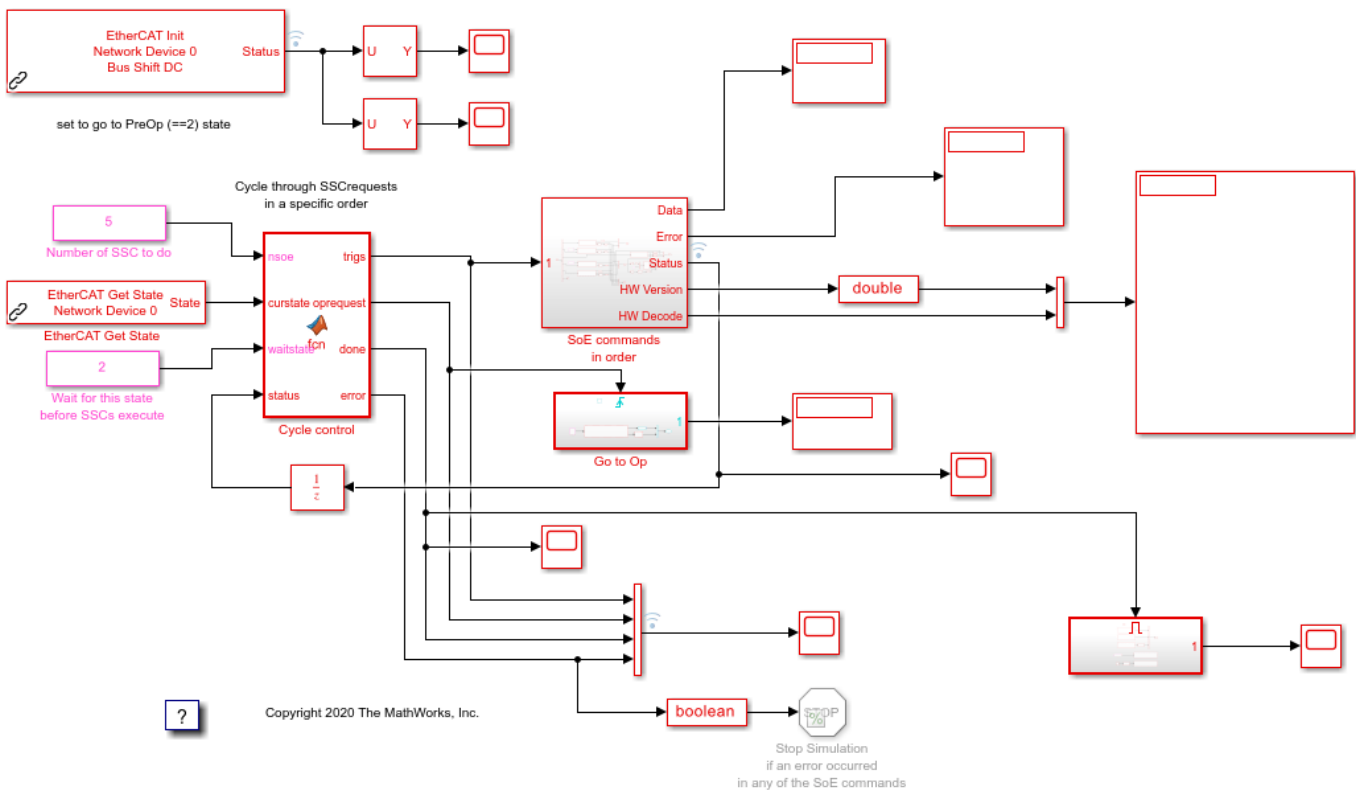


Figure 1: EtherCAT model for sequencing through SoE/SSC commands after pausing initialization at PreOp state

Configure the Model

Open the parameter dialog for the **EtherCAT Init** block and observe the pre-configured values. The EtherCAT slave devices that are daisy chained together with Ethernet cable is a Device, also referred to as an EtherCAT network. The Device Index selects one such chained EtherCAT network. The Ethernet Port Number identifies which Ethernet port to use to access that Device. The EtherCAT Init block connects these two so that other EtherCAT blocks use the Device Index to communicate with the slave devices on that EtherCAT network.

If you only have one connected network of EtherCAT slaves, and you have only reserved one Ethernet port with the Ethernet configuration tool, use Device Index = 0 and Ethernet Port Number = 1.

Create an ENI file for a Different SoE drive

If you need to create a new ENI file you need to use a third-party EtherCAT configurator such as TwinCAT 3 from Beckhoff that you install on a development computer. The EtherCAT configuration (ENI) file preconfigured for this model is `BeckDrive_1ms.xml`.

Each ENI file is specific to the exact network setup for which it was created (for example, the network discovered in step 1 of the configuration file creation process). The configuration file provided for this example is valid if and only if the EtherCAT network consists of one AX5103 drive. If you have a different EtherCAT drive this example still works, but you need to create a new ENI file that uses your drive.

For an overview of the process for creating an ENI file, see “Configure EtherCAT Network by Using TwinCAT 3”.

If you use a different drive, you need to consult the manual for your devices and find the SoE mapping. Using that mapping, you need to change the SSC commands in the **SoE commands in order** subsystem to use objects on your drive.

Build, Download, and Run the Model

To build, download, and run the model:

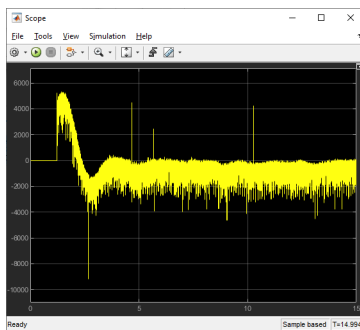
- 1 In the **Simulink Editor**, from the targets list on the **Real-Time** tab, select the target computer on which to run the real-time application.
- 2 Click **Run on Target**.

If you open the two scopes by double clicking each, the data is relayed from the target back to the development computer and displayed.

The model is preconfigured to run for 15 seconds. If you want to run the model longer, pull down the **Run on Target** menu and change the number on the bottom line. Press the green arrow to configure, build, and run.

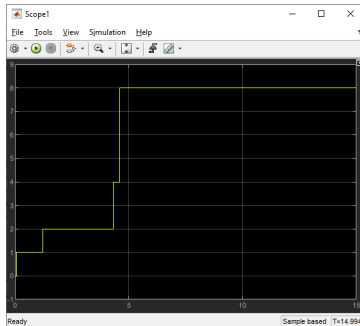
Display the Target Computer Data

If you run the model using the **Run on Target** button, external mode is connected and you can double click the scope blocks and see the data on the development computer. The **Display** blocks also work.

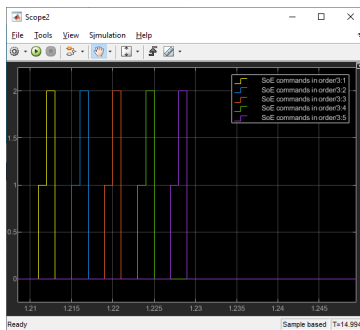


When using **Run on Target**, **Scope** shows the DC timing error between the master code on the target and the first DC enabled slave. Since the error is returned as nanoseconds, this graph shows that the timing difference settles down to the order of 3-5 microseconds (3000 to 5000 nanoseconds)

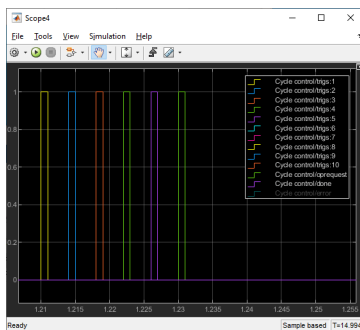
difference between the DC enabled slaves and the target machine running the code. The residual scatter just reflects task scheduling variability in the target computer RTOS.



Scope1 shows the progression of EtherCAT states, from idle to Init to PreOp, SafeOp and finally Op state. SafeOp is only entered briefly and only shows up as the value 4 for a few time steps before the switch to Op state. Since this model uses the distributed clocks mechanism, the switch to Op state only occurs once the timing error settles down.



Scope2 shows the status outputs of the 5 async SDO blocks inside the subsystem. Each SDO block is enabled to write for one time step. The block switches to status = 1 (busy) for a few time steps. On successful completion status = 2 (done), the block switches for one time step. If a block encounters an error, the block switches to status = 3 (error) for one time step. On an error, the **Cycle control** embedded MATLAB code block stops the sequence and sets the error output, which stops the model. In that case, the failing block have output an error code that is displayed on Display1. This display is zoomed into the interval just after state went to PreOp (=2) state.



Scope4 shows several of the outputs from the **Cycle control** block. The first 5 are the enable signals, made true one at a time by **Cycle control**. The **oprequest** output is true for one time step to trigger the request to proceed to Op state. This display is zoomed in to the same interval as in **Scope2**.

When all of the requested SSC commands are complete and the state has progressed to Op state, the **done** signal is set to **true** for the remainder of execution. The rest of your model goes into the **Op State Model** subsystem.

If you need a different number of SSC commands to execute before Op state, you need to edit the **Cycle control** embedded MATLAB code block and modify the persistent array that is currently sized to have length 10, which is larger than the number of SSC commands being requested.

If you run the model from the command line, you can use the Data Inspector, accessible from the toolstrip, to view any signal that has been tagged to log with the **Log Selected Signals** selection found by right clicking on the signal. Those are marked with the dot with two arcs above it in the model editor.

See Also

- “EtherCAT Protocol Sequenced Writing CoE Slave Configuration Variables” on page 16-69
- “EtherCAT Protocol Detect Network Failure and Reset” on page 16-59
- “Modeling EtherCAT Networks”
- “Configure EtherCAT Network by Using TwinCAT 3”

EtherCAT Protocol Sequenced Writing CoE Slave Configuration Variables

This example shows how to use CoE blocks and a simple state machine to write configuration values to variables that can only be written before going to EtherCAT® Op state. For code needed to use the SoE blocks for slaves that understand SoE protocol, see “EtherCAT Protocol Sequenced Writing SoE Slave Configuration Variables” on page 16-64.

For slaves that understand CoE addressing, limited ability to read or write specific objects is somewhat rare. For slaves that understand SoE addressing, this restriction is much more common.

This example also shows distributed clocks synchronization using the bus shift DC mode where the slaves are shifted in time to match the execution time of the master.

Requirements

To run this example, you need an EtherCAT network that consists of the target computer as EtherCAT master device and at least one slave that has CoE addressed objects. The supplied ENI file is for a 5 element slave stack: EK1100+EL1202+EL2202+EL3102+EL4032.

EtherCAT in Simulink® Real-Time™ requires a dedicated network port on the target computer that is reserved for EtherCAT by using the Ethernet configuration tool. Configure the dedicated port for EtherCAT communication, not with an IP address. The dedicated port must be distinct from the port used for the Ethernet link between the development and target computers.

To test this model:

- 1 Connect the port that is reserved for EtherCAT in the target computer to the EtherCAT IN port of the EL1100 interface.
- 2 Make sure the EK1100 is supplied with a 24-volt power source.
- 3 Build and download the model onto the target.

For a complete example that configures the EtherCAT network, configures the EtherCAT master node model, and builds then runs the real-time application, see “Modeling EtherCAT Networks”.

Open the Model

This model illustrates how you can read or write to CoE/SDO objects if they are only writable in EtherCAT PreOp state. You can move the CoE/SDO transfers to EtherCAT SafeOp state by changing the **Initialization end state** in the EtherCAT Init block and by also changing the constant in the **Wait for this state** constant block. These settings direct the state machine to start sending CoE messages when it reaches the initialization end state.

- 1 Init = 1
- 2 PreOp = 2
- 3 SafeOp = 4
- 4 Op = 8

The EtherCAT initialization block requires that the configuration ENI file is present in the current folder.

If you want to modify this model to experiment with it, then copy the example configuration file from the example folder to the current folder. To open the model, in the MATLAB® Command Window, type:

```
open_system(fullfile(matlabroot,'toolbox','slrealtime','examples','slrt_ex_ethercat_asyncCoE_cyc
```

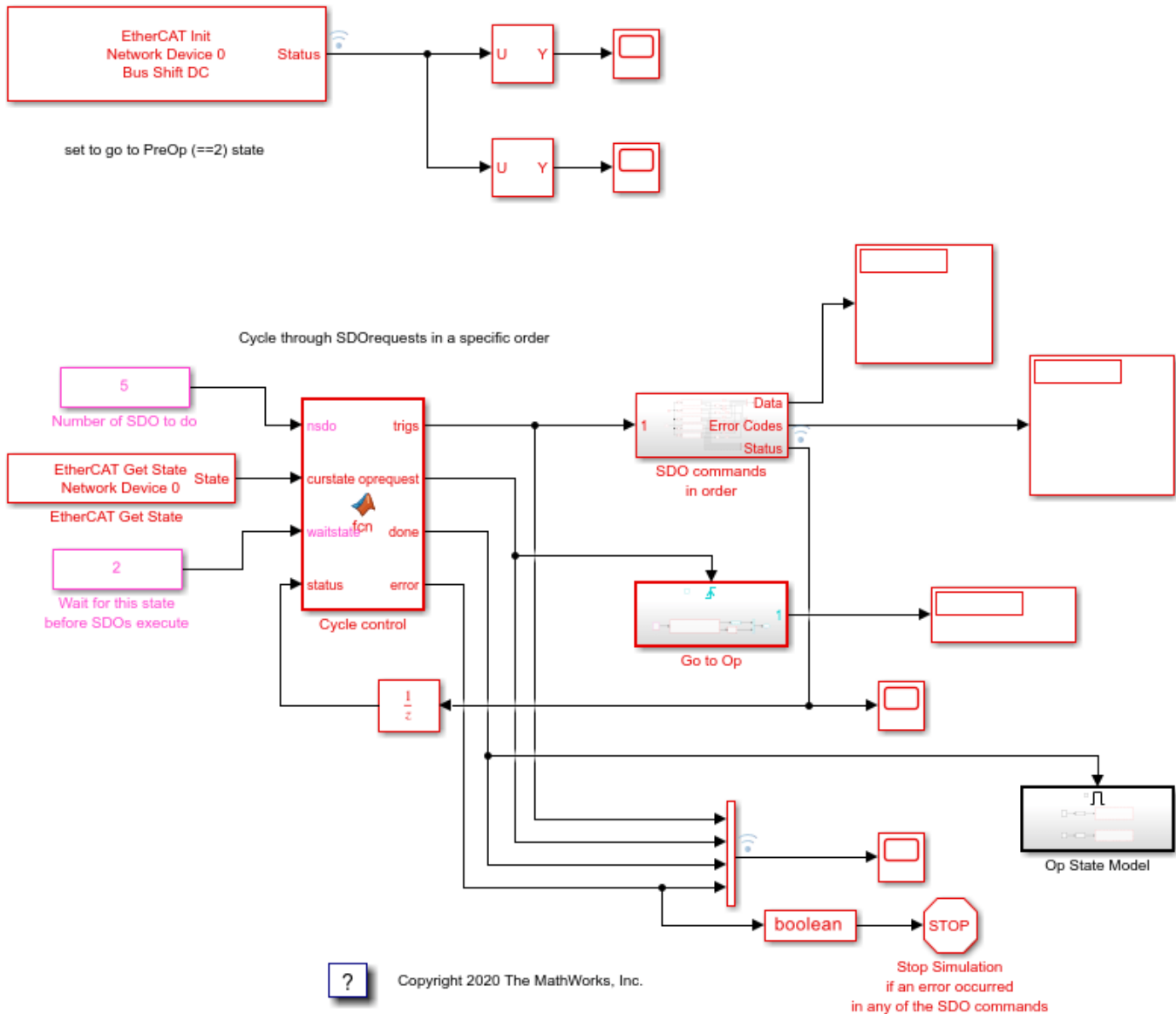


Figure 1: EtherCAT model for sequencing through CoE commands after pausing initialization at PreOp state

Configure the Model

Open the parameter dialog for the **EtherCAT Init** block and observe the pre-configured values. The EtherCAT slave devices that are daisy chained together with Ethernet cable is a Device, also referred to as an EtherCAT network. The Device Index selects one such chained EtherCAT network. The Ethernet Port Number identifies which Ethernet port to use to access that Device. The EtherCAT Init

block connects these two so that other EtherCAT blocks use the Device Index to communicate with the slave devices on that EtherCAT network.

If you only have one connected network of EtherCAT slaves, and you have only reserved one Ethernet port with the Ethernet configuration tool, use Device Index = 0 and Ethernet Port Number = 1.

Create an ENI File for a Different Set of Slaves

If you need to create a new ENI file you need to use a third-party EtherCAT configurator such as TwinCAT 3 from Beckhoff® that you install on a development computer. The EtherCAT configuration (ENI) file preconfigured for this model is `Stack4_BS_1ms.xml`.

Each ENI file is specific to the exact network setup from which it was created (for example, the network discovered in step 1 of the configuration file creation process). The configuration file provided for this example is valid if and only if the EtherCAT network consists of an EK1100+EL1202+EL2202+EL3102+EL4032. If you have a different set of EtherCAT slave devices this example works, but you need to create a new ENI file that uses your devices.

For overview of the process for creating an ENI file, see “Configure EtherCAT Network by Using TwinCAT 3”.

If you use different slave devices, you need to consult the manual for your devices and find the CoE mapping. Using that mapping, you need to change the SDO commands in the **SDO commands in order** subsystem to use objects on your devices.

Build, Download, and Run the Model

To build, download, and run the model:

- 1 In the **Simulink Editor**, from the targets list on the **Real-Time** tab, select the target computer on which to run the real-time application.
- 2 Click **Run on Target**.

If you open the two host side scopes by double clicking each, data is relayed from the target computer to the development computer and is displayed.

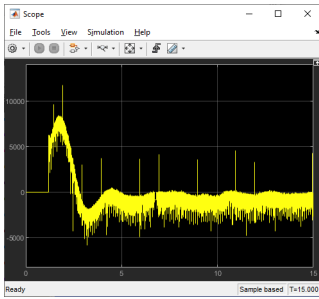
Included in the model is the ability to control the amplitude of the cycling motion. With the **Run on Target** button, the slider is active and connected to the **Amplitude** constant block.

The model is preconfigured to run for 15 seconds. If you want to run the model longer, pull down the **Run on Target** menu and change the number on the bottom line. Press the green arrow to configure, build, and run.

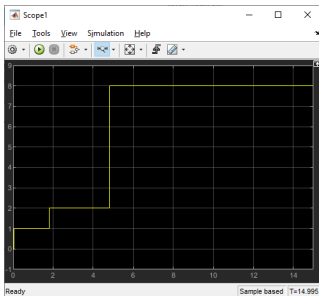
Display the Target Computer Data

If you run the model using the **Run on Target** button, external mode is connected and you can double click the scope blocks and see the data on the development computer. The **Display** blocks also work.

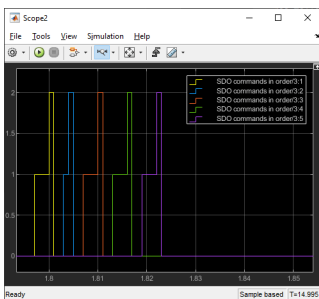
When using **Run on Target**, the **Scope** block shows the DC timing error between the master code on the target computer and the first DC enabled slave. Because the error is returned as nanoseconds, this graph shows that the timing difference settles down to the order of 3-5 microseconds (3000 to 5000 nanoseconds) difference between the DC enabled slaves and the target machine running the code. The residual scatter reflects task scheduling variability in the target computer RTOS.



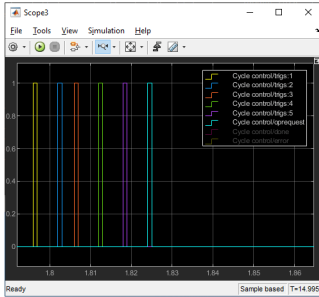
Scope1 shows the progression of EtherCAT states, from idle to Init to PreOp, SafeOp and finally Op state. SafeOp is only entered briefly and only shows up as the value 4 for a few time steps before the switch to Op state. Since this model uses the distributed clocks mechanism, the switch to Op state only occurs once the timing error settles down.



Scope2 shows the status outputs of the 5 async SDO blocks inside the subsystem. Each SDO block is enabled to write for one time step, then switches to status = 1 (busy) for a few time steps, then on successful completion status = 2 (done) for one time step. If a block encounters an error, status = 3 (error) for one time step. On an error, the **Cycle control** embedded MATLAB code block stops the sequence and sets the error output which stops the model. In that case, the failing block has output an error code that is displayed on Display1. This display is zoomed into the interval just after state went to PreOp (=2) state.



Scope3 shows several of the outputs from the **Cycle control** block. The first 5 are the enable signals, made true one at a time by **Cycle control**. Then the **oprequest** output is true for one time step to trigger the request to proceed to Op state. This display is zoomed in to the same interval as in **Scope2**.



When the requested SDO commands are complete and the state has progressed to Op state, the **done** signal is set to **true** for the remainder of execution. The rest of your model goes into the **Op State Model** subsystem.

If you need a different number of SDO commands to execute before Op state, you need to edit the **Cycle control** embedded MATLAB code block and modify the persistent array that is currently sized to have length 5, the same as the number of SDO commands being requesting.

If you run the model from the command line, you can use the Simulation Data Inspector to view any signal that has been marked for signal logging. Signals marked for signal logging have a dot with two arcs above it in the model editor.

See Also

- “EtherCAT Protocol Sequenced Writing SoE Slave Configuration Variables” on page 16-64
- “EtherCAT Protocol Detect Network Failure and Reset” on page 16-59
- “Modeling EtherCAT Networks”
- “Configure EtherCAT Network by Using TwinCAT 3”

Simple ASCII Encoding/Decoding Loopback Test (with Baseboard Blocks)

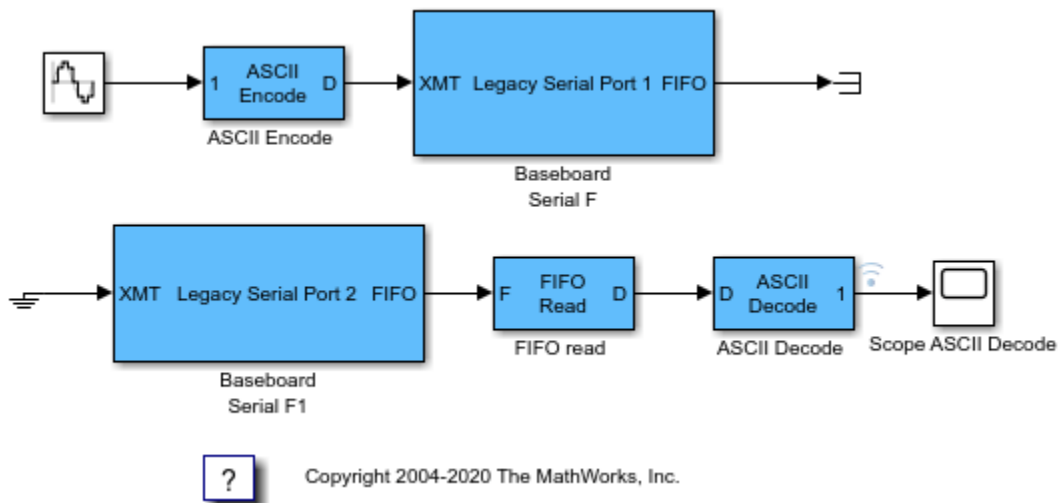
This example model shows how a single floating point number can be converted to ASCII and transmitted over a serial link. The sending serial port and receiving serial port can be in the same system or in different systems.

To test this model:

- 1 The target computer must have two legacy serial ports.
- 2 Connect legacy serial port 1 to legacy serial port 2 with a null modem cable.

This example is configured to use baseboard serial ports (legacy serial port 1 and legacy serial port 2). You can also use legacy serial port 3 and legacy serial port 4 by changing the board setup in the Baseboard blocks. Other serial blocks could be used in place of the Baseboard blocks.

`open_system(fullfile(matlabroot, 'toolbox', 'slrealtime', 'examples', 'slrt_ex_serialbaseboardsimple`



See Also

- “RS-232 Serial Communication”
- “RS-232 Legacy Drivers”

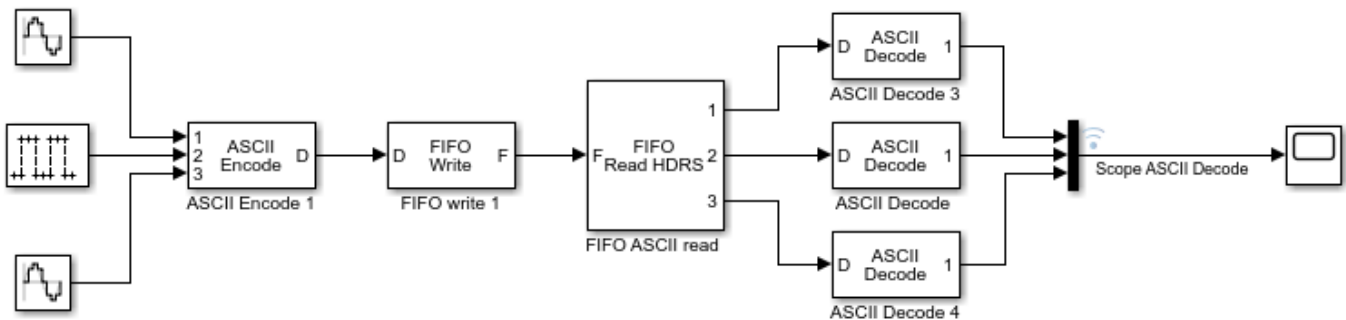
ASCII Encoding/Decoding Loopback Test

This model shows how to send ASCII data over a serial link.

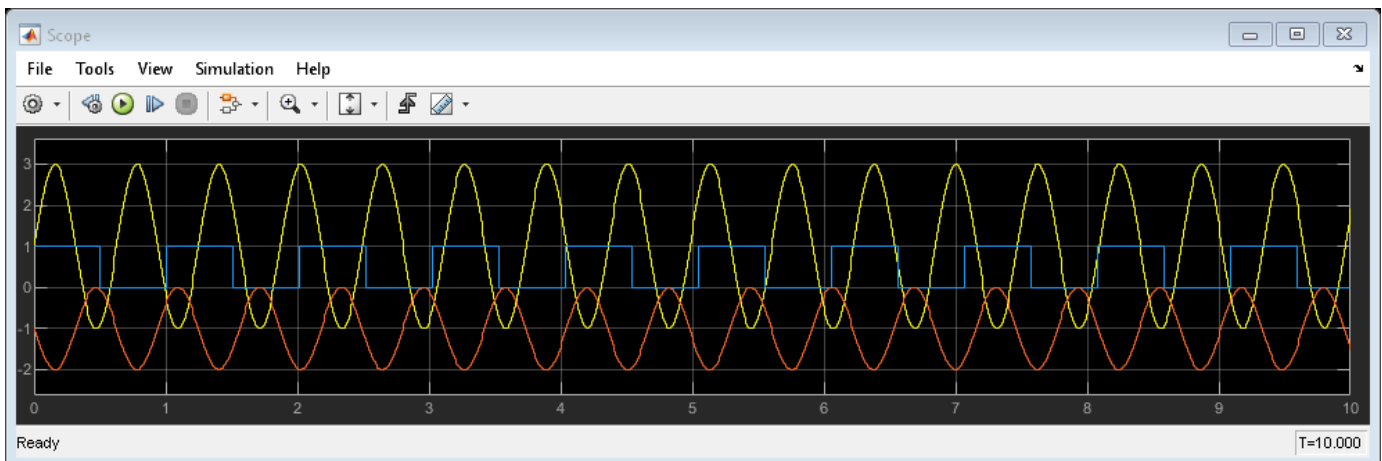
The ASCII Encode block generates a message with three different sub messages along with some extraneous data to show how the FIFO Read HDRS block can remain synchronized to the valid byte stream even in the presence of transmission errors.

The FIFO Read HDRS block can handle an arbitrary number of headers; just add them as strings to the cell array in the block parameters dialog box. The messages must share the same termination string. In this example, it is a carriage return followed by a line feed: "\r\n".

```
open_system(fullfile(matlabroot, 'toolbox', 'slrealtime', 'examples', 'slrt_ex_serialasciitest'))
set_param('slrt_ex_serialasciitest', 'StopTime', '30');
sim('slrt_ex_serialasciitest')
```



Copyright 2004-2019 The MathWorks, Inc.



See Also

- “RS-232 Serial Communication”
- “RS-232 Legacy Drivers”

ASCII Encoding/Decoding Loopback Test (with Baseboard Blocks)

This example model shows how to send ASCII data over a serial link.

The ASCII Encode block generates a message with three different sub messages along with some extraneous data to show how the FIFO Read HDRS block can remain synchronized to the valid byte stream even in the presence of transmission errors.

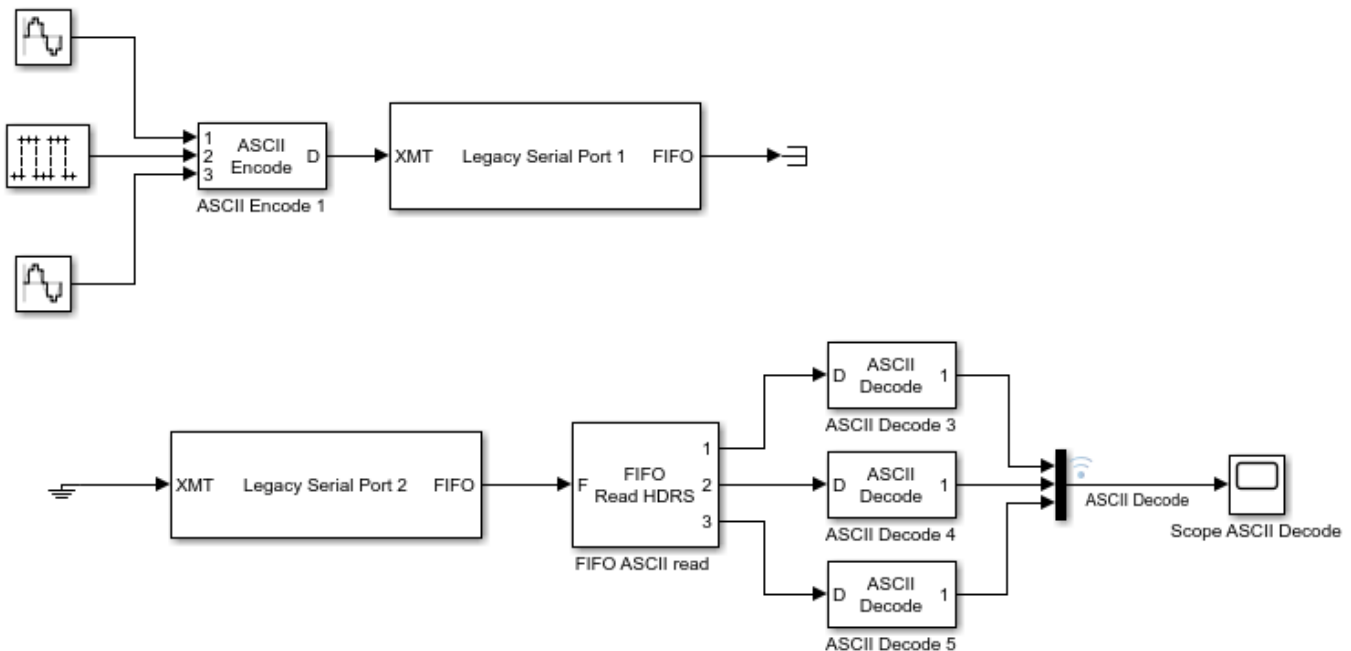
The FIFO Read HDRS block can handle an arbitrary number of headers; just add them as strings to the cell array in the block parameters dialog box. The messages must share the same termination string. In this example, it is a carriage return followed by a line feed: "\r\n".

To test this model:

- 1 The target computer must have two legacy serial ports.
- 2 Connect legacy serial port 1 to legacy serial port 2 with a null modem cable.

This example is configured to use baseboard serial ports (legacy serial port and legacy serial port 2). You can also use legacy serial port 3 and legacy serial port 4 by changing the board setup in the Baseboard blocks. Other serial blocks could be used in place of the Baseboard blocks.

```
open_system(fullfile(matlabroot, 'toolbox', 'slrealtime', 'examples', 'slrt_ex_serialbaseboardasciit
```



Copyright 2004-2019 The MathWorks, Inc.

See Also

- “RS-232 Serial Communication”
- “RS-232 Legacy Drivers”

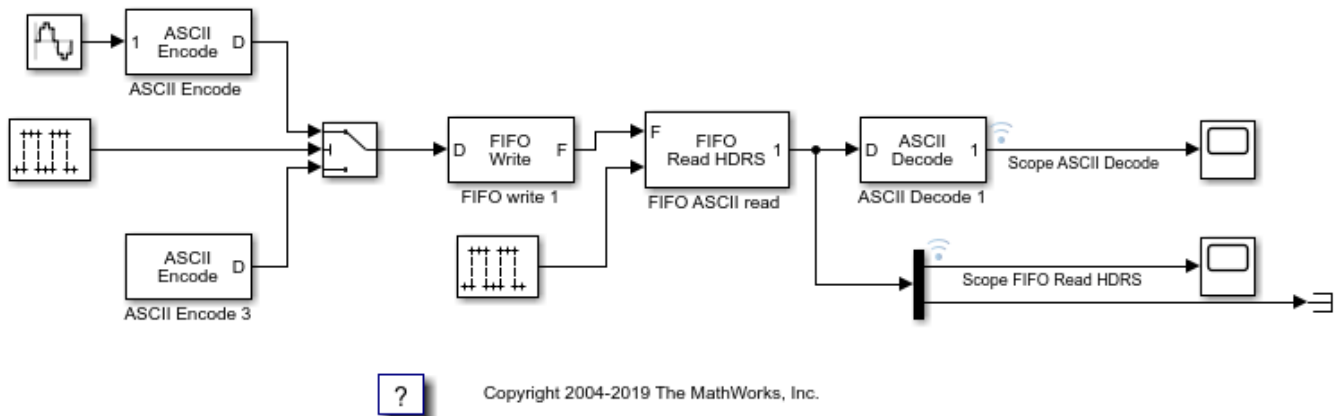
ASCII Encoding/Decoding Resync Loopback Test

This example model shows the ability of the FIFO Read HDRS block to resynchronize after being repeatedly disabled and its the ability to resolve errors such as when a message is only partially complete at the time the read is attempted.

The Switch block alternates between the first and last parts of the message on successive sample times. This mimics a worst case scenario where the model updates before the message construction is complete. As a result, sometimes only part of the message is received. The second pulse generator alternately enables and disables the FIFO Read HDRS block.

Scope 1 graphs the decoded sine wave data received at each time step. When the Pulse Generator1 block outputs a 0, the count from the FIFO Read HDRS block is 0. When it outputs a 1, the read catches up by throwing away extra data and returns the last complete value found in the FIFO. Scope 2 indicates when new data is present.

```
open_system(fullfile(matlabroot,'toolbox','slrealtime','examples','slrt_ex_serialasciisplit'))
set_param('slrt_ex_serialasciisplit','StopTime','30');
sim('slrt_ex_serialasciisplit')
```



See Also

- “RS-232 Serial Communication”
- “RS-232 Legacy Drivers”

ASCII Encoding/Decoding Resync Loopback Test (with Baseboard Blocks)

This model shows the ability of the FIFO Read HDRS block to resynchronize after being repeatedly disabled as well as the ability to resolve errors such as when a message is only partially complete at the time the read is attempted.

The Switch block alternates between the first and last parts of the message on successive sample times. This mimics a worst case scenario where the model updates before the message construction is complete. As a result, sometimes only part of the message is received. The second pulse generator alternately enables and disables the FIFO Read HDRS block.

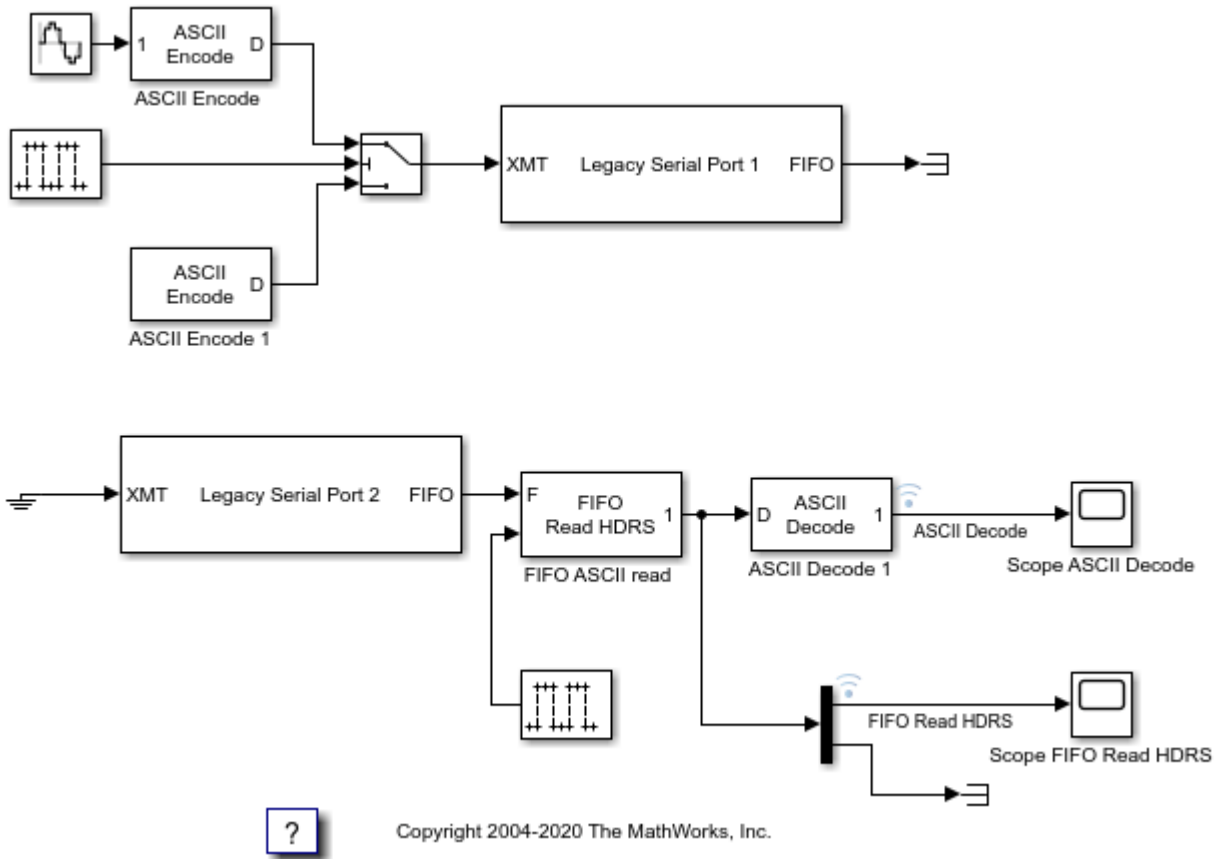
Scope 1 graphs the decoded sine wave data received at each time step. When the Pulse Generator1 block outputs a 0, the count from the FIFO Read HDRS block is 0. When it outputs a 1, the read catches up by throwing away extra data and returns the last complete value found in the FIFO. Scope 2 indicates when new data is present.

To test this model:

- 1 The target computer must have two legacy serial ports.
- 2 Connect legacy serial port 1 to legacy serial port 2 with a null modem cable.

This example is configured to use baseboard serial ports (legacy serial port 1 and legacy serial port 2). You can also use legacy serial port 3 and legacy serial port 4 by changing the board setup in the Baseboard blocks. Other serial blocks could be used in place of the Baseboard blocks.

```
open_system(fullfile(matlabroot, 'toolbox', 'slrealtime', 'examples', 'slrt_ex_serialbaseboardasciis
```



See Also

- “RS-232 Serial Communication”
- “RS-232 Legacy Drivers”

Binary Encoding/Decoding Loopback Test

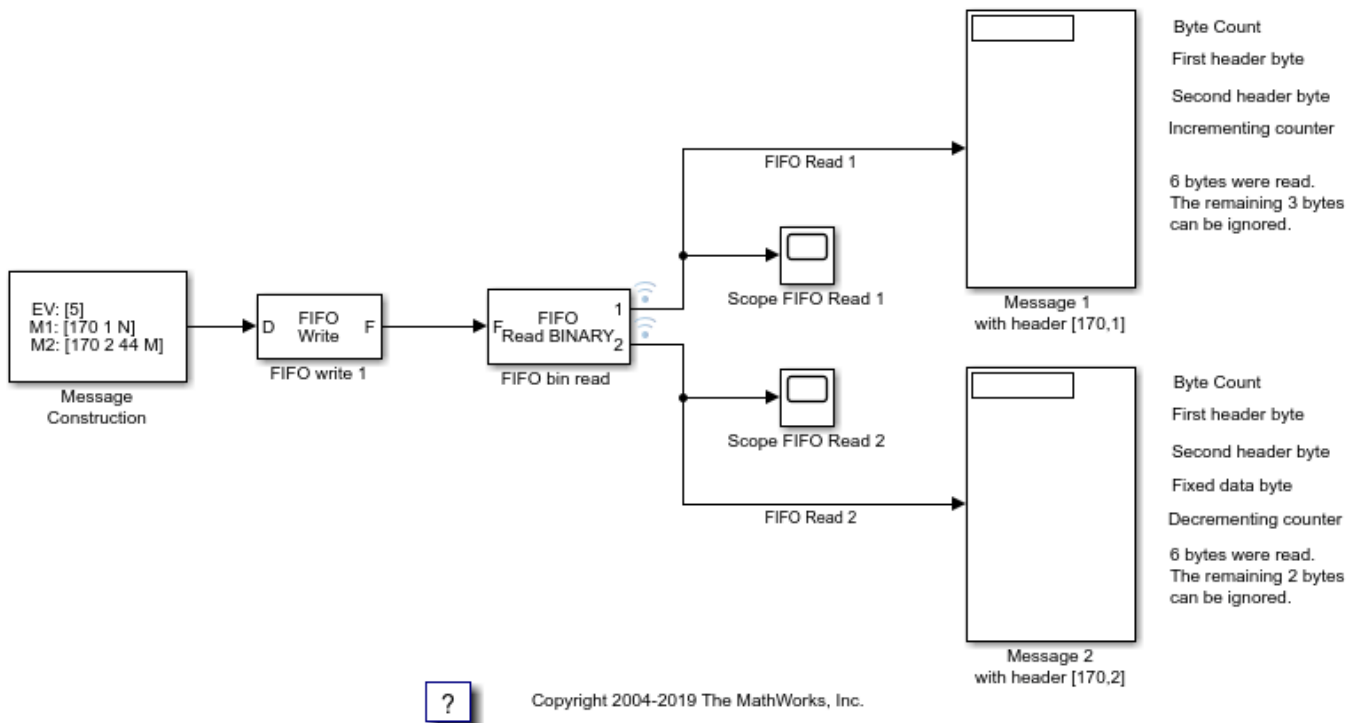
This model shows how to send Binary data over a serial link.

The transmitted data are: [8,5,170,1,N,170,2,44,M]. This byte stream contains two messages along with other elements:

- The first byte, 8, is a count of the remaining number of bytes in the stream.
- The second byte, 5, is an extraneous value (EV).
- [170,1,N] is message 1 (M1).
- [170,2,44,M] is message 2 (M2).
- N and M are numbers between 0 and 255 that are incrementing and decrementing, respectively.

Even though the data stream includes extraneous bytes (5 in this case), the FIFO Read BINARY block can handle and ignore this extra information. Scope 1 displays the received message 1 data. Scope 2 displays the received message 2 data.

```
open_system(fullfile(matlabroot,'toolbox','slrealtime','examples','slrt_ex_serialbinarytest'));
```



See Also

- “RS-232 Serial Communication”
- “RS-232 Legacy Drivers”

Binary Encoding/Decoding Loopback Test (with Baseboard Blocks)

This model shows how to send Binary data over a serial link.

The transmitted data are: [8, 5, 170, 1, N, 170, 2, 44, M]. This byte stream contains two messages along with other elements.

- The first byte, 8, is a count of the remaining number of bytes in the stream.
- The second byte, 5, is an extraneous value (EV).
- [170, 1, N] is message 1 (M1).
- [170, 2, 44, M] is message 2 (M2).
- N and M are numbers between 0 and 255 that are incrementing and decrementing, respectively.

Notice that when the data contains extraneous bytes (5 in this case) the FIFO Read BINARY block can handle and ignore this extra information.

Scope 1 displays the received message 1 data. Scope 2 displays the received message 2 data.

To test this model:

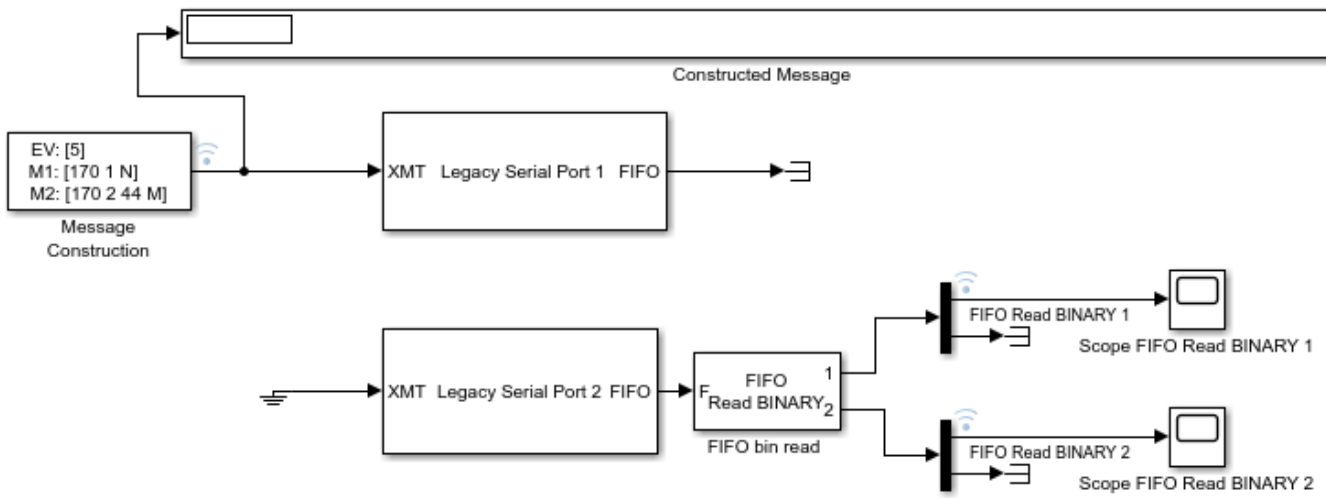
- 1 The target computer must have two legacy serial ports.
- 2 Connect legacy serial port 1 to legacy serial port 2 with a null modem cable.

This example is configured to use baseboard serial ports (legacy serial port 1 and legacy serial port 2). You can also use legacy serial port 3 and legacy serial port 4 by changing the board setup in the Baseboard blocks. Other serial blocks could be used in place of the Baseboard blocks.

```
open_system(fullfile(matlabroot, 'toolbox', 'slrealtime', 'examples', 'slrt_ex_serialbaseboardbinary
```

See Also

- “RS-232 Serial Communication”
- “RS-232 Legacy Drivers”



Copyright 2004-2019 The MathWorks, Inc.

Binary Encoding/Decoding Resync Loopback Test

This model shows the ability of the FIFO Read BINARY block to handle messages that are interrupted and only partially complete. This is a worst case example where every message is interrupted.

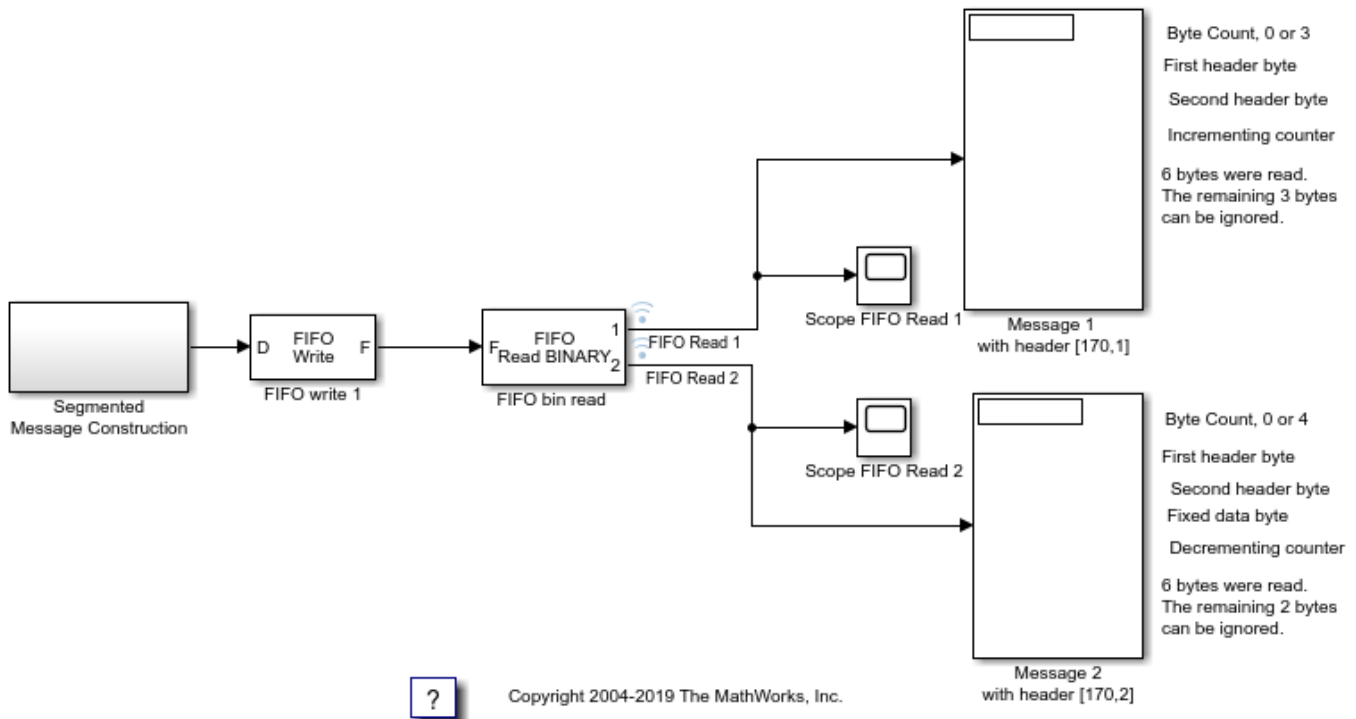
The Segmented Message Constructor subsystem contains blocks that prepare and send only parts of messages on each time step.

On the receive side, the FIFO read BINARY block is looking for two different two-character headers. If it finds $[170, 1]$ it outputs $[3, 170, 1, N]$ on port 1. If it finds $[170, 2]$, it outputs $[4, 170, 2, 44, M]$ to port 2. N and M are numbers between 0 and 255 that are incrementing and decrementing, respectively.

If a message header is not found in the FIFO on a given time step, then that port will output 0. The outputs are padded to the maximum vector size specified in the FIFO Read BINARY block. In this example output vectors are 6 in width. The count in the first element tells how many elements are significant.

Scope 1 displays the received message 1 data. Scope 2 displays the received message 2 data.

```
open_system(fullfile(matlabroot, 'toolbox', 'slrealtime', 'examples', 'slrt_ex_serialbinarysplit'));
```



See Also

- “RS-232 Serial Communication”
- “RS-232 Legacy Drivers”

Binary Encoding/Decoding Resync Loopback Test (with Baseboard Blocks)

This model shows the ability of the FIFO Read BINARY block to handle messages that are interrupted and only partially complete. This is a worst case example where every message is interrupted.

The Segmented Message Constructor subsystem contains blocks that prepare and send only parts of messages on each time step.

On the receive side, the FIFO read BINARY block is looking for two different two-character headers. If it finds [170, 1] it outputs [3, 170, 1, N] on port 1. If it finds [170, 2], it outputs [4, 170, 2, 44, M] to port 2. N and M are numbers between 0 and 255 that are incrementing and decrementing, respectively.

If a message header is not found in the FIFO on a given time step, then that port will output 0. The outputs are padded to the maximum vector size specified in the FIFO Read BINARY block. In this example output vectors are 1024 in width. The count in the first element tells how many elements are significant. The Demux blocks discard the uninteresting parts of the signal.

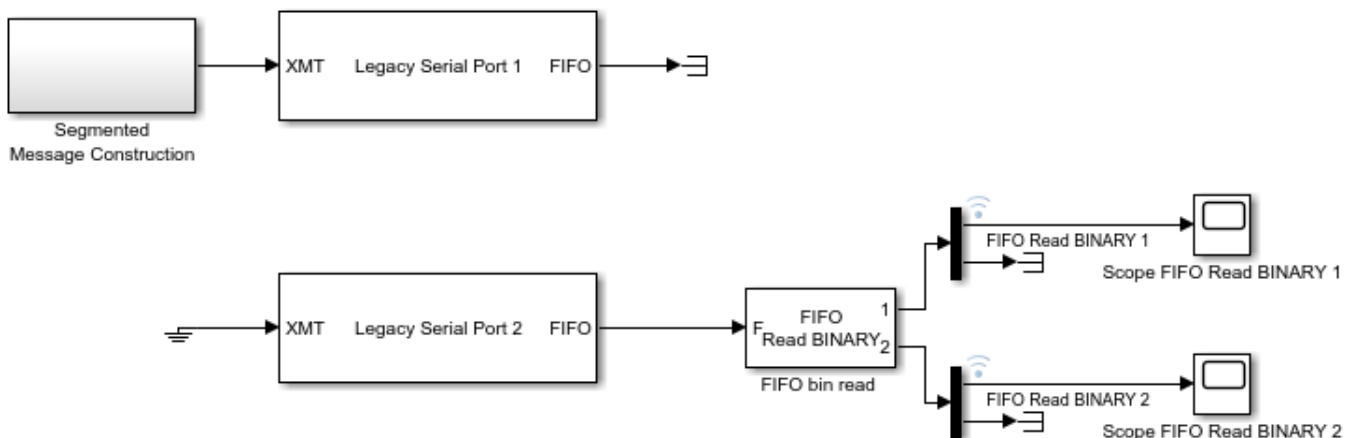
Scope 1 displays the received message 1 data. Scope 2 displays the received message 2 data.

To test this model:

- 1 The target computer must have two legacy serial ports.
- 2 Connect legacy serial port 1 to legacy serial port 2 with a null modem cable.

This example is configured to use baseboard serial ports (legacy serial port 1 and legacy serial port 2). You can also use legacy serial port 3 and legacy serial port 4 by changing the board setup in the Baseboard blocks. Other serial blocks could be used in place of the Baseboard blocks.

```
open_system(fullfile(matlabroot, 'toolbox', 'slrealtime', 'examples', 'slrt_ex_serialbaseboardbinary
```



Copyright 2004-2019 The MathWorks, Inc.

See Also

- “RS-232 Serial Communication”
- “RS-232 Legacy Drivers”

Target to Development Computer Communication by Using TCP

This example shows how to use TCP blocks to send data from the target computer to MATLAB® running on the development computer. This example uses a target computer located at IP address 192.168.7.5.

The TCP Send block in the server real-time application `slrt_ex_target_to_host_TCP` sends data from the target computer to the TCP/IP object that is created in MATLAB on the development computer. The MATLAB m-script sends the received data back to the real-time application.

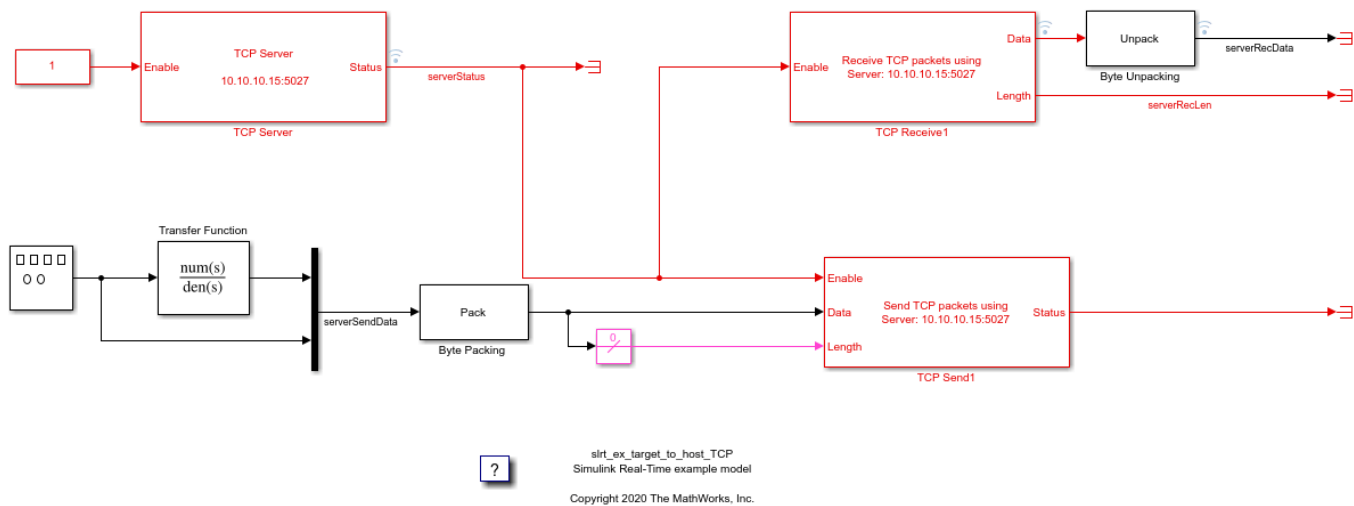
To open this example, in the MATLAB Command Window, type:

```
open_system(fullfile(matlabroot,'toolbox','slrealtime','examples','slrt_ex_target_to_host_TCP'))
```

Open, Build, and Download Server Application

Open the model.

```
model = 'slrt_ex_target_to_host_TCP';
mdlOpen = 0;
systems = find_system('type', 'block_diagram');
if ~any(strcmp(model, systems))
    mdlOpen = 1;
    open_system(fullfile(matlabroot,'toolbox','slrealtime','examples',model));
end
```



Build Model and Download to Target Computer

```
set_param(model, 'RTWVerbose', 'off');
set_param(model, 'StopTime', '10');
targetIP = '192.168.7.5';
set_param([model, '/TCP Server'], 'serverAddress', targetIP);
evalc('slbuild(model)');
tg = slrealtime;
load(tg, model);
```

Close the model.

```
if (mdlOpen)
    bdclose(model);
end
```

Create TCP/IP Object in MATLAB on Development Computer

Create a TCP/IP object and connect the TCP/IP object to the development computer.

```
t = tcpclient(targetIP,5027);
```

Run Real-Time Application on Target Computer

```
start(tg);
pause(3);
```

Read Data Packets and Send Back to Target Computer

Read from the target computer and write back.

```
tic
while (toc<5)
    data = read(t,16);
    write(t,data);
end
```

Stop Real-Time Application on Target Computer

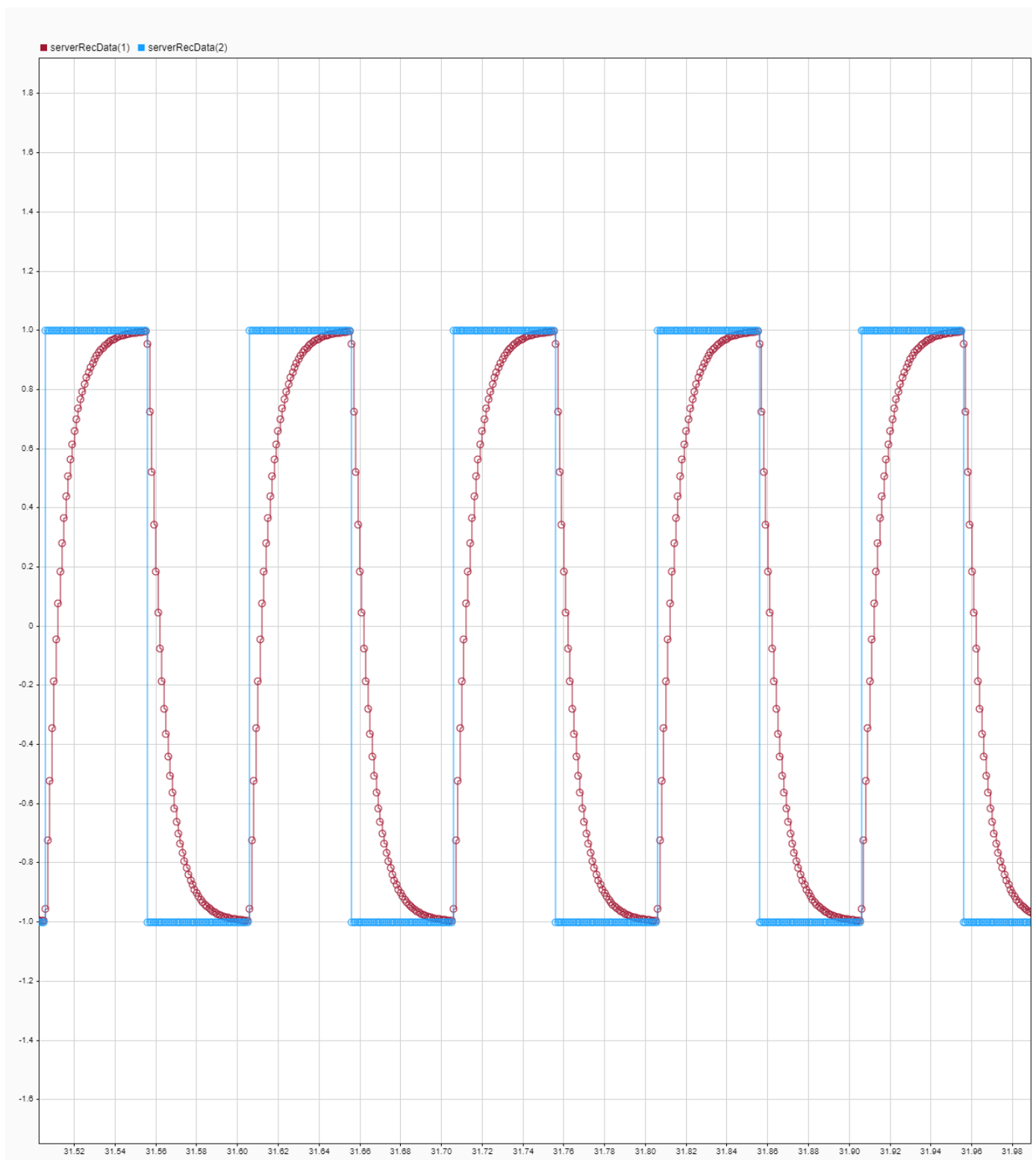
```
stop(tg);
```

Close TCP/IP Object on Development Computer

```
clear t;
```

View Signal Received on Target Computer

```
Simulink.sdi.view();
```



Target to Host Transmission by Using UDP

This example shows how to use UDP blocks to send data from a target computer to a development computer. This example uses a target computer located at IP address 192.168.7.5 and uses a development computer located at IP address 192.168.7.2.

The transmit real-time application `slrt_ex_target_to_host_UDP` runs on the target computer and send signal data to the UDP object that the script creates in MATLAB on the development computer.

When using the UDP protocol for communicating data to or from the target computer, consider these issues:

- The Simulink model on the development computer runs as fast as it can. The model run speed is not synchronized to a real-time clock.
- UDP is a connectionless protocol that does not check to confirm that packets were transmitted. Data packets can be lost or dropped.
- On the target computer, UDP blocks run in a background task that executes each time step after the real-time task completes. If the block cannot run or complete the background task before the next time step, data may not be communicated.
- UDP data packets are transmitted over the Ethernet link between the development and target computers. These transmissions share bandwidth with the Ethernet link.

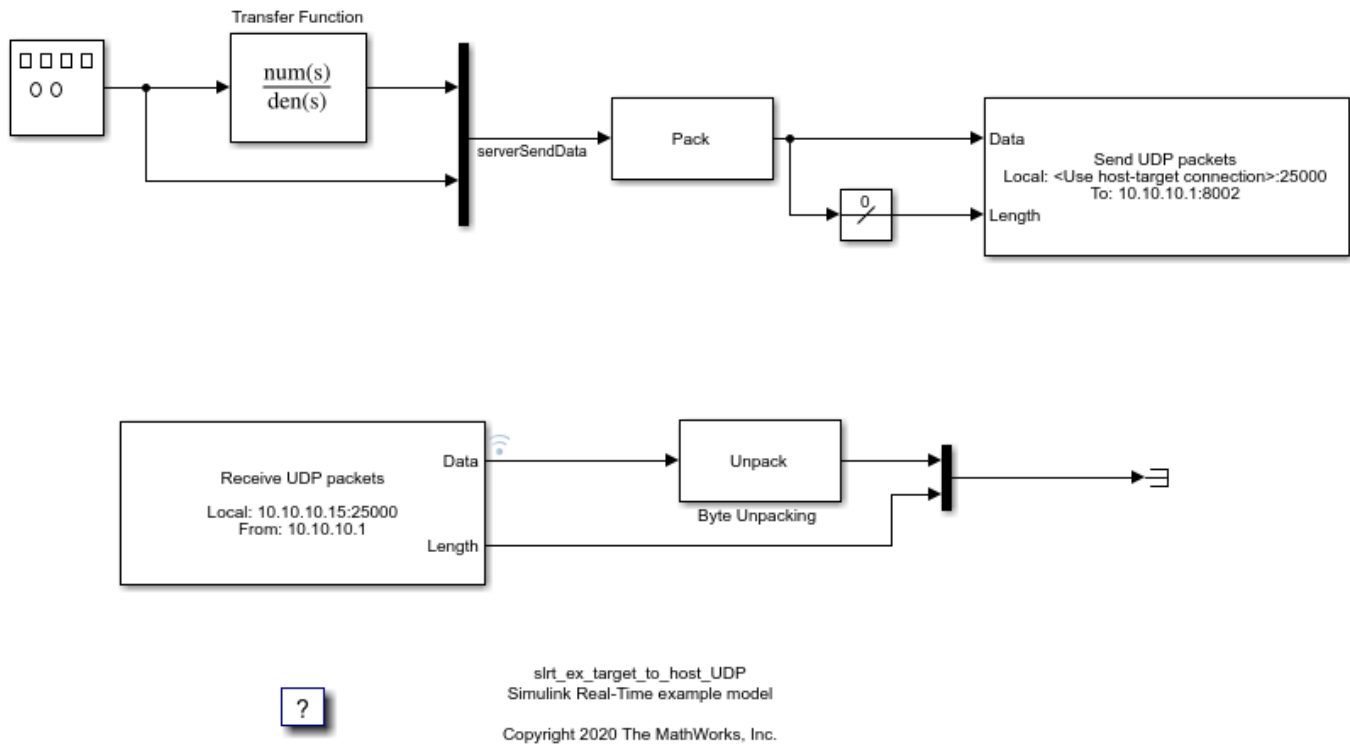
For more information about UDP and Simulink Real-Time, see “UDP Communication Setup”.

Open Model, Build, and Load Real-Time Application

This model drives a first order transfer function with a square wave signal and sends the transfer function input and output signals to the development computer using UDP. To open the model, in the MATLAB Command Window, type:

```
open_system(fullfile(matlabroot,'toolbox','slrealtime','examples','slrt_ex_target_to_host_UDP'));

model = 'slrt_ex_target_to_host_UDP';
mdlOpened = 0;
systems = find_system('type', 'block_diagram');
if ~any(strcmp(model, systems))
    mdlOpened = 1;
    open_system(fullfile(matlabroot,'toolbox','slrealtime','examples',model));
end
```

Build the model and download to the target computer.

- Configure for a non-Verbose build.
- Mark the Byte Unpacking block output for data logging.
- Build and download application.
- Open the Simulation Data Inspector.

This code shows how to mark signals programmatically for data logging. You can also mark signals for data logging in the Simulink Editor. You can view the logged data in in the Simulation Data Inspector.

```
set_param(model, 'RTWVerbose', 'off');
set_param(model, 'StopTime', '10');
targetIP = '192.168.7.5';
set_param([model, '/UDP Receive'], 'ipAddress', targetIP);
hostIP = '192.168.7.2';
set_param([model, '/UDP Send'], 'toAddress', hostIP);
set_param([model, '/UDP Receive'], 'fmAddress', hostIP);
handle = get_param([model, '/Byte Unpacking'], 'PortHandles');
Output = handle.Outputport(1);
Simulink.sdi.markSignalForStreaming(Output, 'on');
evalc('slbuild(model)');
tg = slrealtime;
load(tg, model);
```

Close the model if it is opened.

```
if (mdlOpened)
    bdclose(model);
end
```

Create UDP object in MATLAB on Development Computer

```
uByte = udpport("IPV4","LocalHost",hostIP,"LocalPort",8002);
```

Run Model on Target Computer

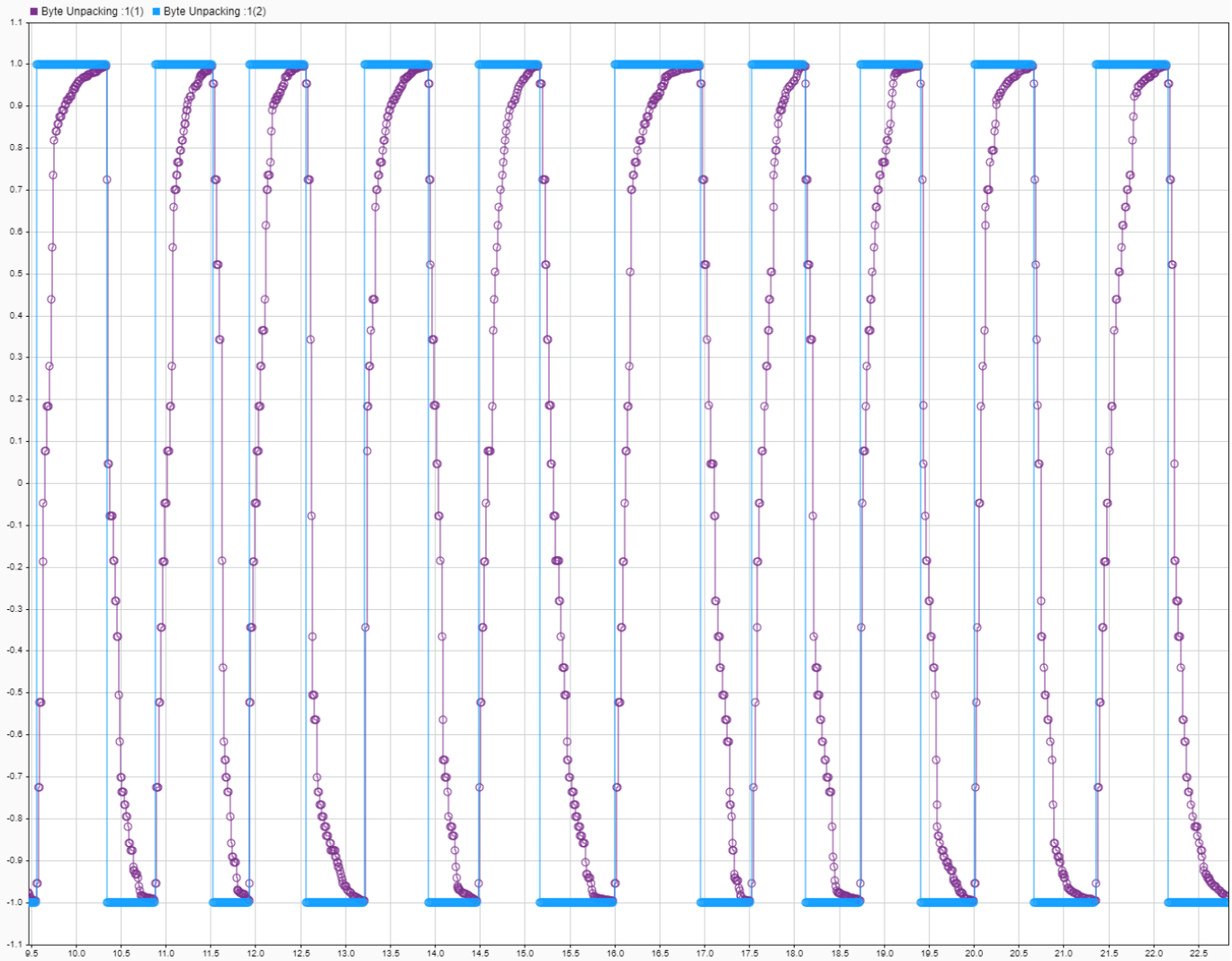
```
start(tg);
```

Read Data and Write Development Computer

```
tic;
while (toc<10)
    data = read(uByte,16);
    write(uByte,data,targetIP,25000);
    data = read(uByte,16);
end
```

View Signals in Simulation Data Inspector

```
Simulink.sdi.view;
```



Disconnect UDP Object on Development Computer

```
clear uByte;
```

Apply 802.1Q VLAN Tag by Using Ethernet Send and Receive Blocks

This example shows how to use Ethernet blocks to send and receive Ethernet packets on a target computer.

On the development computer, a UDP Send block sends a sample packet. On the target computer, this packet is received by an Ethernet Receive block, individual bytes in the payload are manipulated, and the resulting payload is sent out of the target computer by an Ethernet Send block.

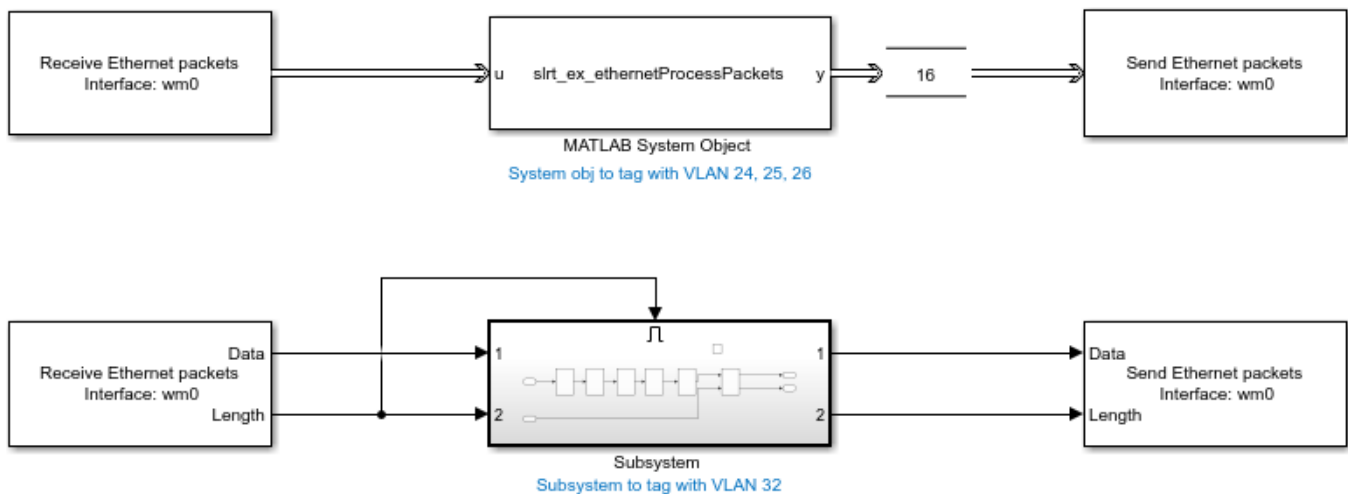
The Ethernet blocks work only on the target computer.

These blocks can work in the default signal input/output mode and a message input/output mode. Both modes are shown in this example.

Set Up Ethernet Send-Receive Model

Open the target model `slrt_ex_ethernetSendReceive`.

```
mdl1 = 'slrt_ex_ethernetSendReceive';
open_system(fullfile(matlabroot, 'toolbox', 'slrealtime', 'examples', mdl1));
```



Copyright 2021-2022 The MathWorks, Inc.

The target model requires a valid `Interface Name` parameter value in the two Receive blocks and the two Send blocks. You can obtain this information on the target computer by using the QNX Neutrino RTOS `ifconfig` command.

This example uses interface name `wm0` for the target IP address `'192.168.7.5'`.

Enter the interface name `wm0` into the four blocks in the model:

```
targetIface = 'wm0';
set_param('slrt_ex_ethernetSendReceive/Ethernet Receive', 'InterfaceName', targetIface)
```

```
set_param('slrt_ex_ethernetSendReceive/Ethernet Receive1', 'InterfaceName', targetIface)
set_param('slrt_ex_ethernetSendReceive/Ethernet Send', 'InterfaceName', targetIface)
set_param('slrt_ex_ethernetSendReceive/Ethernet Send1', 'InterfaceName', targetIface)
```

Operations in the Ethernet Send-Receive Real-Time Application

Every packet sent from the development computer to the target computer is received by each Ethernet Receive block on the target.

The two Receive and corresponding Send blocks demonstrate the operation in signal mode and message mode, where Simulink messages represent the packets.

In signal mode, 'slrt_ex_ethernetSendReceive/Subsystem ' uses Simulink blocks to add a 802.1Q VLAN tag 32 and send it back to the host at a port incremented by 1. If the original sender port was 8001, the loopback destination port is 8002.

In the messages mode, slrt_ex_ethernetSendReceive/MATLAB System Object receives the packets. For each packet it then creates three new packets with the VLAN tags 24, 25, and 26.

Manipulating the Ethernet Header

For details about Ethernet header structure, refer to the standards document for IEEE 802.1Q.

For details about the IPv4 header, refer to RFC 791 for Internet Protocol <https://datatracker.ietf.org/doc/html/rfc791>

These changes to the header occur in the Subsystem and the System object:

- 1 Switch source and destination MAC address: Swap bytes 1-6 with bytes 7-12.
- 2 Switch source and destination IP Address: Swap bytes 27-30 with bytes 31-34.
- 3 Switch source and destination port numbers: Swap bytes 35-36 with bytes 37-38.
- 4 Increment the new destination port number by 1: Add 1 to the value of byte 38.
- 5 Disable checksum verification: Set bytes 41-42 to 0. Without this change, the packets that are sent back to the development computer would be discarded, since checksum value would be incorrect. For details on checksum verification including recalculating new checksums, refer to RFC: 791 for Internet Protocol.
- 6 Add 802.1Q VLAN tag: IEEE 802.1Q adds a 4-byte VLAN tag between the Source/Destination MAC address and Length/Type fields of an Ethernet frame to identify the VLAN to which the frame belongs.

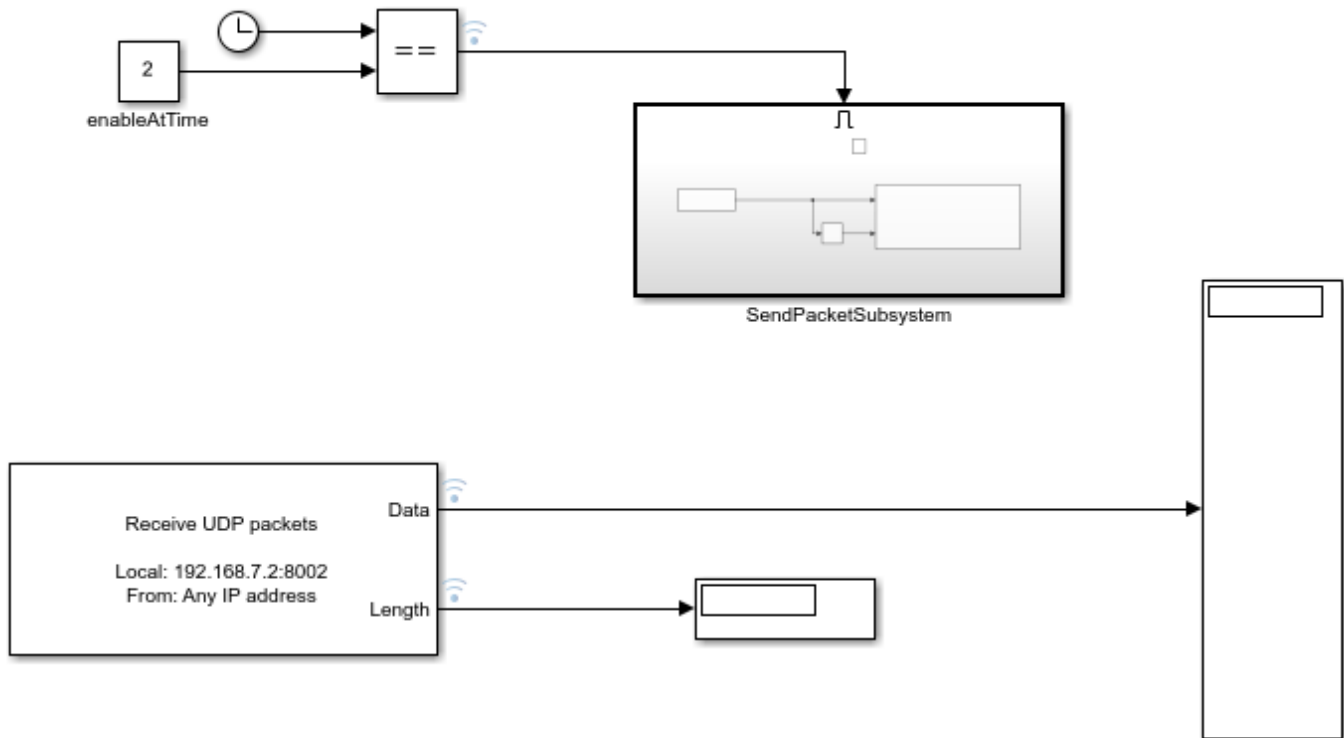
For the VLAN tags:

- 1 Make space for the VLAN tag by shifting bytes 13 onward to the right by 4 to the byte location starting at 17.
- 2 Insert the VLAN tag at byte locations 13-16. For example, to add tag 24, insert 0x81 0x00 0x00 0x18. Here the first 2 bytes correspond to a Tag protocol identifier (TPID), which is a 16-bit field set to a value of 0x8100 to identify the frame as an IEEE 802.1Q-tagged frame. The other 2 bytes set the Tag control information (TCI) to 0x0018, which includes the VLAN Identifier that corresponds to 24.

Open and Set Up Packet Source Model on Development Computer

Open model slrt_ex_udpsend. Set the IP address.

```
developmentIP = '192.168.7.2';
mdl2 = 'slrt_ex_udpsend';
open_system(fullfile(matlabroot, 'toolbox', 'slrealtime', 'examples', mdl2));
set_param('slrt_ex_udpsend/UDP Receive1', 'ipAddress', developmentIP)
```



Copyright 2021 The MathWorks, Inc.



This model uses UDP blocks to send one packet out when the execution time is 2 seconds.

Set the To IP address parameter on `slrt_ex_udpsend/SendPacketSubsystem/UDP Send` to the IP address of the target computer.

```
targetIP = '192.168.7.5';
set_param('slrt_ex_udpsend/SendPacketSubsystem/UDP Send', 'toAddress', targetIP)
```

Run Ethernet Send-Receive Real-Time Application on Target Computer

Run the target model on the target using the **Run on Target** button. Or, in the MATLAB Command Window, type:

```
evalc('slbuild(mdl1)');
tg = slrealtime;
load(tg,mdl1)
start(tg)
```

Simulate the UDP Send Model on the Development Computer

Simulate the model.

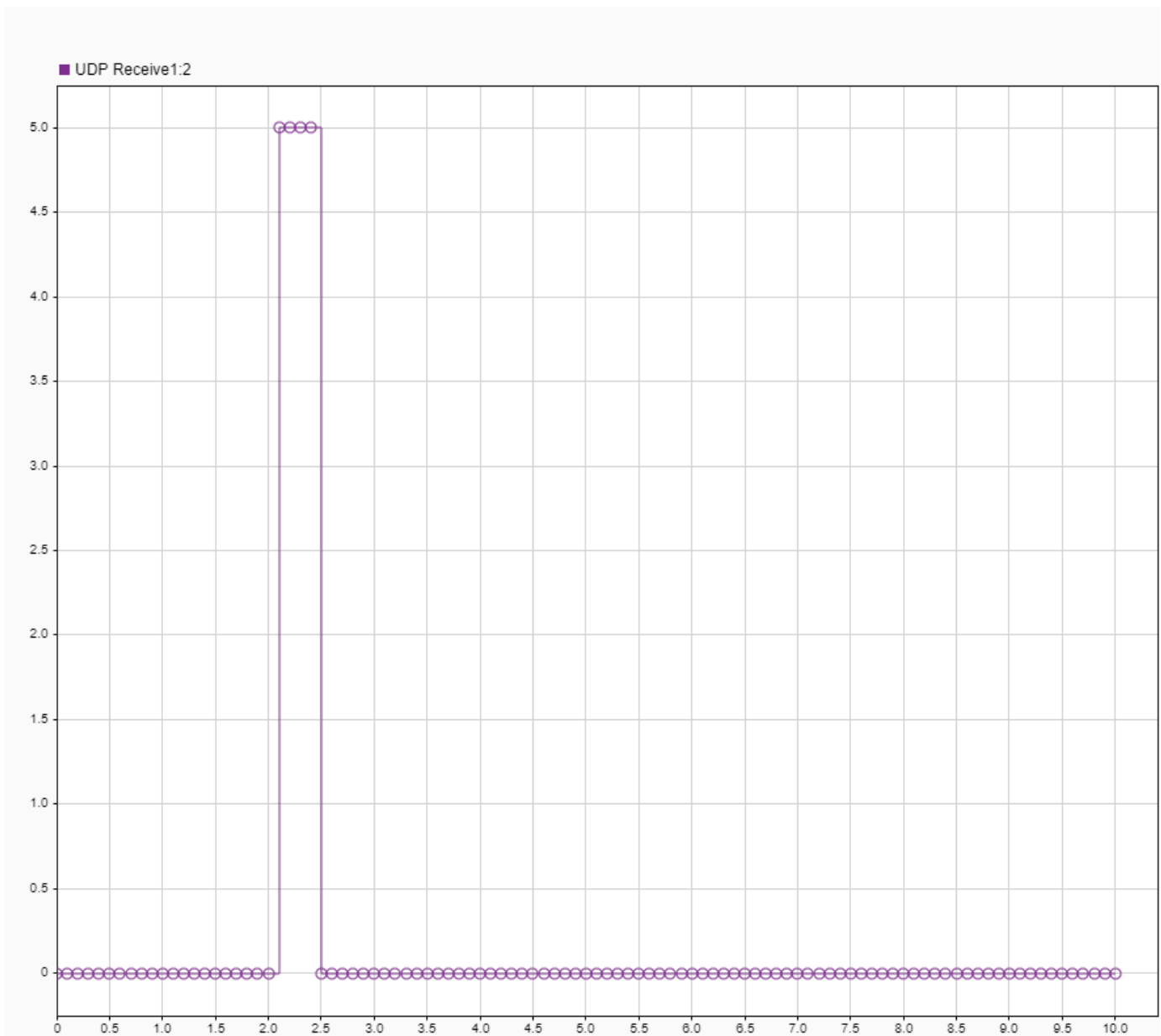
```
set_param('slrt_ex_udpsend', 'SimulationCommand', 'start')
```

Open Simulink Data Inspector

Open the Simulation Data Inspector and observe the new packets created on the target computer. In the MATLAB Command Window, type:

```
Simulink.sdi.view;
```

The Simulation Data Inspector shows that four packets are received by the UDP Receive block through four instances of the data length changing to five (the UDP packet size).



Every simulation sends out just one packet at time = 2 seconds.

The Ethernet send-receive real-time application responds with four packets, which contain the same payload but with VLAN tags 24, 25, 26, and 32.

Windows does not expose the VLAN tags to applications. Due to this using a packet capture program such as Wireshark does not show VLAN tags.

The development computer model shows four UDP packets were received. These are UDP blocks, and Ethernet header information is not output.

For Windows systems that have connections that block the VLAN tag (such as VM Ethernet connections or a network interface between the development and target computers), these connections may prevent the packets from appearing on the output display.

One way to see the tags is by using QNX Neutrino RTOS `tcpdump` command on the target computer while logged in as user `root` by using password `root`. Use the command:

```
tcpdump -i <targetIface> -v -e vlan
```

For `targetIface`, use the interface name `wm0` from the **Set Up Ethernet Send-Receive Model** section.

Close Models

```
bdclose('all');
```


Apply Simulink Real-Time Model Template to Create Real-Time Application

This example shows how to use the Simulink® Real-Time™ template to create a Simulink® model. Starting from the model template provides a new model that has configuration parameters set up for building a real-time application.

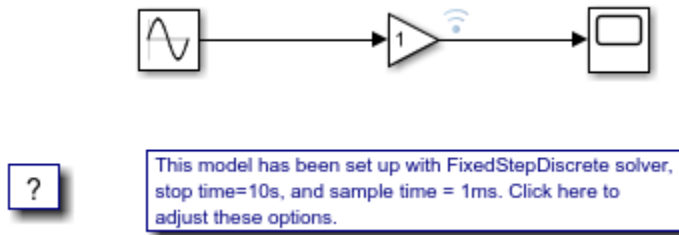
To see the Simulink Real-Time commands for each operation in this example, view the example code.

Create Simulink Model from Template

To create a Simulink model from the Simulink start page, in the MATLAB® Command Window, type:

```
simulink
```

Select the Simulink Real-Time template from the start page and create the `exampleSlrealtimeApp` model. Or, in the Command Window, use the `Simulink.createFromTemplate` command. See code for this script for full syntax.



Blocks, Connections, and Data Logging in the Model

The Simulink Real-Time model template contains a Gain block that connects a Signal Generator to a Scope block. The Gain block output is marked for logging with the Simulation Data Inspector (SDI).

Simulate Real-Time Application and View Logged Data

Build the real-time application, run it on the target computer, and view the logged data:

1. Make sure that the development computer has a connection to the target computer.
2. Build the model and download the real-time application to the target computer. On the **Real-Time** tab, click **Run on Target**. Or, use the `slbuild` command and the `load` command.
3. Run the real-time application and log data by using the **Run on Target** button.
4. Open the Simulation Data Inspector by double-clicking the Simulation Data Inspector icon on the Gain block output signal or by using the `Simulink.sdi.view` command.

More Information

- “Create and Run Real-Time Application from Simulink Model”
- “Configure and Control Real-Time Application by Using Simulink Real-Time Explorer”

- Simulation Data Inspector

Insert Event into Execution Profiling Stream

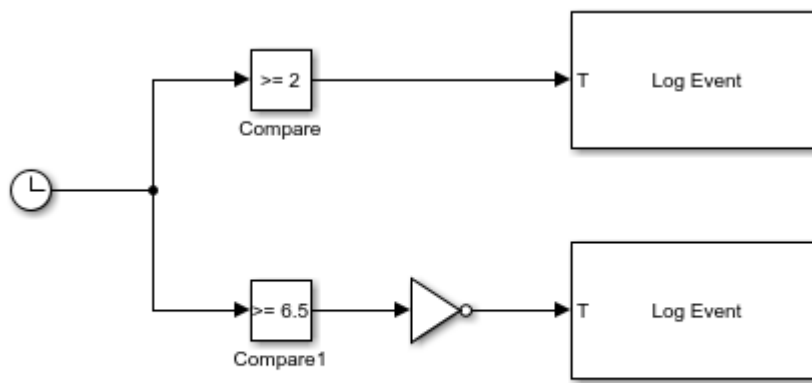
This example shows how to use the Log Event block to insert a user-defined event into the execution profiling event stream. For more information about execution profiling, see “Execution Profiling for Real-Time Applications” on page 10-7.

Open Model

To open the model, in the MATLAB Command Window, type:

```
open_system(fullfile(matlabroot,'toolbox','slrealtime','examples','slrt_ex_log_event'));

mdl = 'slrt_ex_log_event';
mdlOpened = 0;
systems = find_system('type', 'block_diagram');
if ~any(strcmp(mdl, systems))
    mdlOpened = 1;
    open_system(fullfile(matlabroot,'toolbox','slrealtime','examples',mdl));
end
```



Copyright 2020 The MathWorks, Inc.



Set Parameters to Measure Function Execution Times

Open the Configuration Parameters dialog box. Select **Code Generation > Verification**.

For **Measure function execution times**, select **Detailed (all function call sites)**. The **Measure task execution time** check box is checked and locked. Click OK.

Or, in the MATLAB command window, type:

```
set_param('slrt_ex_log_event','CodeProfilingInstrumentation','Detailed');
set_param('slrt_ex_log_event','StopTime','30');
```

Build and Load Real-Time Application

Build the model and download to the target computer.

```
evalc('slbuild mdl');
tg = slrealtime;
load(tg,mdl);
```

Profile Execution

Start the profiler and then execute the real-time application.

```
startProfiler(tg);
start(tg);
pause(20)
stopProfiler(tg);
stop(tg);
```

Display Execution Profile

Retrieve the Profiler data. Display the user-defined event in a table.

```
profiler_data = getProfilerData(tg);
profiler_data.EventTrace.etData
```

```
Processing data on target computer ...
Transferring data from target computer ...
Processing data on host computer ...
```

```
ans =
```

```
2x6 table
```

Channel	Timestamp	Event	Value	CPU	ModelTime
500	4391975917790188	200	200	3	2
1000	4391975917790188	100	700	1	6.5

The Execution Profile plot shows the allocation of execution cycles across the four processors, indicated by the colored horizontal bars. The model sections are listed in the Code Execution Profiling Report. The cores are indicated by the numbers underneath the bars.

Close Model

Close the model if it is opened.

```
if (mdlOpened)
    bdclose(mdl);
end
```

Control Real-Time Application by Using C# Code

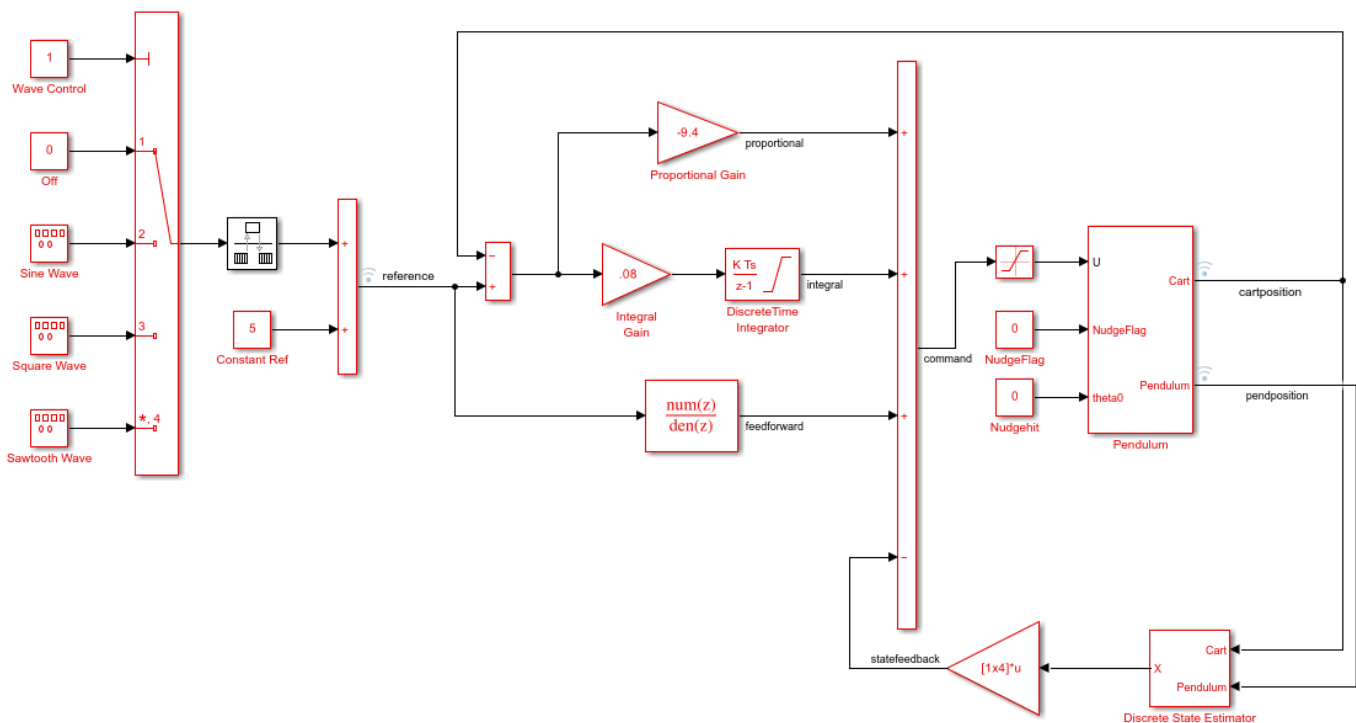
This example shows how to develop a C# program that controls a Simulink® Real-Time™ application by using the functions from the Simulink Real-Time XIL API support package. The C# example code shows how to use the XIL C# API calls to load, run, and stop a Simulink Real-Time application. The example code also shows how to record signal data.

Prepare for API in C# Program

1. Install the Simulink Real-Time Support Package for ASAM® XIL Standard by using the Add On Explorer.

2. Copy model `slrt_ex_pendulum_100Hz` to your working folder.

```
open_system(fullfile(matlabroot, 'toolbox/slrealtime/examples/slrt_ex_pendulum_100Hz.slx'));
```



Model `slrt_ex_pendulum_100Hz`
Simulink Real-Time example model

Copyright 2019-2021 The MathWorks, Inc.

3. Build model `slrt_ex_pendulum_100Hz`.

```
model = 'slrt_ex_pendulum_100Hz';  
evalc('slbuild(model)');
```

4. Create an XIL configuration file. This command uses the default Speedgoat® target machine IP address 192.168.7.5. Replace `full_file_path_to_MLDTX` with the full file path to the `slrt_ex_pendulum_100Hz.mldatx` file.

```
slrealtime.createPortConfigureFile("configFile.xml","192.168.7.5",'full_file_path_to_MLDTX');
```

Create C# Program

1. Open Visual Studio® 2019 and create a project for Console App (.NET core).
2. As project references in visual studio, add `ASAM.XIL.Implementation.Testbench.dll`, `ASAM.XIL.interfaces.dll`, and `MathWorks.ASAM.XIL.Server.dll`. These files are available after you install the support package.

Find `ASAM.XIL.Implementation.Testbench.dll` and `ASAM.XIL.Interfaces.dll` in folder `C:\Program Files (x86)\ASAM e.V\ASAM AE XIL API Standard Assemblies 2.1.0\bin`.

Find `MathWorks.ASAM.XIL.Server.dll` in folder `C:\ProgramData\MATLAB\SupportPackages\<release>\toolbox\slrealtime\xil\src\bin\win64`.

3. Copy the example C# program `myRealTimeAppController.cs` content to your current Visual Studio project. Update the project.

To find file `myRealTimeAppController.cs`, open this example and view the example folder.

4. Build the solution in your Visual Studio project.

Run the C# Program

1. Run your application at the operating system command prompt. Enter:

```
appName configFileFullPath csvFilePath
```

The parts of this command are:

- Application name
- Full file path to your configuration file
- Full file path of a CSV file in which the solution is saved

When you run the application, it loads and runs the Simulink Real-Time application `slrt_ex_pendulum_100Hz.mldtx` on the target computer. While running, the signal data for the signals `slrt_ex_pendulum_100Hz/Pendulum:1` and `slrt_ex_pendulum_100Hz:2` are recorded for about 3 seconds. The data is saved into the CSV file that you selected. When done, the application stops on the target computer.

2. Check signal data saved in the CSV file.

Run Real-Time Application by Using Python Script

This example shows how to call Simulink® Real-Time™ functions from a Python® script to build a real-time application from a model, load and run the application, tune parameter values, and capture signal data.

Set Up MATLAB Session for Python

To set up your MATLAB® session for this example:

1. Open MATLAB and install the MATLAB engine for Python. For more information, see “Call MATLAB from Python”.
2. Convert the MATLAB session into a shared session. In the Command Window, type `matlab.engine.shareEngine`. For more information, see `matlab.engine.shareEngine`.

Set Up Files and Run Python Script

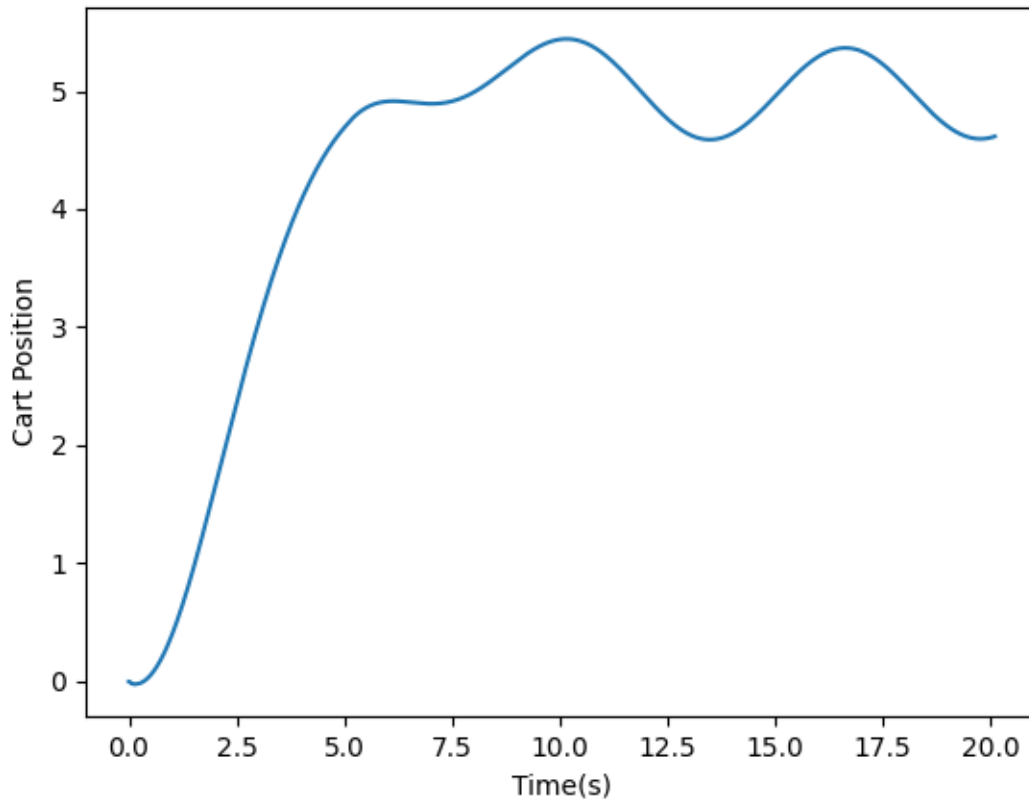
To set up files and run the Python script:

1. Copy model `matlabroot/toolbox/slrealtime/examples/slrt_ex_pendulum_100Hz.slx` to a working folder.
2. Copy Python script `CallingSlrealtimeFromPython.py` to the same working folder.
3. Open an operating system Command Prompt window and make the working folder the current folder for this window.

4. To run the Python script, at the command prompt, type:

```
py CallingSlrealtimeFromPython.py -m [.slx file path] -t [target name]
```

5. The Python script builds the model, runs the real-time application, and generates a plot of the captured signal data for cart position.



Line-by-Line Description of Python Script

The Python script uses the Simulink Real-Time function API to run the real-time application and capture the signal data.

Observe these points in the script:

- To call MATLAB commands from Python, import the `matlab.engine` module.

```
import matlab.engine
```

- Class `modelManagement` handles open and load of the Simulink model, then it builds the model.

```
class modelManagement():
```

- Class `targetManagement` is responsible for handling interactions with the `slrealtime.Target` object in MATLAB. For example, this class loads the application, starts the application, stops the application, gets and sets parameter values, and captures signal data.

```
class targetManagement():
```

- In the main function, the function tries to find all the active, shared MATLAB sessions and connects to the first one. If there is no session, the function opens a new MATLAB session.

```
engs = matlab.engine.find_matlab()
if not engs:
```



```
eng = matlab.engine.start_matlab()
```

```
else:
```

```
eng = matlab.engine.connect_matlab(eng[0])
```

- In the main function, the function instantiates a `modelManagement` object and builds the model.

```
mm = modelManagement(eng, modelFilePath)
```

```
appPath = mm.buildModel()
```

- In the main function, the function instantiates a `targetManagement` object for the given target name.

```
tg = targetManagement(eng, targetName)
```

- In the main function, the function loads the application on the target computer.

```
tg.load(appPath)
```

- In the main function, the function creates a `slrealtime.Instrument` object in MATLAB, adds a signal for cart position to the instrument object, adds this instrument object to the target object, and enables the `BufferData` mode. The live-streamed signal is saved in memory and waits for retrieval. For more information about buffered data mode, see `getBufferedData`.

```
blockPaths = ['slrt_ex_pendulum_100Hz/Pendulum']
```

```
portNumbers = [1]
```

```
tg.captureSignals(appPath, blockPaths, portNumbers)
```

- In the main function, the function starts to run the application on the target computer.

```
tg.start()
```

- In the main function, the function waits for 5 seconds, then sets the Wave Control block parameter `Value` to 2. This setting causes the cart to move in a sinusoidal pattern. The value is read again to make sure that the parameter value has been successfully updated.

```
tg.setparam('slrt_ex_pendulum_100Hz/Wave Control', 'Value', 2)
```

```
newValue = tg.getparam('slrt_ex_pendulum_100Hz/Wave Control', 'Value')
```

```
assert newValue == 2
```

- In the main function, the function waits for 15 seconds and stops the application.

```
time.sleep(15)
```

```
tg.stop()
```

- In the main function, the function retrieves the signal data and transfers the data back to Python.

```
[t, data] = tg.getCapturedSignals('slrt_ex_pendulum_100Hz/Pendulum', 1)
```

- In the main function, the function removes the previously added instrument object from the target object, leaving the target object in a clean state.

```
tg.removeInstrument()
```

- In the main function, the function plots the captured signal data against the time. In the resulting plot, you can see that the cart position is stabilized at 5 around 5 seconds, and then the cart starts to move in a sinusoidal pattern as expected after 5 seconds.

```
plt.plot(t, data)
```

```
plt.xlabel('Time(s)')
```

```
plt.ylabel('Cart Position')  
plt.show()
```

Hello World! Example External Code Integration for Simulink Real-Time

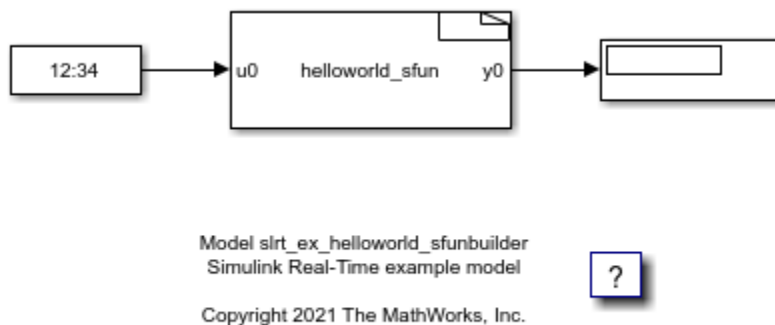
This example shows how to use an S-Function Builder block for external code integration. The example adds a hello message to the system log.

Before running this example, install the Simulink Real-Time Target Support Package. The support package includes the tools that compile the code that runs on the target computer.

Open the Model

Use the **Open Model** button to open the `slrt_ex_helloworld_sfunbuilder` model.

```
open_system(fullfile(matlabroot, 'toolbox', 'slrealtime', 'examples', 'slrt_ex_helloworld_sfunbuilder', 'slrt_ex_helloworld_sfunbuilder.slrtm'))
```



Open the S-Function Block

Double-click the `helloworld_sfun` S-Function block. The S-Function Builder opens and displays the S-function code.

```
/* Includes_BEGIN */
#ifdef SIMULINK_REAL_TIME
#include "slrt_log.hpp"
#endif
/* Includes_END */

/* Externs_BEGIN */
/* extern double func(double a); */
/* Externs_END */

void helloworld_sfun_Start_wrapper(SimStruct *S)
{
/* Start_BEGIN */

/* Start_END */
}

void helloworld_sfun_Outputs_wrapper(const real_T *u0,
                                     real_T *y0,
                                     SimStruct *S)
```

```
{
/* Output_BEGIN */
// Create custom message
static char hellormsg[100];
sprintf(hellormsg,"Hello World! t=%f \n",*u0);
// Use macros for platform dependent code
#ifdef SIMULINK_REAL_TIME
slrealtime::log_info(hellormsg);
#else
ssPrintf(hellormsg);
#endif

// Generic platform independent code
*y0 = *u0;
/* Output_END */
}

void helloworld_sfun_Terminate_wrapper(SimStruct *S)
{
/* Terminate_BEGIN */
/*
 * Custom Terminate code goes here.
 */
/* Terminate_END */
}
```

Build Model and Run Real-Time Application

Before building the model, you can run the model on your desktop and view the output message in the Simulink Real-Time system log viewer.

When you are ready to build the model, on the Simulink Editor **Real-Time** tab, connect to the target computer and click **Run on Target**. Or, in the MATLAB Command Window, type:

```
tg = slrealtime;
connect(tg);
model = 'slrt_ex_helloworld_sfunbuilder';
evalc('slbuild(model)');
load(tg,model);
start(tg);
pause(20);
stop(tg);
```

View Message in Status Log

Open the target computer status log and view the Hello World! message. On the Simulink Editor **Real-Time** tab, select **Prepare > SLRT Explorer**. Then, select the **System Log Viewer** tab. Or, in the MATLAB Command Window, type:

```
slrtLogViewer;
```

The viewer shows the Hello World! messages in the system log.

Timestamp	Message	Se...	Categ...
25-06-2021 22:22:15...	Loading model lamp	info	0
25-06-2021 22:22:15...	Ready to start	info	0
25-06-2021 22:22:17...	Starting model lamp	info	0
25-06-2021 22:22:34...	TET 0 avg: 1.467e-06 min: 1.301e-06 max: 2.209e-06	info	0
25-06-2021 22:22:34...	Stopping model lamp at 17s	info	0
27-06-2021 21:22:33...	Loading model helloworld	info	0
27-06-2021 21:22:33...	Ready to start	info	0
27-06-2021 21:22:40...	Starting model helloworld	info	0
27-06-2021 21:22:41...	Hello World! t=0.000000	info	100
27-06-2021 21:22:42...	Hello World! t=1.000000	info	100
27-06-2021 21:22:43...	Hello World! t=2.000000	info	100
27-06-2021 21:22:44...	Hello World! t=3.000000	info	100
27-06-2021 21:22:45...	Hello World! t=4.000000	info	100
27-06-2021 21:22:46...	Hello World! t=5.000000	info	100
27-06-2021 21:22:47...	Hello World! t=6.000000	info	100
27-06-2021 21:22:48...	Hello World! t=7.000000	info	100
27-06-2021 21:22:49...	Hello World! t=8.000000	info	100
27-06-2021 21:22:50...	Hello World! t=9.000000	info	100
27-06-2021 21:22:51...	Hello World! t=10.000000	info	100
27-06-2021 21:22:51...	Stopping model helloworld at 10s	info	0
27-06-2021 21:22:51...	TET 0 avg: 2.8876e-05 min: 2.7718e-05 max: 5.3294e-05	info	0

Close All Files

```
bdclose('all');
```

Control Color of Lamp on Instrument Panel

This example shows how to control the color of a lamp indicator on an instrument panel that connects to a Simulink Real-Time application.

The example operations are:

- Create uifigure, add lamps, and add labels
- Open model and build real-time application
- Connect lamps and add instrument
- Observe color cycle of lamps
- Remove instrument
- Close model

Create uifigure and Add Components

```
f = uifigure;
lamp1 = uilamp(f);
lamp1.Position = [10 300 20 20];
tlabel1 = uilabel(f);
tlabel1.Position = [40 298 100 22];
tlabel1.Text = 'Lamp 1';
lamp2 = uilamp(f);
lamp2.Position = [10 200 20 20];
tlabel2 = uilabel(f);
tlabel2.Position = [40 198 100 22];
tlabel2.Text = 'Lamp 2';
```



Build and Run Real-Time Application

```
open_system(fullfile(matlabroot,'toolbox','slrealtime','examples','slrt_ex_lamp_instrument'));
model = 'slrt_ex_lamp_instrument';
evalc('slbuild(model)');
tg = slrealtime;
load(tg,model);
start(tg);
pause(2);
inst = slrealtime.Instrument;
inst.connectScalar(lamp1, 'lamp1', 'Property', 'Color', 'Callback', @setLampColor);
inst.connectScalar(lamp2, 'lamp2', 'Property', 'Color');
addInstrument(tg,inst);
pause(10);
stop(tg);
removeInstrument(tg,inst);
bdclose('all');

function color = setLampColor(~,d)
    switch uint8(d)
        case 5
            color = 'green';
        case 4
            color = 'yellow';
        case 3
            color = 'cyan';
        case 2
```

```
        color = 'magenta';  
    case 1  
        color = 'red';  
    otherwise  
        color = 'white';  
    end  
end  
end
```

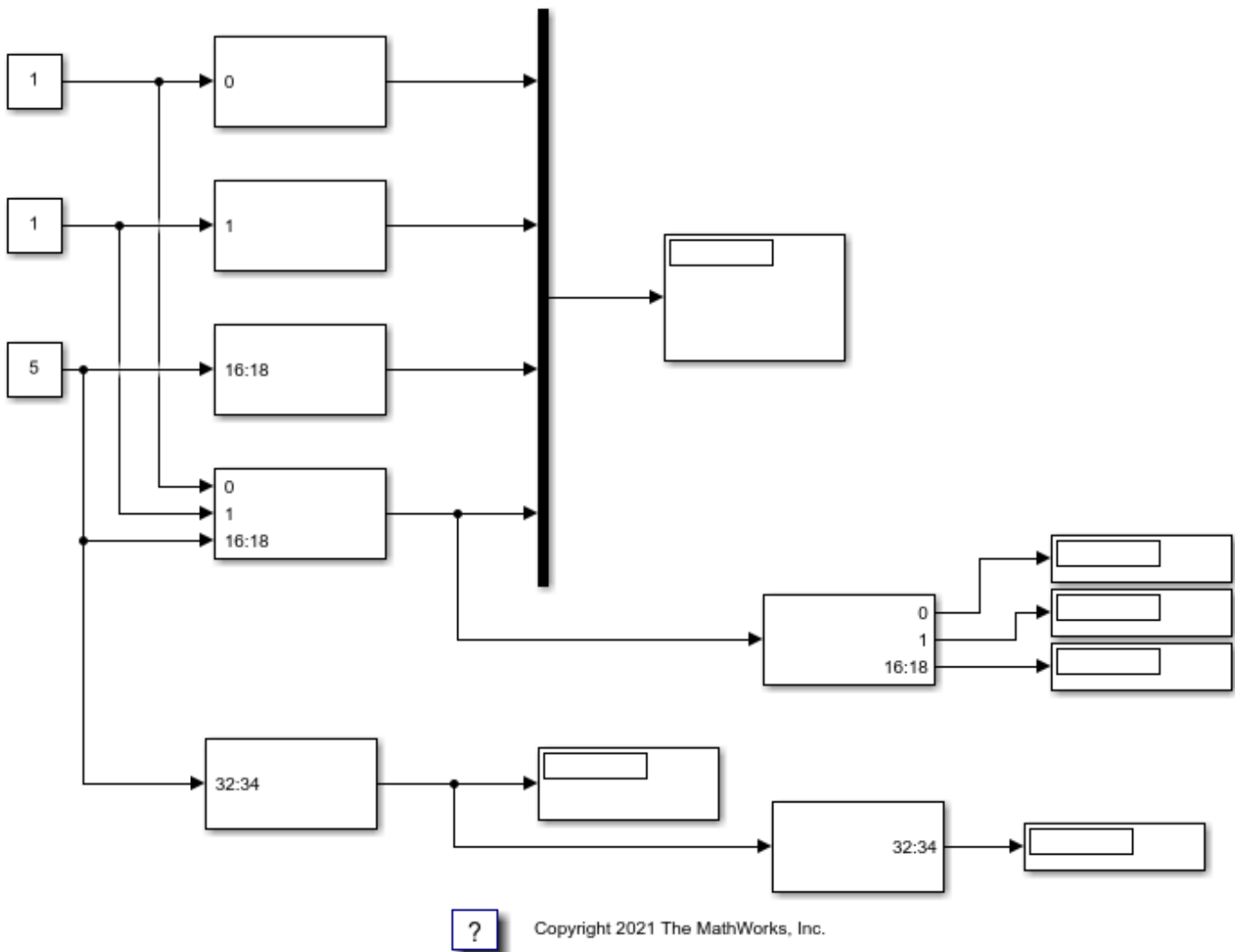

Configure Input and Output Ports for Bit Packing and Unpacking

This example shows how to configure Bit Packing blocks and Bit Unpacking blocks.

Open Model

The model uses Bit Packing, Bit Unpacking, and Display blocks to show how to use the input and output port configuration syntax. Open the model and double-click blocks to view the configuration.

```
model = 'slrt_ex_bit_pack_unpack';
open_system(fullfile(matlabroot, 'toolbox', 'slrealtime', 'examples', model));
```



Settings for Bit Packing and Bit Unpacking Blocks

To demonstrate the syntax for configuring block input and output ports, the model `slrt_ex_bit_pack_unpack` has a set of Bit Packing blocks and Bit Unpacking blocks that use these settings:

slrt_ex_bit_pack_unpack/Bit Packing

- Bit patterns: {[0]}
- Output port (packed) data type: uint32
- Output port (packed) dimensions: [1]

slrt_ex_bit_pack_unpack/Bit Packing1

- Bit patterns: {[1]}
- Output port (packed) data type: uint32
- Output port (packed) dimensions: [1]

slrt_ex_bit_pack_unpack/Bit Packing2

- Bit patterns: {[16:18]}
- Output port (packed) data type: uint32
- Output port (packed) dimensions: [1]

slrt_ex_bit_pack_unpack/Bit Packing3

- Bit patterns: {0, 1, [16:18]}
- Output port (packed) data type: uint32
- Output port (packed) dimensions: [1]

slrt_ex_bit_pack_unpack/Bit Packing4

- Bit patterns: {[32:34]}
- Output port (packed) data type: uint32
- Output port (packed) dimensions: [2]

slrt_ex_bit_pack_unpack/Bit Unpacking

- Bit patterns: {0, 1, [16:18]}
- Input port (packed) data type: uint32
- Input port (packed) dimensions: [1]
- Output port (unpacked) data types (cell array): {'uint8', 'uint16', 'uint32'}
- Output port (unpacked) dimensions (cell array): {1,1,1}
- Sign extended: 'On'

slrt_ex_bit_pack_unpack/Bit Unacking1

- Bit patterns: {[32:34]}
- Input port (packed) data type: uint32

- Input port (packed) dimensions: [2]
- Output port (unpacked) data types (cell array): {'uint8'}
- Output port (unpacked) dimensions (cell array): {1}
- Sign extended: 'On'

Build Model and Run Real-Time Application

To show the unpacked output in the Display blocks, build the model and run the real-time application.

```
evalc('slbuild(model)');  
tg = slrealtime;  
load(tg,model);  
start(tg);  
pause(10);  
stop(tg);
```

Close Model

```
bdclose('all');
```

Run Real-Time Simulation of Permanent Magnet Synchronous Motor

This example shows how to run a real-time simulation of a permanent magnet synchronous motor (PMSM) that is externally controlled at high switching frequency. The real-time application runs on a Speedgoat® target computer that has an IO334 I/O module installed. To open the models in this example and build the real-time application requires these products:

- MATLAB®
- Simulink®
- Simulink Coder™
- Simulink Real-Time™
- Simulink Real-Time Target Support Package
- Speedgoat I/O Blockset
- Speedgoat HDL Coder™ Integration Package
- Speedgoat IO334-325K I/O module with IO334-21 plugin

HDL Coder is required for design customizations.

You can use the Simulink Real-Time model and its corresponding bitstream file to simulate a permanent magnet synchronous motor system that is externally controlled in closed-loop at high switching frequency (100kHz) with sufficiently small time step (50 nanoseconds).

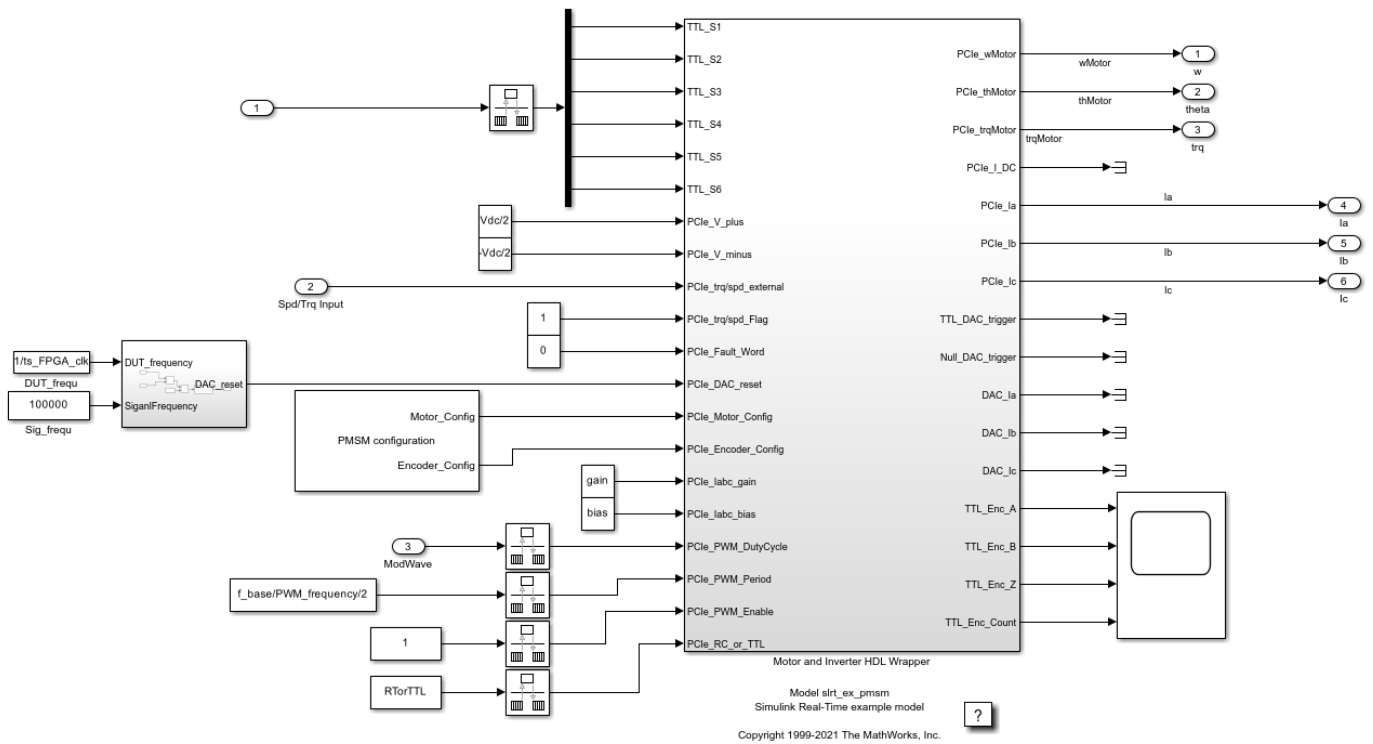
Simulink Model for Permanent Magnet Synchronous Motor System

The permanent Magnet synchronous motor system model is a physical system in Simulink. The system consists of three components, an inverter block, a permanent magnet synchronous motor block, and an incremental encoder block. The HDL-compatible Simulink implementation of the subcycle-averaging inverter block reads the PWM duty-cycle from "Gates" inport and DC voltage input from "V_plus" and "V_minus", and outputs the line voltages from output "Vll_abc". You can set its fault flag from Fault_Word port.

The permanent magnet synchronous motor block is an HDL Coder compatible implementation of a three-phase exterior permanent magnet synchronous motor (PMSM) with sinusoidal back electromotive force. It takes the input line voltage from the subcycle-averaging inverter block and outputs shaft angular velocity, shaft angular position, three phase currents, and motor torque. This block can also take an external speed or torque as input.

The Simulink implementation of quadrature encoder converts the angular position of the motor shaft to digital pulses. To see more details, open the model:

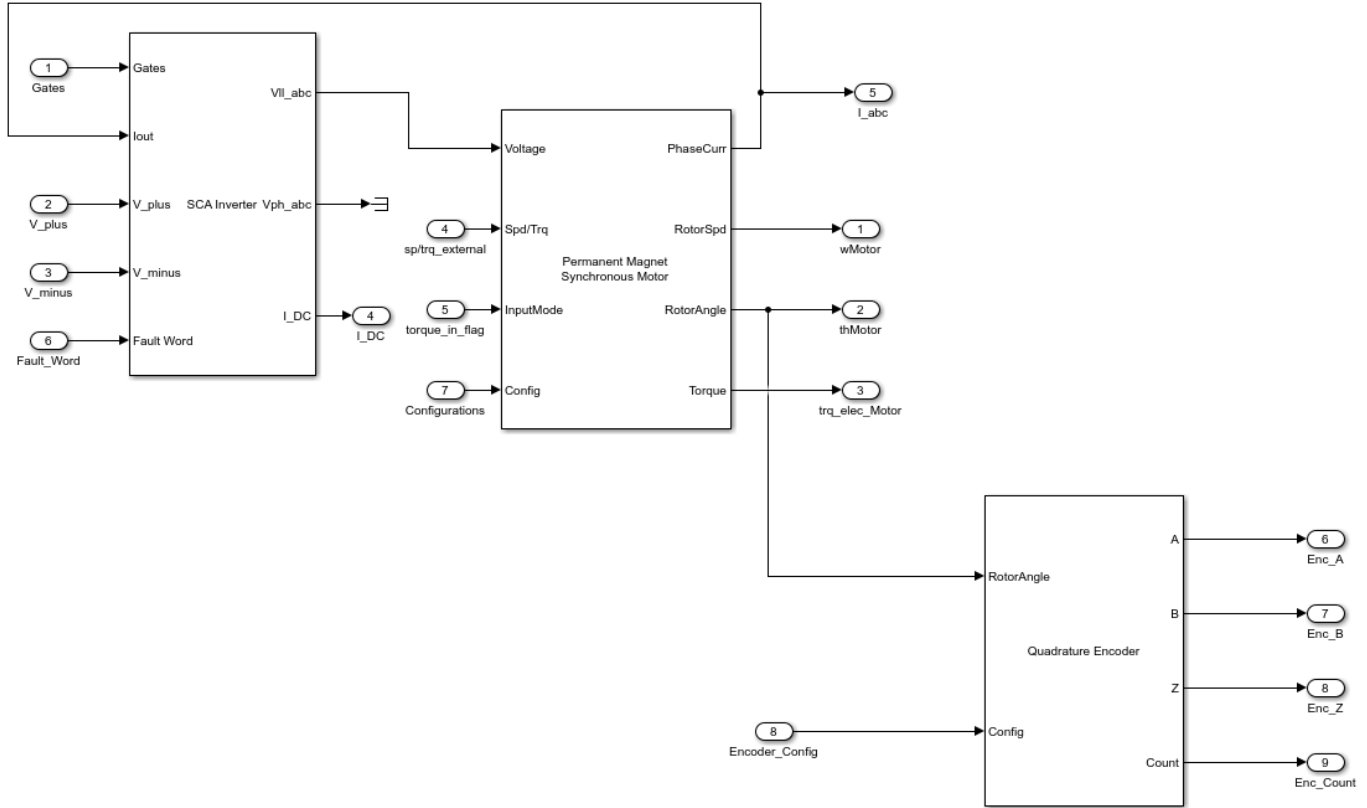
```
model = 'slrt_ex_pmsm';  
open_system(fullfile(matlabroot, 'toolbox', 'slrealtime', 'examples', model));
```



```
set_param(model, 'SimulationCommand', 'Update');
```

```
load_system('slrt_ex_pmsm')
```

```
open_system('slrt_ex_pmsm/Motor and Inverter HDL Wrapper/Motor and Inverter Mathematical Models')
```



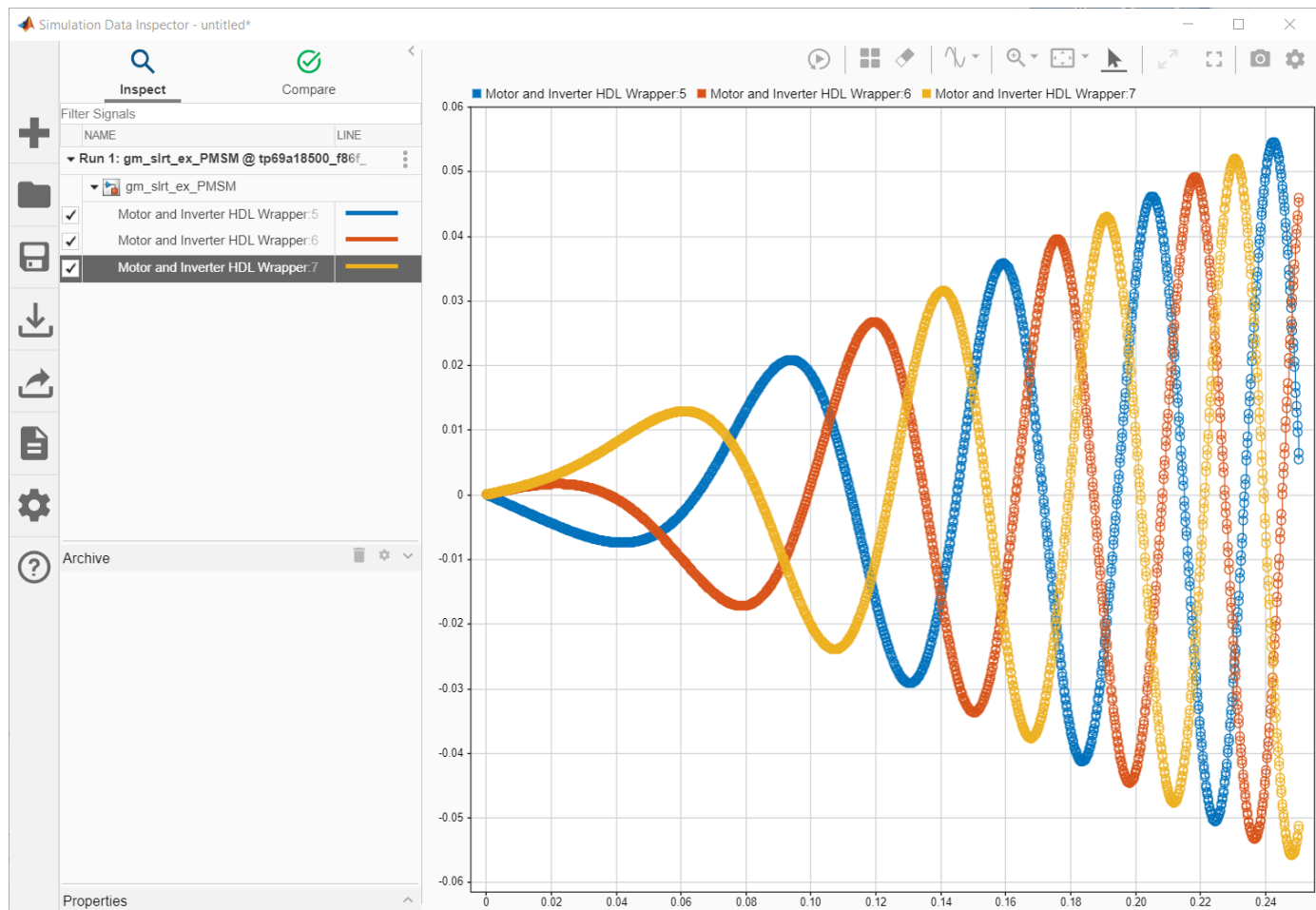
This model verifies the three-phase output currents and encoder's digital signals resulting from the input torque signal. To see how the model works, simulate the model.

```
sim(model);
```

Real-Time Simulation

This example also provides the corresponding Simulink Real-Time interface model and the bitstream generated by using the HDL Workflow Advisor, which you can download to a Speedgoat FPGA I/O 334-21 target.

```
model_gm = 'slrt_ex_pmsm_gm';
open_system(fullfile(matlabroot, 'toolbox', 'slrealtime', 'examples', model_gm));
```

More Information

The FPGA bitstream file `MotorAndInverterHDLWrapper.mcs` is available in the Simulink Real-Time examples folder.

```
cd(fullfile(matlabroot,'toolbox','slrealtime','examples'))
```

Close All Open Files

```
bdclose all;
```


Apply Persistent Variables in Real-Time Applications

This example shows how to apply persistent variables in real-time applications.

In a model, you save variables on target computer whose values persist when the real-time application stops and even when the target computer is shut down by using **Persistent Variable Write** blocks. At real-time application start, you direct the real-time application to read these persistent variables by using **Persistent Variable Read** blocks.

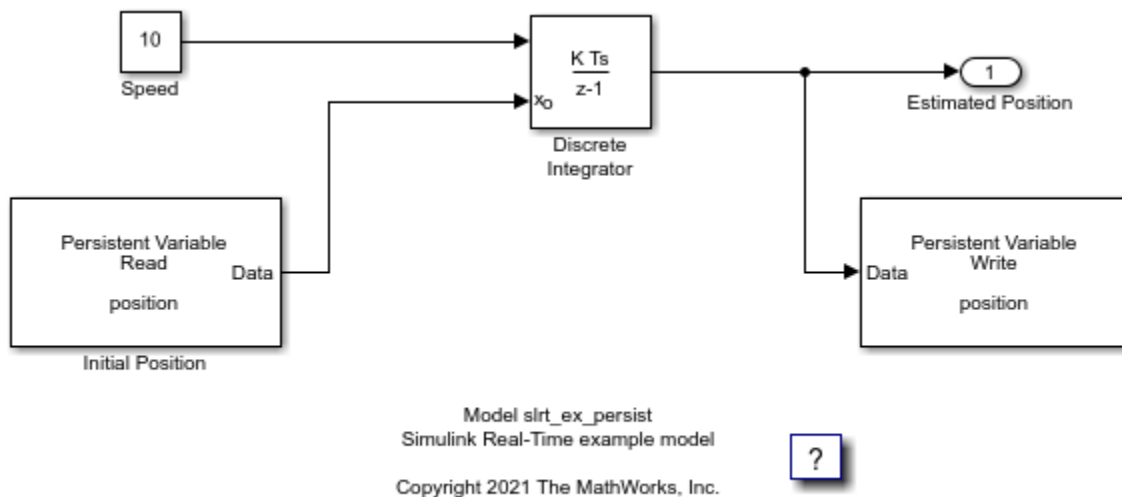
In MATLAB®, you can get or set the values of persistent variables on the target computer by using the `getPersistentVariables` function and the `setPersistentVariables` function.

Examine Persistent Variables in the Model

The model `slrt_ex_persist` computes the final position by using information from the `speed` input and `initial position` input. The Persistent Variable Read block variable `position` provides the initial value for the first run of the application from variable default value when the variable `position` does not exist on target computer. When the application stops, the Persistent Variable Write block variable `position` stores the final position from the run. The real-time application uses this value as the initial position for the next run.

To open the model and examine the Persistent Variable block values, in the Command Window, type:

```
model = 'slrt_ex_persist';
open_system(fullfile(matlabroot, 'toolbox', 'slrealtime', 'examples', model));
```



Build and Load the Real-Time Application

To build and load the real-time application, in the Command Window, type:

```
tg = slrealtime;
connect(tg);
evalc('slbuild(model)');
load(tg,model);
```

Initial Value of Persistent Variable

The initial value of the persistent variable `position` when it does not exist on target computer is 0.

The value is set by the Default value parameter of the Persistent Variable Read block.

Observe How Persistent Variable Changes in First Run

To run the real-time application, in the Command Window, type:

```
start(tg);
```

Check the value of the persistent variable after the stop time expires and the real-time application stops.

```
pause(5);  
myPersistVars = getPersistentVariables(tg)
```

```
myPersistVars =  
    struct with fields:  
        position: 20
```

Observe How Persistent Variable Changes in Second Run

To run the real-time application, in the Command Window, type:

```
load(tg,model);  
start(tg);
```

Check the value of the persistent variable after the stop time expires and the real-time application stops.

```
pause(5);  
myPersistVars = getPersistentVariables(tg)
```

```
myPersistVars =  
    struct with fields:  
        position: 40
```

Clear the Persistent Variable Values

Because the persistent variable values remain on the target computer after the real-time application stops, you must clear the retained values if the retained values are not needed. These steps show a way to clear the `position` persistent variable values.

```
myNewPersistVars = rmfield(myPersistVars,'position');  
setPersistentVariables(tg,myNewPersistVars);  
myPersistVars = getPersistentVariables(tg)
```

```
myPersistVars =
```

```
[]
```

You can also remove all persistent variable values by using this command.

```
setPersistentVariables(tg, []);
```

Preserve Persistent Variable Data by Safe Shutdown of Target Computer

The previous steps demonstrate how Persistent Variable values are stored when the real-time application stops and are reloaded when the real-time application starts. These variables are also retained when the target computer is shut down.

Target computers can handle being shut down by turning off power to the computer, but using this approach is not the best practice for the target computer. Also, if you just turn off the target computer while the real-time application is running, you can lose the last few seconds of data for the Persistent Variables.

To preserve all persistent variable data and safely shut down the target computer:

- 1.** On the target computer, stop the real-time application (for example, `stop(tg)`). The values for persistent variables are stored.
- 2.** Open a system terminal window.
- 3.** On the development computer, for user `slrt` and target computer IP address `192.168.7.5`, type command: `ssh slrt@192.168.7.5`
- 4.** Complete the login with password: `slrt`
- 5.** At the target computer system prompt that appears in the terminal window, shut down the target computer by using QNX Neutrino command: `shutdown -S system`
- 6.** After the shutdown command runs, you can safely turn off power to the target computer.

For more shutdown command info, see `shutdown` in the QNX Neutrino documentation.

Close All Files

```
bdclose('all');
```

Communicate with Data Distribution Service (DDS) Middleware

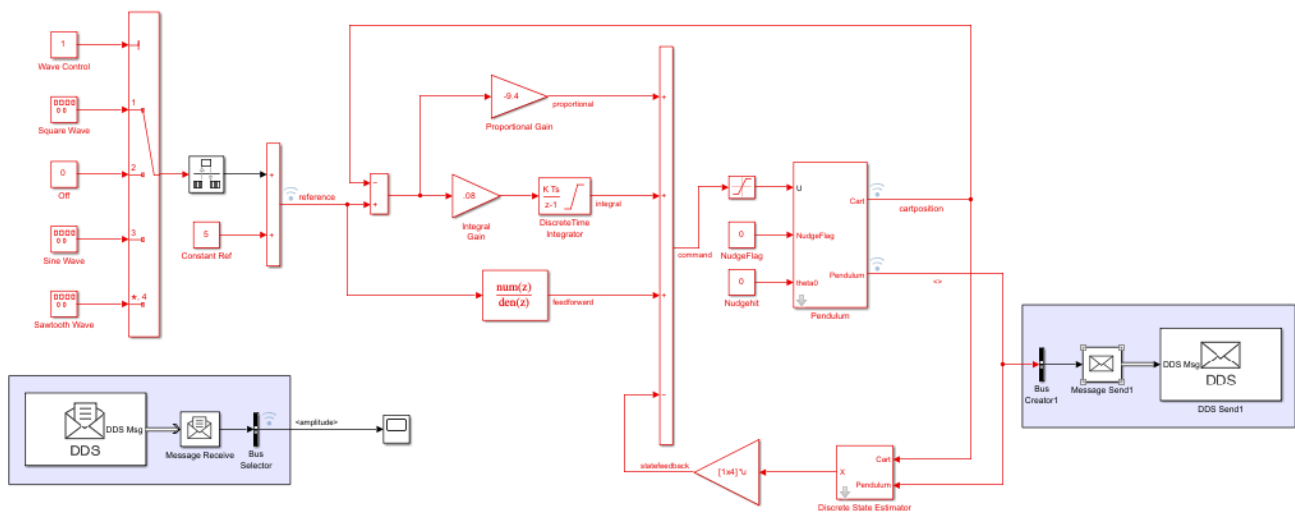
This example shows how to operate communications between a Data Distribution Service (DDS) middleware application on the development computer and a real-time application on the target computer. The Simulink® Real-Time™ DDS Send block and DDS Receive block pass messages between the middleware application and the real-time application. In this example, the DDS Send and DDS Receive blocks send and receive data on the application deployed on target computer.

Build the `slrt_ex_pendulum_100Hz_dds` model, and deploy the application on a target computer. Start running the application, and observe the data from DDS Send block is received by the DDS Receive block by using the Simulink Data Inspector.

Open Example Model

Open the model. In the Command Window, type

```
open_system('slrt_ex_pendulum_100Hz_dds');
```



Model `slrt_ex_pendulum_100Hz_dds`
Simulink Real-Time example model

Copyright 2019-2022 The MathWorks, Inc.

Build Model and Download to Target Computer

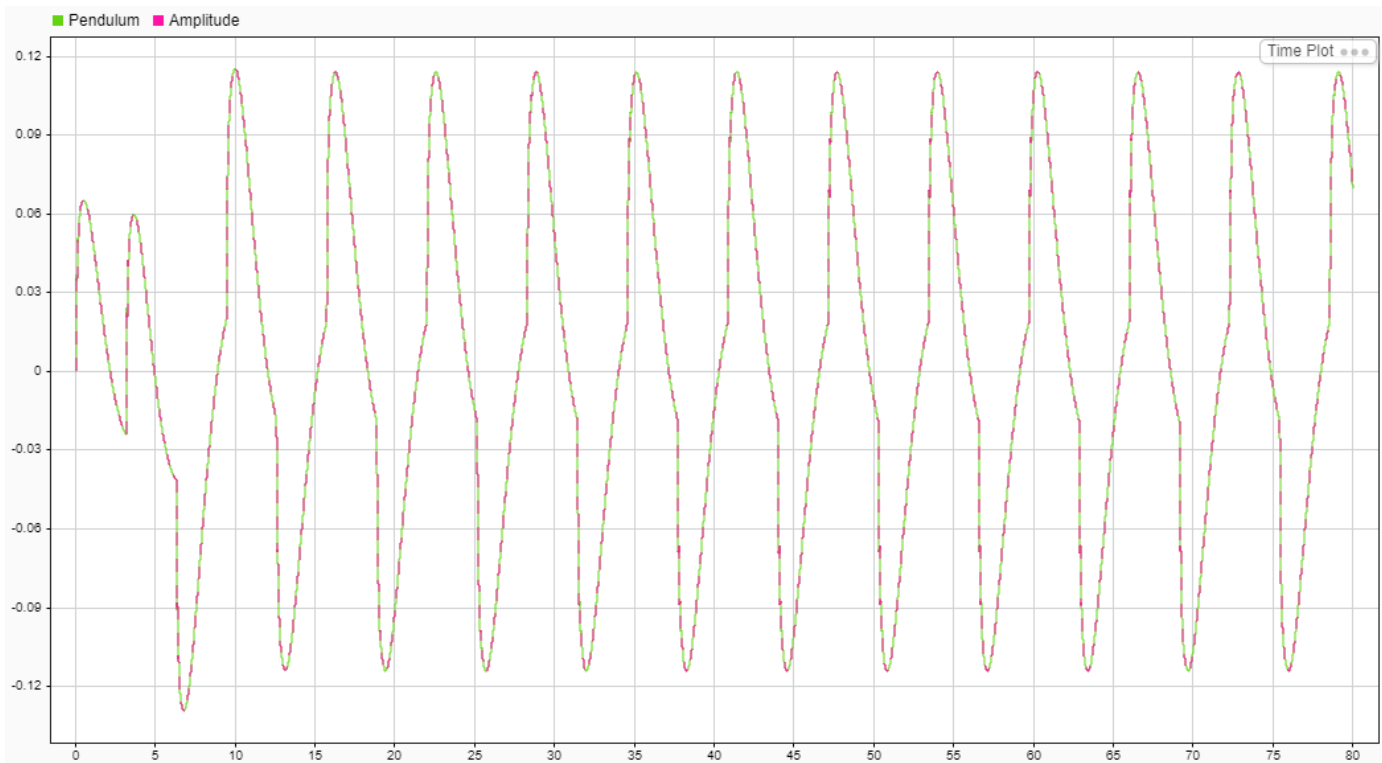
To build the model and download the real-time application, type:

```
tg=slrealtime('TargetPC1');
evalc('slbuild(''slrt_ex_pendulum_100Hz_dds'')');
load(tg,'slrt_ex_pendulum_100Hz_dds');
start(tg);
```

View Signals From DDS Send and DDS Receive Blocks

To view the signals, open the Simulation Data Inspector. In the Command Window, type:

```
Simulink.sdi.view();
```



Close the App and Models

Close the Simulation Data Inspector. In the Command Window, type:

```
Simulink.sdi.close;
```

Close the open models. In the Command Window, type:

```
bdclose ('all');
```


Troubleshooting

Solutions have been worked out for some common errors and problems that can occur when you are using Simulink Real-Time software. For more information, see “Find Simulink Real-Time Support” on page 17-35.

- “Troubleshooting Basics” on page 17-2
- “Troubleshoot Missing Real-Time Tab” on page 17-4
- “Troubleshoot Communication Failure Through Firewall (Windows)” on page 17-5
- “Troubleshoot Cannot Load Shared Object on Target Computer” on page 17-13
- “Troubleshoot Signal Data Logging from Nonvirtual Bus, Fixed-Point, and Multidimensional Signals” on page 17-15
- “Troubleshoot Signal Data Logging from Inport in Referenced Model” on page 17-17
- “Troubleshoot Signal Data Logging from Inport in Referenced Model in Test Harness” on page 17-19
- “Troubleshoot Signal Data Logging from Send and Receive Blocks” on page 17-21
- “Troubleshoot Signals for Streaming or File Logging” on page 17-22
- “Troubleshoot Folder Names with Spaces or Special Characters Halt Model Builds” on page 17-23
- “Troubleshoot Model Links to Static Libraries or Shared Objects” on page 17-24
- “Troubleshoot Build Error for Accelerator Mode” on page 17-26
- “Troubleshoot Long Build Times for Real-Time Application” on page 17-27
- “Troubleshoot Working with Persistent Variables” on page 17-29
- “Troubleshoot Unsatisfactory Real-Time Performance” on page 17-30
- “Troubleshoot Overloaded CPU from Executing Real-Time Application” on page 17-32
- “Troubleshoot Gaps in Streamed Data” on page 17-34
- “Find Simulink Real-Time Support” on page 17-35
- “Install Simulink Real-Time Software Updates” on page 17-36

Troubleshooting Basics

For questions or issues about your installation of the Simulink Real-Time product, refer to these guidelines and tips.

For more specific troubleshooting solutions, go to the MathWorks® Support website MathWorks Help Center website. The troubleshooting suggestions address these areas:

- Troubleshooting System Configuration
 - “Troubleshoot Communication Failure Through Firewall (Windows)” on page 17-5
 - “Troubleshoot Cannot Load Shared Object on Target Computer” on page 17-13
 - “Troubleshoot Vector CANape Operation” on page 4-13
 - “Troubleshoot ETAS Inca Operation” on page 4-17
 - “Troubleshoot System Upgrade for R2020b”
- Troubleshooting Model Preparation
 - “Troubleshoot Missing Real-Time Tab” on page 17-4
 - “Troubleshoot Folder Names with Spaces or Special Characters Halt Model Builds” on page 17-23
 - “Troubleshoot Model Links to Static Libraries or Shared Objects” on page 17-24
 - “Troubleshoot Build Error for Accelerator Mode” on page 17-26
 - “Troubleshoot Long Build Times for Real-Time Application” on page 17-27
 - “Troubleshoot Working with Persistent Variables” on page 17-29
 - “Troubleshoot Model Upgrade for R2020b”
 - “Troubleshoot S-Function Build Upgrade for R2020b”
- Troubleshooting Control and Instrumentation
 - “Troubleshoot Parameters Not Accessible by Name” on page 7-72
 - “Troubleshoot Signals Not Accessible by Name” on page 7-70
 - “Troubleshoot Signal Data Logging from Nonvirtual Bus, Fixed-Point, and Multidimensional Signals” on page 17-15
 - “Troubleshoot Signal Data Logging from Inport in Referenced Model” on page 17-17
 - “Troubleshoot Signal Data Logging from Inport in Referenced Model in Test Harness” on page 17-19
 - “Troubleshoot Signal Data Logging from Send and Receive Blocks” on page 17-21
 - “Troubleshoot Signals for Streaming or File Logging” on page 17-22
- Troubleshooting Performance Optimization
 - “Troubleshoot Unsatisfactory Real-Time Performance” on page 17-30
 - “Troubleshoot Overloaded CPU from Executing Real-Time Application” on page 17-32
 - “Troubleshoot Gaps in Streamed Data” on page 17-34
 - “Troubleshoot System Upgrade for R2020b”
 - “Troubleshoot Model Upgrade for R2020b”
 - “Troubleshoot MATLAB API Call Upgrade for R2020b”

- More Troubleshooting: Simulink Real-Time Support
 - “Find Simulink Real-Time Support” on page 17-35
 - “Install Simulink Real-Time Software Updates” on page 17-36

Troubleshoot Missing Real-Time Tab

Where is the **Real-Time** tab? I do not see this tab in the Simulink editor.

What This Issue Means

From the model configuration, the Simulink editor determines which tabs to display. The editor displays the **Real-Time** tab for models that are configured for Simulink Real-Time.

Try This Workaround

To configure your model for Simulink Real-Time, in Simulink Editor, from the **Apps** tab, click **Simulink Real-Time**.

This operation changes the code generation target to `slrealtime.tlc` for the model. After changing the configuration, the Simulink editor displays the **Real-Time** tab for the model.

See Also

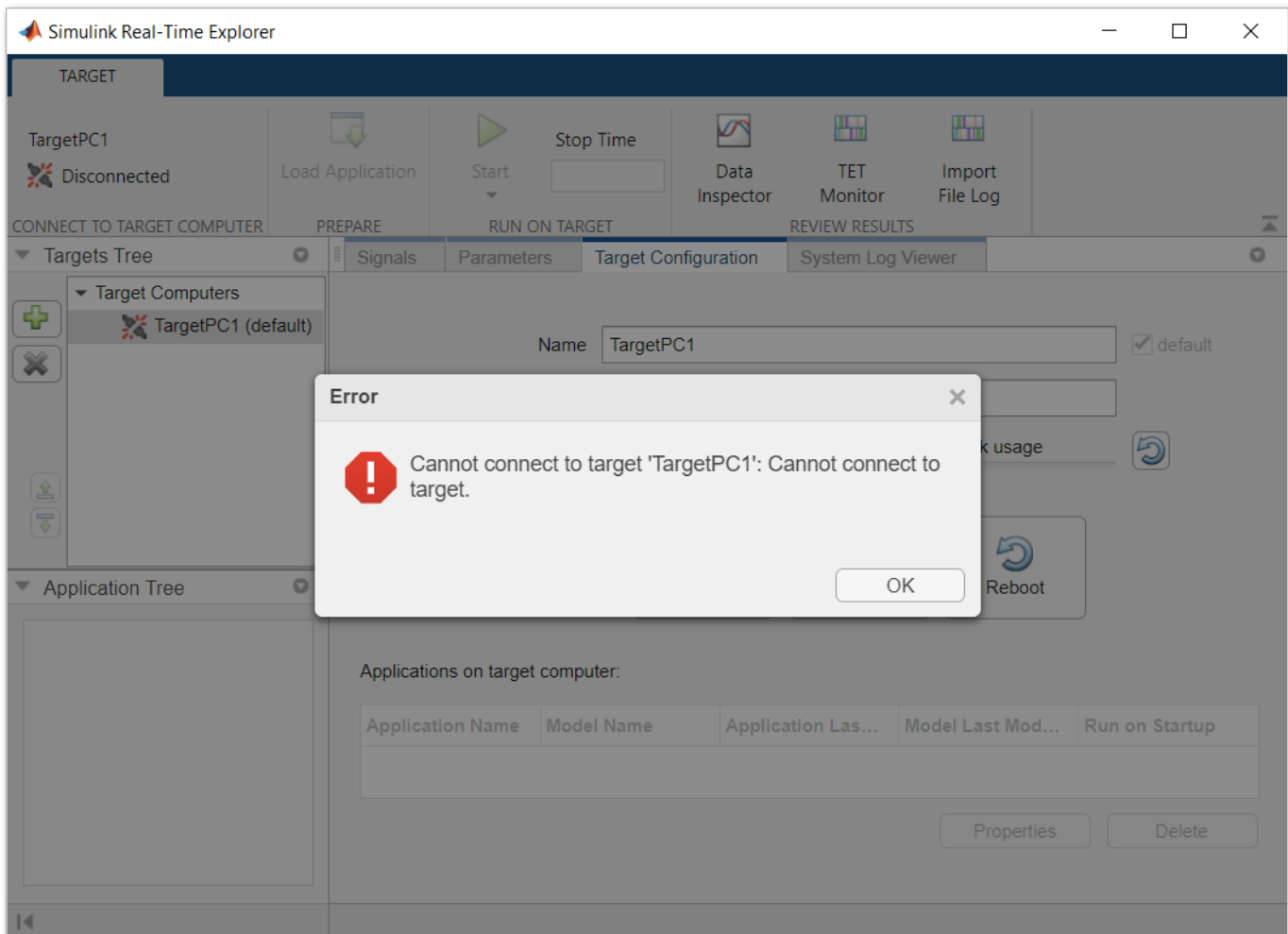
More About

- “Create and Run Real-Time Application from Simulink Model”

Troubleshoot Communication Failure Through Firewall (Windows)

When I attempt to connect to the target computer by using Simulink Real-Time Explorer, I see this error message.

Error: Cannot connect to target 'TargetPC1': Cannot connect to target.



Even though the connection fails, clicking on the **Update** button or **Reboot** button in Simulink Real-Time Explorer works. These operations indicate that the target computer can be reached through the Ethernet port.

What This Issue Means

In R2020b and later releases, Simulink Real-Time uses a protocol for the development-to-target computer connection that is blocked by default in Windows Defender Firewall for networks classified as Public. Windows also classifies all Ethernet connections as Public by default.

If you do not select the correct options when first running MATLAB, it can be possible to ping, update, and reboot the target computer from MATLAB. But, these incorrect option selections prevent

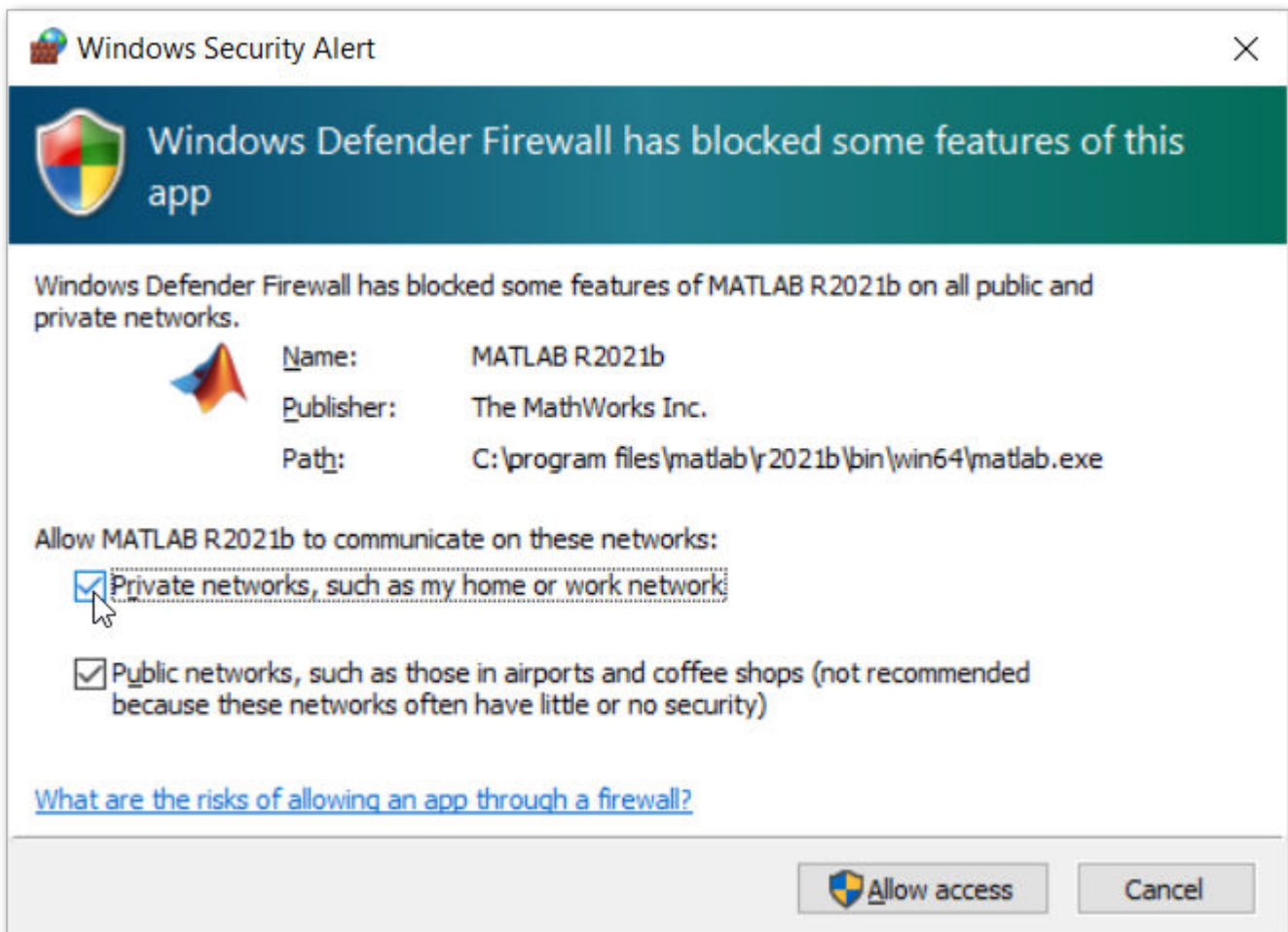
communication connection to the target computer. This communication connection is necessary to load and run real-time applications on the target computer.

Try These Workarounds

Resolve this issue by allowing MATLAB to communicate on all types of networks. Apply this setting when prompted on first connection or apply this setting later through the Windows Defender Firewall **Allow an app through Windows Firewall** selection. If that is not possible due to privilege restrictions, the issue can also be resolved by changing the classification of the Ethernet interface used for development-to-target computer connection from Public to Private.

Allow MATLAB for Public and Private Networks by Using Prompt

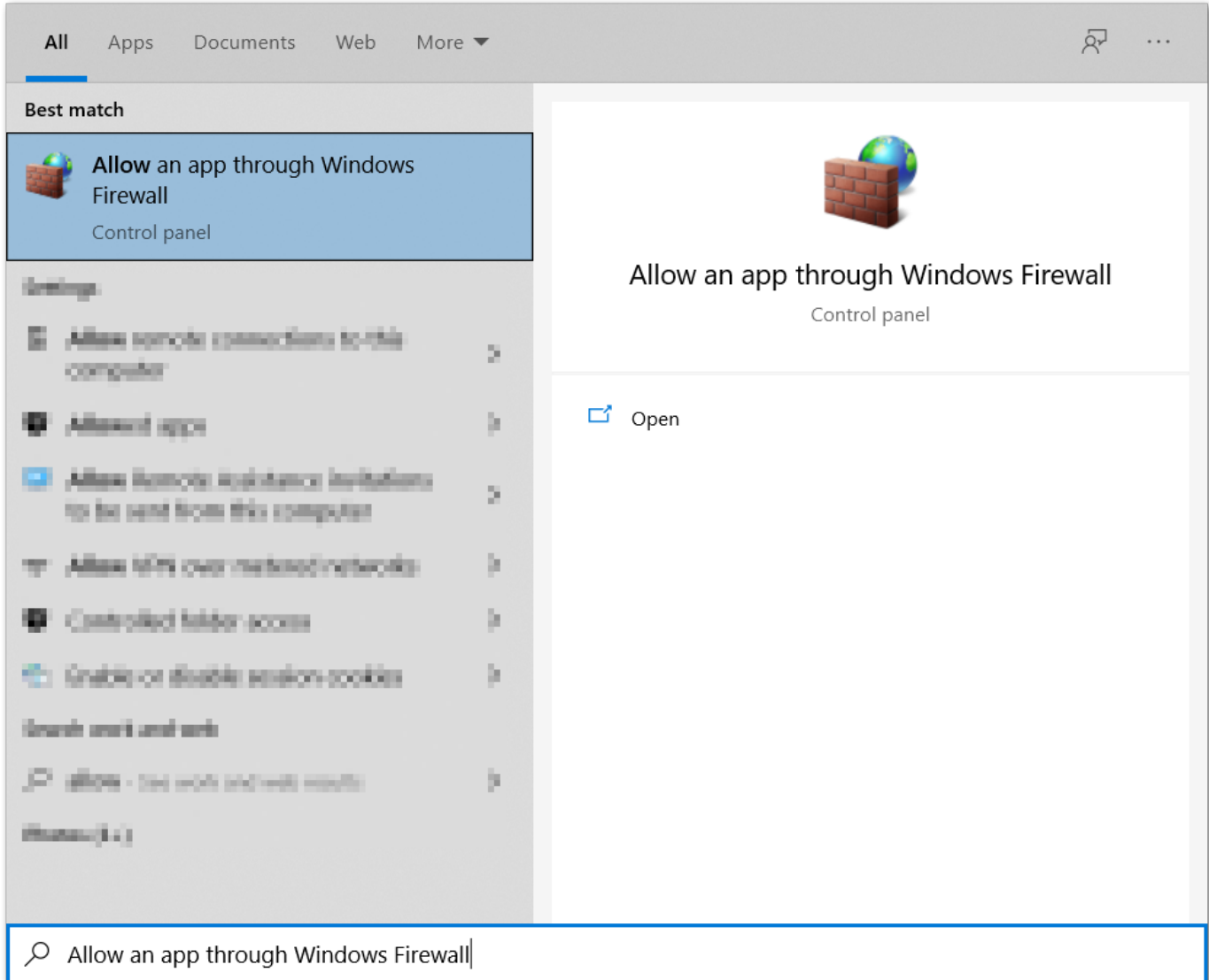
When you first try to connect to a target computer, Windows Defender Firewall prompts you to allow MATLAB to communicate on Private and Public networks. Make sure that both **Private** and **Public** options are selected. Only one is selected by default.



Click **Allow access**.

Manually Allow MATLAB for Public and Private Networks

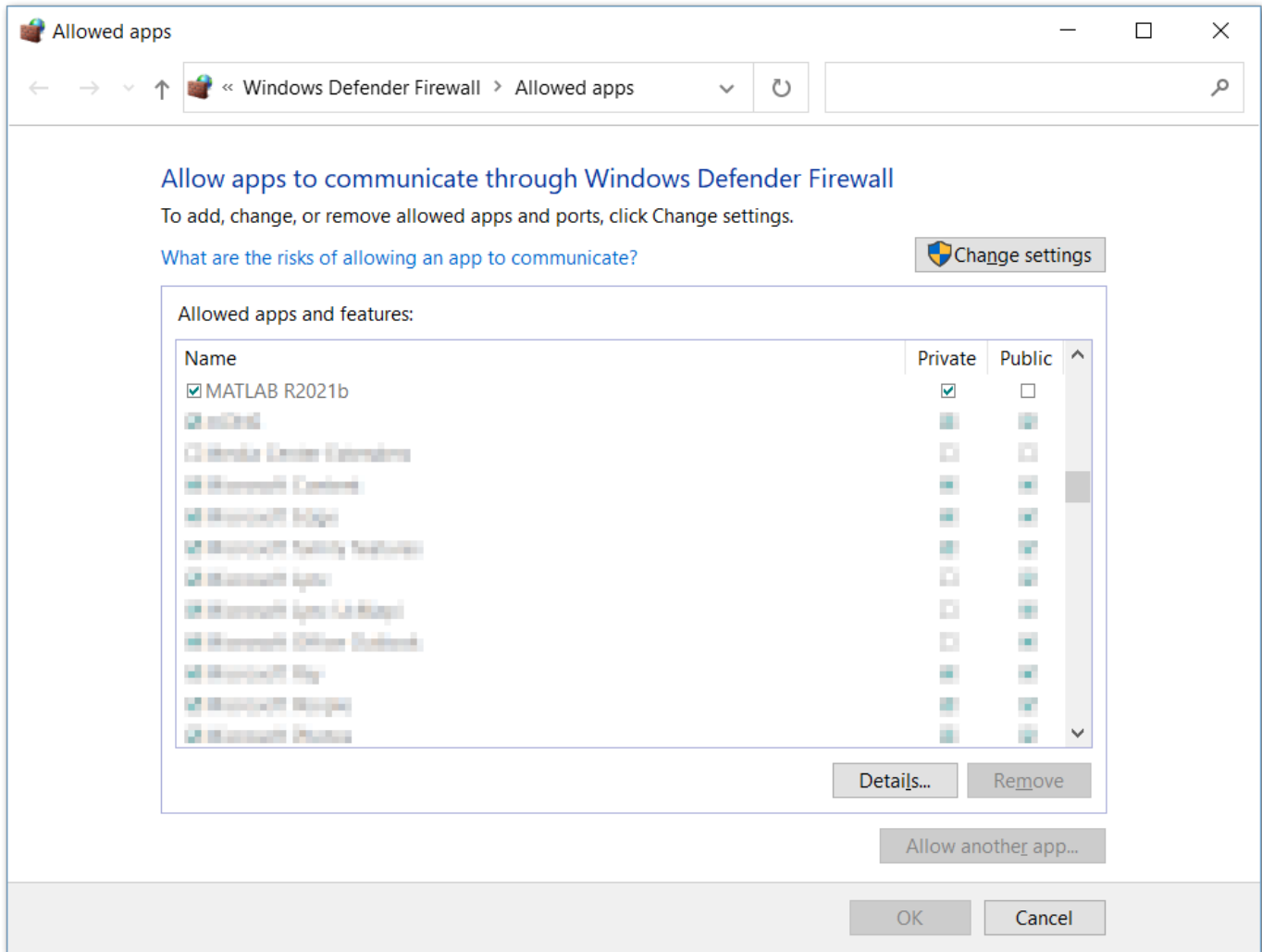
From the Windows Start menu, search Allow an app through Windows Firewall.



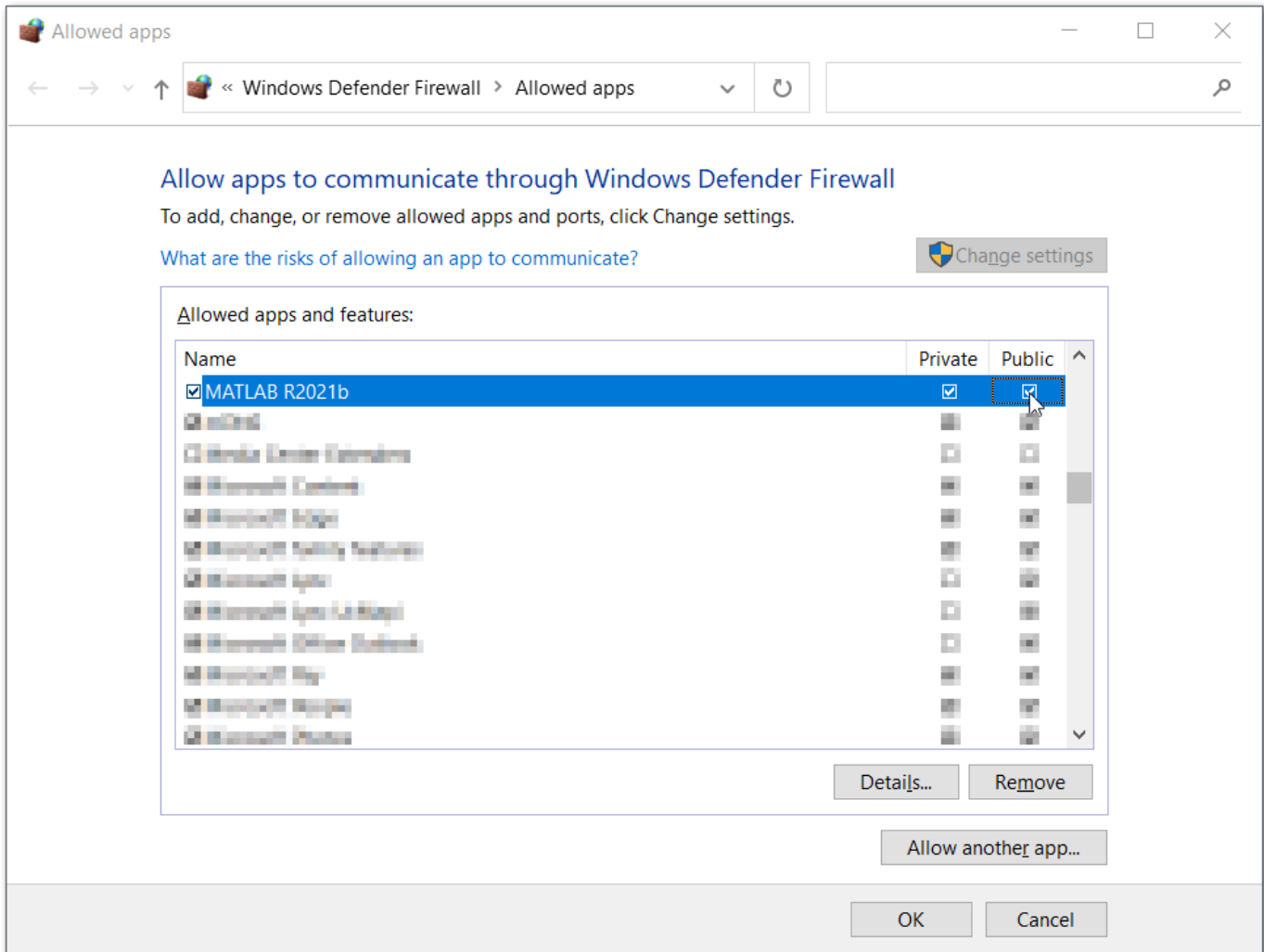
Click the **Allow an app through Windows Firewall** option.

Scroll down in the **Allowed apps and features** list and find the MATLAB release that you are using.

If both **Private** and **Public** check boxes are selected, see “Configure Development-to-Target Computer Ethernet Interface as Private” on page 17-9.



Click the **Change settings** button and confirm the security dialog. Make sure that the **Private** and **Public** boxes are selected. A **Domain** option or others may be available, but these options are not relevant for the MATLAB network access configuration.



Configure Development-to-Target Computer Ethernet Interface as Private

You can use the Windows UI or PowerShell command window to configure development-to-target computer Ethernet interface as private.

Windows UI Method

The Windows UI method is straightforward, but may not be available depending on your system configuration. If not, use the PowerShell Command Window Method.

- 1 Right-click on the Wi-Fi or Ethernet network icon in the lower right corner of the Windows taskbar (next to the clock).
- 2 Select **Open Network & Internet** settings.
- 3 Under the **Ethernet** section, click **Properties**.
- 4 Select the radio button for **Private**.

PowerShell Command Window Method

- 1** Right-click on the Windows **Start** menu and click the **Windows PowerShell (Admin)** selection.
- 2** Run the command `Get-NetConnectionProfile`.
- 3** Find the name of the Ethernet interface that you are using for development-to-target computer communication.
- 4** Enter the command `Set-NetConnectionProfile`.
- 5** Use the interface name that you find as the `Name` argument.
- 6** Confirm the changes by running `Get-NetConnectionProfile`.

For example, if the interface is named `Unidentified network`, enter the command:

```
Set-NetConnectionProfile -Name "Unidentified network" -NetworkCategory "Private"
```



```

Administrator: Windows PowerShell
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

Try the new cross-platform PowerShell https://aka.ms/pscore6

PS C:\Windows\system32> Get-NetConnectionProfile

Name                : Unidentified network
InterfaceAlias      : Ethernet
InterfaceIndex      : 15
NetworkCategory     : Public
IPv4Connectivity    : NoTraffic
IPv6Connectivity    : NoTraffic

Name                : Ethernet
InterfaceAlias      : Ethernet
InterfaceIndex      : 15
NetworkCategory     : Public
IPv4Connectivity    : Internet
IPv6Connectivity    : NotSet

PS C:\Windows\system32> Set-NetConnectionProfile -Name "Unidentified network" -NetworkCategory "Private"
PS C:\Windows\system32> Get-NetConnectionProfile

Name                : Unidentified network
InterfaceAlias      : Ethernet
InterfaceIndex      : 15
NetworkCategory     : Private
IPv4Connectivity    : NoTraffic
IPv6Connectivity    : NoTraffic

Name                : Ethernet
InterfaceAlias      : Ethernet
InterfaceIndex      : 15
NetworkCategory     : Public
IPv4Connectivity    : Internet
IPv6Connectivity    : NotSet

PS C:\Windows\system32>

```

Confirm Successful Configuration

To confirm successful configuration, in Simulink Real-Time Explorer or in the Simulink Editor on the **Real-Time** tab, click the **Disconnected** button. Confirm that the button label changes to **Connected**.

If the label does not change to **Connected**, the connection problem persists. Contact a systems administrator for further assistance. Administrator credentials may be required to configure the Windows Defender Firewall, or there may be another firewall on the development computer that

requires configuration. A systems administrator may need to allow communications on specific ports or add more specific firewall rules.

See Also

More About

- “Enable Development Computer Communication (Windows)”

External Websites

- MathWorks Help Center website
- What ports does a Simulink Real-Time target use to communicate with the host?
- How do I configure Windows Defender Firewall for Simulink Real-Time (SLRT)?

Troubleshoot Cannot Load Shared Object on Target Computer

When I load and run on the target computer a real-time application that depends on a shared object (.so), the real-time application cannot run and load the library. In the system log, I see a message like this error:

```
ldd:FATAL: Could not load library xyz.so
```

What This Issue Means

An error loading a shared object can indicate some issue with missing or corrupt library dependencies on the target computer. The issue could be:

- The download to the target computer has modified or has removed some required files on the target computer.
- The download to the target computer put the library in a location that is not accessible when the real-time application runs.

Try This Workaround

These workarounds explore the possible issues.

Check for Issues with Required Files

To check for this issue, connect to the target computer, and then try to build, load, and run example model `slrt_ex_osc`. If working with the default target computer, in the MATLAB Command Window, type:

```
tg = slrealtime;  
connect(tg);  
open_system(fullfile(matlabroot,'toolbox','slrealtime','examples','slrt_ex_osc'));  
slbuild('slrt_ex_osc');  
load('slrt_ex_osc');  
start('slrt_ex_osc');
```

If you can successfully connect to the target computer and build, load, and run the real-time application, there is no issue with files from the Simulink Real-Time Target Support Package on the target computer.

If you cannot complete those operations successfully, update the target computer software by using the `force` option. If working with the default target computer, in the MATLAB Command Window, type:

```
update(tg,'force',true);
```

After the software update, connect to the target computer and try to build, load, and run the real-time application..

Check Location of Shared Object on Target Computer

To check for this issue, use SSH or FTP to examine the location of the shared object file on the target computer. For more information, see “Execute Target Computer RTOS Commands at Target Computer Command Line” on page 9-3.

After you build the real-time application that links to a shared object, you must install the real-time application and the shared object on the target computer. Put the shared objects in a location on the target computer where they can be found and loaded at run time. The recommended locations are `/lib`, `/usr/lib`, or `/usr/local/lib`. Root access is required to copy or modify files in these locations.

See Also

`update | slrealtime.getSupportInfo`

More About

- “Troubleshoot Model Links to Static Libraries or Shared Objects” on page 17-24
- “Execute Target Computer RTOS Commands at Target Computer Command Line” on page 9-3
- “External Code Integration of Libraries and C/C++ Code with Simulink Real-Time Models” on page 11-2

Troubleshoot Signal Data Logging from Nonvirtual Bus, Fixed-Point, and Multidimensional Signals

My models sometimes use signals in nonvirtual buses, signals with fixed-point data types, and multidimensional signals that have a number of dimensions greater than two. I want to view signal data from these signals in the Simulation Data Inspector. I do not see data for these signals when I select them in Simulink Real-Time Explorer for streaming to the Simulation Data Inspector.

What This Issue Means

There are some guidelines to data logging signals in nonvirtual buses, signals with fixed-point data types, and multidimensional signals that have a number of dimensions greater than two:

- When these signals are marked for logging with the Simulation Data Inspector, the signal data displays in the Simulation Data Inspector.
- When these signals are connected to File Log blocks, the signal data displays in the Simulation Data Inspector.
- When these signals are selected for dynamic streaming with an instrument object—either by selecting the signals in Simulink Real-Time Explorer or adding the signals by using the Application object API, the signal data does not display in the Simulation Data Inspector or in App Designer instrument panel applications.

Try This Workaround

There are workarounds to get signals in nonvirtual buses, signals with fixed-point data types, and multidimensional signals (that have a number of dimensions greater than two) to display in the Simulation Data Inspector.

Signals in Nonvirtual Buses

To get signals in nonvirtual buses to display in the Simulation Data Inspector, mark the signals for data logging in the model or connect the signals to File Log blocks.

To instrument signals in nonvirtual buses to stream to an Instrument object, use the `BusElement` argument in the `addSignal`, `connectLine`, or `connectScalar` methods.

Signals with Fixed-Point Data Types

To get signals with fixed-point data types to display in the Simulation Data Inspector, mark the signals for data logging in the model or connect the signals to File Log blocks.

Multidimensional Signal

To get signals in multidimensional signals (that have a number of dimensions greater than two) to display in the Simulation Data Inspector, mark the signals for data logging in the model or connect the signals to File Log blocks.

See Also

Bus Creator | `fixdt` | `addSignal` | `connectLine` | `connectScalar`

Related Examples

- “Parameter Tuning and Data Logging” on page 16-2

More About

- “Create Nonvirtual Buses Within a Component”
- “Fixed-Point Data in MATLAB and Simulink”
- “Signal Basics”
- “Variable-Size Signal Basics”

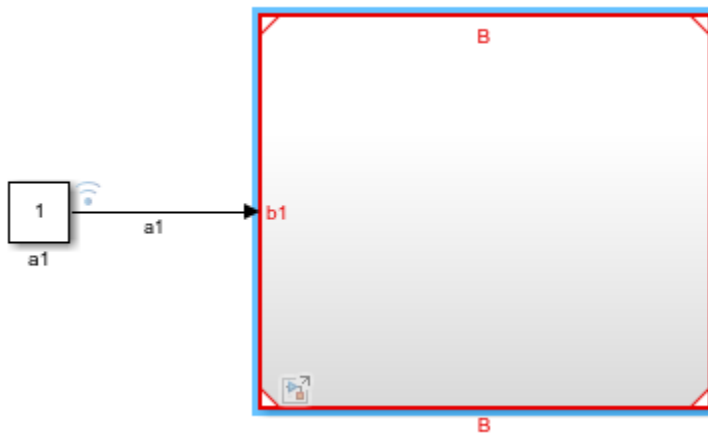
Troubleshoot Signal Data Logging from Inport in Referenced Model

My model contains referenced models. The referenced models have root-inport signals that would be helpful to log and stream to the Simulation Data Inspector for visualization. When I mark these signals in the model for logging, I see this warning in the diagnostic viewer when I build the model:

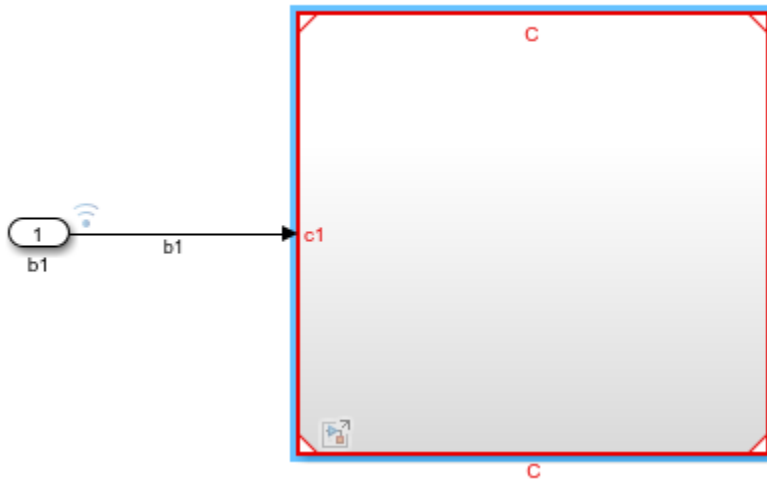
```
Warning: Streaming to the SDI is not available for signal at  
<sldiag objui="outport" objparam="1" objname="{ 'A/B',  
'B/b1'}">output port 1</sldiag> of block '{ 'A/B', 'B/b1'}'.  
Add a SignalCopy block at that port and instrument the  
SignalCopy output port.
```

What This Issue Means

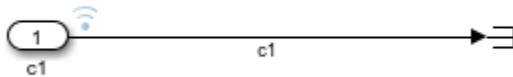
The warning message reports that the root-inport signals of referenced models are not available for streaming. Within top model A, the referenced model root inports that generate this warning message appear in referenced model A/B and referenced model A/B/C.



Top Model A



Referenced Model A/B



Referenced Model A/B/C

Try This Workaround

To instrument a root-inport signal in a referenced model and stream the signal to the Simulation Data Inspector, you can connect the signal to a Signal Conversion block that you configure as a Signal Copy block. Mark the output of the Signal Copy block for logging to the Simulation Data Inspector.

See Also

Related Examples

- “Trace or Log Data with the Simulation Data Inspector” on page 7-21

More About

- “Signal Logging and Streaming Basics” on page 7-26

Troubleshoot Signal Data Logging from Inport in Referenced Model in Test Harness

I created a Simulink Test test harness for my Simulink Real-Time model. The model has a referenced model that contains an inport whose signal I have marked for data logging in the Simulation Data Inspector. During testing, I see this error:

```
Warning: Streaming to the SDI is not available for signal at
<sldiag objui="outport" objparam="1"
objname="{ 'Example_Harness1/Example', 'Example/Input' }">output
port 1</sldiag> of block '{ 'Example_Harness1/Example',
'Example/Input' }'. Add a SignalCopy block at that port and
instrument the SignalCopy output port.
```

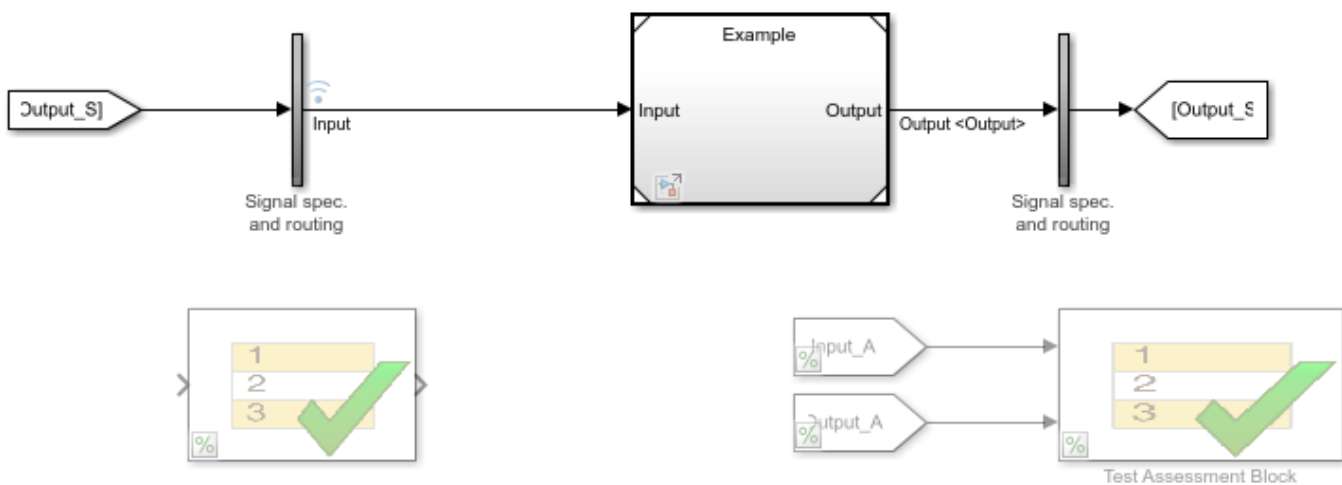
What This Issue Means

It is not possible to stream signal data from the referenced model inport for logging from within the test harness.

Try This Workaround

Mark the input signals to the model block for logging. This model provides an example workaround. For more information, see the Simulink Test documentation.

The **input** signal is logged to the model block because the marking the signal for logging inside the referenced model causes a warning that the signal is unavailable.



See Also

Related Examples

- “Test Real-Time Application in Simulink Test” on page 15-2

Troubleshoot Signal Data Logging from Send and Receive Blocks

My model uses Send and Receive blocks. I want to view signal data from the message line (output of send or input of receive) in the Simulation Data Inspector. I see unexpected data when I select a message line in Simulink Real-Time Explorer for streaming to the Simulation Data Inspector.

What This Issue Means

There are some guidelines to data logging message line signals:

- Message line signals that are marked for logging with the Simulation Data Inspector display the data accurately in the Simulation Data Inspector.
- Message line signals that are connected to File Log blocks display the data accurately in the Simulation Data Inspector.
- Message line signals that are selected for dynamic streaming with an instrument object—either by selecting the signals in Simulink Real-Time Explorer or adding the signals by using the Application object API—do not display the data accurately in the Simulation Data Inspector or in App Designer instrument panel applications.

For more information about message lines, see “Animate and Understand Sending and Receiving Messages”.

Try This Workaround

To get accurate display of message line signals in the Simulation Data Inspector, mark the signals for data logging in the model or connect the signals to File Log blocks.

See Also

File Log

Related Examples

- “Animate and Understand Sending and Receiving Messages”

More About

- “Data Logging with Simulation Data Inspector (SDI)” on page 7-14

Troubleshoot Signals for Streaming or File Logging

There are signals selected for streaming or connected to File Log blocks in my model that generate an error that starts with:

```
Cannot stream signal signal_name.
```

What This Issue Means

This error message for signals selected for streaming or connected to File Log blocks could indicate that the signal has one or more of these issues:

- The signal is not available in application.
- The signal does not use globally accessible memory in application.
- The signal connects to a Send/MessageSend block.
- The signal has inherited sample time.
- The signal is discontinuous.

Try These Workarounds

The workarounds for these issues vary. Try these.

Workaround for Signal Not Available

Make sure that these signal types are not being monitored, traced, or logged by name in the real-time application:

- Virtual or bus signals (including signals from bus creator blocks and virtual blocks)
- Signals that Simulink optimizes away
- Signals of complex or multiword data types
- Blocks without alphanumeric names

Workaround for Signal Not Global Available or Discontinuous Signal

To resolve, try inserting a Signal Copy block (a Signal Conversion block in Signal Copy mode) into the signals that you want to stream. Log the copied signal output instead. If you use a Dashboard block, connect it to the output signal of the Signal Copy block.

Workaround for Signal Connected to Message Block

To resolve, try streaming or file logging the input signal to the Send/MessageSend block. The output of the block (a message) cannot be streamed or logged.

Workaround for Signal Has Inherited Sample Time

To resolve, change the signal sample time from inherited to a value. Signals with inherited sample time cannot be streamed or logged.

Troubleshoot Folder Names with Spaces or Special Characters Halt Model Builds

When a space character appears the file path, my Simulink Real-Time model build reports an error:

```
Simulink Real-Time model build cannot use a file path with spaces for model build directory.
```

When a special character, such as an open parenthesis character "(", appears in the file path, my model build reports an error:

```
Error(s) encountered while building "xxxx"
```

What This Issue Means

For the `Simulink Real-Time model build ...` message or for the `Error(s) encountered while building ...` message, message indicates that a space character or special character appears in the file path. The QNX Neutrino toolchain for the code generation target is not compatible with file paths that contain spaces or special characters, the model build halts and does not output a real-time application.

Try This Workaround

Try these workaround options to resolve the model build errors.

Create a Build Folder

Create a folder name that does not have spaces or special characters in it. Build your model in that folder.

Map the Build Folder

Map the folder name or path that has spaces or special characters in it to a folder name or path without spaces or special characters. Build your model in the mapped folder.

See Also

More About

- “Build Process Support for File and Folder Names”

Troubleshoot Model Links to Static Libraries or Shared Objects

Some model build and runtime issues occur when I link my real-time application to static libraries (.a) or shared objects (.so).

What This Issue Means

When building or running real-time a application that links to static link libraries (.a) or shared object libraries (.so), there are some considerations that help you use libraries that are compatible with the QNX Neutrino RTOS on the target computer. These recommendations are helpful when troubleshooting library usage, including:

- A link to a library from QNX Neutrino RTOS that is available in the Simulink Real-Time target support package
- A shared object that is included in the model through an FMU block
- A custom static library or shared object that is linked to the real-time application

Try This Workaround

These workarounds explore the possible issues.

Link to Accessible Libraries or Objects

When you link to a static library or shared object, the library or object must be:

- Compatible with the QNX Neutrino RTOS
- Accessible to the toolchain at build time

Check to ensure that you have followed the guidelines for library compatibility. See “External Code Integration of Libraries and C/C++ Code with Simulink Real-Time Models” on page 11-2. Check that the toolchain can access the library at build time. Remember that the QNX Neutrino cannot process spaces in the path to files.

Install Shared Objects on Target Computer

After you build the real-time application that links to a shared object, install both the real-time application and the shared object on the target computer. Copy the shared objects to a location on the target computer where they can be found and loaded at runtime. The recommended locations are /lib, /usr/lib, or /usr/local/lib. Root access is required to copy files to these locations.

Rebuild Real-Time Application When Rebuilding Static Libraries

Because linked static libraries are included in the real-time application, when you modify and rebuild a static library, you rebuild any real-time applications that include that library. After rebuilding the static library and the real-time application, reinstall the real-time application on the target computer.

See Also

FMU

More About

- “Build Support for S-Functions”
- “Compile Source Code for Functional Mock-up Units” on page 3-3
- “External Code Integration of Libraries and C/C++ Code with Simulink Real-Time Models” on page 11-2
- “Troubleshoot Cannot Load Shared Object on Target Computer” on page 17-13

External Websites

- MathWorks Help Center website

Troubleshoot Build Error for Accelerator Mode

I get a build error when building a model in accelerator mode or rapid accelerator mode when the model contains Simulink Real-Time blocks (for example, model blocks that represent hardware).

What This Issue Means

Simulink Real-Time does not support accelerator mode or rapid accelerator mode simulation of models with blocks that represent hardware. For example, if you open the `slrt_ex_serialasciitest` model, change the Simulink mode to rapid accelerator, and run the model, Simulink displays this error:

```
Unable to build a standalone executable to simulate the model  
'slrt_ex_serialasciitest' in rapid accelerator mode.
```

This error occurs because accelerator mode and rapid accelerator mode produce compiled code that runs on the development computer, not on the Simulink Real-Time target computer. Any blocks that access hardware report a build error if you compile the model by using accelerator mode or rapid accelerator mode.

Try This Workaround

Change the simulation mode to normal mode or external mode.

See Also

More About

- “How Acceleration Modes Work”
- “Simulink Real-Time Options Pane”

External Websites

- MathWorks Help Center website

Troubleshoot Long Build Times for Real-Time Application

The model build process for my Simscape Multibody™ models is slow and uses an unexpected amount of memory.

What This Issue Means

The default QNX Neutrino compiler switches for Simulink Real-Time apply optimizations that lead to long build times or slow builds for some complex models, such as Simscape Multibody models.

Try This Workaround

To improve the real-time application build speed, change the compiler switch selections from the default selections by adding the `-fdisable-rtl-sched2` switch for the C/C++ compiler:

- 1 Open your Simulink Real-Time model.
- 2 In the Simulink Editor, from the **Real-Time** tab, select **Hardware Settings**.
- 3 Select **Code Generation > Build configuration > Specify**
- 4 Click the **C Compiler** options and add option `-fdisable-rtl-sched2`.
- 5 Click the **C++ Compiler** options and add option `-fdisable-rtl-sched2`.
- 6 Click **Apply** and **OK**.

After updating the compiler options, the options appear as shown.

Tool	Options
C Compiler	<code>-c -V\$(QCC_TARGET) -g -O2 -fwrapv -fdisable-rtl-sched2</code>
Linker	<code>-V\$(QCC_TARGET) -g -std=gnu++14 -stdlib=libstdc++</code>
Shared Library Linker	<code>-V\$(QCC_TARGET) -shared -Wl,--no-undefined -g</code>
C++ Compiler	<code>-c -V\$(QCC_TARGET) -g -std=gnu++14 -stdlib=libstdc++ -O2 -fwrapv -fdisable-rtl-sched2</code>
C++ Linker	<code>-V\$(QCC_TARGET) -g -std=gnu++14 -stdlib=libstdc++</code>
C++ Shared Library Linker	<code>-V\$(QCC_TARGET) -shared -Wl,--no-undefined -g</code>
Archiver	<code>ruvs</code>
Make Tool	<code>-f \$(MAKEFILE)</code>

If you prefer to use a programmatic approach to update these compiler switches, you could use this code.

```
% add a compiler flag '-fdisable-rtl-sched2'

set_param(modelName, 'BuildConfiguration', 'Specify');
options = get_param(modelName, 'CustomToolchainOptions');
ccompiler_idx = find(strcmp(options, 'C Compiler'));
cppcompiler_idx = find(strcmp(options, 'C++ Compiler'));
```

```
options{ccompiler_idx+1} = ...  
  [options{ccompiler_idx+1} '-fdisable-rtl-sched2'];  
options{cppcompiler_idx+1} = ...  
  [options{cppcompiler_idx+1} '-fdisable-rtl-sched2'];  
set_param(modelName, 'CustomToolchainOptions', options);
```

See Also

External Websites

- QNX Momentics IDE 7.1 User's Guide
- QNX Momentics IDE 7.1 User's Guide, Utilities Reference

Troubleshoot Working with Persistent Variables

When I run the `getPersistentVariables` function or `setPersistentVariables` function, I see this error:

```
Cannot parse the file that stores persistent variables from
target computer. To clear the issue, delete all persistent
variables on target computer. For more information, see
Troubleshoot Working with Persistent Variables.
```

What This Issue Means

This error message indicates that the file on the target computer that stores the persistent variable values is corrupted or unreadable.

Try This Workaround

To resolve this issue, clear the persistent variable values that are stored on the target computer.

- 1 On the development computer, create a `Target` object `tg` and connect to the target computer.

```
tg = slrealtime;
connect(tg);
```

- 2 Use the `setPersistentVariables` function to clear the persistent variable values that are stored on the target computer.

```
setPersistentVariables(tg, []);
```

See Also

[getPersistentVariables](#) | [setPersistentVariables](#) | [Persistent Variable Read](#) | [Persistent Variable Write](#)

Related Examples

- “Apply Persistent Variables in Real-Time Applications” on page 16-123

Troubleshoot Unsatisfactory Real-Time Performance

I want some recommended methods to improve unsatisfactory real-time application performance.

What This Issue Means

Run-time performance and reduce the task execution time (TET) of a model depend on model design, target computer capacity, and target computer utilization.

Try This Workaround

You can improve run-time performance and reduce the task execution time (TET) of a model with these methods.

Enable Compile with GCC `-ffast-math` Option

The **Compile with GCC `-ffast-math`** option enables the GCC compiler `-ffast-math` option when compiling real-time application code. This option is disabled by default for Simulink Real-Time models.

By enabling the **Compile with GCC `-ffast-math`** option, you provide the compiler with more flexibility to optimize floating-point math at the expense of deviating from the IEEE-754 floating-point standard.

For more information about the `-ffast-math` option, see the Semantics of Floating-Point Math in GCC and gcc.gnu.org/wiki/FloatingPointMath/

Run Performance Tools

Use these performance tools:

- To profile execution of a real-time application, use the `startProfiler` command.
- To run the profiler and plot the results, use the `plot` function.

For more information, see “Execution Profiling for Real-Time Applications” on page 10-7.

Use a Multicore Target Computer

You can improve run-time performance by configuring your model to take advantage of your multicore target computer:

- 1 Partition the model into subsystems according to the physical requirements of the system that you are modeling. Set the block sample rates within each subsystem to the slowest rate that meets the physical requirements of the system.
- 2 In the Configuration Parameters dialog box, on the **Solver** pane, select the check box for **Treat each discrete rate as a separate task**.
- 3 Click **Configure Tasks**, and then select the **Enable explicit model partitioning for concurrent behavior** check box.
- 4 Create tasks and triggers, and then explicitly assign subsystem partitions to the tasks. See “Partition Your Model Using Explicit Partitioning” and “Multicore Programming with Simulink”.
- 5 Run the real-time application.

Note Do not use MATLAB System blocks in the top level of Simulink Real-Time models in which task execution is explicitly partitioned. These blocks generate a TLC error when building the real-time application, for example:

```
"Unable to find TLCBlockSID within the Block scope"
```

Minimize the Model

You can improve run-time performance by minimizing your model to make more memory and CPU cycles available for the real-time application:

- 1 On the **Solver** pane, increase **Fixed-step size (fundamental sample time)**. Executing with a short sample time can overload the CPU.
- 2 Use polling mode. See “Execution Modes” on page 8-2.
- 3 Reduce the number of I/O channels in the model.

Contact Technical Support

For additional guidance, refer to these sources:

- MathWorks Tech Support: MathWorks Help Center website
- MATLAB Answers: www.mathworks.com/matlabcentral/answers/?term=Simulink+Real-Time
- MATLAB Central: www.mathworks.com/matlabcentral

For Speedgoat hardware issues, contact Speedgoat Tech Support: www.speedgoat.com/knowledge-center.

See Also

Compile with GCC `-ffast-math`

Related Examples

- “Concurrent Execution on Simulink Real-Time” on page 16-11

More About

- “Execution Profiling for Real-Time Applications” on page 10-7
- “Partition Your Model Using Explicit Partitioning”
- “Execution Modes” on page 8-2
- “Find Simulink Real-Time Support” on page 17-35
- “Multicore Programming with Simulink”

External Websites

- MathWorks Help Center website
- www.speedgoat.com/products-services
- www.speedgoat.com/knowledge-center
- gcc.gnu.org/wiki/FloatingPointMath/

Troubleshoot Overloaded CPU from Executing Real-Time Application

Some issue is producing a CPU overload when executing a real-time application.

What This Issue Means

A CPU overload indicates that the CPU is unable to complete processing a model time step before restarting for the next time step.

When this error occurs, the Simulink Real-Time RTOS halts model execution and the Target object property TargetStatus shows an error, for example:

```
mCPUOverload: Sub-rate exception: Overload limit (0) exceeded in 0.02s rate with 1 overloads
```

If you allow the overload, model execution continues until the allowed overload limit is reached. If the model continues to run after a CPU overload, the time step lasts as long as the time required to finish the execution. This behavior delays the next time step.

Model design or target computer resources cause CPU overloads. Possible reasons are:

- The target computer is too slow or the model sample time is too small.
- The model is too complex (algorithmic complexity).
- I/O latency, where each I/O channel used introduces latency into the system. I/O latency can cause the execution time to exceed the model time step.

To find latency values for Speedgoat boards, contact Speedgoat technical support.

Try This Workaround

The Simulink Real-Time RTOS usually halts model execution when it encounters a CPU overload. You can configure the Simulink Real-Time model to allow CPU overloads. Use this capability to support long initializations and for overload diagnosis. You also can try to reduce overloads by improving application performance and enabling the **Compile with GCC -ffast-math** option.

Permit Long Initialization Time

For some real-time applications, normal initialization can extend beyond the first sample time. Use the SLRT Overload Options block to increase the number of startup time steps to ignore overloads. By default, only the first time step ignores overloads.

Note Allowing the target computer CPU to overload can cause incorrect results, especially for multirate models. Use the SLRT Overload Options block only for diagnosis. When your diagnosis is complete, turn off these options.

Enable Compile with GCC -ffast-math Option

The **Compile with GCC -ffast-math** option enables the GCC compiler `-ffast-math` option when compiling real-time application code. This option is disabled by default for Simulink Real-Time models.

By enabling the **Compile with GCC -ffast-math** option, you provide the compiler with more flexibility to optimize floating-point math at the expense of deviating from the IEEE-754 floating-point standard.

For more information about the `-ffast-math` option, see the Semantics of Floating-Point Math in GCC and gcc.gnu.org/wiki/FloatingPointMath/

Force Polling Mode

The **Force polling mode** option enables polling mode — instead of interrupt-driven mode — for clocking the real-time application. Enabling this option can help reduce CPU overloads if:

- The target computer has at least four CPU cores.
- The CPU overload is caused by sporadic TET spikes.

See Also

Compile with GCC -ffast-math

Related Examples

- “Monitor CPU Overload Rate” on page 10-3

More About

- “CPU Overload” on page 10-2

External Websites

- MathWorks Help Center website

Troubleshoot Gaps in Streamed Data

A real-time application is producing a live streaming overload while attempting to stream signal data at a high rate.

What This Issue Means

Live streaming from a real-time application does not guarantee all the data appears in the Simulation Data Inspector. Live stream instrumentation runs at a lower priority than the real-time application. So, data sent by live streaming could be dropped if the host-target connection cannot keep up.

If a live stream overload occurs, you could see noticeable gaps in the data in the Simulation Data Inspector or see that some timesteps are lost when you export data from the Simulation Data Inspector.

Try This Workaround

The issue is caused by high data rates and live streaming of data.

To workaround the issue:

- Modify the real-time application to decrease the data rate for live streaming data. To do this, you could increase the sample rate, instrument fewer signals, or increase the decimation of instrumented signals.
- Change the real-time application to use file logging instead of live streaming. File logging is capable of logging higher data rates without dropping data.

See Also

Related Examples

- “Parameter Tuning and Data Logging” on page 16-2

More About

- “Trace or Log Data with the Simulation Data Inspector” on page 7-21

External Websites

- MathWorks Help Center website

Find Simulink Real-Time Support

For support with Speedgoat target machines or the Speedgoat I/O Blockset, contact Speedgoat support:

www.speedgoat.com/knowledge-center

For support on general MATLAB or Simulink issues, see MathWorks Support:

www.mathworks.com/support

For support on Simulink Real-Time issues, see:

- Simulink Real-Time Support:

[MathWorks Help Center website](#)

- Simulink Real-Time Answers:

www.mathworks.com/matlabcentral/answers/?term=Simulink+Real-Time

www.mathworks.com/matlabcentral/answers/?term=xPC+Target

- Simulink Real-Time Central File Exchange:

www.mathworks.com/matlabcentral/fileexchange/?term=Simulink+Real-Time

www.mathworks.com/matlabcentral/fileexchange/?term=xPC+Target

After searching these resources, if you still cannot solve your issue:

- For online or phone support, contact MathWorks technical support directly.

Install Simulink Real-Time Software Updates

The general procedure for updating Simulink Real-Time is:

- 1 Navigate to the MathWorks download page:
www.mathworks.com/downloads
- 2 Navigate to the page for the Simulink Real-Time software version that you want. Download the software version to your development computer.
- 3 Install and integrate the new release software.

After updating Simulink Real-Time, to re-create your Simulink Real-Time target settings:

- 1 In the MATLAB Command Window, type `slrtExplorer`.
- 2 On the **Targets Tree** pane, select a target computer node.
- 3 Click the **Target Configuration** tab.
- 4 Click **Change IP Address** and select the IP Address and Netmask for communication method between your development and target computer. For more information, see “Target Computer Settings”. Click **OK**.
- 5 Click the **Disconnected** link, toggling it to **Connected**.
- 6 Repeat steps 2 through 5 for each target computer.
- 7 Build each model that you want to execute. In the Simulink Editor, on the **Real-Time** tab, click **Run on Target**.

See Also

More About

- “Target Computer Settings”

External Websites

- www.mathworks.com/downloads
- www.speedgoat.com/knowledge-center