

AUTOSAR Blockset

User's Guide



MATLAB® & SIMULINK®

R2023a



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

AUTOSAR Blockset User's Guide

© COPYRIGHT 2019–2023 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

March 2019	Online only	New for Version 2.0 (Release 2019a)
September 2019	Online only	Revised for Version 2.1 (Release 2019b)
March 2020	Online only	Revised for Version 2.2 (Release 2020a)
September 2020	Online only	Revised for Version 2.3 (Release 2020b)
March 2021	Online only	Revised for Version 2.4 (Release 2021a)
September 2021	Online only	Revised for Version 2.5 (Release 2021b)
March 2022	Online only	Revised for Version 2.6 (Release 2022a)
September 2022	Online only	Revised for Version 3.0 (Release 2022b)
March 2023	Online only	Revised for Version 3.1 (Release 2023a)

1

Overview of AUTOSAR Support

AUTOSAR Blockset Product Description	1-2
AUTOSAR Standard	1-3
Comparison of AUTOSAR Classic and Adaptive Platforms	1-5
Classic Platform	1-5
Adaptive Platform	1-7
AUTOSAR Software Components and Compositions	1-11
Workflows for AUTOSAR	1-13
Simulink Originated (Bottom-Up) Workflow	1-13
Round-Trip Workflow	1-13
AUTOSAR Workflow Samples	1-15
Develop AUTOSAR Software Component Model	1-16
Prerequisites	1-16
Example Model	1-16
What You Will Learn	1-16
Create Algorithmic Model Content That Represents AUTOSAR Software Component Behavior	1-18
Configure Elements of AUTOSAR Software Component for Simulink	
Modeling Environment	1-20
Set Up Initial Component Configuration	1-20
Customize Component Configuration	1-22
Configure AUTOSAR Software Component Elements from AUTOSAR	
Standard Perspective	1-22
Simulate AUTOSAR Software Component	1-24
Optional: Generate AUTOSAR Software Component Code (Requires Embedded Coder)	1-25
Develop AUTOSAR Adaptive Software Component Model	1-28
Prerequisites	1-28
Example Model	1-28
What You Will Learn	1-28
Create Algorithmic Model Content That Represents AUTOSAR Adaptive Software Component Behavior	1-30

Configure Elements of AUTOSAR Adaptive Software Component for Simulink Modeling Environment	1-33
Set Up an Initial Component Configuration	1-33
Customize Component Configuration	1-34
Configure AUTOSAR Adaptive Software Component Elements from AUTOSAR Standard Perspective	1-35
Simulate AUTOSAR Adaptive Software Component	1-37
Optional: Generate AUTOSAR Adaptive Software Component Code (Requires Embedded Coder)	1-38
Develop AUTOSAR Software Architecture Model	1-41
Prerequisites	1-41
Example Model	1-41
What You Will Learn	1-41
Create AUTOSAR Software Architecture Model	1-43
Add AUTOSAR Compositions and Components and Link Component Implementations	1-45
Add Compositions and Components to Architecture Canvas	1-45
Define Component Behavior by Linking Implementation Models	1-46
Complete Architecture Model Top Level	1-48
Simulate Components in AUTOSAR Architecture	1-50
Optional: Generate and Package Composition ARXML and Component Code (Requires Embedded Coder)	1-52

Modeling Patterns for AUTOSAR Components

2

Simulink Modeling Patterns for AUTOSAR	2-2
Model AUTOSAR Software Components	2-3
About AUTOSAR Software Components	2-3
Implementation Considerations	2-3
Rate-Based Components	2-6
Function-Call-Based Components	2-7
Multi-Instance Components	2-8
Startup, Reset, and Shutdown	2-8
Modeling Patterns for AUTOSAR Runnables	2-10
Model AUTOSAR Runnables Using Exported Functions	2-18
Model AUTOSAR Communication	2-21
About AUTOSAR Communication	2-21
Sender-Receiver Interface	2-21
Queued Sender-Receiver Interface	2-23
Client-Server Interface	2-24

Mode-Switch Interface	2-25
Nonvolatile Data Interface	2-28
Parameter Interface	2-29
Trigger Interface	2-29
Model AUTOSAR Component Behavior	2-31
AUTOSAR Elements for Modeling Component Behavior	2-31
Runnables	2-31
Inter-Runnable Variables	2-32
Included Data Type Sets	2-32
System Constants	2-33
Per-Instance Memory	2-34
Static and Constant Memory	2-34
Shared and Per-Instance Parameters	2-35
Port Parameters	2-35
Model AUTOSAR Variants	2-37
Variants for Ports and Runnables	2-37
Variants for Runnable Implementations	2-38
Variants for Array Sizes	2-38
Predefined Variants and System Constant Value Sets	2-38
Model AUTOSAR Nonvolatile Memory	2-40
Implicit Access to AUTOSAR Nonvolatile Memory	2-40
Explicit Access to AUTOSAR Nonvolatile Memory	2-41
Model AUTOSAR Data Types	2-43
About AUTOSAR Data Types	2-43
Enumerated Data Types	2-44
Structure Parameters	2-45
Data Types	2-45
CompuMethod Categories for Data Types	2-48
Model AUTOSAR Calibration Parameters and Lookup Tables	2-50
AUTOSAR Calibration Parameters	2-50
Calibration Parameters for STD_AXIS, FIX_AXIS, and COM_AXIS Lookup Tables	2-50

AUTOSAR Component Creation

3

Create AUTOSAR Software Component in Simulink	3-2
Create Mapped AUTOSAR Component with Quick Start	3-2
Create Mapped AUTOSAR Component with Simulink Start Page	3-5
Create and Configure AUTOSAR Software Component	3-8
Import AUTOSAR XML Descriptions Into Simulink	3-13
Create ARXML Importer Object	3-14
Import Software Component and Create Model	3-14
Import Software Composition and Create Models	3-15
Import Component or Composition External Updates Into Model	3-16

Import Shared Element Packages into Component Model	3-16
Import AUTOSAR Software Component with Multiple Runnables	3-18
Import AUTOSAR Component to Simulink	3-19
Import AUTOSAR Software Composition with Atomic Software Components (Classic Platform)	3-24
Import AUTOSAR Software Component Updates	3-25
Update Model with AUTOSAR Software Component Changes	3-25
AUTOSAR Update Report Section Examples	3-26
Import and Reference Shared AUTOSAR Element Definitions	3-29
Import AUTOSAR Package into Component Model	3-31
AUTOSAR ARXML Importer	3-35
Round-Trip Preservation of AUTOSAR XML File Structure and Element Information	3-37
Limitations and Tips	3-40
Cannot Save Importer Objects in MAT-Files	3-40
ApplicationRecordDataType and ImplementationDataType Element Names Must Match	3-40

AUTOSAR Component Development

4

AUTOSAR Component Configuration	4-3
Configure AUTOSAR Elements and Properties	4-8
AUTOSAR Elements Configuration Workflow	4-8
Configure AUTOSAR Atomic Software Components	4-9
Configure AUTOSAR Ports	4-12
Configure AUTOSAR Runnables	4-21
Configure AUTOSAR Inter-Runnable Variables	4-25
Configure AUTOSAR Parameters	4-26
Configure AUTOSAR Communication Interfaces	4-27
Configure AUTOSAR Computation Methods	4-40
Configure AUTOSAR SwAddrMethods	4-42
Configure AUTOSAR XML Options	4-43
Map AUTOSAR Elements for Code Generation	4-50
Simulink to AUTOSAR Mapping Workflow	4-50
Map Entry-Point Functions to AUTOSAR Runnables	4-52
Map Imports and Outports to AUTOSAR Sender-Receiver Ports and Data Elements	4-53
Map Model Workspace Parameters to AUTOSAR Component Parameters	4-54
Map Data Stores to AUTOSAR Variables	4-56

Map Block Signals and States to AUTOSAR Variables	4-58
Map Data Transfers to AUTOSAR Inter-Runnable Variables	4-60
Map Function Callers to AUTOSAR Client-Server Ports and Operations .	4-61
Specify C Type Qualifiers for AUTOSAR Static and Constant Memory ...	4-61
Specify Default Data Packaging for AUTOSAR Internal Variables	4-62
Map Calibration Data for Submodels Referenced from AUTOSAR	
Component Models	4-65
Submodel Data Mapping Workflow	4-65
Map Submodel Parameters to AUTOSAR Component Parameters	4-68
Map Submodel Data Stores to AUTOSAR Variables	4-69
Map Submodel Signals and States to AUTOSAR Variables	4-70
Generate Submodel Data Macros for Verification and Deployment	4-73
Incrementally Update AUTOSAR Mapping After Model Changes	4-74
Design and Simulate AUTOSAR Components and Generate Code	4-77
Configure AUTOSAR Packages	4-84
AR-PACKAGE Structure	4-84
Configure AUTOSAR Packages and Paths	4-85
Control AUTOSAR Elements Affected by Package Path Modifications ...	4-88
Export AUTOSAR Packages	4-89
AR-PACKAGE Location in Exported ARXML Files	4-91
Configure AUTOSAR Package for Component, Interface, CompuMethod, or SwAddrMethod	4-94
Configure AUTOSAR Sender-Receiver Communication	4-96
Configure AUTOSAR Sender-Receiver Interface	4-96
Configure AUTOSAR Provide-Require Port	4-97
Configure AUTOSAR Receiver Port for IsUpdated Service	4-99
Configure AUTOSAR Sender-Receiver Data Invalidation	4-100
Configure AUTOSAR S-R Interface Port for End-To-End Protection	4-103
Configure AUTOSAR Receiver Port for DataReceiveErrorEvent	4-106
Configure AUTOSAR Sender-Receiver Port ComSpecs	4-108
Configure AUTOSAR Queued Sender-Receiver Communication	4-112
Simulink Workflow for Modeling AUTOSAR Queued Send and Receive	4-113
Configure AUTOSAR Sender and Receiver Components for Queued Communication	4-114
Implement AUTOSAR Queued Send and Receive Messaging	4-115
Configure Simulation of AUTOSAR Queued Sender-Receiver Communication	4-118
Simulate N-to-1 AUTOSAR Queued Sender-Receiver Communication ..	4-119
Simulate Event-Driven AUTOSAR Queued Sender-Receiver Communication	4-122
Implement AUTOSAR Queued Send and Receive By Using Stateflow Messaging	4-126
Configure AUTOSAR Ports By Using Simulink Bus Ports	4-138
Model AUTOSAR Ports By Configuring Simulink Bus Ports	4-138
Model AUTOSAR Interfaces By Typing Bus Ports with Bus Objects	4-140

Configure AUTOSAR Client-Server Communication	4-142
Configure AUTOSAR Server	4-142
Configure AUTOSAR Client	4-150
Configure AUTOSAR Client-Server Error Handling	4-156
Concurrency Constraints for AUTOSAR Server Runnables	4-159
Configure and Map AUTOSAR Server and Client Programmatically ...	4-161
Configure AUTOSAR Mode-Switch Communication	4-163
Configure Mode Receiver Port and Mode-Switch Event for Mode User .	4-163
Configure Mode Sender Port and Mode Switch Point for Application Mode Manager	4-166
Configure AUTOSAR Nonvolatile Data Communication	4-169
Configure AUTOSAR Port Parameters for Communication with Parameter Component	4-171
Configure Receiver for AUTOSAR External Trigger Event Communication	4-175
Configure AUTOSAR Runnables and Events	4-178
Configure AUTOSAR Runnable Execution Order	4-181
Configure AUTOSAR Initialize, Reset, or Terminate Runnables	4-187
Add Top-Level Asynchronous Trigger to Periodic Rate-Based System .	4-193
Configure AUTOSAR Initialization Runnable (R4.1)	4-196
Configure Disabled Mode for AUTOSAR Runnable Event	4-198
Configure Internal Data Types for AUTOSAR IncludedDataTypeSets ..	4-199
Configure AUTOSAR Per-Instance Memory	4-201
Configure Block Signals and States as AUTOSAR Typed Per-Instance Memory	4-201
Configure Data Stores as AUTOSAR Typed Per-Instance Memory	4-202
Configure Data Stores to Preserve State Information at Startup and Shutdown	4-204
Configure AUTOSAR Static Memory	4-206
Configure Block Signals and States as AUTOSAR Static Memory	4-206
Configure Data Stores as AUTOSAR Static Memory	4-207
Configure AUTOSAR Constant Memory	4-210
Configure AUTOSAR Shared or Per-Instance Parameters	4-212
Configure Model Workspace Parameters as AUTOSAR Shared Parameters	4-212
Configure Model Workspace Parameters as AUTOSAR Per-Instance Parameters	4-213
Configure Variants for AUTOSAR Elements	4-217

Configure Variants for AUTOSAR Runnable Implementations	4-220
Export Variation Points for AUTOSAR Calibration Data	4-223
Configure Dimension Variants for AUTOSAR Array Sizes	4-225
Control AUTOSAR Variants with Predefined Value Combinations	4-227
Configure Postbuild Variant Conditions for AUTOSAR Software Components	4-229
Configure Variant Parameter Values for AUTOSAR Elements	4-232
Specify Variant Parameters at Precompile Time	4-232
Specify Variant Parameters at Postbuild Time	4-233
Configure AUTOSAR CompuMethods	4-236
Configure AUTOSAR CompuMethod Properties	4-236
Create AUTOSAR CompuMethods	4-237
Configure CompuMethod Direction for Linear Functions	4-238
Export CompuMethod Unit References	4-239
Modify Linear Scaling for SCALE_LINEAR_AND_TEXTTABLE CompuMethod	4-240
Configure Rational Function CompuMethod for Dual-Scaled Parameter	4-241
Configure AUTOSAR Data Types Export	4-244
Control Application Data Type Generation	4-244
Configure DataTypeMappingSet Package and Name	4-245
Initialize Data with ApplicationValueSpecification	4-246
Configure AUTOSAR Internal Data Constraints Export	4-246
Automatic AUTOSAR Data Type Generation	4-248
Configure Parameters and Signals for AUTOSAR Calibration and Measurement	4-250
Configure Subcomponent Data for AUTOSAR Calibration and Measurement	4-255
Configure AUTOSAR Data for Calibration and Measurement	4-262
About Software Data Definition Properties (SwDataDefProps)	4-262
Configure SwCalibrationAccess	4-262
Configure DisplayFormat	4-264
Configure SwAddrMethod	4-267
Configure SwAlignment	4-270
Export SwImplPolicy	4-271
Export SwRecordLayout for Lookup Table Data	4-271
Configure Lookup Tables for AUTOSAR Calibration and Measurement	4-273
Configure STD_AXIS Lookup Tables by Using Lookup Table Objects ...	4-273
Configure COM_AXIS Lookup Tables by Using Lookup Table and Breakpoint Objects	4-276
Configure FIX_AXIS Lookup Tables by Using Simulink Parameter Objects	4-281
Configure Array Layout for Multidimensional Lookup Tables	4-284

Parameterizing Instances of Reusable Referenced Model Lookup Tables and Breakpoints	4-285
Exporting Lookup Table Constants as Record Value Specification	4-288
Exporting AdminData Record Layout Annotations	4-290
Configure and Map AUTOSAR Component Programmatically	4-293
AUTOSAR Property and Map Functions	4-293
Tree View of AUTOSAR Configuration	4-293
Properties of AUTOSAR Elements	4-294
Specify AUTOSAR Element Location	4-297
AUTOSAR Property and Map Function Examples	4-299
Configure AUTOSAR Software Component	4-300
Configure AUTOSAR Interfaces	4-311
Configure AUTOSAR XML Export	4-319
Limitations and Tips	4-321
AUTOSAR Client Block in Referenced Model	4-321

AUTOSAR Code Generation

5

Generate AUTOSAR C Code and XML Descriptions	5-2
Configure AUTOSAR Code Generation	5-7
Select AUTOSAR Classic Schema	5-7
Specify Maximum SHORT-NAME Length	5-8
Configure AUTOSAR Compiler Abstraction Macros	5-8
Root-Level Matrix I/O	5-9
Inspect AUTOSAR XML Options	5-9
Generate AUTOSAR C and XML Files	5-9
Code Generation with AUTOSAR Code Replacement Library	5-12
Code Replacement Library for AUTOSAR Code Generation	5-12
Find Supported AUTOSAR Library Routines	5-12
Configure Code Generator to Use AUTOSAR 4.0 Code Replacement Library	5-13
AUTOSAR 4.0 Library Host Code Verification	5-13
Code Replacement Library Checks	5-14
AUTOSAR Code Replacement Library Example for IFL/IFX Function Replacement	5-14
Required Algorithm Property Settings for IFL/IFX Function and Block Mappings	5-16
Verify AUTOSAR C Code with SIL and PIL	5-29
Integrate Generated Code for Multi-Instance Software Components ...	5-31
Import and Simulate AUTOSAR Code from Previous Releases	5-32
Limitations and Tips	5-33
Generate Code Only Check Box	5-33

AUTOSAR Compiler Abstraction Macros (Classic Platform)	5-33
Preservation of Bus Element Dimensions in Exported ARXML and Code	5-33
C++11 Style Scoped Enum Classes Generated for AUTOSAR Adaptive Applications	5-33

AUTOSAR Adaptive Software Component Modeling

6

Model AUTOSAR Adaptive Software Components	6-2
Create and Configure AUTOSAR Adaptive Software Component	6-6
Import AUTOSAR Adaptive Software Descriptions	6-12
Import AUTOSAR Adaptive Components to Simulink	6-13
Import AUTOSAR Package into Adaptive Component Model	6-17
Configure AUTOSAR Adaptive Elements and Properties	6-21
AUTOSAR Elements Configuration Workflow	6-21
Configure AUTOSAR Adaptive Software Components	6-22
Configure AUTOSAR Adaptive Service Interfaces and Ports	6-25
Configure AUTOSAR Adaptive Persistent Memory Interfaces and Ports . .	6-30
Configure AUTOSAR Adaptive XML Options	6-33
Map AUTOSAR Adaptive Elements for Code Generation	6-37
Simulink to AUTOSAR Mapping Workflow	6-37
Map Inports and Outports to AUTOSAR Service Ports and Events	6-39
Map Data Stores to AUTOSAR Persistent Memory Ports and Data Elements	6-39
Configure AUTOSAR Adaptive Software Components	6-41
Model AUTOSAR Adaptive Service Communication	6-50
Model Client-Server Communication	6-53
Configure Memory Allocation for AUTOSAR Adaptive Service Data	6-60
Configure AUTOSAR Adaptive Service Discovery Modes	6-62
Configure AUTOSAR Adaptive Service Instance Identification	6-64
Model AUTOSAR Adaptive Persistent Memory	6-66
Generate AUTOSAR Adaptive C++ Code and XML Descriptions	6-68
Configure AUTOSAR Adaptive Code Generation	6-73
Select AUTOSAR Adaptive Schema	6-73
Specify Maximum SHORT-NAME Length	6-74
Specify XCP Slave Transport Layer	6-74

Specify XCP Slave IP Address	6-74
Specify XCP Slave Port	6-75
Enable XCP Slave Message Verbosity	6-75
Use Custom XCP Slave	6-75
Inspect AUTOSAR Adaptive XML Options	6-76
Customize Class Name and Namespace in Generated Code	6-76
Configure Run-Time Logging Behavior	6-76
Generate AUTOSAR Adaptive C++ and XML Files	6-77
Configure AUTOSAR Adaptive Data for Run-Time Calibration and Measurement	6-80
Configure XCP Communication Interface in Generated Code	6-80
Configure AUTOSAR Adaptive Model for External Mode Simulation ...	6-82
Build Library or Executable from AUTOSAR Adaptive Model	6-83
Build Out of the Box Linux Executable from AUTOSAR Adaptive Model	6-86
Configure Run-Time Logging for AUTOSAR Adaptive Executables	6-89
Logging to Console	6-89
Logging to File	6-90
Logging to Network	6-90
Get Started with Embedded Coder Support Package for Linux Applications	6-92
Event Communication Between AUTOSAR Adaptive Applications Using Message Polling	6-96
Event Communication Between AUTOSAR Adaptive Applications Using Message Triggering	6-100

AUTOSAR Composition and ECU Software Simulation

7

Import AUTOSAR Composition to Simulink	7-2
Combine and Simulate AUTOSAR Software Components	7-7
Import AUTOSAR Composition as Model (Classic Platform)	7-7
Create Composition Model for Simulating AUTOSAR Components	7-8
Alternatives for AUTOSAR System-Level Simulation	7-9
Model AUTOSAR Basic Software Service Calls	7-12
Configure Calls to AUTOSAR Diagnostic Event Manager Service	7-14
Configure Calls to AUTOSAR Function Inhibition Manager Service	7-18
Model Function Inhibition	7-18
Scope Failures to Operation Cycles	7-23
Control Function Availability During Failure or For Testing	7-23

Configure Service Calls for Function Inhibition	7-24
Configure Calls to AUTOSAR NVRAM Manager Service	7-28
Configure AUTOSAR Basic Software Service Implementations for Simulation	7-33
Simulate AUTOSAR Basic Software Services and Run-Time Environment	7-36
Configure and Simulate AUTOSAR Function Inhibition Service Calls ..	7-49
Simulate and Verify AUTOSAR Component Behavior by Using Diagnostic Fault Injection	7-53

AUTOSAR Software Architecture Modeling

8

Create AUTOSAR Architecture Models	8-2
Add and Connect AUTOSAR Classic Components and Compositions	8-4
Add and Connect Classic Component Blocks	8-4
Add and Connect Composition Blocks to a Classic Model	8-6
Add and Connect AUTOSAR Adaptive Components and Compositions ..	8-10
Add and Connect Adaptive Component Blocks	8-10
Add and Connect Composition Blocks to an Adaptive Model	8-13
Import AUTOSAR Composition from ARXML	8-16
Import AUTOSAR Composition By Using AUTOSAR Importer App	8-16
Import AUTOSAR Composition By Calling importFromARXML	8-18
Create Profiles Stereotypes and Views for AUTOSAR Architecture Analysis	8-20
Create Profiles and Stereotypes	8-20
View Component or Composition Dependencies	8-20
Create Custom Views for Analysis	8-22
Link AUTOSAR Components to Requirements	8-25
Define AUTOSAR Component Behavior by Creating or Linking Models	8-27
Create Simulink Behavior Based on Block Interface	8-27
Link to Implementation Model	8-30
Create Model from ARXML Component Description	8-33
Configure AUTOSAR Scheduling and Simulation	8-38
Simulate Basic Software Service Calls	8-38
Connect a Test Harness	8-38
Schedule Component Runnables	8-40

Generate and Package AUTOSAR Composition XML Descriptions and Component Code	8-43
Configure Composition XML Options	8-43
Export Composition XML and Component Code	8-45
Export Composition ECU Extract	8-47
Author AUTOSAR Classic Compositions and Components in Architecture Model	8-50
Import AUTOSAR Composition into Architecture Model	8-63
Configure AUTOSAR Architecture Model Programmatically	8-67
Programmatically Create and Configure Architecture Model	8-67
Manage Shared Interfaces and Data Types for AUTOSAR Architecture Models	8-71
Create Interface Dictionary	8-71
Design Data Types and Interfaces by Using Interface Dictionary	8-72
Link Interface Dictionary to Architecture Model	8-74
Apply Interfaces to Architecture Model in Simulink Environment	8-77
Deploy Interface Dictionary	8-78
Limitations	8-79
Create AUTOSAR Architecture from a Component in System Composer Model	8-81

Overview of AUTOSAR Support

- “AUTOSAR Blockset Product Description” on page 1-2
- “AUTOSAR Standard” on page 1-3
- “Comparison of AUTOSAR Classic and Adaptive Platforms” on page 1-5
- “AUTOSAR Software Components and Compositions” on page 1-11
- “Workflows for AUTOSAR” on page 1-13
- “AUTOSAR Workflow Samples” on page 1-15
- “Develop AUTOSAR Software Component Model” on page 1-16
- “Create Algorithmic Model Content That Represents AUTOSAR Software Component Behavior” on page 1-18
- “Configure Elements of AUTOSAR Software Component for Simulink Modeling Environment” on page 1-20
- “Simulate AUTOSAR Software Component” on page 1-24
- “Optional: Generate AUTOSAR Software Component Code (Requires Embedded Coder)” on page 1-25
- “Develop AUTOSAR Adaptive Software Component Model” on page 1-28
- “Create Algorithmic Model Content That Represents AUTOSAR Adaptive Software Component Behavior” on page 1-30
- “Configure Elements of AUTOSAR Adaptive Software Component for Simulink Modeling Environment” on page 1-33
- “Simulate AUTOSAR Adaptive Software Component” on page 1-37
- “Optional: Generate AUTOSAR Adaptive Software Component Code (Requires Embedded Coder)” on page 1-38
- “Develop AUTOSAR Software Architecture Model” on page 1-41
- “Create AUTOSAR Software Architecture Model” on page 1-43
- “Add AUTOSAR Compositions and Components and Link Component Implementations” on page 1-45
- “Simulate Components in AUTOSAR Architecture” on page 1-50
- “Optional: Generate and Package Composition ARXML and Component Code (Requires Embedded Coder)” on page 1-52

AUTOSAR Blockset Product Description

Design and simulate AUTOSAR software

AUTOSAR Blockset provides apps and blocks for developing AUTOSAR Classic and Adaptive software using Simulink® models. You can design and map Simulink models to software components using the AUTOSAR Component Designer app. Alternatively, the blockset lets you generate new Simulink models for AUTOSAR by importing software component and composition descriptions from AUTOSAR XML (ARXML) files.

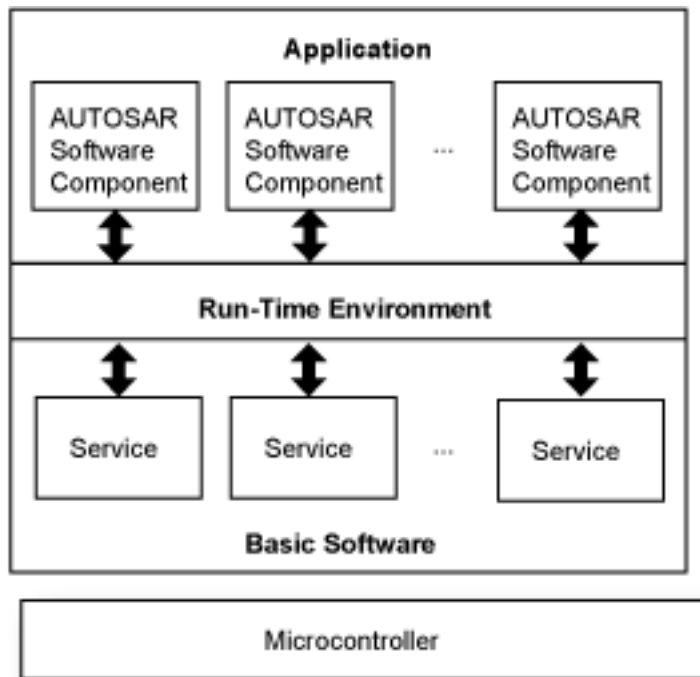
AUTOSAR Blockset provides blocks and constructs for AUTOSAR library routines and Basic Software (BSW) services, including NVRAM and Diagnostics. By simulating the BSW services together with your application software model, you can verify your AUTOSAR ECU software without leaving Simulink.

AUTOSAR Blockset lets you create AUTOSAR architecture models in Simulink (requires System Composer™). In the AUTOSAR architecture model, you can author software compositions, components with interfaces, data types, profiles, and stereotypes. You can add simulation behavior, including BSW service components. Alternatively, you can round-trip (import and export) software descriptions via ARXML files.

AUTOSAR Blockset supports C and C++ production code generation (with Embedded Coder®). It is qualified for use with the ISO 26262 standard (with IEC Certification Kit).

AUTOSAR Standard

Simulink software supports *AUTomotive Open System ARchitecture* (AUTOSAR), an open and standardized automotive software architecture consisting of three layers of software: Application, Run-Time Environment (RTE), and Basic Software.



Automobile manufacturers, suppliers, and tool developers jointly develop components of the Application layer. The standard refers to the components as AUTOSAR software components. They interact with the Run-Time Environment layer. The Run-Time Environment layer enables communication between:

- Components of the Application layer
- The Basic Software layer and components of the Application layer

The Basic Software layer provides shared common system services that components of the Application layer use.

The AUTOSAR standard addresses:

- **Architecture**—A layered software architecture decouples application software from the execution platform. Standard interfaces between AUTOSAR software components and the run-time environment allow reuse or relocation of components within the Electronic Control Unit (ECU) topology of a vehicle.

The standard defines variations of the software architecture called AUTOSAR platforms: Classic Platform and Adaptive Platform. For more information, see “Comparison of AUTOSAR Classic and Adaptive Platforms” on page 1-5.

- Methodology—Configuration description files define system information that ECUs share, system information that is unique to specific ECUs, and basic software information specific to an ECU.
- Foundation—Requirements and specifications shared between AUTOSAR platforms that support platform interoperability.
- Application Interfaces—Provide a standardized exchange format by specifying interfaces for typical automotive applications and specifying interfaces between the layers of software.

See Also

More About

- <https://www.autosar.org>
- “Comparison of AUTOSAR Classic and Adaptive Platforms” on page 1-5
- “Modeling Patterns”
- “Workflows for AUTOSAR” on page 1-13
- <https://www.mathworks.com/automotive/standards/autosar.html>

Comparison of AUTOSAR Classic and Adaptive Platforms

The AUTOSAR standard defines variations of the software architecture called AUTOSAR platforms: Classic Platform (CP) and Adaptive Platform (AP).

When you choose which platform to use for designing and implementing an AUTOSAR software component, review the information in this table for guidance.

AUTOSAR Platform Comparison

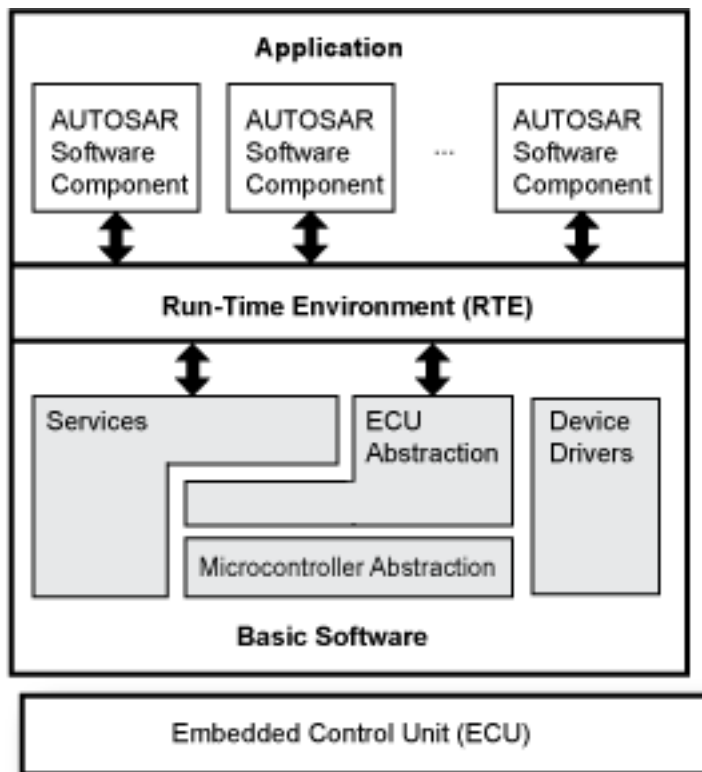
Goal or Feature	Classic Platform	Adaptive Platform
Use cases	Embedded systems	High performance computing, communication with external resources, and flexible deployment
Programming language	C	C++
Operating system	Bareboard	POSIX
Real-time requirements	Hard	Soft
Computing power	Low	High
Communication	Signal-based	Event-based, service-oriented
Safety and security	Supported	Supported
Dynamic updating	Not available	Incremental deployment and run-time configuration changes
Level of standardization	High—detailed specifications	Low—APIs and semantics
Agile development	No	Yes

Classic Platform

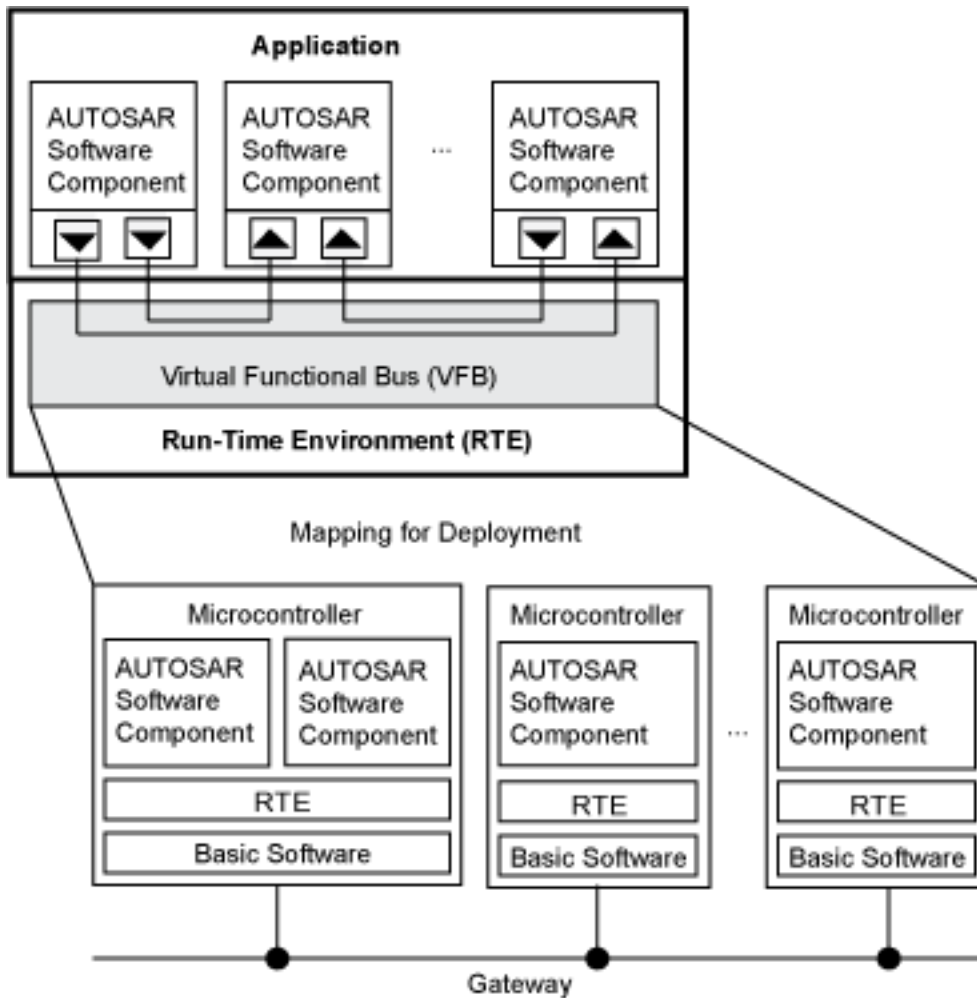
The Classic Platform addresses requirements of deeply embedded electronic control units (ECUs) that control electrical output signals based on input signals and information from other ECUs connected to a vehicle network. Typically, you design and implement the control software for a specific type of vehicle, which does not change during the lifetime of the vehicle.

The Run-Time Environment (RTE) layer of the software architecture handles communication between AUTOSAR software components in the Application layer and between AUTOSAR software components and services provided by the Basic Software layer. The Basic Software layer consists of:

- Services, such as system, memory, and communication services
- Device drivers
- ECU abstraction
- Microcontroller abstraction



The Classic Platform uses a virtual functional bus (VFB) to support hardware-independent development and usage of AUTOSAR application software. The bus consists of abstract representations of RTEs for specific ECUs, decoupling AUTOSAR software components in the Application layer of the architecture from the architecture infrastructure. AUTOSAR software components and the bus communicate by using dedicated ports. You configure an application by mapping component ports to the RTE representations of the system ECUs.



Adaptive Platform

The Adaptive Platform is a distributed computing and service-oriented architecture (SOA). The platform provides high-performance computing, message-based communication mechanisms, and flexible software configuration for supporting applications, such as automated driving and infotainment systems. Software based on this platform can:

- Meet strict integrity and security requirements
- Address environment perception and behavioral response planning
- Integrate a vehicle into the back end or infrastructure of an external system
- Address changes to external systems because you can update the software during the lifetime of a vehicle

The RTE layer of the software architecture includes the C++ standard library. It supports communication between AUTOSAR software components in the Application layer and between AUTOSAR software components and software provided by the Basic Software layer. The Basic Software layer consists of system foundation software and services. AUTOSAR software components in the Application layer communicate with each other, with nonplatform services, and with foundation

software and services by responding to event-driven messages. Software components interact with software in the Basic Software layer by using C++ application programming interfaces (APIs).

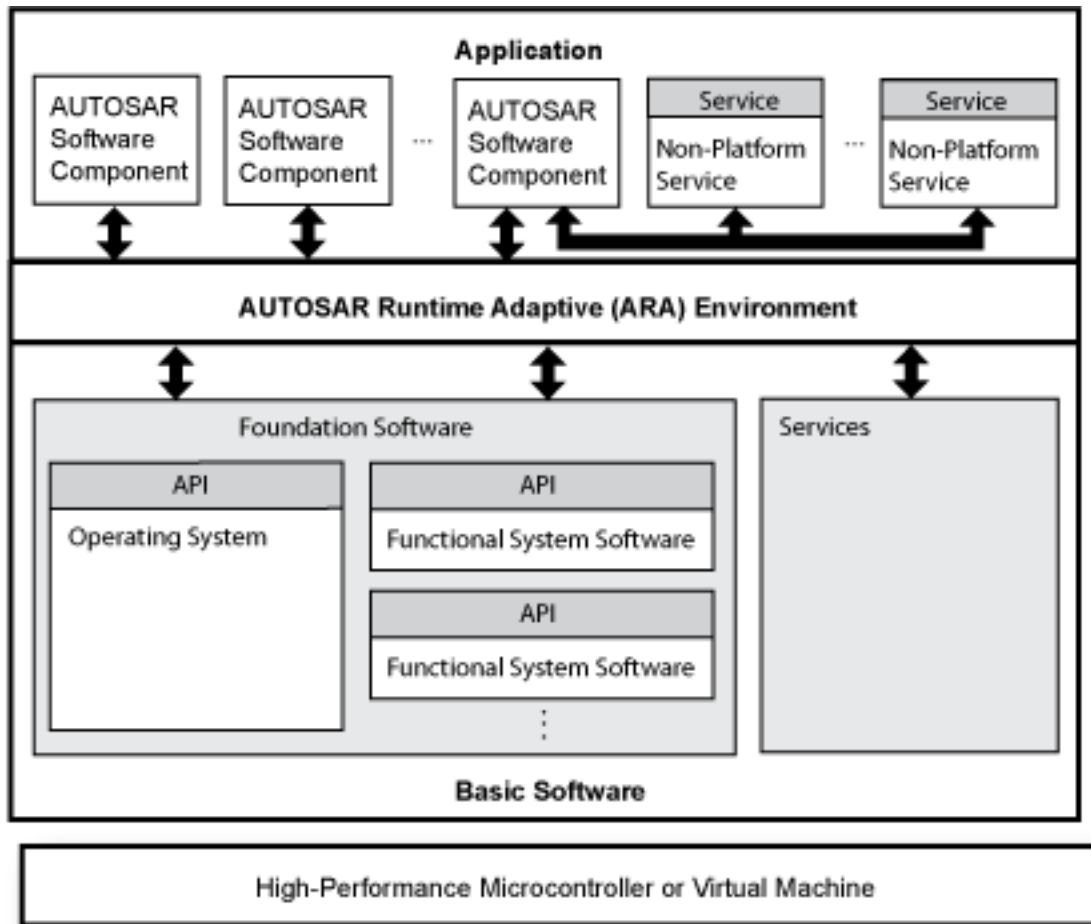
Foundation software includes the POSIX operating system and software for system management tasks, such as:

- Execution management
- Communication management
- Time synchronization
- Identity access management
- Logging and tracing

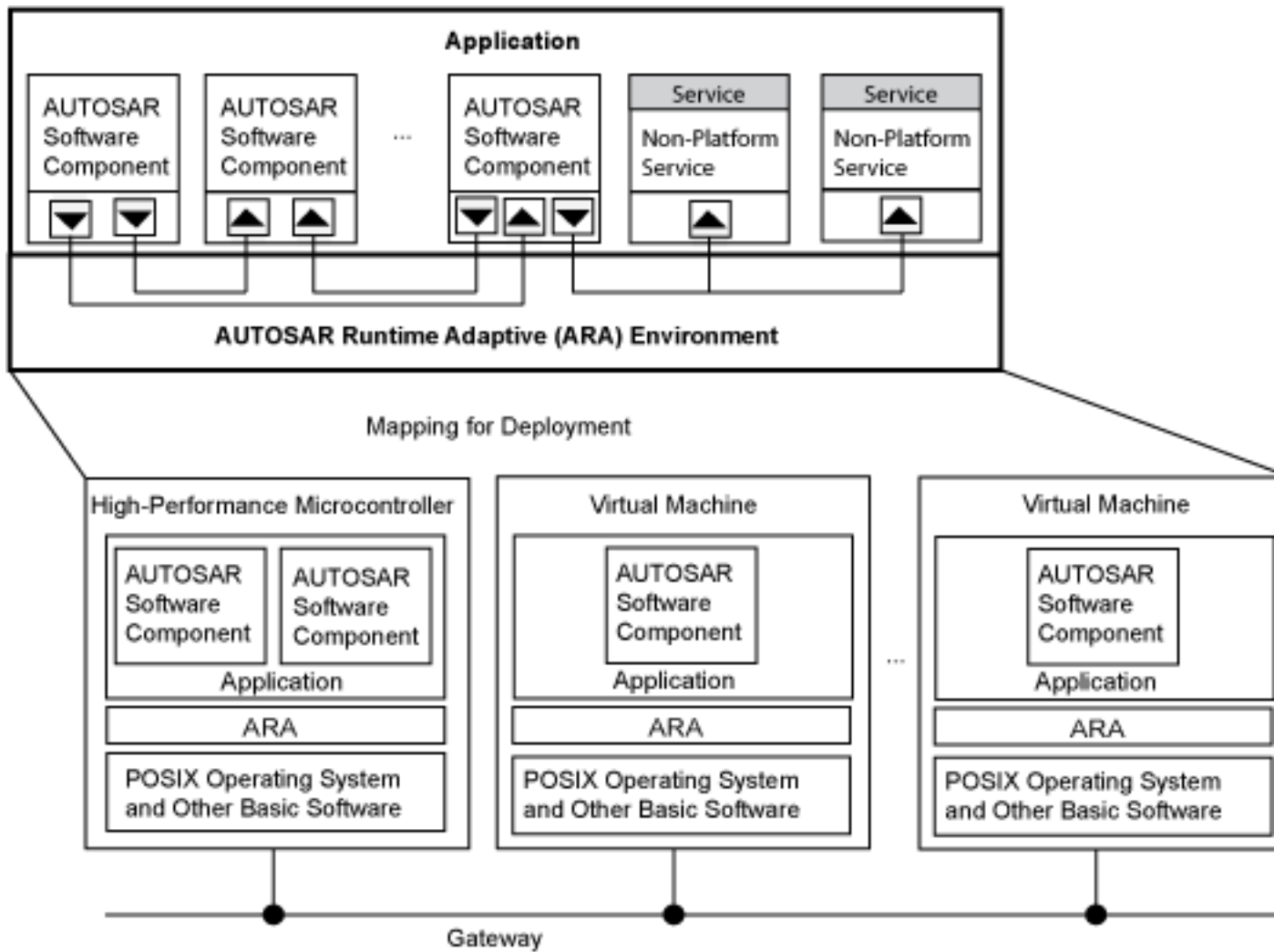
Examples of services include:

- Update and configuration management
- Diagnostics
- Signal-to-service mapping
- Network management

ECU hardware on which a single instance of an Adaptive Platform application runs is a machine. A machine might be one or more chips or a virtual hardware component. The hardware can be a single chip that hosts one or more machines or multiple chips that host a single machine.



The Adaptive Platform supports hardware-independent development and usage of AUTOSAR application software. Abstract representations of RTEs for specific ECUs (microcontrollers, high-performance microcontrollers, and virtual machines) decouple AUTOSAR software components in the Application layer of the architecture from the architecture infrastructure. AUTOSAR software components and foundation software and services communicate by using dedicated ports. You configure an application by mapping component ports to the RTE representations of the system ECUs.



See Also

More About

- <https://www.autosar.org>

AUTOSAR Software Components and Compositions

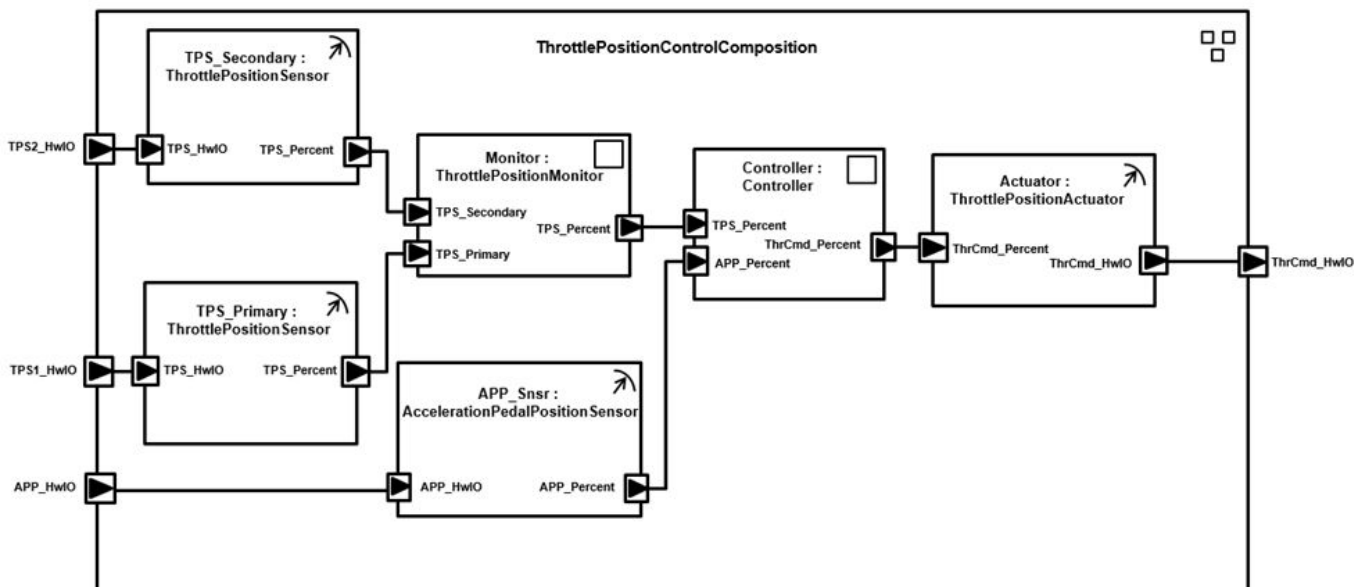
AUTOSAR software components are reusable building blocks of AUTOSAR software. An AUTOSAR software component encapsulates one or more algorithms and communicates with its environment through well-defined ports. For example, a throttle application might include AUTOSAR software components that represent sensors for throttle and acceleration pedal sensors, a throttle position monitor, a controller, and an actuator.

An AUTOSAR software component connects to an AUTOSAR runtime environment for communicating with other software components and software in the Basic Software layer of the AUTOSAR software architecture. You can reuse and relocate software components between ECUs.

In Simulink, you represent AUTOSAR software components with Simulink model components, such as Model, subsystem, and Simulink Function blocks.

AUTOSAR compositions are AUTOSAR software components that aggregate groups of software components that have related functionality. A composition is a system abstraction that facilitates scalability and helps to manage complexity when designing the logical representation of a software application.

This figure shows a composition for throttle position control.



The composition consists of software components that represent:

- Two throttle position sensors
- Throttle position monitor
- Acceleration pedal position sensor
- Controller
- Throttle position actuator

If you are using the AUTOSAR Classic Platform, you can model an AUTOSAR software composition by importing an ARXML description of a composition into Simulink or by using an AUTOSAR architecture model to author a software composition (requires System Composer).

See Also

More About

- <https://www.autosar.org>
- “Develop AUTOSAR Software Component Model” on page 1-16
- “Develop AUTOSAR Adaptive Software Component Model” on page 1-28
- “Model AUTOSAR Software Components” on page 2-3
- “Model AUTOSAR Adaptive Software Components” on page 6-2
- “Import AUTOSAR XML Descriptions Into Simulink” on page 3-13
- “Import AUTOSAR Adaptive Software Descriptions” on page 6-12
- “Create AUTOSAR Software Component in Simulink” on page 3-2
- “Import AUTOSAR Composition to Simulink” on page 7-2
- “Import AUTOSAR Software Composition with Atomic Software Components (Classic Platform)” on page 3-24
- “Author AUTOSAR Classic Compositions and Components in Architecture Model” on page 8-50

Workflows for AUTOSAR

To develop AUTOSAR software components in Simulink, you create a Simulink representation of an AUTOSAR software component. AUTOSAR component creation can start from an existing Simulink design or from an AUTOSAR XML (ARXML) component description created in another development environment.

In a Simulink originated (*bottom-up*) workflow, you take an existing Simulink design or algorithm and map it into an AUTOSAR software component model.

In a *round-trip* workflow, you import an AUTOSAR component description created by an authoring tool in another development environment. Importing the component specification into Simulink creates an AUTOSAR software component model.

In this section...

“Simulink Originated (Bottom-Up) Workflow” on page 1-13

“Round-Trip Workflow” on page 1-13

Simulink Originated (Bottom-Up) Workflow

In a Simulink originated, or *bottom-up*, workflow, you take a design or algorithm that originated in Simulink and configure it into an AUTOSAR software component model. To get started, use the AUTOSAR Component Quick Start or AUTOSAR model templates on the Simulink Start Page. For more information, see “Create AUTOSAR Software Component in Simulink” on page 3-2.

You develop the component design and behavior in Simulink. For example, you configure AUTOSAR software component elements, map Simulink model elements to AUTOSAR software component elements, develop component behavior algorithms, and simulate the component behavior.

Using Simulink Coder™ and Embedded Coder, you can generate AUTOSAR-compliant XML descriptions and C or C++ code from the component model. You can test the code in Simulink or integrate the descriptions and code in an AUTOSAR run-time environment.

Round-Trip Workflow

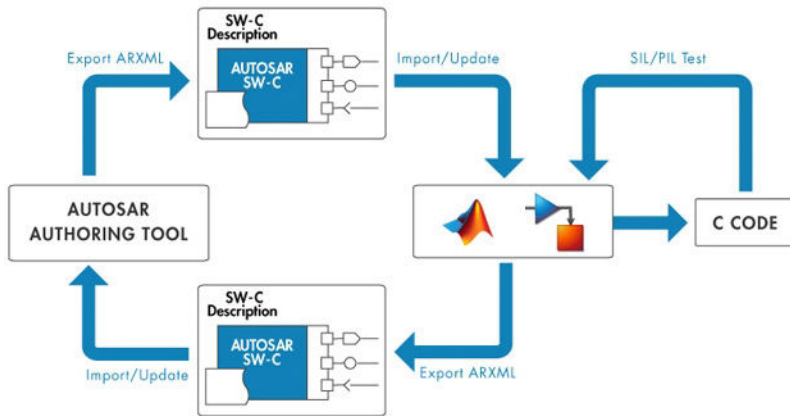
In a *round-trip* workflow, you import an AUTOSAR software component description created in another development environment into Simulink. Simulink can import AUTOSAR-compliant XML descriptions exported by common AUTOSAR authoring tools (AATs). Importing the XML description of an AUTOSAR software component creates a Simulink model representation of the component. For more information, see “Import AUTOSAR XML Descriptions Into Simulink” on page 3-13 or “Import AUTOSAR Adaptive Software Descriptions” on page 6-12.

As with a Simulink originated design, you develop the component design and behavior in Simulink. For example, you configure AUTOSAR software component elements, map Simulink model elements to AUTOSAR software component elements, develop component behavior algorithms, and simulate the component behavior.

Using Simulink Coder and Embedded Coder, you can generate AUTOSAR-compliant XML descriptions and C or C++ code from the component model for testing or integration.

In a round-trip workflow, you deliver the generated description files and code back to the originating AAT. Using the AAT, merge your Simulink design work with other components and systems. If you

further modify the component in the other development environment, use the AAT to export updated XML specifications. In your Simulink environment, import the new descriptions and update your component model to reflect the changes. For more information, see “Import AUTOSAR Software Component Updates” on page 3-25.



To support the round trip of AUTOSAR elements between an AAT and Simulink, ARXML import preserves imported AUTOSAR XML file structure and content for ARXML export. For more information, see “Round-Trip Preservation of AUTOSAR XML File Structure and Element Information” on page 3-37.

See Also

Related Examples

- “Create AUTOSAR Software Component in Simulink” on page 3-2
- “Import AUTOSAR XML Descriptions Into Simulink” on page 3-13
- “Import AUTOSAR Software Component Updates” on page 3-25
- “Import AUTOSAR Adaptive Software Descriptions” on page 6-12
- “Round-Trip Preservation of AUTOSAR XML File Structure and Element Information” on page 3-37

AUTOSAR Workflow Samples

Example	How to ...
“Create and Configure AUTOSAR Software Component” on page 3-8	Create an AUTOSAR software component model from an algorithm model.
“Import AUTOSAR Component to Simulink” on page 3-19	Create Simulink model from XML description of AUTOSAR software component.
“Design and Simulate AUTOSAR Components and Generate Code” on page 4-77	Develop AUTOSAR components by implementing behavior algorithms, simulating components and compositions, and generating component code.
“Modeling Patterns for AUTOSAR Runnables” on page 2-10	Use Simulink models, subsystems, and functions to model AUTOSAR atomic software components and their runnable entities (runnables).
“Create and Configure AUTOSAR Adaptive Software Component” on page 6-6	Create an AUTOSAR adaptive software component model from an algorithm model.
“Import AUTOSAR Adaptive Components to Simulink” on page 6-13	Create Simulink models from XML descriptions of AUTOSAR adaptive software components.
“Simulate AUTOSAR Basic Software Services and Run-Time Environment” on page 7-36	Simulate AUTOSAR component calls to Basic Software memory and diagnostic services using reference implementations.
“Generate AUTOSAR C Code and XML Descriptions” on page 5-2	With Embedded Coder software, generate AUTOSAR-compliant C code and export AUTOSAR XML (ARXML) descriptions from AUTOSAR component model.
“Generate AUTOSAR Adaptive C++ Code and XML Descriptions” on page 6-68	With Embedded Coder software, generate AUTOSAR-compliant C++ code and export AUTOSAR XML (ARXML) descriptions from AUTOSAR adaptive component model.
“Author AUTOSAR Classic Compositions and Components in Architecture Model” on page 8-50	With System Composer software, use an architecture model to develop AUTOSAR compositions and components for the Classic Platform.
“AUTOSAR Property and Map Function Examples” on page 4-299	Programmatically add AUTOSAR elements to a model, configure AUTOSAR properties, and map Simulink elements to AUTOSAR elements.

See Also

More About

- “Workflows for AUTOSAR” on page 1-13
- <https://www.autosar.org>

Develop AUTOSAR Software Component Model

Prerequisites

This tutorial assumes that you are familiar with the basics of the AUTOSAR standard and Simulink. The code generation portion of this tutorial assumes that you have knowledge of Embedded Coder basics.

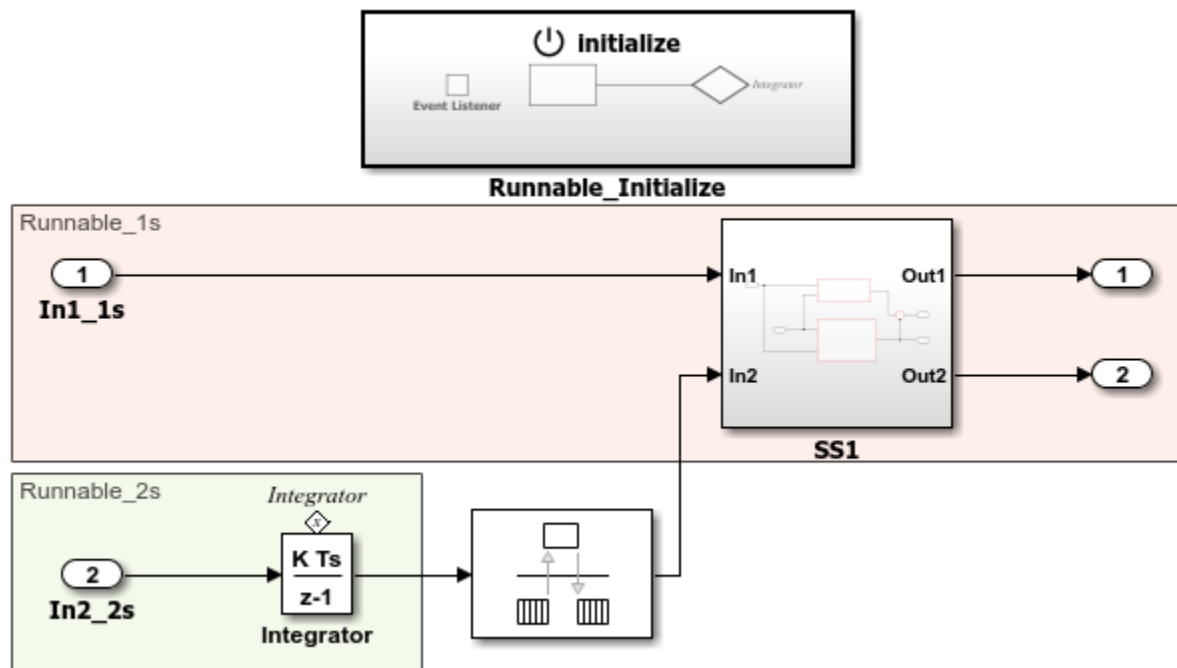
To complete this tutorial, you must have:

- MATLAB®
- Simulink

An optional part of the tutorial requires Simulink Coder and Embedded Coder software.

Example Model

The tutorial uses example models `swc` and `autosar_swc`.



What You Will Learn

You will learn how to:

- 1 Create algorithmic model content that represents AUTOSAR software component behavior.
- 2 Configure elements of an AUTOSAR software component for the Simulink modeling environment.
- 3 Simulate the AUTOSAR software component.
- 4 Optionally, generate AUTOSAR software component code.

To start the tutorial, see “Create Algorithmic Model Content That Represents AUTOSAR Software Component Behavior” on page 1-18.

Create Algorithmic Model Content That Represents AUTOSAR Software Component Behavior

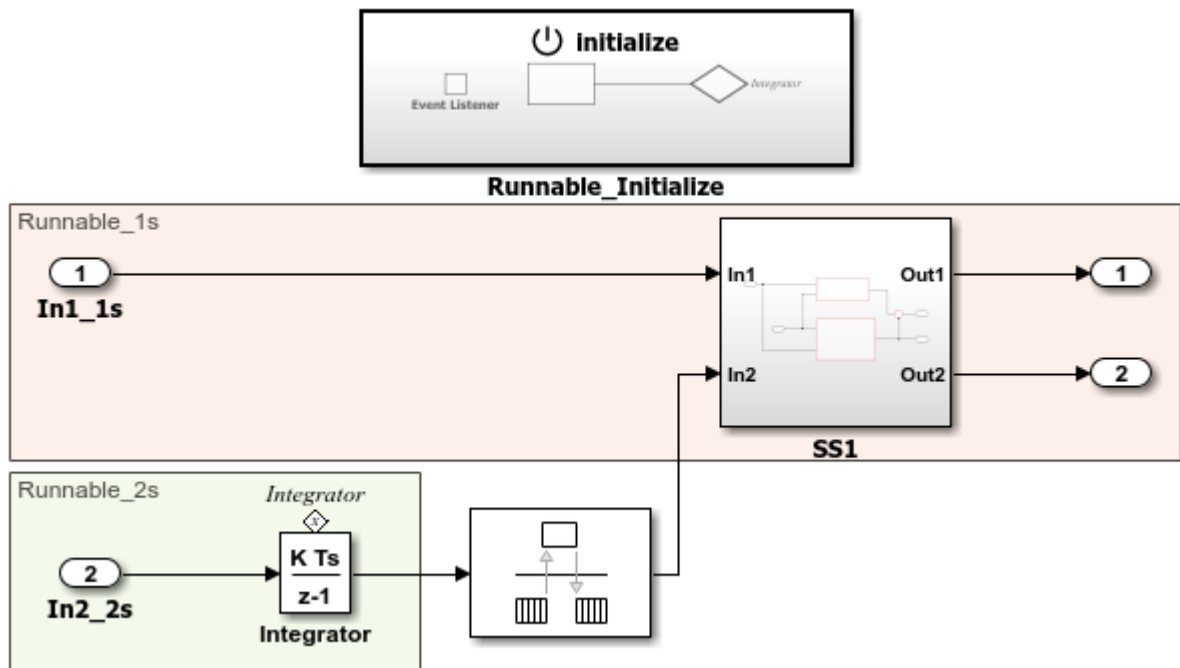
AUTOSAR Blockset software supports AUTOSAR software component modeling for the AUTOSAR Classic Platform. To develop an AUTOSAR software component in Simulink, create a Simulink model that represents the AUTOSAR software component. Initiate the model creation in one of these ways:

- Import an existing AUTOSAR XML (ARXML) component description into the Simulink environment as a model. You import a component description by using the AUTOSAR ARXML importer.
- Rework an existing Simulink model into a representation of the AUTOSAR software component.
- Starting from an AUTOSAR Blockset model template, create a Simulink model.

After creating an initial model design, refine the algorithmic content.

This tutorial uses example model `autosar_sw_c` to show a sample model representation of an AUTOSAR software component.

- 1 Open model `autosar_sw_c`.



- 2 Explore the model components. The model consists of:
 - Periodic runnable `Runnable_1s`, which is configured with a sample rate of 1 second (`In1_1s`).
 - Periodic runnable `Runnable_2s`, which is configured with a sample rate of 2 seconds (`In2_2s`).
 - Initialize Function block, `Runnable_Initialize`, which initializes the integrator in `Runnable_2s` to a value of 1.
- 3 Explore the model configuration.

Model configuration parameter **System target file** is set to `autosar.tlc`. That system target file setting enables use of AUTOSAR Blockset software.

To maximize execution efficiency, the model is configured for multitasking mode. Solver settings are:

- **Type** is set to Fixed-step.
- **Solver** is set to discrete (no continuous states).
- **Fixed-step size (fundamental sample time)** is set to auto.
- **Treat each discrete rate as a separate task** is selected.

In the Simulink Editor, you can enable sample time color-code by selecting the **Debug** tab and selecting **Diagnostics > Information Overlays > Colors**. A sample time legend shows the implicit rate grouping. Red represents the fastest discrete rate. Green represents the second fastest discrete rate. Yellow represents a mixture of the two rates.

Because the model has multiple rates and the **Solver** parameter **Treat each discrete rate as a separate task** is selected, the model simulates in multitasking mode. The model handles the rate transition for `In2_2s` explicitly by using the Rate Transition block.

The Rate Transition block parameter **Ensure deterministic data transfer** is cleared to facilitate integration into an AUTOSAR run-time environment.

Generated code for the model schedules subrates in the model. For this model, the rate for Inport block `In2_2s`, the green rate, is a subrate. The generated code properly transfers data between tasks that run at different rates.

Next, configure elements of the AUTOSAR software component for use in the Simulink modeling environment.

See Also

Related Examples

- “Modeling Patterns”

Configure Elements of AUTOSAR Software Component for Simulink Modeling Environment

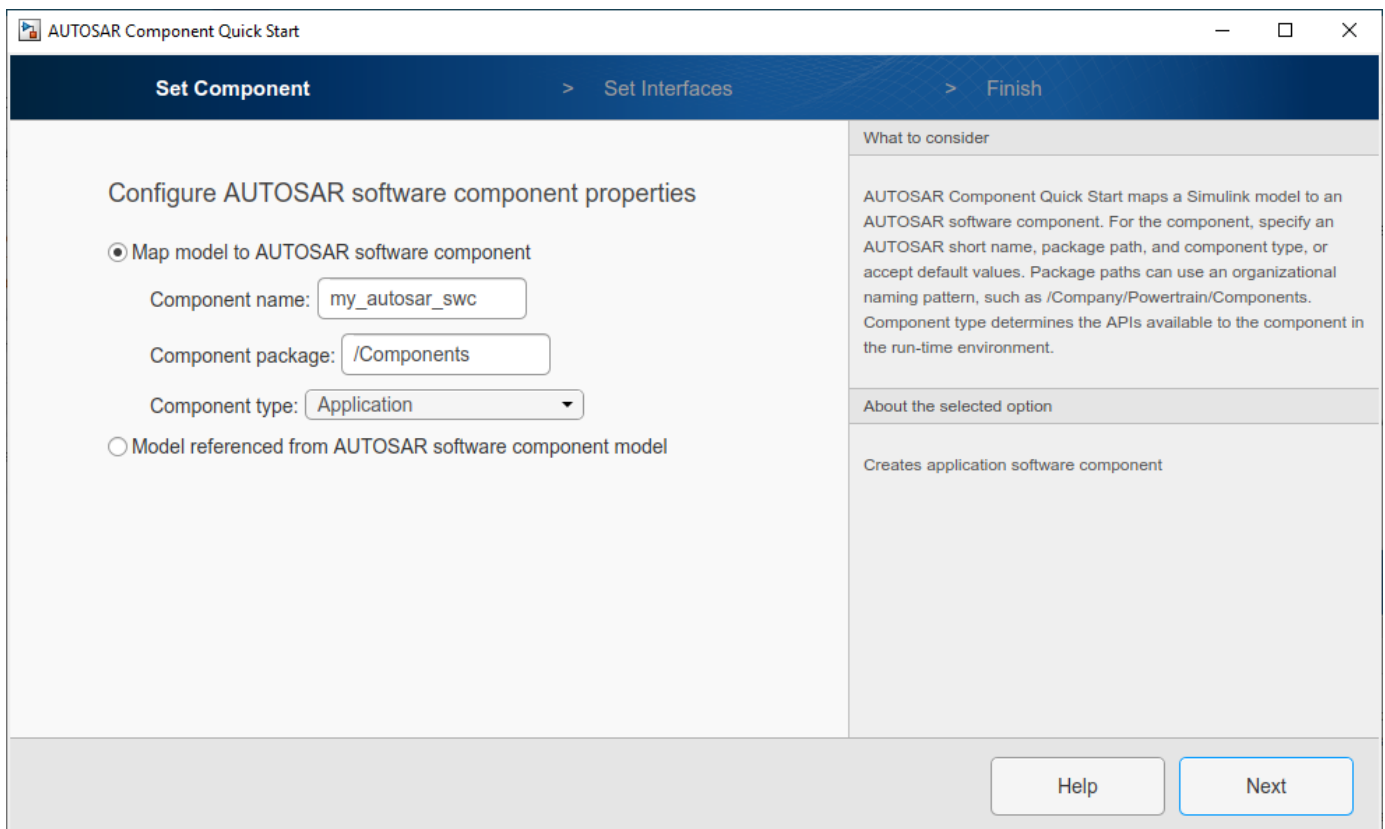
After you create a representation of the AUTOSAR software component in the Simulink Editor, configure elements of the software component for use in Simulink. The configuration maps AUTOSAR software component elements to Simulink modeling elements.

AUTOSAR Blockset software reduces the effort of setting up a configuration by providing an AUTOSAR Component Quick Start tool. If necessary, you can modify the initial configuration by using the Code Mappings editor and the AUTOSAR Dictionary.

Set Up Initial Component Configuration

Set up an initial configuration of an AUTOSAR software component by using the AUTOSAR Component Quick Start tool.

- 1 Open example model `swc`, an unconfigured version of `autosar_swc`.
- 2 Save a copy of the example model to a writable folder on your current MATLAB search path. Name the file `my_autosar_swc.slx`.
- 3 Set model configuration parameter **System target file** to `autosar.tlc`.
- 4 Run the AUTOSAR Component Quick Start tool. From the **Apps** tab, open the AUTOSAR Component Designer app. When you open the app for an unmapped model that is configured with an AUTOSAR system target file, the AUTOSAR Component Quick Start tool runs.

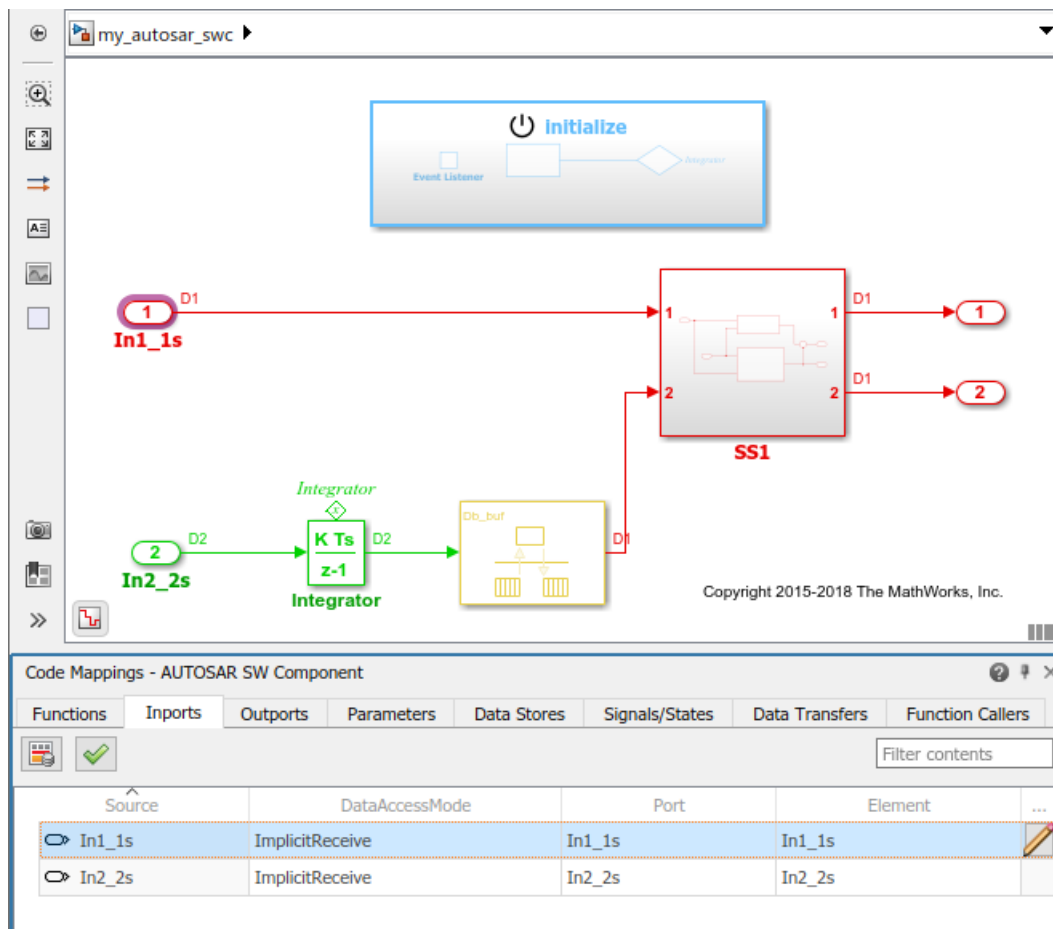


- 5 Advance through the steps of the AUTOSAR Component Quick Start tool. Each step prompts you for input that the tool uses to configure your AUTOSAR software component for the Simulink environment.
- The name, package, and type of the AUTOSAR software component that you are configuring.
 - Whether you want to use default properties based on the model or import AUTOSAR software component properties from an ARXML file.

For this tutorial, use the defaults.

After you click **Finish**, the tool:

- Creates a mapping between elements of the AUTOSAR software component and Simulink model elements.
- Opens the model in the Simulink Editor AUTOSAR Code perspective. The AUTOSAR Code perspective displays the model and directly below the model, the Code Mappings editor.
- Displays the AUTOSAR software component mappings in the Code Mappings editor, which you can use to customize the configuration.



- 6 Save the model.


Customize Component Configuration

The AUTOSAR Component Quick Start tool sets up an initial configuration for an AUTOSAR software component. To refine or make changes to an existing component configuration, use the Code Mappings editor and the AUTOSAR Dictionary.

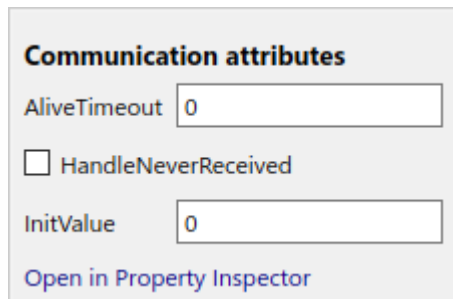
In a tabbed table format, the Code Mappings editor displays Simulink model elements, such as entry-point functions, inports, outports, and data transfers. Use the editor to map Simulink model elements to AUTOSAR software component elements. AUTOSAR software component elements are defined in the AUTOSAR standard. They include runnable entities, ports, and inter-runnable variables (IRVs).

- 1 If not already open, open model `my_autosar_sw`.
- 2 In the Code Mappings editor, select the **Inports** tab.
- 3 Select model inport `In1_1s`. Selecting the inport highlights the corresponding element in the model. The inport is mapped to AUTOSAR port `In1_1s` and data element `In1_1s` with data access mode `ImplicitReceive`.

In each Code Mappings editor tab, you can select model elements and modify their AUTOSAR mapping and attributes. Modifications are reflected in generated ARXML descriptions and C code.

- 4 Modify attribute settings for a mapped model element. For this tutorial, modify communication attributes for inport `In1_1s`. Click the  icon and change:


- **AliveTimeout** from 0 to 30
- **HandleNeverReceived** from cleared to selected
- **InitValue** from 0 to 1



- 5 Save the model.

Configure AUTOSAR Software Component Elements from AUTOSAR Standard Perspective

Configure AUTOSAR software component elements from the perspective of the AUTOSAR standard by using the AUTOSAR Dictionary.

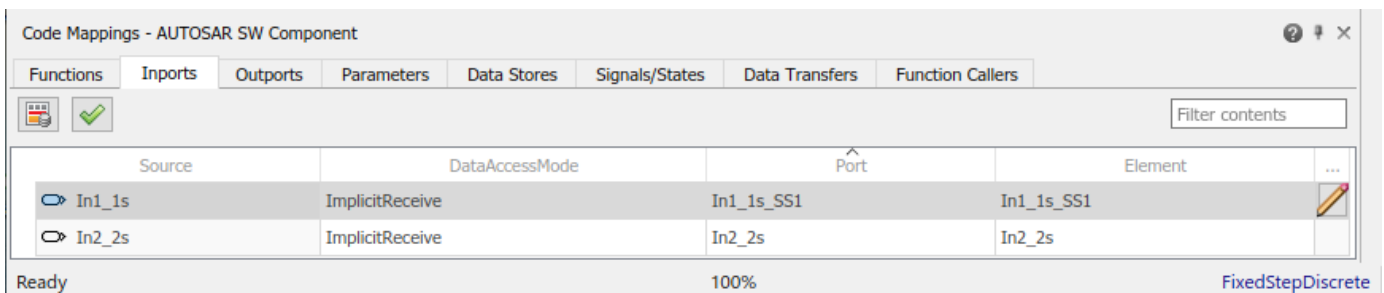
- 1 If not already open, open model `my_autosar_sw`.
- 2 Open the AUTOSAR Dictionary. In the Code Mappings editor, click the AUTOSAR Dictionary button . The AUTOSAR Dictionary opens in the AUTOSAR view that corresponds to the Simulink element that you last selected and mapped in the Code Mappings editor. If you selected

and mapped a Simulink inport, the dictionary opens in ReceiverPorts view and displays the AUTOSAR port that you mapped the inport to.

In a tree format, the AUTOSAR Dictionary displays the mapped AUTOSAR software component and its elements, communication interfaces, computation methods, software address methods, and XML options.

- 3 Use the AUTOSAR Dictionary to further customize component configurations. In the ReceiverPorts view, select port `In1_1s`, the AUTOSAR receiver port to which the Simulink inport was mapped. An attributes panel appears, showing the attributes settings for that element.
- 4 In the AUTOSAR Dictionary, rename the AUTOSAR receiver port `In1_1s` to `In1_1s_SS1`. To initiate the edit, double-click the **Name** value field.

The Code Mappings editor reflects the name change.



- 5 Save the model.

Next, simulate the AUTOSAR software component.


See Also

Related Examples

- “Create AUTOSAR Software Component in Simulink” on page 3-2
- “Map AUTOSAR Elements for Code Generation” on page 4-50
- “Configure AUTOSAR Elements and Properties” on page 4-8

Simulate AUTOSAR Software Component

After you configure the AUTOSAR software component model for use in the Simulink environment, simulate model `my_autosar_sw`, which you configured in “Configure Elements of AUTOSAR Software Component for Simulink Modeling Environment” on page 1-20.

- 1** If not already open, open your configured version of model `my_autosar_sw`.
- 2** In the Simulink Editor, click the Simulate button .

If you have access to Simulink Coder and Embedded Coder software, next, generate code for the AUTOSAR model.

Optional: Generate AUTOSAR Software Component Code (Requires Embedded Coder)

If you have access to Simulink Coder and Embedded Coder software, you can build an AUTOSAR model. When you build an AUTOSAR model, the code generator produces C code that complies with the AUTOSAR standard and ARXML descriptions.

- 1 If not already open, open your configured version of model `my_autosar_sw`.
- 2 Initiate code generation by pressing **Ctrl+B**. The code generator produces C code and ARXML files. The generated code complies with the AUTOSAR standard so that you can schedule the code with the AUTOSAR run-time environment.

The code generator also produces and displays a code generation report.

- 3 In the code generation report, review the generated code. In your current MATLAB folder, the `my_autosar_sw_autosar_rtw` folder contains the primary files listed in this table.

Generated Code Files

Files	Description
<code>my_autosar_sw.c</code>	Contains entry points for the code that implements the model algorithm. This file includes rate scheduling code.
<code>my_autosar_sw.h</code>	Declares model data structures and a public interface to the model entry points and data structures.
<code>rtwtypes.h</code>	Defines data types, structures, and macros that the generated code requires.
<code>my_autosar_sw_component.arxml</code> <code>my_autosar_sw_datatype.arxml</code> <code>my_autosar_sw_implementation.arxml</code> <code>my_autosar_sw_interface.arxml</code>	Contain elements and objects that represent AUTOSAR software components, ports, interfaces, data types, and packages. You integrate ARXML files into an AUTOSAR run-time environment. You can import ARXML files into the Simulink environment by using the AUTOSAR ARXML importer tool.

- 4 Open and review the Code Interface Report. This information is captured in the ARXML files. The run-time environment generator uses the ARXML descriptions to interface the code into an AUTOSAR run-time environment.

Entry-point functions:

- Initialization entry-point function — `void my_autosar_sw_Init(void)`. At startup, call this function once.
- Output and update entry-point function — `void my_autosar_sw_Step(void)`. Call this function periodically at the fastest rate in the model. For this model, call the function every second. To achieve real-time execution, attach this function to a timer.
- Output and update entry-point function — `void my_autosar_sw_Step1(void)`. Call this function periodically at the second fastest rate in the model. For this model, call the function every 2 seconds. To achieve real-time execution, attach this function to a timer.

The entry-point functions are also accessible in the Code Mappings editor, on the **Functions** tab. You call these generated functions from external code or from a version of a generated main function that you modify. If required, you can change the name of a function. For the base-rate step function of a rate-based model and for step functions for export function models, you can customize the function name and arguments.

Input ports:

- Block In1_1s — Require port, interface: sender-receiver of type real-T of 1 dimension
- Block In2_2s — Require port, interface: sender-receiver of type real-T of 1 dimension

Output ports:

- Block Out1 — Provide port, interface: sender-receiver of type real-T of 1 dimension
- Block Out2 — Provide port, interface: sender-receiver of type real-T of 1 dimension

- 5 Check whether the configuration changes that you made appear in the generated code by using the Code panel in the Code perspective. To open the Code panel, on the AUTOSAR tab, click **View Code**. The Code panel opens to the right of the model. In the search field, type In1_1s_SS1, the new name for AUTOSAR software component port In1_1s. Then, click the arrow button to advance to instances of the name in the ARXML file my_autosar_sw_component.arxml. Verify that the settings of communication attributes that you modified for the AUTOSAR software component port appear correctly.

```

<PORTS>
  <R-PORT-PROTOTYPE UUID="7b644055-8cb0-500d-612e-c39
    <SHORT-NAME>In1_1s_SS1</SHORT-NAME>
    <REQUIRED-COM-SPECS>
      <NONQUEUED-RECEIVER-COM-SPEC>
        <DATA-ELEMENT-REF DEST="VARIABLE-DATA-F
        <HANDLE-OUT-OF-RANGE>NONE</HANDLE-OUT-C
        <USES-END-TO-END-PROTECTION>>false</USES
        <ALIVE-TIMEOUT>30</ALIVE-TIMEOUT>
        <ENABLE-UPDATE>>false</ENABLE-UPDATE>
        <HANDLE-NEVER-RECEIVED>>true</HANDLE-NEV
        <HANDLE-TIMEOUT-TYPE>NONE</HANDLE-TIMEC
        <INIT-VALUE>
          <NUMERICAL-VALUE-SPECIFICATION>
            <SHORT-LABEL>DefaultInitValue_D
            <VALUE>1</VALUE>

```

- 6 Use the Code perspective Code panel to explore other aspects of the generated code. For example, if you select file my_autosar_sw.c, and then click in the search field, a list of links to code elements, including the entry-point functions, appears. Use the links to quickly navigate to key areas of the generated C code.

See Also

Related Examples

- “Configure AUTOSAR Code Generation” on page 5-7

Develop AUTOSAR Adaptive Software Component Model

Prerequisites

This tutorial assumes that you are familiar with the basics of the AUTOSAR standard and Simulink. The code generation portion of this tutorial assumes that you have knowledge of Embedded Coder basics.

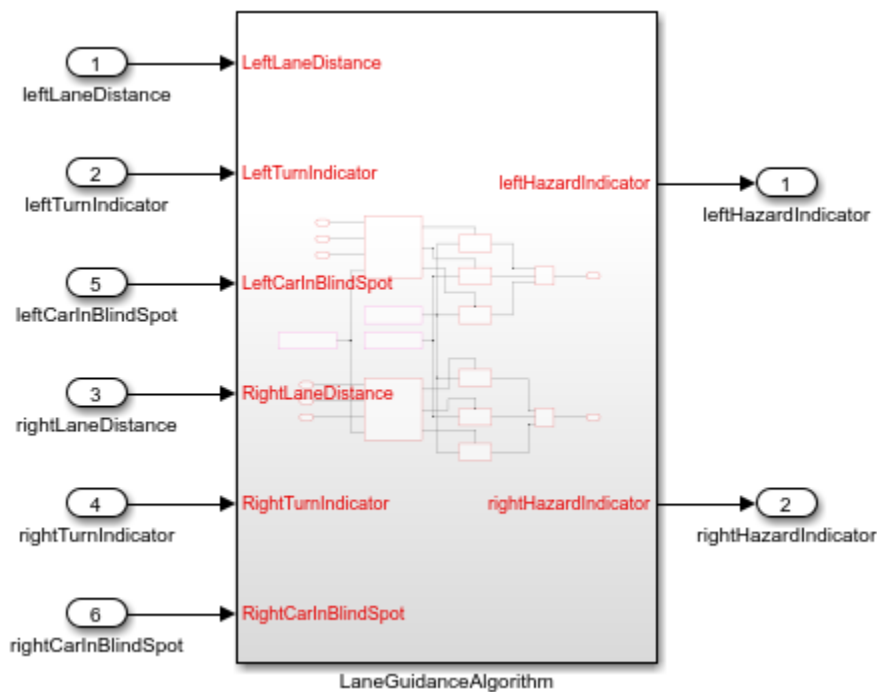
To complete this tutorial, you must have:

- MATLAB
- Simulink

An optional part of the tutorial requires Simulink Coder and Embedded Coder software.

Example Model

The tutorial uses example models LaneGuidance and autosar_LaneGuidance.



What You Will Learn

You will learn how to:

- 1 Create algorithmic model content that represents AUTOSAR adaptive software component behavior.
- 2 Configure elements of an AUTOSAR adaptive software component for the Simulink modeling environment.

- 3** Simulate the AUTOSAR adaptive software component.
- 4** Optionally, generate AUTOSAR adaptive software component code.

To start the tutorial, see “Create Algorithmic Model Content That Represents AUTOSAR Adaptive Software Component Behavior” on page 1-30.

Create Algorithmic Model Content That Represents AUTOSAR Adaptive Software Component Behavior

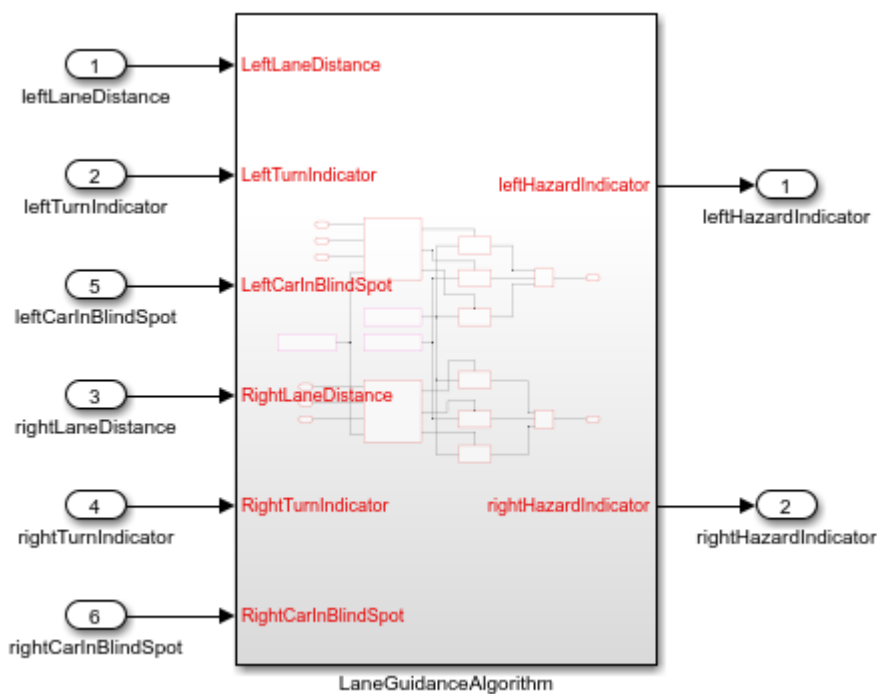
AUTOSAR Blockset software supports AUTOSAR software component modeling for the AUTOSAR Adaptive Platform. To develop an AUTOSAR adaptive software component in Simulink, create a Simulink model that represents the AUTOSAR adaptive software component. Initiate the model creation in one of these ways:

- Import an existing AUTOSAR XML (ARXML) component description into the Simulink environment as a model. You import a component description by using the AUTOSAR ARXML importer.
- Rework an existing Simulink model into a representation of the AUTOSAR adaptive software component.
- Starting from an AUTOSAR Blockset model template, create a Simulink model.

After creating an initial model design, refine the algorithmic content.

This tutorial shows a sample model representation of an AUTOSAR adaptive software component.

- 1 Open model LaneGuidance.



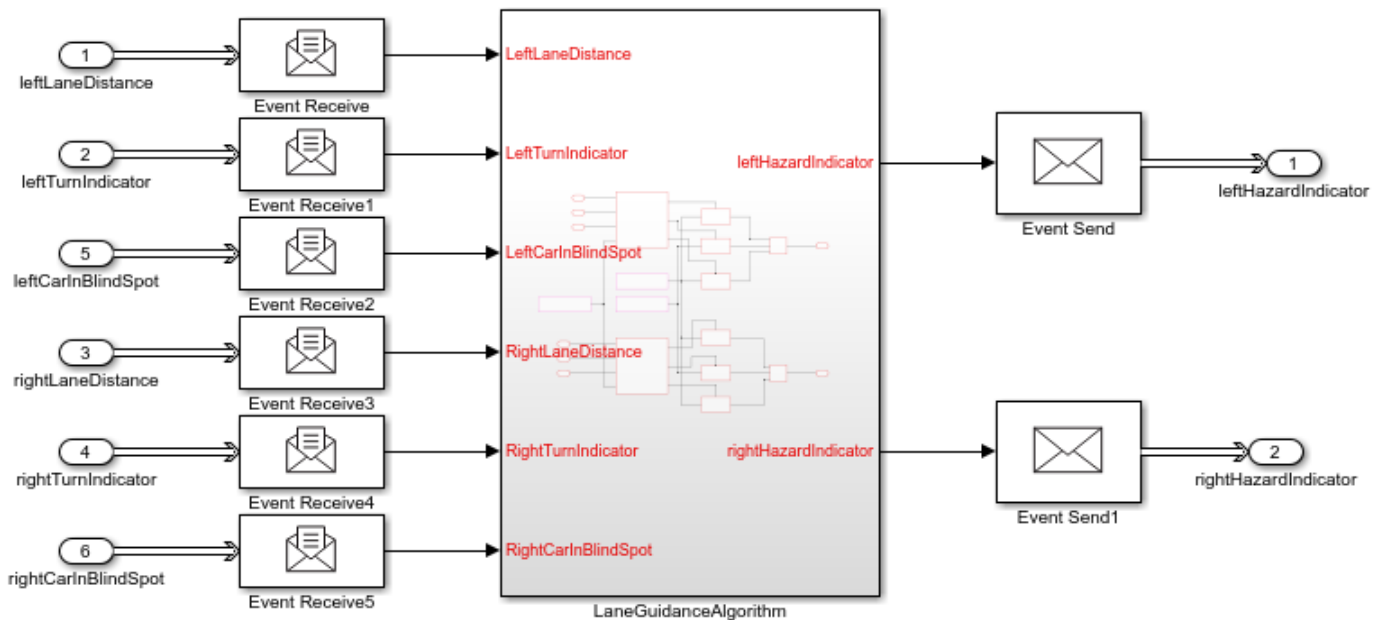
- 2 Explore the model. It consists of a subsystem, LaneGuidanceAlgorithm. The subsystem has six inports, which represent required ports of the AUTOSAR adaptive software component: leftLaneDistance, leftTurnIndicator, leftCarInBlindSpot, rightLaneDistance, rightTurnIndicator, and rightCarInBlindSpot. Two outports represent provider ports: leftHazardIndicator and rightHazardIndicator.
- 3 Set model configuration parameter **System target file** to `autosar_adaptive.tlc`. That system target file setting enables use of AUTOSAR Blockset software and affects other model configuration parameter settings. For example:

- **Language** is set to C++.
 - **Generate code only** is selected.
 - **Toolchain** is set to AUTOSAR Adaptive | CMake.
 - **Code interface packaging** is set to C++ class.
- 4 At the top level of the model, set up event-based communication. An AUTOSAR adaptive software component provides and consumes services. Each component contains:
- An algorithm that performs tasks in response to received events
 - Required and provided ports, each associated with a service interface
 - Service interfaces, with associated events and associated namespaces

AUTOSAR Blockset provides Event Receive and Event Send blocks to make the necessary event and signal connections.

- After each root inport, add an Event Receive block, which converts an input event to a signal while preserving the signal values and data type.
- Before each root outport, add an Event Send block, which converts an input signal to an event while preserving the signal values and data type.

To expedite the block insertion, you can copy the event blocks from the completed version of the example model `autosar_LaneGuidance`.



- 5 Explore the model configuration. Solver settings are:
- **Type** is set to Fixed-step.
 - **Solver** is set to auto (Automatic solver selection).
 - **Fixed-step size (fundamental sample time)** is set to 1/10.
 - **Periodic same time constraint** is set to Unconstrained.

In the Simulink Editor, you can enable sample time color-code by selecting the **Debug** tab and selecting **Diagnostics > Information Overlays > Colors**. A sample time legend shows the

implicit rate grouping. The legend for this model shows that the model uses a single rate of 0.1 second. The model simulates in single-tasking mode.

- 6 Save the model to a writable folder on your current MATLAB search path. Name the file `my_autosar_LaneGuidance.slx`.

Next, configure elements of the AUTOSAR adaptive software component for use in the Simulink modeling environment.

See Also

Related Examples

- “Modeling Patterns”

Configure Elements of AUTOSAR Adaptive Software Component for Simulink Modeling Environment

After you create a representation of the AUTOSAR adaptive software component in the Simulink Editor, configure elements of the software component for use in Simulink. The configuration maps AUTOSAR adaptive software component elements to Simulink modeling elements.

AUTOSAR Blockset software reduces the effort of setting up a configuration by providing an AUTOSAR Component Quick Start tool. If necessary, you can modify the initial configuration by using the Code Mappings editor and the AUTOSAR Dictionary.

Set Up an Initial Component Configuration

Set up an initial configuration of an AUTOSAR adaptive software component by using the AUTOSAR Component Quick Start tool.

- 1 Open your saved version of the example model `my_autosar_LaneGuidance`.
- 2 Run the AUTOSAR Component Quick Start tool. From the **Apps** tab, open the AUTOSAR Component Designer app. When you open the app for an unmapped model that is configured with an AUTOSAR system target file, the AUTOSAR Component Quick Start tool runs.

The screenshot shows the 'AUTOSAR Component Quick Start' dialog box. The main section is titled 'Set Component' and contains the following text and inputs:

- Configure AUTOSAR software component properties
- Map model to AUTOSAR software component (Adaptive)
- Component name:
- Component package:

On the right side, there is a 'What to consider' section with the following text:

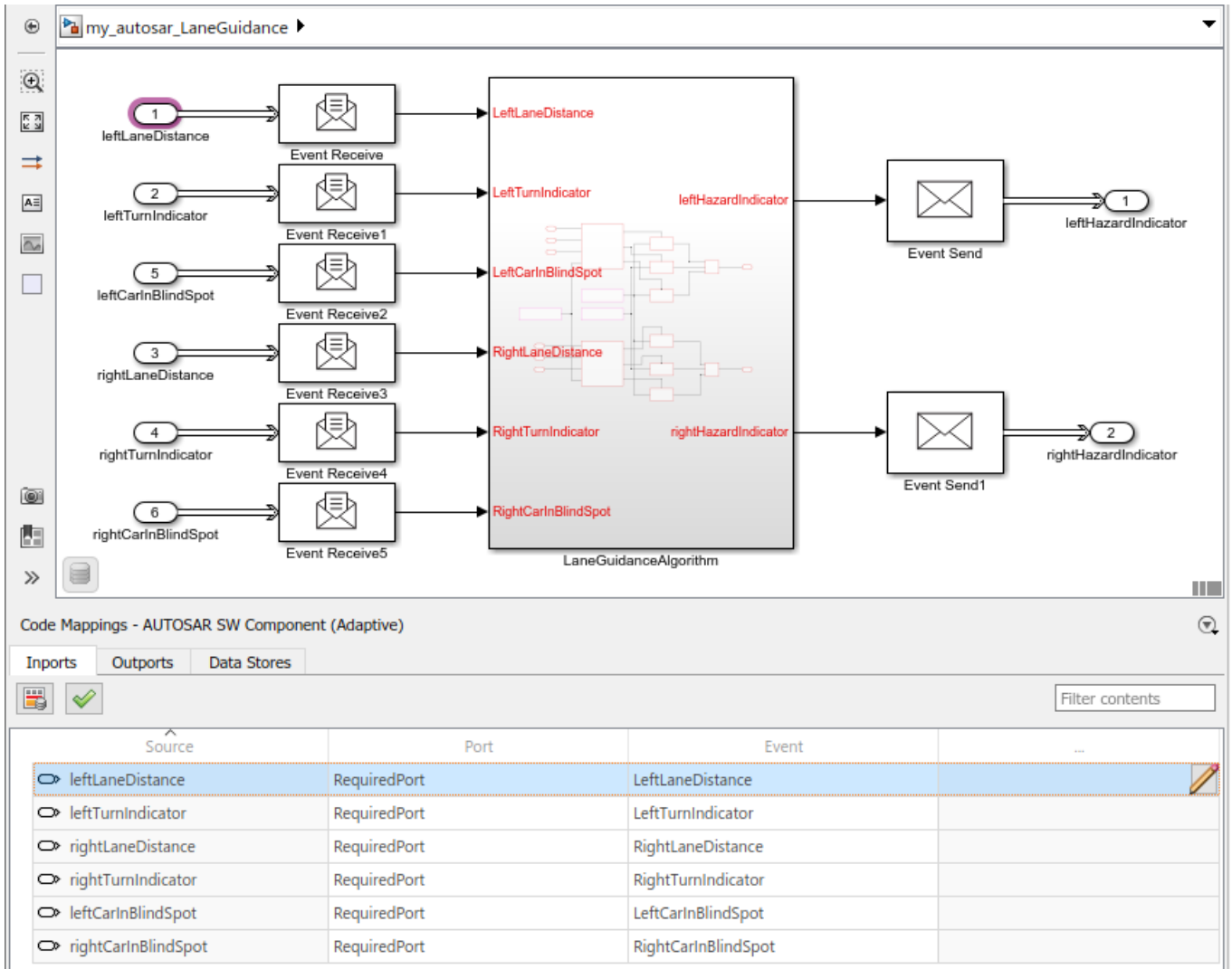
AUTOSAR Component Quick Start maps a Simulink model to an AUTOSAR software component. For the component, specify an AUTOSAR short name, package path, and component type, or accept default values. Package paths can use an organizational naming pattern, such as `/Company/Powertrain/Components`. Component type determines the APIs available to the component in the run-time environment.

At the bottom of the dialog, there are two buttons: 'Help' and 'Next'.

- 3 Advance through the steps of the AUTOSAR Component Quick Start tool. Each step prompts you for input that the tool uses to configure your AUTOSAR software component for the Simulink environment. For this tutorial, use the defaults.

After you click **Finish**, the tool:

- Creates a mapping between elements of the AUTOSAR adaptive software component and Simulink model elements.
- Opens the model in the Simulink Editor AUTOSAR Code perspective. The AUTOSAR Code perspective displays the model and directly below the model, the Code Mappings editor.
- Displays the AUTOSAR software component mappings in the Code Mappings editor, which you can use to customize the configuration.



4 Save the model.

Customize Component Configuration

The AUTOSAR Component Quick Start tool sets up an initial configuration for an AUTOSAR adaptive software component. To refine or make changes to an existing component configuration, use the Code Mappings editor and the AUTOSAR Dictionary.

In a tabbed table format, the Code Mappings editor displays Simulink model inports and outports. Map Simulink inports and outports to AUTOSAR adaptive software component ports in the editor. AUTOSAR adaptive software component ports are defined in the AUTOSAR standard.


- 1 If not already open, open model `my_autosar_LaneGuidance`.
- 2 In the Code Mappings editor, examine the mapping of Simulink inports and outports to AUTOSAR ports and events. In each tab, you can select model elements and modify their AUTOSAR mapping and attributes. Modifications are reflected in generated ARXML descriptions and C code.

Select the **Inports** tab. For each Simulink inport, the editor lists the corresponding AUTOSAR port type and event. For example, Simulink inport `leftLaneDistance` is mapped to an AUTOSAR required port and event `LeftLaneDistance`.

- 3 With a Code Mappings editor row selected, open the Property Inspector. Check whether you need to reconfigure data types or other attributes of model data. For example, verify that event data is configured correctly for the design. For this tutorial, make no changes.

Configure AUTOSAR Adaptive Software Component Elements from AUTOSAR Standard Perspective

Configure AUTOSAR software component elements from the perspective of the AUTOSAR standard perspective by using the AUTOSAR Dictionary.

- 1 If not already open, open model `my_autosar_LaneGuidance`.
- 2 Open the AUTOSAR Dictionary. In the Code Mappings editor, click the AUTOSAR Dictionary button . The AUTOSAR Dictionary opens in the AUTOSAR view, which corresponds to the Simulink element that you last selected and mapped in the Code Mappings editor. If you selected and mapped a Simulink inport, the dictionary opens in RequiredPorts view and displays the AUTOSAR port that you mapped the inport to.

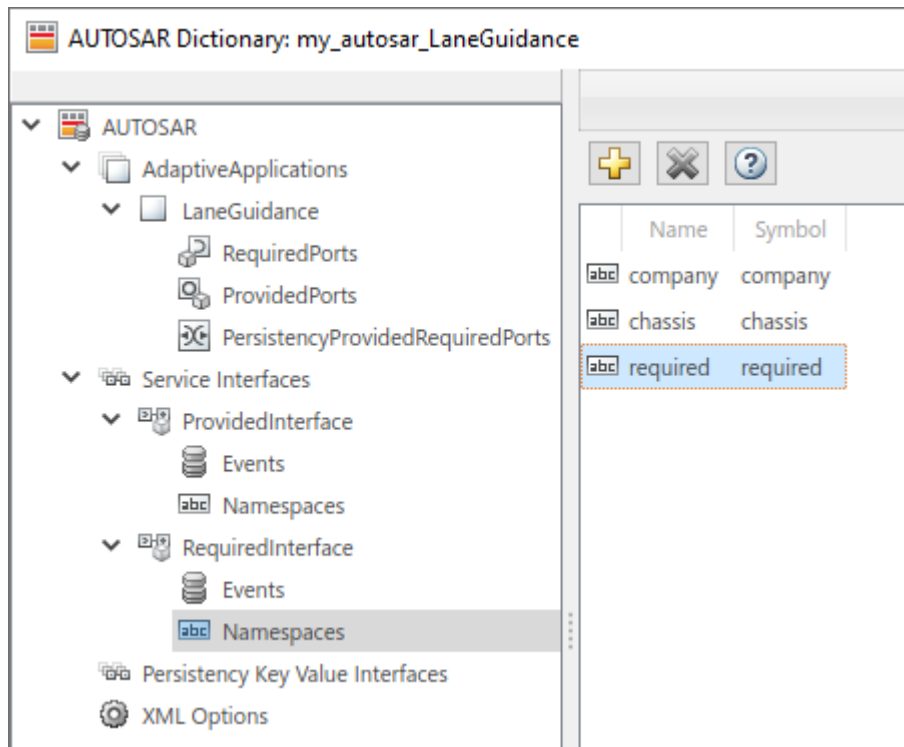
In a tree format, the AUTOSAR Dictionary displays the mapped AUTOSAR software component and its elements, interfaces, and XML options.

- 3 Use the AUTOSAR Dictionary to further customize component configurations. For example, you can use the dictionary to:
 - Expand service interface nodes to examine AUTOSAR events created during the default component mapping.
 - Define a unique namespace for each service interface. The code generator uses the defined namespaces when producing C++ code for the model.
 - Configure characteristics of exported AUTOSAR XML.

In the left pane of the dictionary, expand the tree nodes and explore what is defined for the model.

- 4 For this tutorial, add namespaces for service interfaces `ProvidedInterface` and `RequiredInterface`.
 - a In the left pane of the dictionary, expand the **Service Interfaces** and **ProvidedInterface** nodes.
 - b Select **Namespaces**.

- c In the right pane, click the plus sign.
- d Set **Name** and **Symbol** to company.
- e Add namespace entries for chassis and provided.
- f Add company, chassis, and required namespace entries for the **RequiredInterface** node.



- 5 Close the dictionary.
- 6 Save the model.

Next, simulate the AUTOSAR software component.


See Also

Related Examples

- “Create AUTOSAR Software Component in Simulink” on page 3-2
- “Map AUTOSAR Adaptive Elements for Code Generation” on page 6-37
- “Configure AUTOSAR Adaptive Elements and Properties” on page 6-21

Simulate AUTOSAR Adaptive Software Component

After you configure the AUTOSAR adaptive software component model for use in the Simulink environment, simulate model `my_autosar_LaneGuidance`, which you configured in “Configure Elements of AUTOSAR Adaptive Software Component for Simulink Modeling Environment” on page 1-33.

- 1** If not already open, open your configured version of model `my_autosar_LaneGuidance`.
- 2** In the Simulink Coder Editor, click the Simulate button .

If you have access to Simulink Coder and Embedded Coder software, next, generate code for the AUTOSAR model.

Optional: Generate AUTOSAR Adaptive Software Component Code (Requires Embedded Coder)

If you have access to Simulink Coder and Embedded Coder software, you can build an AUTOSAR adaptive model. When you build an AUTOSAR adaptive model, the code generator produces C++ code that complies with the AUTOSAR standard for the Adaptive Platform and ARXML descriptions.

- 1 If not already open, open your configured version of model `my_autosar_LaneGuidance`.
- 2 Initiate code generation by pressing **Ctrl+B**. The code generator produces C++ code and ARXML files. The generated code complies with the AUTOSAR standard so that you can schedule the code with the AUTOSAR run-time environment.

The code generator also produces and displays a code generation report.

- 3 In the code generation report, review the generated code. In your current MATLAB folder, the `my_autosar_LaneGuidance_autosar_adaptive` folder contains the primary files listed in this table.

Generated Code Files

Files	Description
<code>my_autosar_LaneGuidance.cpp</code>	Contains entry points for the code that implements the model algorithm. This file includes the rate scheduling code.
<code>my_autosar_LaneGuidance.h</code>	Declares model data structures and a public interface to the model entry points and data structures.
<code>rtwtypes.h</code>	Defines data types, structures, and macros that the generated code requires.
<code>my_autosar_LaneGuidance.arxml</code> <code>my_autosar_LaneGuidance_ExecutionManifest.arxml</code> <code>my_autosar_LaneGuidance_ServiceInstanceManifest.arxml</code>	The main ARXML file contains elements and objects that represent AUTOSAR software components, ports, interfaces, data types, and packages. The manifest files provide deployment-related and service-configuration information. You integrate ARXML files into an AUTOSAR run-time environment. You can import ARXML files into the Simulink environment by using the AUTOSAR ARXML importer tool.
<code>main.cpp</code> <code>MainUtils.hpp</code>	Provide a framework for running adaptive software component service code.

- 4 Open and review the Code Interface Report. This information is captured in the ARXML files. The run-time environment generator uses the ARXML descriptions to interface the code into an AUTOSAR run-time environment.

Entry-point functions

- Initialization entry-point function — `void my_autosar_LaneGuidanceModelClass::initialize()`. At startup, call this function once.

- Output entry-point function — `void my_autosar_LaneGuidanceModelClass::step()`. Call this function periodically, every 0.1 seconds.
- Termination entry-point function — `void my_autosar_LaneGuidanceModelClass::terminate()`. At shutdown, call this function once.

Input ports:

- Block `leftLaneDistance` — Require port, interface: sender-receiver of type `real-T` of 1 dimension
- Block `leftTurnIndicator` — Require port, interface: sender-receiver of type `real-T` of 1 dimension
- Block `rightLaneDistance` — Require port, interface: sender-receiver of type `real-T` of 1 dimension
- Block `rightTurnIndicator` — Require port, interface: sender-receiver of type `real-T` of 1 dimension
- Block `leftCarInBlindSpot` — Require port, interface: sender-receiver of type `real-T` of 1 dimension
- Block `rightCarInBlindSpot` — Require port, interface: sender-receiver of type `real-T` of 1 dimension

Output ports:

- Block `leftHazardIndicator` — Port defined externally of type `real-T` of 1 dimension
 - Block `rightHazardIndicator` — Port defined externally of type `real-T` of 1 dimension
- 5** Check whether the configuration changes that you made appear in the generated code by using the Code panel in the Code perspective. To open the Code panel, on the AUTOSAR tab, click **View Code**. The Code panel opens to the right of the model.

With file `my_autosar_LaneGuidance.cpp` selected, in the search field, type `company` (one of the namespace values that you defined for the service interfaces). The Code view highlights instances of `company`, showing how the namespace symbols are applied in the code.

```
203 // Model initialize function
204 void my_autosar_LaneGuidanceModelClass::initialize()
205 {
206 {
207     ara::com::ServiceHandleContainer< company::chassis::provided::proxy::
208         RequiredInterfaceProxy::HandleType > handles;
209     handles = company::chassis::provided::proxy::RequiredInterfaceProxy::
210         FindService(ara::com::InstanceIdentifier("1"));
211     if (handles.size() > 0U) {
212         RequiredPort = std::make_shared< company::chassis::provided::proxy::
213             RequiredInterfaceProxy >(*handles.begin());
214
215         // Subscribe event
216         RequiredPort->leftLaneDistance.Subscribe(1U);
217         RequiredPort->leftTurnIndicator.Subscribe(1U);
218         RequiredPort->leftCarInBlindSpot.Subscribe(1U);
219         RequiredPort->rightLaneDistance.Subscribe(1U);
220         RequiredPort->rightTurnIndicator.Subscribe(1U);
221         RequiredPort->rightCarInBlindSpot.Subscribe(1U);
222     }
223
224     ProvidedPort = std::make_shared< company::chassis::provided::skeleton::
225         ProvidedInterfaceSkeleton >(ara::com::InstanceIdentifier("2"), ara::com:
226         MethodCallProcessingMode::kPoll);
227     ProvidedPort->OfferService();
228 }
229 }
```

- 6 Use the Code perspective Code panel to explore other aspects of the generated code. For example, if you select file `my_autosar_LaneGuidance.cpp`, and then click in the search field, a list of links to code elements appear. Use the links to quickly navigate to key areas of the generated code.

See Also

Related Examples

- “Configure AUTOSAR Adaptive Code Generation” on page 6-73

Develop AUTOSAR Software Architecture Model

Prerequisites

This tutorial assumes that you are familiar with the basics of the AUTOSAR standard and Simulink. The code generation portion of this tutorial assumes that you have knowledge of Embedded Coder basics.

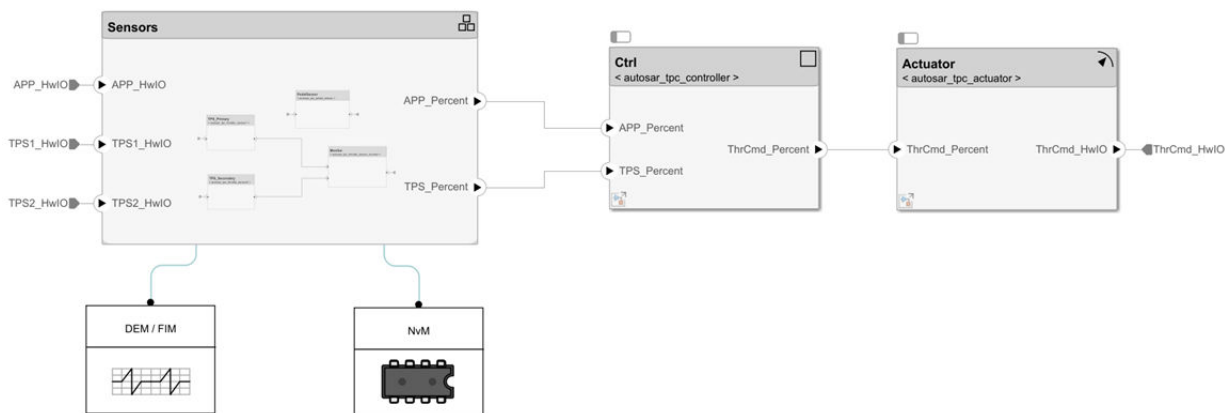
To complete this tutorial, you must have:

- MATLAB
- Simulink
- System Composer

An optional part of the tutorial requires Simulink Coder and Embedded Coder software.

Example Model

The tutorial uses example model `autosar_tpc_composition` and several supporting models that implement AUTOSAR component behavior. To open the models in a local working example folder, click this [autosar_tpc_composition](#) link or enter the MATLAB command `openExample('autosar_tpc_composition')`.



What You Will Learn

You will learn how to:

- 1 Create a software architecture canvas for developing AUTOSAR compositions and components.
- 2 Add and connect AUTOSAR compositions and components and add Simulink behavior to components.
- 3 Simulate the behavior of aggregated components in an AUTOSAR architecture model.
- 4 Optionally, export composition and component AUTOSAR XML files and generate component code from an AUTOSAR architecture model.

The tutorial focuses on how to develop an AUTOSAR classic architecture model, but note that the workflow for developing an adaptive architecture model is the same.

To start the tutorial, see “Create AUTOSAR Software Architecture Model” on page 1-43.

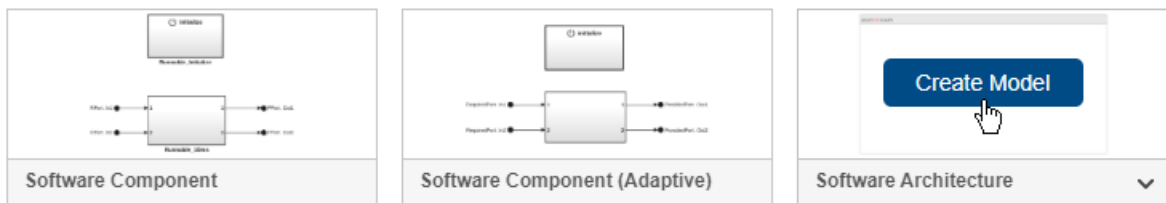
Create AUTOSAR Software Architecture Model

To begin developing AUTOSAR compositions and components in a software architecture canvas, create an AUTOSAR architecture model (requires System Composer).

- 1 Open a local working folder containing example models required for this tutorial. Click this `autosar_tpc_composition` link or enter the MATLAB command `openExample('autosar_tpc_composition')`. After you open the folder, you can close the `autosar_tpc_composition` model or leave it open for reference.
- 2 Open the Simulink Start Page by entering the MATLAB command `simulink`.

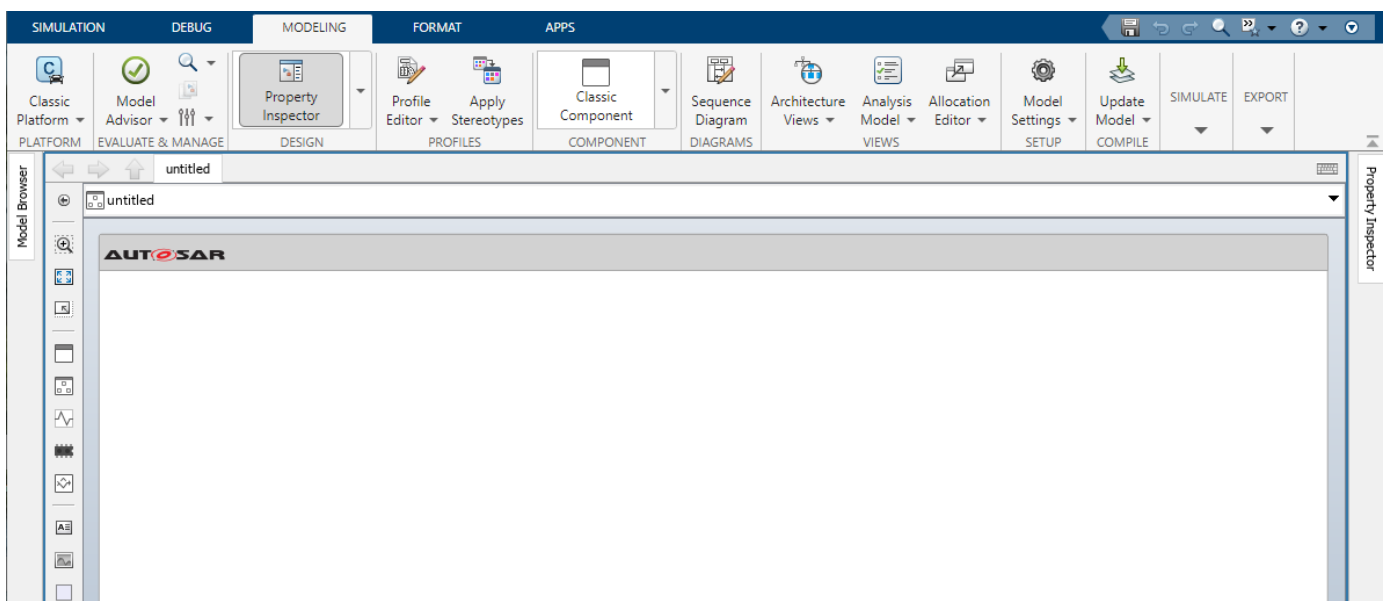
On the **New** tab, scroll down to AUTOSAR Blockset and expand the list of model templates. Place your cursor over the **Software Architecture** template and click **Create Model**.

▼ AUTOSAR Blockset



A new AUTOSAR architecture model opens.

- In the Simulink Toolstrip, the **Modeling** tab supports common tasks for architecture modeling.
- To the left of the model window, the palette includes icons for adding different types of AUTOSAR components to the model: Classic Component, Software Composition, and for Basic Software (BSW) modeling, Diagnostic Service Component and NVRAM Service Component.
- The composition editor provides a view of AUTOSAR software architecture based on the AUTOSAR Virtual Function Bus (VFB). The model canvas initially is empty.



This tutorial constructs a throttle position control application. Perform the steps in a new architecture model or refer to example model `autosar_tpc_composition`, which shows the end result.

Next, add and connect AUTOSAR compositions and components and add Simulink behavior to components.

See Also

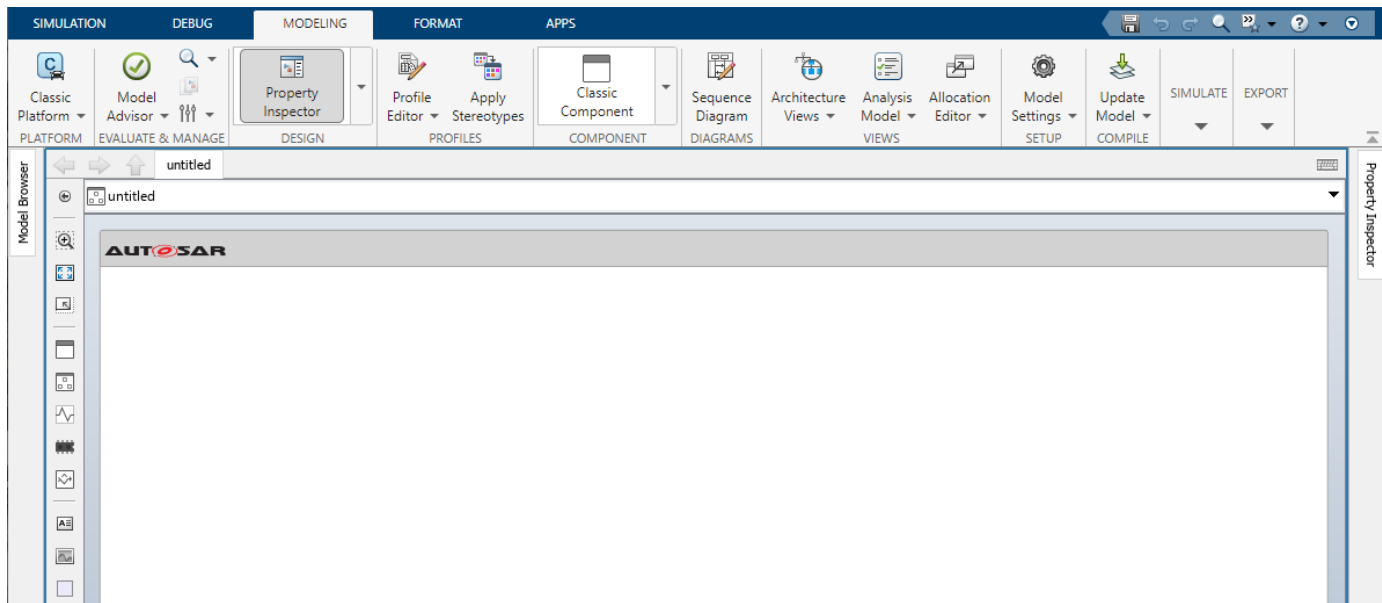
Related Examples

- “Create AUTOSAR Architecture Models” on page 8-2

Add AUTOSAR Compositions and Components and Link Component Implementations

After you create an AUTOSAR architecture model, you begin authoring the top level of an AUTOSAR software design. Use the composition editor and the Simulink Toolstrip **Modeling** tab to add and connect AUTOSAR compositions and components.

In the previous step, you opened a local working example folder and created an empty AUTOSAR architecture model. If necessary, repeat the step to open the working folder and create the empty model.



As you construct a throttle position control application, you can refer to example model `autosar_tpc_composition`, which shows the end result.

Add Compositions and Components to Architecture Canvas

Typically, an AUTOSAR composition contains a set of AUTOSAR components and compositions with a shared purpose. As part of constructing a throttle position control application, this tutorial places four sensor components in a sensors composition. Note that the model is configured for classic architecture modeling by setting the **Platform** selection on the toolstrip to **Classic Platform**.

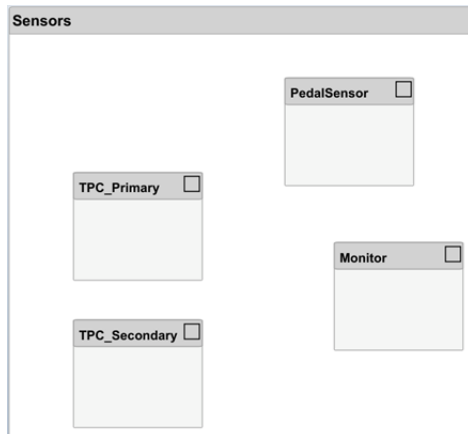
To add the sensors composition and its components to the AUTOSAR architecture model:

- 1 In the architecture model canvas, add a Software Composition block and name it Sensors. For example, on the **Modeling** tab, select **Software Composition** and insert a Software Composition block in the canvas. In the highlighted name field, enter `Sensors`.



- 2 To populate a composition, you open the Software Composition block and add software components.

Open the Sensors block so that the model canvas shows the composition content. Inside the composition, add AUTOSAR software components named TPS_Primary, TPS_Secondary, Monitor, and PedalSensor. For example, on the **Modeling** tab, you can select **Classic Component** to create each one.



Next, you add require and provide ports to the components, and then connect the component ports to other component blocks or to composition root ports. To add component require and provide ports, this tutorial links software component blocks to implementation models in which the ports are already defined.

Define Component Behavior by Linking Implementation Models

The behavior of an AUTOSAR application is defined by its AUTOSAR software components. After you insert software component blocks in an AUTOSAR architecture model, you can add Simulink behavior to the components. For each software component block, you can:

- Create a model based on the block interface.
- Link to an implementation model.
- Create a model from an AUTOSAR XML (ARXML) component description.

For convenience, this tutorial provides a Simulink implementation model for each AUTOSAR component:

- `autosar_tpc_throttle_sensor1.slx` for component TPS_Primary
- `autosar_tpc_throttle_sensor2.slx` for component TPS_Secondary
- `autosar_tpc_throttle_sensor_monitor.slx` for component Monitor
- `autosar_tpc_pedal_sensor.slx` for component PedalSensor

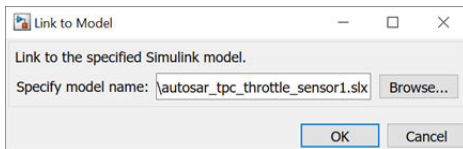
To add Simulink behavior to the components:

- 1 In the architecture model, open the Sensors composition block if it is not already open. Inside the composition, link each AUTOSAR sensor component to a Simulink model that implements its behavior.

For example, select the TPS_Primary component block, place your cursor over the displayed ellipsis, and select the cue **Link to Model**.

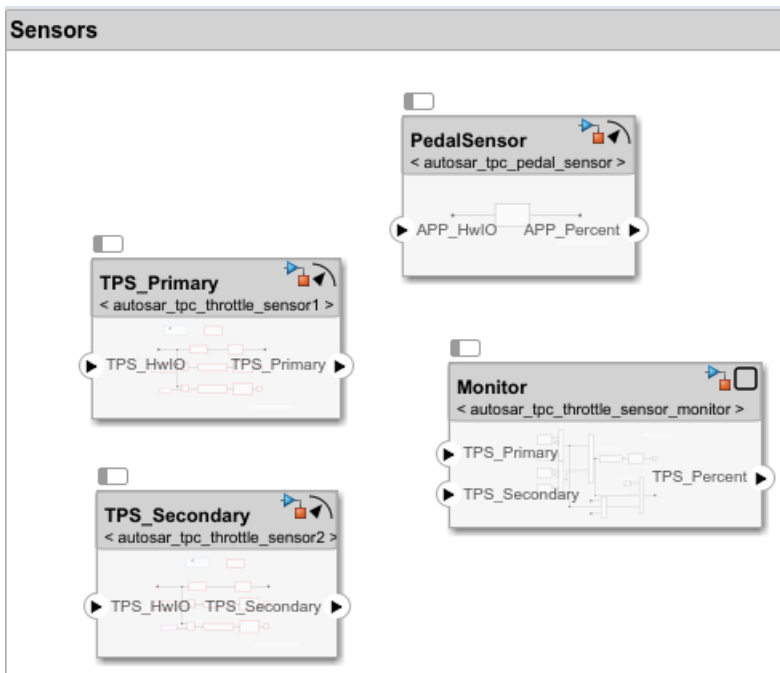


In the Link to Model dialog box, browse to the implementation model `autosar_tpc_throttle_sensor1.slx`.



To link the component to the implementation model, click **OK**.

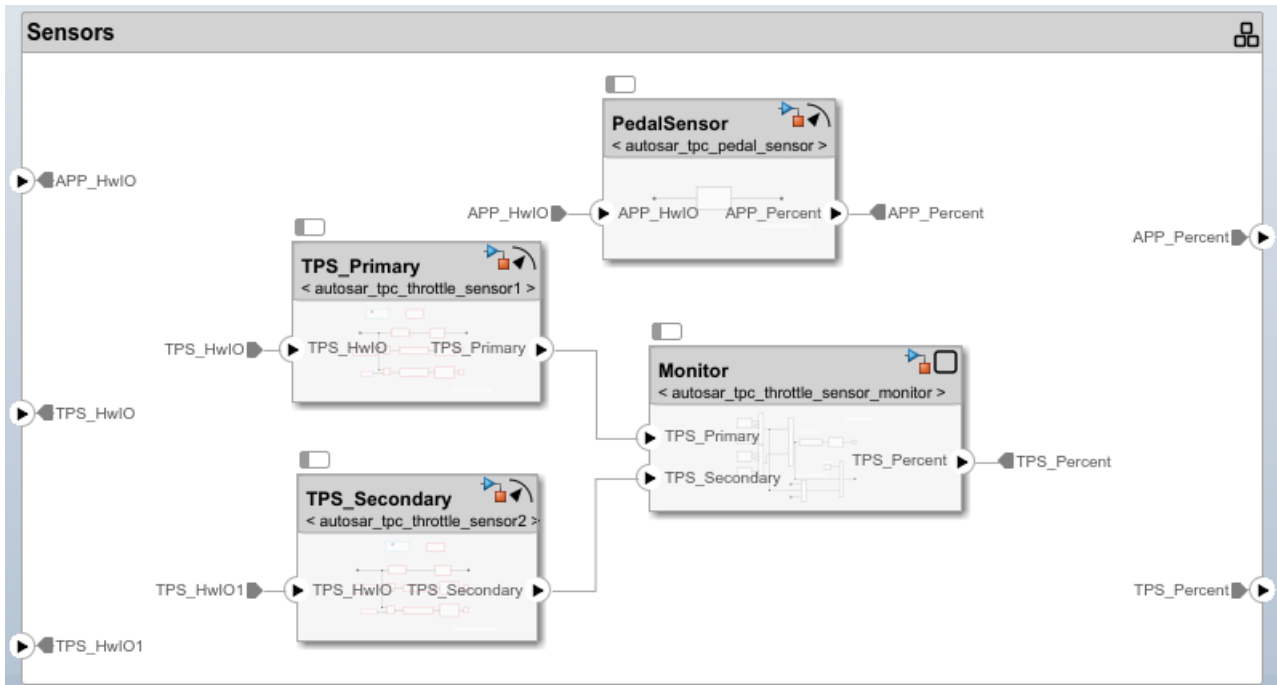
- 2 Link components TPS_Secondary, Monitor, and PedalSensor to their implementation models. After you link each model, you can resize the associated component block to better display the component ports.



Linking a software component block to a specified implementation model updates the block and model interfaces to match. If you link to a model that uses root Inport and Outport blocks, the software converts the model signal ports to bus ports. To view the model content, open the component block.

- 3 Connect the components to each other and to composition root ports.

- To interconnect components, drag a line from a component provider port to another component receiver port.
- To connect components to Sensors composition root ports, drag from a component port to the Sensors composition boundary.

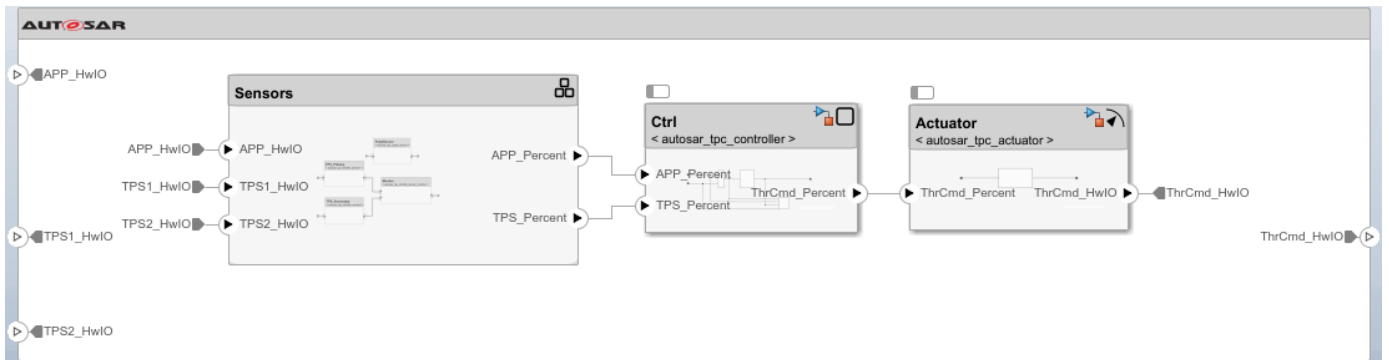


- 4 Optionally, to exactly match the root port naming in example model `autosar_tpc_composition`, rename ports `TPS_HwIO` and `TPS_HwIO1` to `TPS1_HwIO` and `TPS2_HwIO`.

Complete Architecture Model Top Level

To complete the throttle position control application:

- 1 Return to the top level of the architecture model. Add two Classic Component blocks and name them `Ctrl` and `Actuator`.
- 2 Link the AUTOSAR components `Ctrl` and `Actuator` to their Simulink implementation models, `autosar_tpc_controller.slx` and `autosar_tpc_actuator.slx`.
- 3 Connect the `Sensors` composition, `Ctrl` component, and `Actuator` component to each other and to the architecture model boundary.



- 4 To check for interface or data type issues, update the architecture model. On the **Modeling** tab, select **Update Model**. If any issues are found, compare your model with example model `autosar_tpc_composition`.
- 5 Save the model with a unique name, such as `myTPC_Composition.slx`.

Next, simulate the behavior of the aggregated components in the AUTOSAR architecture model.

See Also

Related Examples

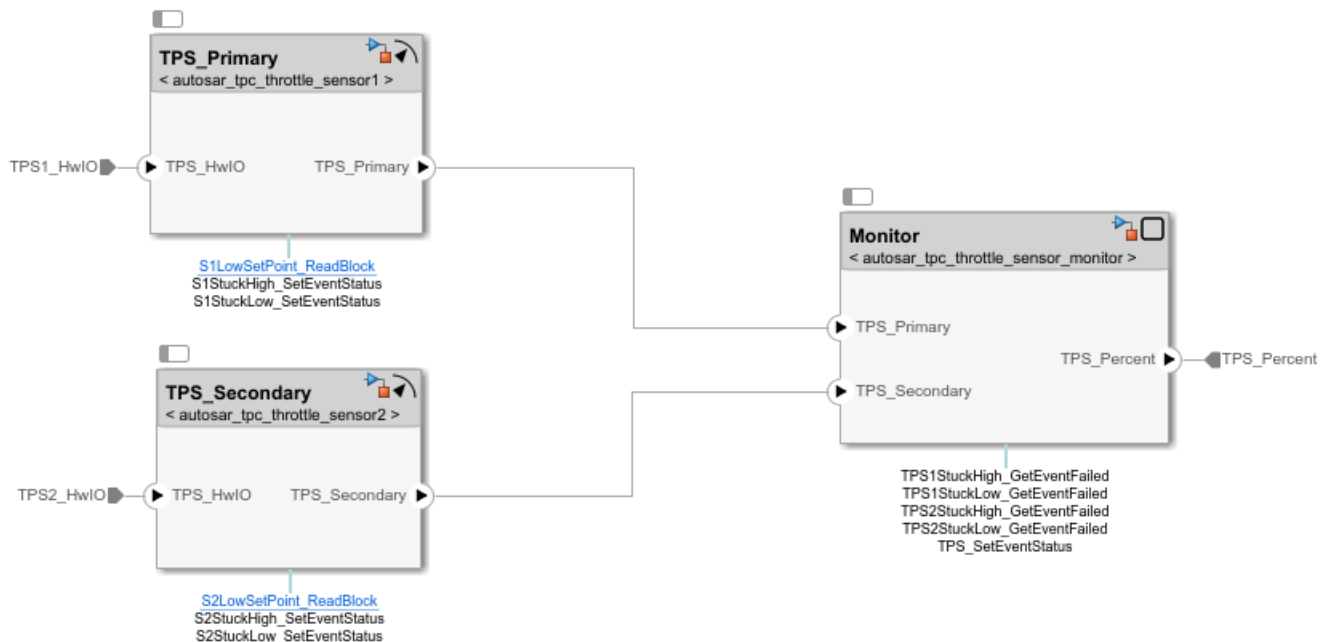
- “Add and Connect AUTOSAR Classic Components and Compositions” on page 8-4
- “Define AUTOSAR Component Behavior by Creating or Linking Models” on page 8-27

Simulate Components in AUTOSAR Architecture

To simulate the behavior of the aggregated components in an AUTOSAR architecture model, go to the top level of the architecture model and click **Run**.

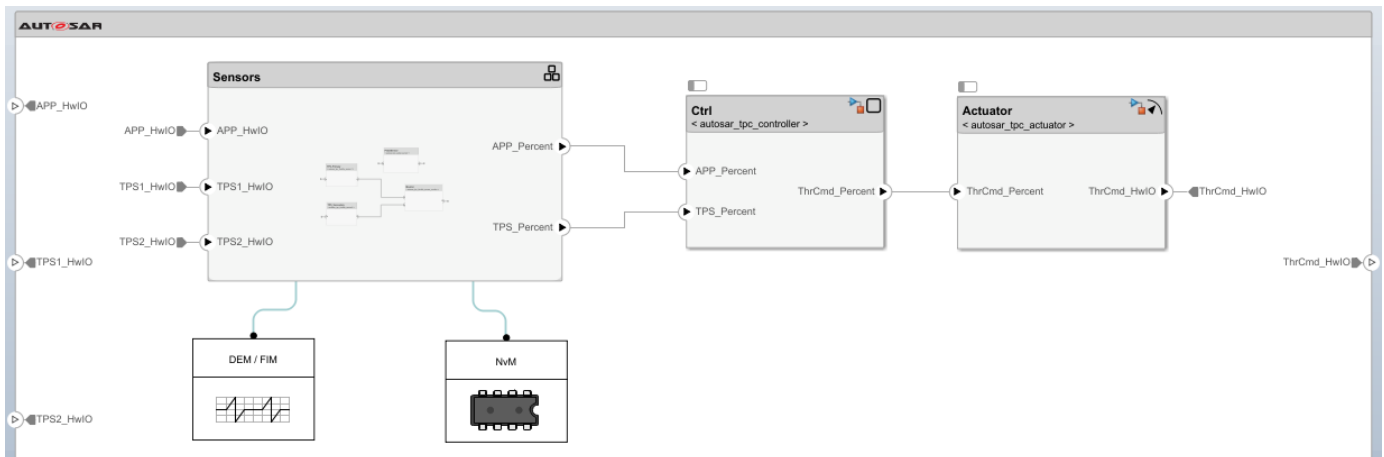
If you try to run the architecture model constructed in this tutorial, an error message reports that a function definition was not found for a Basic Software (BSW) function caller block. Three of the component implementation models contain BSW function calls that require BSW service implementations.

To view those function calls, open your architecture model, for example, `myTPC_Composition.slx`. On the **Debug** tab, select **Information Overlays > Function Connectors**. This selection lists function connectors for each model that contains function calls. To see the models with BSW function calls, open the Sensors composition.



The models contain function calls to Diagnostic Event Manager (Dem) and NVRAM Manager (NvM) services. Before the application can be simulated, you must add Diagnostic Service Component and NVRAM Service Component blocks to the top model.

- 1 Return to the top level of the architecture model and select the **Modeling** tab. To add the service implementation blocks, select and place an instance of **Diagnostic Service Component** and an instance of **NVRAM Service Component**. To wire the function callers to the BSW service implementations, update the model.



- After adding DEM/FIM and NvM service blocks to a model, check the mapping of the BSW function-caller client ports to BSW service IDs. Dem client ports map to Dem service event IDs and NvM client ports map to NvM service block IDs. For this tutorial, update the Dem mapping. Open the DEM/FIM block dialog box, select the **RTE** tab, and enter the event ID values shown. Click **OK**. For more information about BSW ID mapping, see “Simulate AUTOSAR Basic Software Services and Run-Time Environment” on page 7-36.

Diagnostic Service Component
Configure AUTOSAR Diagnostic Services and Runtime Environment (RTE) for emulation.

RTE Dem FIM

Update diagram to refresh table.

Filter contents

Client Port	ID	ID Type
S1StuckHigh	1	
S1StuckLow	2	
S2StuckHigh	3	
S2StuckLow	4	
TPS	5	
TPS1StuckHigh	1	
TPS1StuckLow	2	
TPS2StuckHigh	3	
TPS2StuckLow	4	

- The architecture model is now ready to be simulated. Click **Run**.

Next, if you have access to Embedded Coder software, you can export composition and component AUTOSAR XML files and generate component code from the AUTOSAR architecture model.

See Also

Related Examples

- “Configure AUTOSAR Scheduling and Simulation” on page 8-38

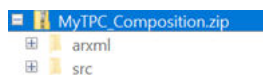
Optional: Generate and Package Composition ARXML and Component Code (Requires Embedded Coder)

If you have access to Simulink Coder and Embedded Coder software, you can export composition and component AUTOSAR XML (ARXML) files and generate component code from an AUTOSAR architecture model. Optionally, you can create a ZIP file to package build artifacts for the model hierarchy, for example, for relocation to a testing or integration environment.

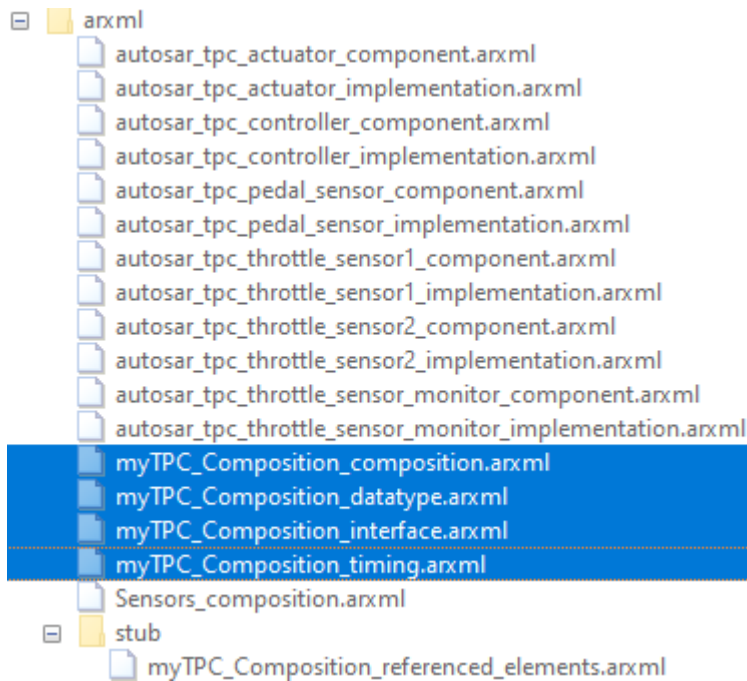
- 1 Open the architecture model constructed in this tutorial or open example model `autosar_tpc_composition`.
- 2 Optionally, to prepare for exporting ARXML, you can examine and modify XML options. On the **Modeling** tab, select **Export > Configure XML Options**. XML options specified at the architecture model level are inherited during export by each component in the model.
- 3 To generate and package code for the throttle position control application, on the **Modeling** tab, select **Export > Generate Code and ARXML**. In the Export Composition dialog box, specify the name of the ZIP file in which to package the generated files. To begin the export, click **OK**.

As the architecture model builds, you can view the build log in the Diagnostic Viewer. First the component models build, each as a standalone top-model build. Finally, composition ARXML is exported. When the build is complete, the current folder contains build folders for the architecture model and each component model in the hierarchy, and the specified ZIP file.

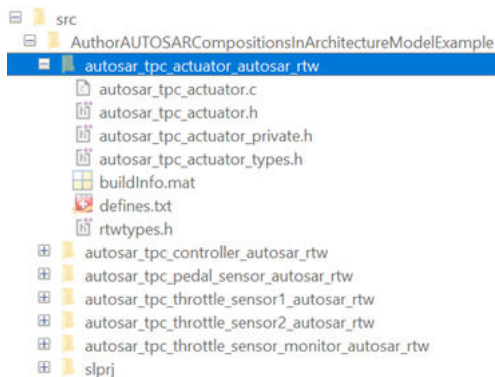
- 4 Expand the ZIP file. Its content is organized in `arxml` and `src` folders.



- 5 Examine the `arxml` folder. Each AUTOSAR component has component and implementation description files, while the architecture model has composition, datatype, interface, and timing description files. The composition file includes XML descriptions of the composition, component prototypes, and composition ports and connectors. The datatype, interface, and timing files aggregate elements from the entire architecture model hierarchy.



- 6 Examine the `src` folder. Each component model has a build folder that contains artifacts from a standalone model build.



See Also

Related Examples

- “Generate and Package AUTOSAR Composition XML Descriptions and Component Code” on page 8-43

Modeling Patterns for AUTOSAR Components

- “Simulink Modeling Patterns for AUTOSAR” on page 2-2
- “Model AUTOSAR Software Components” on page 2-3
- “Modeling Patterns for AUTOSAR Runnables” on page 2-10
- “Model AUTOSAR Runnables Using Exported Functions” on page 2-18
- “Model AUTOSAR Communication” on page 2-21
- “Model AUTOSAR Component Behavior” on page 2-31
- “Model AUTOSAR Variants” on page 2-37
- “Model AUTOSAR Nonvolatile Memory” on page 2-40
- “Model AUTOSAR Data Types” on page 2-43
- “Model AUTOSAR Calibration Parameters and Lookup Tables” on page 2-50

Simulink Modeling Patterns for AUTOSAR

The following topics present Simulink modeling patterns for common AUTOSAR elements. You can use these modeling patterns when developing models for the AUTOSAR Classic Platform.

- “Model AUTOSAR Software Components” on page 2-3
- “Modeling Patterns for AUTOSAR Runnables” on page 2-10
- “Model AUTOSAR Runnables Using Exported Functions” on page 2-18
- “Model AUTOSAR Communication” on page 2-21
- “Model AUTOSAR Component Behavior” on page 2-31
- “Model AUTOSAR Variants” on page 2-37
- “Model AUTOSAR Nonvolatile Memory” on page 2-40
- “Model AUTOSAR Data Types” on page 2-43
- “Model AUTOSAR Calibration Parameters and Lookup Tables” on page 2-50

Model AUTOSAR Software Components

In Simulink, you can flexibly model the structure and behavior of software components for the AUTOSAR Classic Platform. Components can contain one or multiple runnable entities, and can be single-instance or multi-instance. To design the internal behavior of components, you can use Simulink modeling styles, such as rate-based and function-call-based.

In this section...

“About AUTOSAR Software Components” on page 2-3

“Implementation Considerations” on page 2-3

“Rate-Based Components” on page 2-6

“Function-Call-Based Components” on page 2-7

“Multi-Instance Components” on page 2-8

“Startup, Reset, and Shutdown” on page 2-8

About AUTOSAR Software Components

An AUTOSAR application is made up of interconnected *software components* (SWCs). Each software component encapsulates a functional implementation of automotive behavior, with well-defined connection points to the outside world.

In Simulink, you can model:

- *Atomic* software components — An atomic software component runs on exactly one automotive electronic control unit (ECU), and cannot be split into smaller software components.
- *Parameter* software components — A parameter software component represents memory containing AUTOSAR calibration parameters, and provides parameter data to connected atomic software components.

The main focus of AUTOSAR modeling in Simulink is atomic software components. For information about parameter software components, see “Model AUTOSAR Calibration Parameters and Lookup Tables” on page 2-50.

Note Do not confuse *atomic* in this context with the Simulink concept of atomic subsystems.

An AUTOSAR atomic software component interacts with other AUTOSAR software components or system services via well-defined connection points called *ports*. One or more *runnable entities* (runnables) implement the behavior of the component.

Implementation Considerations

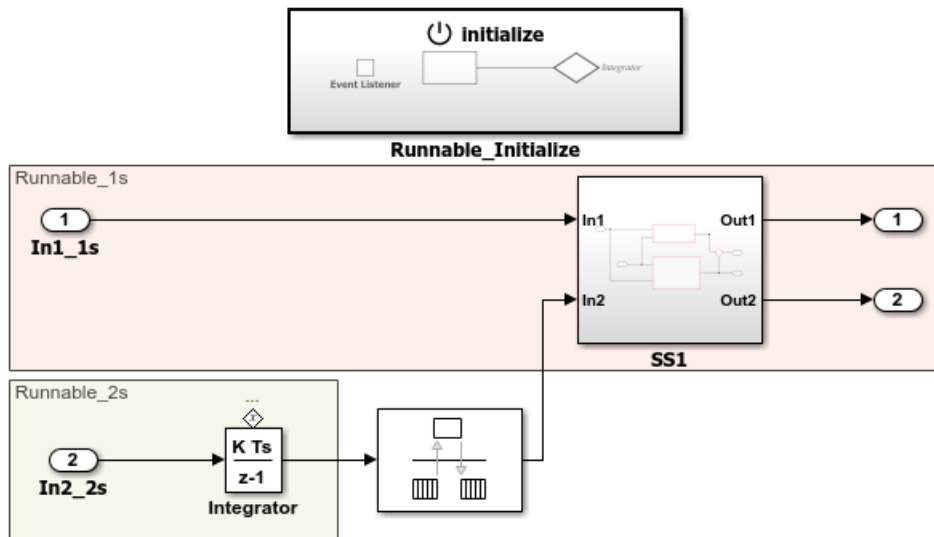
To develop an AUTOSAR atomic software component in Simulink, you create an initial Simulink representation of an AUTOSAR component, as described in “Component Creation”. You can either import an AUTOSAR component description from ARXML files or, in an existing model, build a default AUTOSAR component based on the model content. The resulting representation includes:

- Simulink blocks, connections, and data that model AUTOSAR elements such as ports, runnables, inter-runnable variables (IRV), and parameters.

- Stored properties, defined in the AUTOSAR standard, for AUTOSAR elements in the software component.
- A mapping of Simulink elements to AUTOSAR elements.

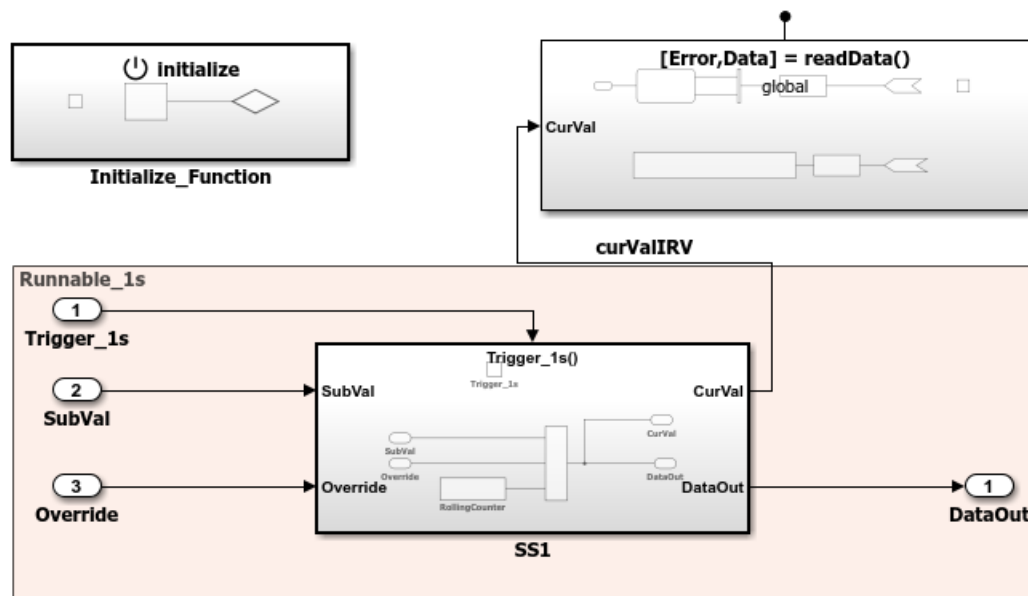
Usually, the Simulink representation of an AUTOSAR component is a rate-based model, in which periodic runnables are modeled as atomic subsystems with periodic rates.

Consider the AUTOSAR example model `autosar_sw_c`. This model shows a rate-based implementation of an AUTOSAR atomic software component. The model implements periodic runnables using multiple rates. An Initialize Function block initializes the component.



However, if your component design requires server functions or periodic function calls, you can use a function-call-based model. The model can contain Simulink Function blocks or function-call subsystems with periodic rates.

Consider the AUTOSAR example model `autosar_sw_c_slfcns`. This model shows a function-call-based implementation of an AUTOSAR atomic software component. The model uses a Simulink Function block and a periodic function-call subsystem at root level. An Initialize Function block initializes the component.



If your AUTOSAR software component design contains periodic runnables, you must decide whether your component requires a rate-based or function-call-based modeling approach. Before you create an initial Simulink representation of your AUTOSAR component, specify how to model periodic runnables:

- If you are importing an AUTOSAR component description from ARXML files using `arxml.importer` object function `createComponentAsModel`, set the property `ModelPeriodicRunnablesAs` to `AtomicSubsystem` (default) for rate-based or `FunctionCallSubsystem` for function-call based.
- If you are building a default AUTOSAR component in an existing model, populate the model with rate-based or function-call-based content.
 - For rate-based modeling, create model content with one or more periodic rates. To model an AUTOSAR inter-runnable variable, use a Rate Transition block that handles data transfers between blocks operating at different rates. The resulting component has N periodic step runnables, where N is the number of discrete rates in the model. Events that represent rate-based interrupts initiate execution of the periodic step runnables, using rate monotonic scheduling.
 - For function-call based modeling create function-call subsystems at the top level of a model, or for client-server modeling, create Simulink Function blocks. Add root model inports and outports. To model an AUTOSAR inter-runnable variable, use a signal line to connect function-call subsystems. The resulting component has N exported-function or server runnables where N is the number of function-call subsystems or Simulink Function blocks at the top level of the model. Events that represent function calls initiate execution of the function-based runnables.

Select rate-based modeling, the default, unless your design requires function-call-based modeling.

Conditions in your AUTOSAR software component can prevent use of rate-based modeling if, for example, the AUTOSAR software component contains:

- A server runnable
- An inter-runnable variable (IRV) that multiple runnables read or write

- A periodic runnable with a rate that is not a multiple of the fastest rate
- Multiple runnables that access the same read or write data at different rates
- A periodic runnable that other events also trigger
- Multiple periodic runnables that are triggered at the same period

For examples of different ways to model AUTOSAR software components, see “Rate-Based Components” on page 2-6, “Function-Call-Based Components” on page 2-7, and “Modeling Patterns for AUTOSAR Runnables” on page 2-10.

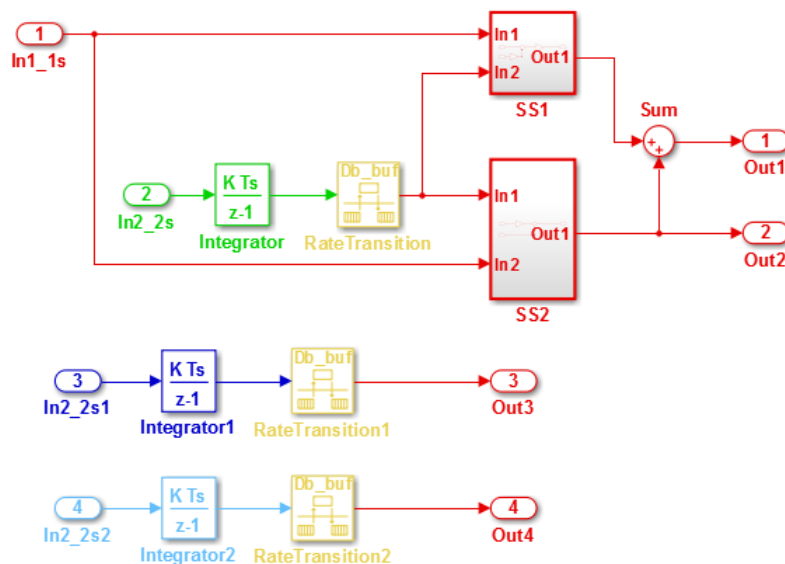
Rate-Based Components

You can model AUTOSAR multirunnables using Simulink rate-based, multitasking modeling. First you create or import model content with multiple periodic rates. You can:

- Create a software component with multiple periodic runnables in Simulink.
- Import a software component with multiple periodic runnables from ARXML files into Simulink. Use `arxml.importer` object function `createComponentAsModel` with property `ModelPeriodicRunnablesAs` set to `AtomicSubsystem`.
- Migrate an existing rate-based, multitasking Simulink model to the AUTOSAR target.

Root model inports and outports represent AUTOSAR ports, and Rate Transition blocks represent AUTOSAR inter-runnable variables (IRVs).

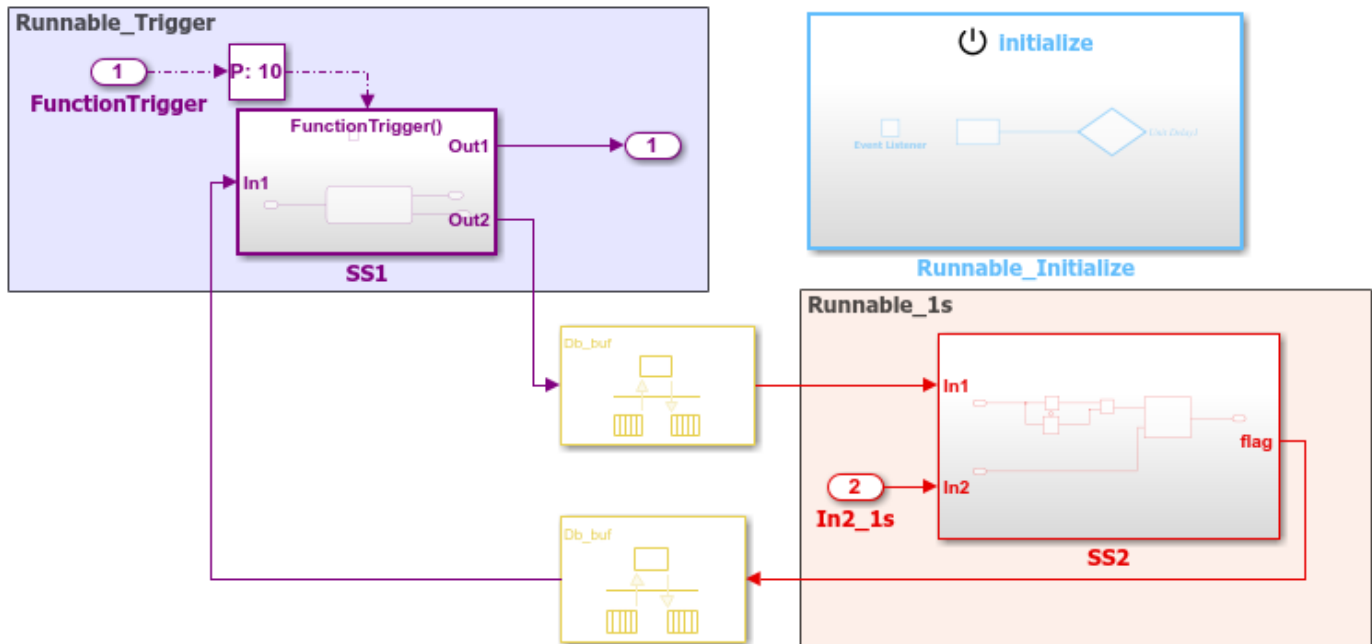
Here is an example of a rate-based, multitasking model that is suitable for simulation and AUTOSAR code generation. (This example uses the example model `mMultitasking_4rates.slx`.) The model represents an AUTOSAR software component. The colors displayed when you update the model (if colors are enabled on the **Debug** tab, under **Diagnostics > Information Overlays**) represent the different periodic rates present. The Rate Transition blocks represent AUTOSAR IRVs.



When you generate code, the model C code contains rate-grouped model step functions corresponding to AUTOSAR runnables, one for each discrete rate in the model. (The periodic step functions must be called in the manner of a rate-monotonic scheduler.) For more information, see “Modeling Patterns for AUTOSAR Runnables” on page 2-10.

A rate-based AUTOSAR software component can include both periodic and asynchronous runnables. For example, in the JMAAB type beta architecture, an asynchronous trigger runnable interacts with periodic rate-based runnables.

Consider the AUTOSAR example model `autosar_sw_cncalls`. This model shows a rate-based implementation of an AUTOSAR atomic software component that includes an asynchronous (triggered) function-call subsystem at root level. An Initialize Function block initializes the component.



For more information, see “Add Top-Level Asynchronous Trigger to Periodic Rate-Based System” on page 4-193.

Function-Call-Based Components

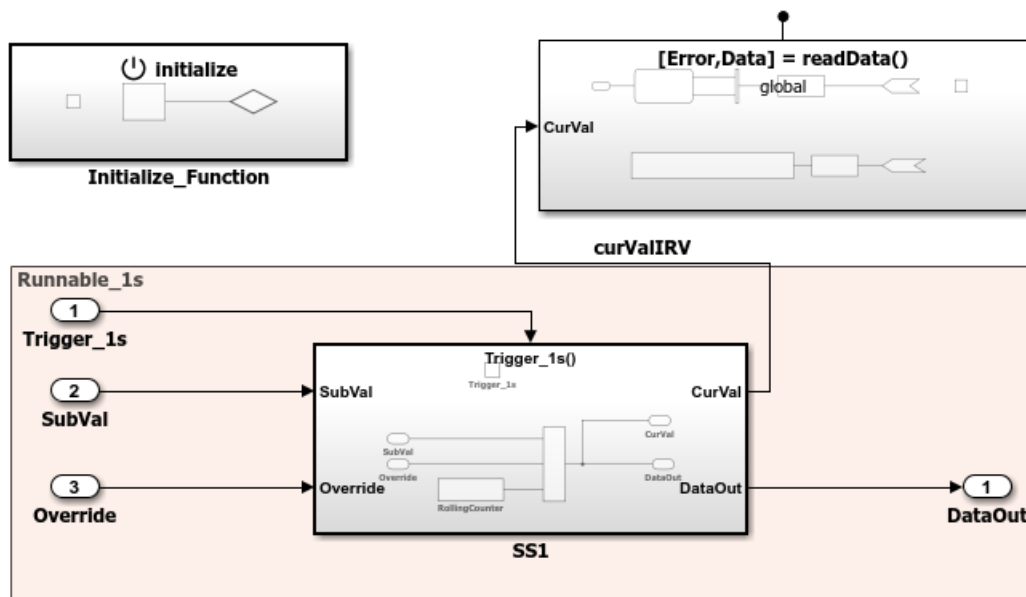
You can model AUTOSAR multirunnables using Simulink function-call subsystems—or (for client-server modeling) Simulink Function blocks—at the top level of a model. First you create or import model content with multiple functions. You can:

- Create a software component with multiple runnables modeled as function-call subsystems or Simulink Function blocks in Simulink.
- Import a software component with multiple runnables from ARXML files into Simulink. Use `arxml.importer` object function `createComponentAsModel` with property `ModelPeriodicRunnablesAs` set to `FunctionCallSubsystem`.
- Migrate an existing function-based Simulink model to the AUTOSAR target.

Root model inports and outports represent AUTOSAR ports, and signal lines connecting function-call subsystems represent AUTOSAR inter-runnable variables (IRVs).

Here is an example of a function-call-based model, with multiple runnable entities, that is suitable for simulation and AUTOSAR code generation. (This example uses AUTOSAR example model

autosar_swc_slfcns.) The model represents an AUTOSAR software component. The function-call subsystem labeled SS1 and the Simulink Function block readData represent runnables that implement its behavior. An Initialize Function block initializes the component. The signal line curValIRV represents an AUTOSAR IRV.



When you generate code, the model C code includes callable model entry-point functions corresponding to AUTOSAR runnables, one for each top-model function-call subsystem or Simulink Function block. For more information, see “Modeling Patterns for AUTOSAR Runnables” on page 2-10.

Multi-Instance Components

You can model multi-instance AUTOSAR SWCs in Simulink. For example, you can:

- Map and configure a Simulink model as a multi-instance AUTOSAR SWC, and validate the configuration. Use the **Reusable** function setting of the model parameter **Code interface packaging** (Simulink Coder).
- Generate C code with reentrant runnable functions and multi-instance RTE API calls. You can access external I/O, calibration parameters, and per-instance memory, and use reusable subsystems in multi-instance mode.
- Verify AUTOSAR multi-instance C code with software-in-the-loop (SIL) and processor-in-the-loop (PIL) simulations.
- Import and export multi-instance AUTOSAR SWC description XML files.

Startup, Reset, and Shutdown

AUTOSAR applications sometimes require complex logic to execute during system initialization, reset, and termination sequences. To model startup, reset, and shutdown processing in an AUTOSAR software component, use the Simulink blocks Initialize Function and Terminate Function.

The Initialize Function and Terminate Function blocks can control execution of a component in response to initialize, reset, or terminate events. You can place the blocks at any level of a model

hierarchy. Each nonvirtual subsystem can have its own set of initialize, reset, and terminate functions. In a lower level model, Simulink aggregates the content of the functions with corresponding instances in the parent model.

The Initialize Function and Terminate Function blocks contain an Event Listener block. To specify the event type of the function—**Initialize**, **Reset**, or **Terminate**—use the **Event type** parameter of the Event Listener block. In addition, the function block reads or writes the state of conditions for other blocks. By default, the Initialize Function block initializes the block state with the State Writer block. Similarly, the Terminate Function block saves the block state with the State Reader block. When the function is triggered, the value of the state variable is written to or read from the specified block.

AUTOSAR models can use the blocks to model potentially complex AUTOSAR startup, reset, and shutdown sequences. The subsystems work with any AUTOSAR component modeling style. (However, software-in-the-loop simulation of AUTOSAR initialize, reset, or terminate runnables works only with exported function modeling.)

In an AUTOSAR model, you map each Simulink initialize, reset, or terminate entry-point function to an AUTOSAR runnable. For each runnable, configure the AUTOSAR event that activates the runnable. In general, you can select any AUTOSAR event type except `TimingEvent`.

For more information, see “Configure AUTOSAR Initialize, Reset, or Terminate Runnables” on page 4-187.

See Also

Rate Transition | Simulink Function | Initialize Function | Terminate Function | Event Listener | State Writer | State Reader

Related Examples

- “Import AUTOSAR XML Descriptions Into Simulink” on page 3-13
- “Modeling Patterns for AUTOSAR Runnables” on page 2-10
- “Configure AUTOSAR Runnables and Events” on page 4-178
- “Configure AUTOSAR Initialize, Reset, or Terminate Runnables” on page 4-187
- “Add Top-Level Asynchronous Trigger to Periodic Rate-Based System” on page 4-193
- “Configure AUTOSAR Code Generation” on page 5-7

More About

- “AUTOSAR Component Configuration” on page 4-3

Modeling Patterns for AUTOSAR Runnables

Use Simulink® models, subsystems, and functions to model AUTOSAR atomic software components and their runnable entities (runnables).

Multiple Periodic Runnables Configured for Multitasking

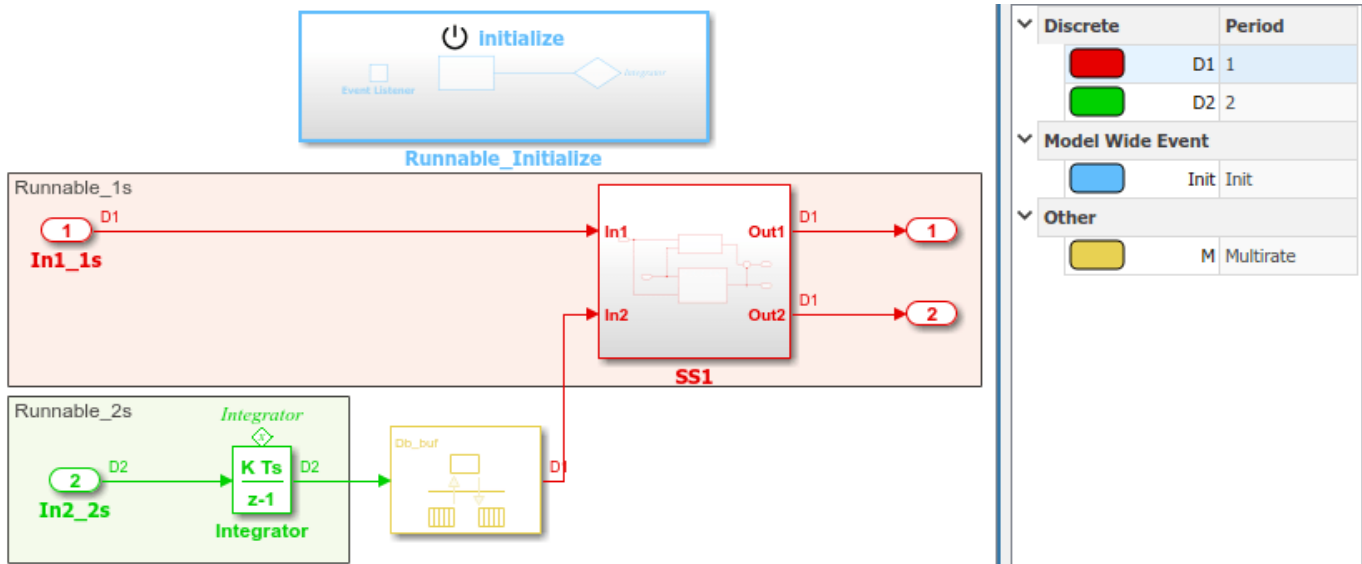
Open the example model `autosar_sw.slx`.

```
open_system('autosar_sw')
```

The model shows the implementation of an AUTOSAR atomic software component (ASWC). Two periodic runnables, `Runnable_1s` and `Runnable_2s`, are modeled with multiple sample rates: 1 second (`In1_1s`) and 2 seconds (`In2_2s`). To maximize execution efficiency, the model is configured for multitasking.

The model includes an Initialize Function block, which initializes the integrator in `Runnable_2s` to a value of 1.

To display color-coded sample rates with annotations and a legend, on the **Debug** tab, select **Diagnostics > Information Overlays > Colors**.



Relevant Model Configuration Parameter Settings

- **Solver > Type** set to Fixed-step.
- **Solver > Solver** set to discrete (no continuous states).
- **Solver > Fixed-step size (fundamental sample time)** set to auto.
- **Solver > Treat each discrete rate as a separate task** selected.

Scheduling

In the model window, enable sample time color-coding by selecting the **Debug** tab and selecting **Diagnostics > Information Overlays > Colors**. The sample time legend shows the implicit rate

grouping. Red represents the fastest discrete rate. Green represents the second fastest discrete rate. Yellow represents the mixture of the two rates.

Because the model has multiple rates and the **Solver** parameter **Treat each discrete rate as a separate task** is selected, the model simulates in multitasking mode. The model handles the rate transition for In2_2s explicitly with the Rate Transition block.

The Rate Transition block parameter **Ensure deterministic data transfer** is cleared to facilitate integration into an AUTOSAR run-time environment.

The generated code for the model schedules subrates in the model. In this example, the rate for Inport block In2_2s, the green rate, is a subrate. The generated code properly transfers data between tasks that run at the different rates.

Generate Code and Report (Embedded Coder)

If you have Simulink Coder and Embedded Coder software, generate code and a code generation report. The example model generates a report.

Generated code complies with AUTOSAR so that you can schedule the code with the AUTOSAR run-time environment.

Review Generated Code

In the code generation report, review the generated code.

- `autosar_sw.c` contains entry points for the code that implements the model algorithm. This file includes the rate scheduling code.
- `autosar_sw.h` declares model data structures and a public interface to the model entry points and data structures.
- `autosar_sw_private.h` contains local `define` constants and local data required by the model and subsystems.
- `autosar_sw_types.h` provides forward declarations for the real-time model data structure and the parameters data structure.
- `rtwtypes.h` defines data types, structures, and macros that the generated code requires.
- `autosar_sw_component.arxml`, `autosar_sw_datatype.arxml`, `autosar_sw_implementation.arxml`, and `autosar_sw_interface.arxml` contain elements and objects that represent AUTOSAR software components, ports, interfaces, data types, and packages. You integrate ARXML files into an AUTOSAR run-time environment. You can import ARXML files into the Simulink environment by using the AUTOSAR ARXML importer tool.
- `Compiler.h`, `Platform_Types.h`, `Rte_ASWC.h`, `Rte_Type.h`, and `Std_Types.h` contain stub implementations of AUTOSAR run-time environment functions. Use these files to test the generated code in Simulink, for example, in software-in-the-loop (SIL) or processor-in-the-loop (PIL) simulations of the component under test.

Code Interface

Open and review the Code Interface Report. This information is captured in the ARXML files. The run-time environment generator uses the ARXML descriptions to interface the code into an AUTOSAR run-time environment.

Input ports:

- Require port, interface: sender-receiver of type `real-T` of 1 dimension
- Require port, interface: sender-receiver of type `real-T` of 1 dimension

Entry-point functions:

- Initialization entry-point function, `void Runnable_Initialize(void)`. At startup, call this function once.
- Output and update entry-point function, `void Runnable_1s(void)`. Call this function periodically at the fastest rate in the model. For this model, call the function every second. To achieve real-time execution, attach this function to a timer.
- Output and update entry-point function, `void Runnable_2s(void)`. Call this function periodically at the second fastest rate in the model. For this model, call the function every 2 seconds. To achieve real-time execution, attach this function to a timer.

Output ports:

- Provide port, interface: sender-receiver of type `real-T` of 1 dimension
- Provide port, interface: sender-receiver of type `real-T` of 1 dimension

Multiple Runnables Configured as Periodic-Rate Runnable and Asynchronous Function-Call Runnable

Open the example model `autosar_swc_fcncalls.slx`.

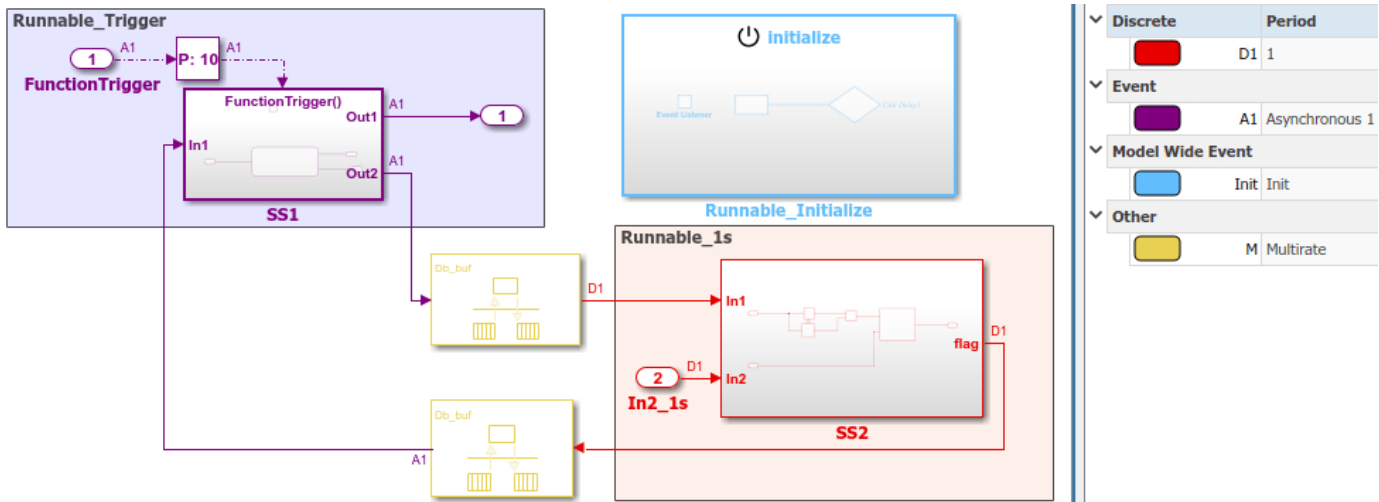
```
open_system('autosar_swc_fcncalls')
```

The model shows the implementation of an AUTOSAR atomic software component (ASWC). The model uses an asynchronous function-call runnable, `Runnable_Trigger`, which is triggered by an external event. The model also includes a periodic rate-based runnable, `Runnable_1s`. The Rate Transition blocks represent inter-runnable variables (IRVs).

Use this approach to model the JMAAB complex control model type beta architecture. In JMAAB type beta modeling, at the top level of a control model, you place function layers above scheduling layers.

The model includes an Initialize Function block, which initializes the unit delay in `Runnable_1s` to a value of 0.

To display color-coded sample rates with annotations and a legend, on the **Debug** tab, select **Diagnostics > Information Overlays > Colors**.



Relevant Model Configuration Parameter Settings

- **Solver > Type** set to Fixed-step.
- **Solver > Solver** set to discrete (no continuous states).
- **Solver > Fixed-step size (fundamental sample time)** set to 1.
- **Solver > Treat each discrete rate as a separate task** cleared.

Scheduling

In the model window, enable sample time color-coding by selecting the **Debug** tab and selecting **Diagnostics > Information Overlays > Colors**. The sample time legend shows the implicit rate grouping. Red represents the discrete rate. Magenta represents the asynchronous function trigger. Yellow represents the mixture of two rates.

The asynchronous trigger runnable runs at asynchronous rates (the **Sample time type** parameter of the function-call subsystem Trigger block is set to [triggered]) while the periodic rate runnable runs at the specified discrete rate. The generated code manages the rates by using single-tasking assumptions. For models with one discrete rate, the code generator does not produce scheduling code because there is only a single rate to execute. Use this technique for a single-rate application when you have one periodic runnable.

The model handles transitions between the asynchronous and discrete rates of the connected runnables with the two Rate Transition blocks. The Rate Transition block parameter **Ensure deterministic data transfer** is cleared to facilitate integration into an AUTOSAR run-time environment.

Generate Code and Report (Embedded Coder)

If you have Simulink Coder and Embedded Coder software, generate code and a code generation report. The example model generates a report.

Generated code complies with AUTOSAR so that you can schedule the code with the AUTOSAR run-time environment.

Review Generated Code

In the code generation report, review the generated code.

- `autosar_swc_fcncalls.c` contains entry points for the code that implements the model algorithm. This file includes the rate scheduling code.
- `autosar_swc_fcncalls.h` declares model data structures and a public interface to the model entry points and data structures.
- `autosar_swc_fcncalls_private.h` contains local `define` constants and local data required by the model and subsystems.
- `autosar_swc_fcncalls_types.h` provides forward declarations for the real-time model data structure and the parameters data structure.
- `rtwtypes.h` defines data types, structures, and macros that the generated code requires.
- `autosar_swc_fcncalls_component.arxml`, `autosar_swc_fcncalls_datatype.arxml`, `autosar_swc_fcncalls_implementation.arxml`, and `autosar_swc_fcncalls_interface.arxml` contain elements and objects that represent AUTOSAR software components, ports, interfaces, data types, and packages. You integrate ARXML files into an AUTOSAR run-time environment. You can import ARXML files into the Simulink environment by using the AUTOSAR ARXML importer tool.
- `Compiler.h`, `Platform_Types.h`, `Rte_ASWC.h`, `Rte_Type.h`, and `Std_Types.h` contain stub implementations of AUTOSAR run-time environment functions. Use these files to test the generated code in Simulink, for example, in software-in-the-loop (SIL) or processor-in-the-loop (PIL) simulations of the component under test.

Code Interface

Open and review the Code Interface Report. This information is captured in the ARXML files. The run-time environment generator uses the ARXML descriptions to interface the code into an AUTOSAR run-time environment.

Input port:

- Require port, interface: sender-receiver of type `real-T` of 1 dimension

Entry-point functions:

- Initialization entry-point function, `void Runnable_Initialize(void)`. At startup, call this function once.
- Simulink function, `void Runnable_1s(void)`. Call this function periodically at the fastest rate in the model. For this model, call the function every second. To achieve real-time execution, attach this function to a timer.
- Exported function, `void Runnable_Trigger(void)`. Call this function at any time from an external trigger.

Output port:

- Provide port, interface: sender-receiver of type `real-T` of 1 dimension

Multiple Runnables Configured As Function-Call Subsystem and Simulink Function

Open the example model `autosar_swc_slfcns.slx`.

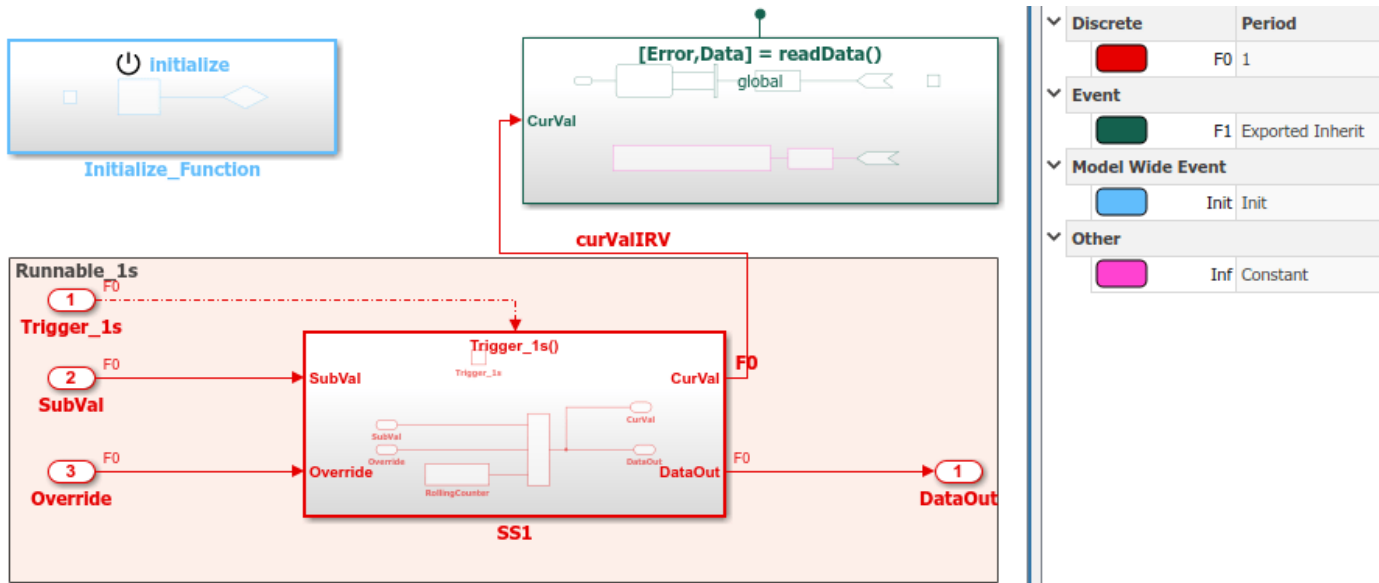
```
open_system('autosar_swc_slfcns')
```

The model shows the implementation of an AUTOSAR atomic software component (ASWC). The model includes one periodic rate runnable, `Runnable_1s`, that uses a function-call subsystem, `SS1`.

The model also includes a Simulink function, `readData`, to provide a value (`CurVal`) to clients that request it.

The model includes an Initialize Function block, which initializes the unit delay in subsystem `RollingCounter` to a value of 0.

To display color-coded sample rates with annotations and a legend, on the **Debug** tab, select **Diagnostics > Information Overlays > Colors**.



Use function-call subsystems:

- When it is difficult or not possible to specify system events in a Simulink model.
- To achieve complex multirate scheduling of runnables. Model each rate as a separate function-call subsystem.

Relevant Model Configuration Parameter Settings

- **Solver > Type** set to Fixed-step.
- **Solver > Solver** set to discrete (no continuous states).
- **Solver > Fixed-step size (fundamental sample time)** set to 1.
- **Solver > Treat each discrete rate as a separate task** selected.

Scheduling

In the model window, enable sample time color-coding by selecting the **Debug** tab and selecting **Diagnostics > Information Overlays > Colors**. The sample time legend shows the implicit rate grouping. Red identifies the discrete rate. Magenta identifies rates inherited from exported functions, indicating their execution is outside the context of Simulink scheduling.

Your execution framework must schedule the generated function code and handle data transfers between functions.

Generate Code and Report (Embedded Coder)

If you have Simulink Coder and Embedded Coder software, generate code and a code generation report. The example model generates a report.

The code generator:

- Produces an AUTOSAR runnable for the function-call subsystem at the root level of the model.
- Implements signal connections between runnables as AUTOSAR inter-runnable variables (IRVs).

Generated code complies with AUTOSAR so that you can schedule the code with the AUTOSAR run-time environment.

Review Generated Code

In the code generation report, review the generated code.

- `autosar_swc_slfcns.c` contains entry points for the code that implements the model algorithm. This file includes the rate scheduling code.
- `autosar_swc_slfcns.h` declares model data structures and a public interface to the model entry points and data structures.
- `autosar_swc_slfcns_private.h` contains local `define` constants and local data required by the model and subsystems.
- `autosar_swc_slfcns_types.h` provides forward declarations for the real-time model data structure and the parameters data structure.
- `readData_private.h` contains local `define` constants and local data required by the Simulink function.
- `rtwtypes.h` defines data types, structures, and macros that the generated code requires.
- `autosar_swc_slfcns_component.arxml`, `autosar_swc_slfcns_datatype.arxml`, `autosar_swc_slfcns_implementation.arxml`, and `autosar_swc_slfcns_interface.arxml` contain elements and objects that represent AUTOSAR software components, ports, interfaces, data types, and packages. You integrate ARXML files into an AUTOSAR run-time environment. You can import ARXML files into the Simulink environment by using the AUTOSAR ARXML importer tool.
- `Compiler.h`, `Platform_Types.h`, `Rte_ASWC.h`, `Rte_Type.h`, and `Std_Types.h` contain stub implementations of AUTOSAR run-time environment functions. Use these files to test the generated code in Simulink, for example, in software-in-the-loop (SIL) or processor-in-the-loop (PIL) simulations of the component under test.

Code Interface

Open and review the Code Interface Report. This information is captured in the ARXML files. The run-time environment generator uses the ARXML descriptions to interface the code into an AUTOSAR run-time environment.

Input ports:

- Require port, interface: sender-receiver of type `uint16-T` of 1 dimension
- Require port, interface: sender-receiver of type `real-T` of 1 dimension

Entry-point functions:

- Initialization entry-point function, `void Runnable_Init(void)`. At startup, call this function once.

- Exported function, `void Runnable_1s(void)`. Call this function periodically, every second.
- Simulink function, `Std_ReturnType readData(real_T Data[2])`. Call this function at any time.

Output ports:

- Provide port, interface: sender-receiver of type `uint16-T` of 1 dimension

Related Links

- “Model AUTOSAR Software Components” on page 2-3
- “Component Creation”
- “Code Generation”

Model AUTOSAR Runnables Using Exported Functions

Use Simulink® exported functions to model AUTOSAR runnables.

Multiple Periodic Runnables Configured for Function Export

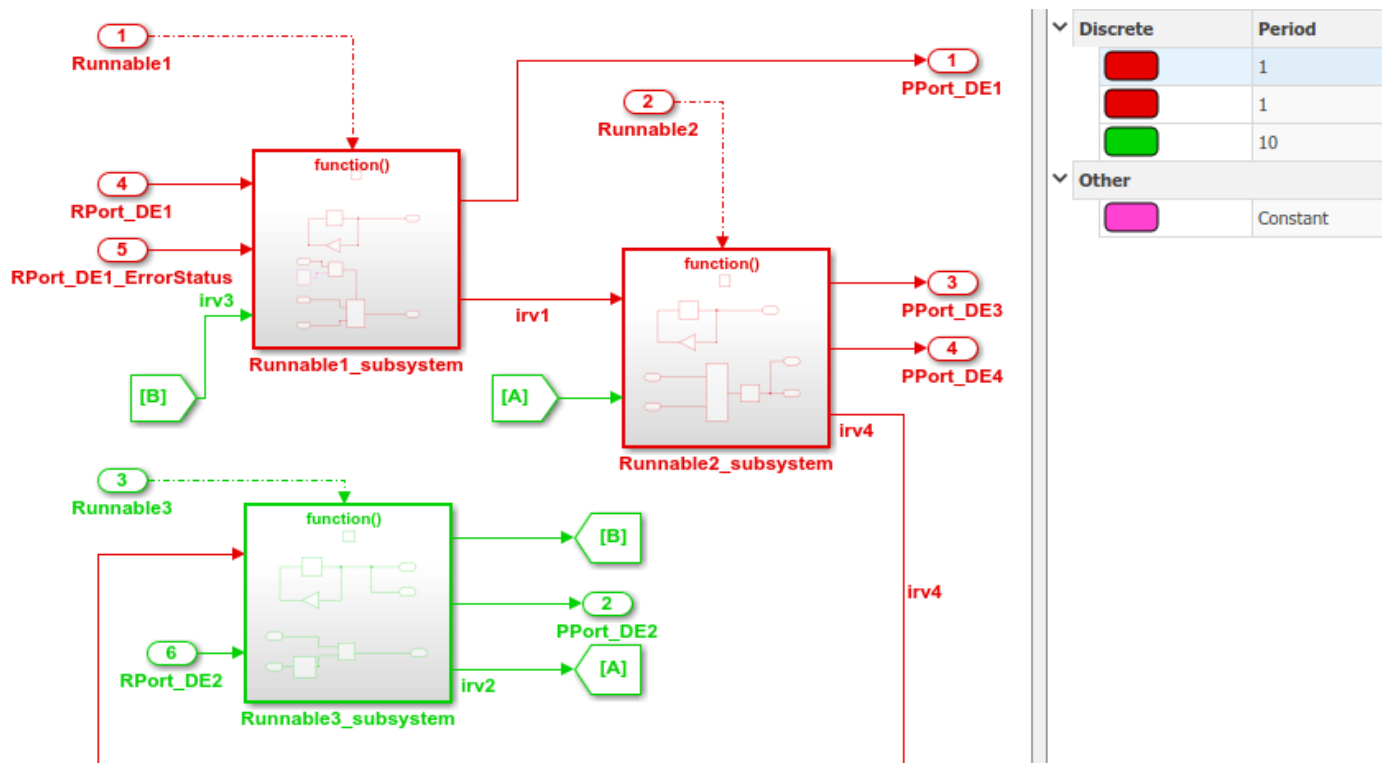
Open the example model `autosar_swc_expfncns.slx`.

```
open_system('autosar_swc_expfncns')
```

The model shows the implementation of an AUTOSAR atomic software component (ASWC) using export-function modeling. Export-function models are Simulink models that generate code for independent functions. You can integrate the independent function code with an external environment and scheduler. Functions typically are defined using Function-Call Subsystem and Simulink Function blocks.

This model implements three AUTOSAR periodic runnables using Function-Call Subsystem blocks that have periodic rates. The runnables have sample rates of 1 second, 1 second, and 10 seconds, respectively. To display color coded sample rates with annotations and a legend, on the **Debug** tab, select **Diagnostics > Information Overlays > Colors**.

Simulink signal lines model AUTOSAR inter-runnable variables (IRVs), which connect the runnables.



Generate AUTOSAR Component Code and XML Descriptions (Embedded Coder)

If you have Simulink Coder and Embedded Coder software, you can generate algorithmic C code and AUTOSAR XML (ARXML) component descriptions. You can test the generated code in Simulink or integrate the code and descriptions into an AUTOSAR run-time environment.

For example, to build the `autosar_swc_expfcns` component model, open the model. Press **Ctrl+B** or enter the MATLAB command `slbuild('autosar_swc_expfcns')`. When the build completes, a code generation report opens.

In the code generation report, select the **Code Interface Report** section, and examine the **Entry-Point Functions** table.

Function: [Runnable1](#)

Prototype	void Runnable1(void)
Description	Exported function: <Root>/Runnable1
Timing	Must be called periodically, every 1 second
Arguments	None
Return value	None
Header file	

Function: [Runnable2](#)

Prototype	void Runnable2(void)
Description	Exported function: <Root>/Runnable2
Timing	Must be called periodically, every 1 second
Arguments	None
Return value	None
Header file	

Function: [Runnable3](#)

Prototype	void Runnable3(void)
Description	Exported function: <Root>/Runnable3
Timing	Must be called periodically, every 10 seconds
Arguments	None
Return value	None
Header file	

In the generated code, each root-level function-call Inport block generates a void-void function. From generated file `autosar_swc_expfcns.c`, here is the generated code for `Runnable1`.

```
78 /* Model step function for TID1 */
79 void Runnable1(void)          /* Explicit Task: Runnable1 */
80 {
81   /* RootInportFunctionCallGenerator: '<Root>/RootFcnCall_InsertedFor_Runnable1_at_outport_1' incorporates:
82    * SubSystem: '<Root>/Runnable1_subsystem'
83    */
84   /* Output: '<Root>/PPort_DE1' incorporates:
85    * UnitDelay: '<S1>/Delay'
86    */
87   Rte_IWrite_Runnable1_PPort_DE1(rtDWork.Delay_DSTATE_a);
88
89   /* Gain: '<S1>/Gain' incorporates:
90    * UnitDelay: '<S1>/Delay'
91    */
92   rtDWork.Delay_DSTATE_a = (sint8)-rtDWork.Delay_DSTATE_a;
93
94   /* Outputs for Enabled SubSystem: '<S1>/Subsystem' incorporates:
95    * EnablePort: '<S4>/Enable'
96    */
97   /* RelationalOperator: '<S1>/Data_Valid' incorporates:
98    * Inport: '<Root>/RPort_DE1_ErrorStatus'
99    */
100  if (Rte_IStatus_Runnable1_RPort_DE1() == 0) {
101    /* Sum: '<S4>/Add' incorporates:
102     * Inport: '<Root>/RPort_DE1'
103     * SignalConversion: '<S1>/TmpSignal_ConversionAtIn2Output1'
104     * SignalConversion: '<S3>/TmpSignal_ConversionAtTicToc_irvInport1'
105     */
106    rtB.Add = Rte_IRead_Runnable1_RPort_DE1() + (float64)
107      Rte_IrvIRead_Runnable1_IRV3();
108  }
109
110  /* End of RelationalOperator: '<S1>/Data_Valid' */
111  /* End of Outputs for SubSystem: '<S1>/Subsystem' */
112
113  /* SignalConversion: '<S1>/TmpSignal_ConversionAtAdderInport1' incorporates:
114   * SignalConversion: '<S1>/OutputBufferForAdder'
115   */
116  Rte_IrvIWrite_Runnable1_IRV1(rtB.Add);
117
118  /* End of Outputs for RootInportFunctionCallGenerator: '<Root>/RootFcnCall_InsertedFor_Runnable1_at_outport_1' */
119 }
```

Related Links

“Export-Function Models Overview”

Model AUTOSAR Communication

In Simulink, for the Classic Platform, you can model AUTOSAR sender-receiver (S-R), client-server (C-S), mode-switch (M-S), nonvolatile (NV) data, parameter, and trigger communication.

About AUTOSAR Communication

AUTOSAR software components provide well-defined connection points called *ports*. There are three types of AUTOSAR ports:

- Require (In)
- Provide (Out)
- Combined Provide-Require (InOut — introduced in AUTOSAR schema version 4.1)

AUTOSAR ports can reference the following kinds of interfaces:

- Sender-Receiver
- Client-Server
- Mode-Switch
- Nonvolatile Data
- Parameter
- Trigger

The following figure shows an AUTOSAR software component with four ports representing the port and interface combinations for Sender-Receiver and Client-Server interfaces.



A Require port that references a Mode-Switch interface is called a mode-receiver port.

Sender-Receiver Interface

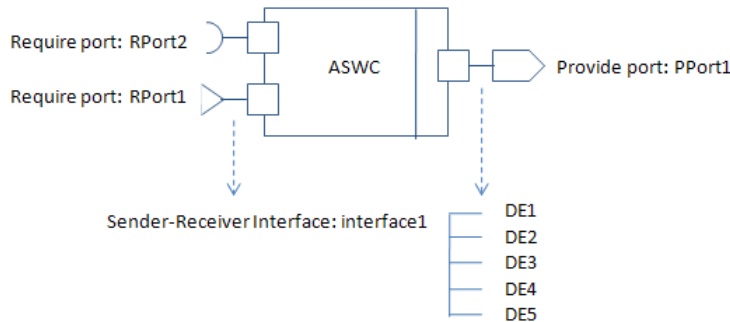
In AUTOSAR port-based sender-receiver (S-R) communication, AUTOSAR software components read and write data to other components or services. To implement S-R communication, AUTOSAR software components define:

- An AUTOSAR sender-receiver interface with data elements.
- AUTOSAR provide and require ports that send and receive data.

In Simulink, you can:

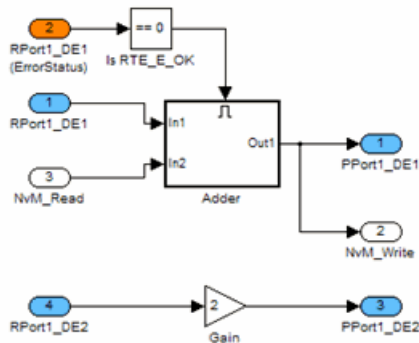
- 1 Create AUTOSAR S-R interfaces and ports by using the AUTOSAR Dictionary.
- 2 Model AUTOSAR provide and require ports by using Simulink root-level outports and inports.
- 3 Map the outports and inports to AUTOSAR provide and require ports by using the Code Mappings editor.

A Sender-Receiver Interface consists of one or more data elements. Although a Require, Provide, or Provide-Require port can reference a Sender-Receiver Interface, the AUTOSAR software component does not necessarily access all of the data elements. For example, consider the following figure.



The AUTOSAR software component has a Require and Provide port that references the same Sender-Receiver Interface, Interface1. Although this interface contains data elements DE1, DE2, DE3, DE4, and DE5, the component does not utilize all of the data elements.

The following figure is an example of how you model, in Simulink, an AUTOSAR software component that accesses data elements.



ASWC accesses data elements DE1 and DE2. You model data element access as follows:

- For Require ports, use Simulink inports. For example, RPort1_DE1 and RPort1_DE2.
- For Provide ports, use Simulink outports. For example, PPort1_DE1 and PPort1_DE2.
- For Provide-Require ports (schema 4.1 or higher), use a Simulink inport and outport pair with matching data type, dimension, and signal type. For more information, see “Configure AUTOSAR Provide-Require Port” on page 4-97.

ErrorStatus is a value that the AUTOSAR Runtime Environment (RTE) returns to indicate errors that the communication system detects for each data element. You can use a Simulink inport to model error status, for example, RPort1_DE1 (*ErrorStatus*).

Use the AUTOSAR Dictionary and the Code Mappings editor to specify the AUTOSAR settings for each inport and outport. For more information, see “Configure AUTOSAR Sender-Receiver Communication” on page 4-96.

Queued Sender-Receiver Interface

In AUTOSAR queued sender-receiver (S-R) communication, AUTOSAR software components read and write data to other components or services. Data sent by an AUTOSAR sender software component is added to a queue provided by the AUTOSAR Runtime Environment (RTE). Newly received data does not overwrite existing unread data. Later, a receiver software component reads the data from the queue.

To implement queued S-R communication, AUTOSAR software components define:

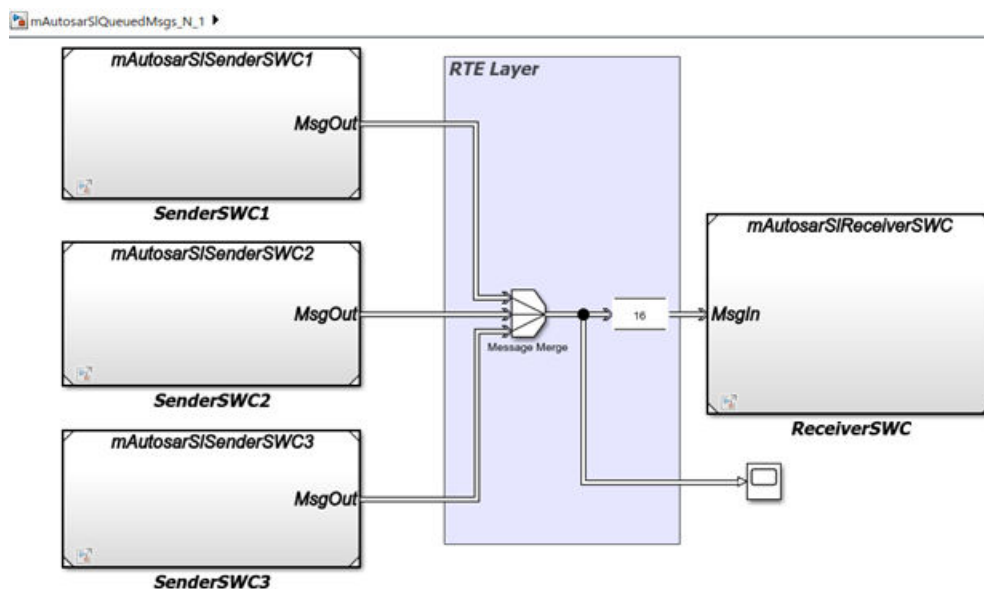
- An AUTOSAR sender-receiver interface with data elements.
- AUTOSAR provide and require ports that send and receive queued data.

In Simulink, you can:

- 1 Create AUTOSAR queued S-R interfaces and ports by using the AUTOSAR Dictionary.
- 2 Model AUTOSAR provide and require ports by using Simulink root-level outports and inports.
- 3 Map the outports and inports to AUTOSAR provide and require ports by using the Code Mappings editor. Set the AUTOSAR data access modes to `QueuedExplicitSend` or `QueuedExplicitReceive`.

To model sending and receiving AUTOSAR data using a queue, use Simulink Send and Receive blocks. If your queued S-R communication implementation involves states or requires decision logic, use Stateflow® charts. You can handle errors that occur when the queue is empty or full. You can specify the size of the queue. For more information, see “Simulink Messages Overview”.

You can simulate AUTOSAR queued sender-receiver (S-R) communication between component models, for example, in a composition-level simulation. Data senders and receivers can run at different rates. Multiple data senders can communicate with a single data receiver.



To get started, you can import components with queued S-R interfaces and ports from ARXML files into Simulink, or use Simulink to create the interfaces and ports. For more information, see “Configure AUTOSAR Queued Sender-Receiver Communication” on page 4-112.

Client-Server Interface

AUTOSAR allows client-server communication between:

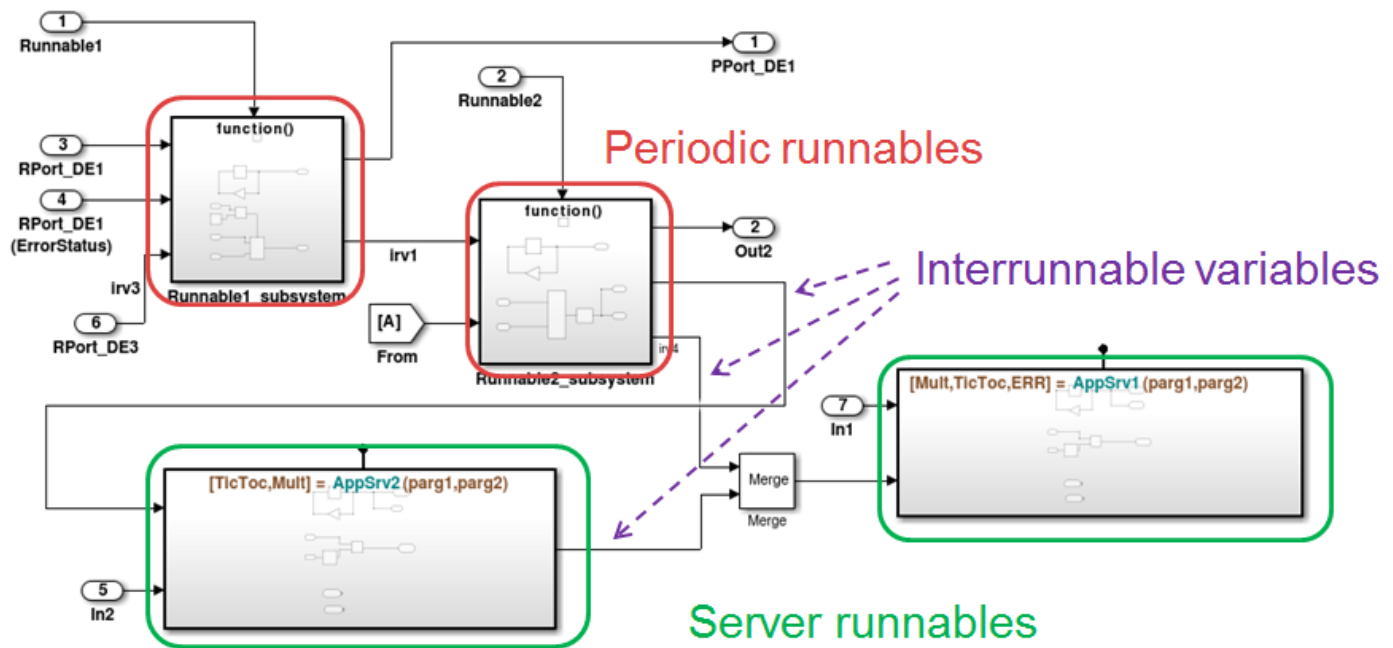
- Application software components
- An application software component and Basic Software

An AUTOSAR Client-Server Interface defines the interaction between a software component that *provides* the interface and a software component that *requires* the interface. The component that provides the interface is the server. The component that requires the interface is the client.

To model AUTOSAR clients and servers in Simulink, for simulation and code generation:

- To model AUTOSAR servers, use Simulink Function blocks at the root level of a model.
- To model AUTOSAR client invocations, use Function Caller blocks.
- Use the function-call-based modeling style to create interconnected Simulink functions, function-calls, and root model inports and outports at the top level of a model.

This diagram illustrates a function-call framework in which Simulink Function blocks model AUTOSAR server runnables, Function Caller blocks model AUTOSAR client invocations, and Simulink data transfer lines model AUTOSAR inter-runnable variables (IRVs).



The high-level workflow for developing AUTOSAR clients and servers in Simulink is:

- 1 Model server functions and caller blocks in Simulink. For example, create Simulink Function blocks at the root level of a model, with corresponding Function Caller blocks that call the functions. Use the Simulink toolset to simulate and develop the blocks.
- 2 In the context of a model configured for AUTOSAR, map and configure the Simulink functions to AUTOSAR server runnables. Validate the configuration, simulate, and generate C code and ARXML files from the model.

- 3 In the context of another model configured for AUTOSAR, map and configure function caller blocks to AUTOSAR client ports and AUTOSAR operations. Validate the configuration, simulate, and generate C code and ARXML files from the model.
- 4 Integrate the generated C code into a test framework for testing, for example, with SIL simulation. (Ultimately, the generated C code and ARXML files are integrated into the AUTOSAR Runtime Environment (RTE).)

For more information, see “Configure AUTOSAR Client-Server Communication” on page 4-142.

Mode-Switch Interface

AUTOSAR mode-switch (M-S) communication relies on a mode manager and connected mode users. The mode manager is an authoritative source for software components to query the current mode and to receive notification when the mode changes. A mode manager can be provided by AUTOSAR Basic Software (BSW) or implemented as an AUTOSAR software component. A mode manager implemented as a software component is called an application mode manager. A software component that queries the mode manager and receives notifications of mode changes is a mode user.

- “Mode User” on page 2-25
- “Application Mode Manager” on page 2-27

Mode User

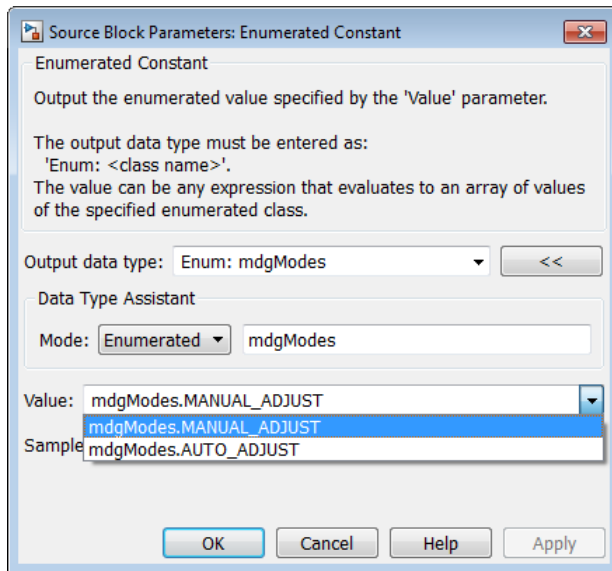
To model an AUTOSAR mode user software component in Simulink:

- Create an AUTOSAR mode-switch interface.
- Create an AUTOSAR mode receiver port and map it to a Simulink inport.
- For an initialization or other AUTOSAR runnable in the model, specify a mode-switch event to trigger the runnable.

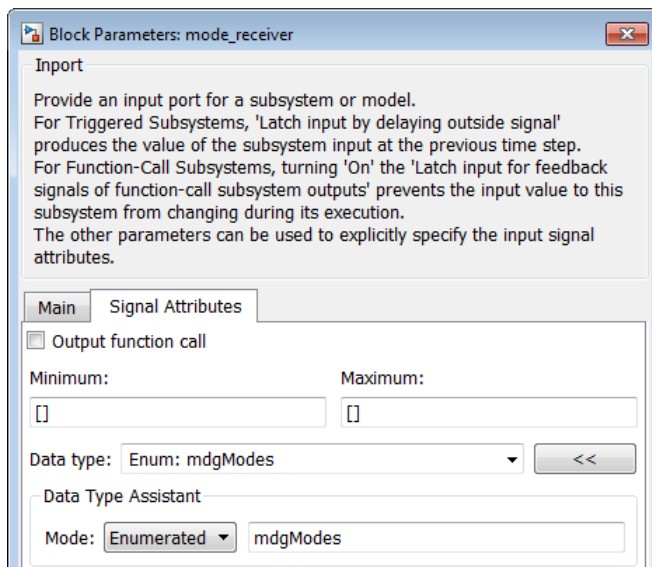
To model an AUTOSAR software component mode-receiver port, general steps can include:

- 1 Declare a mode declaration group — a group of mode values — using Simulink enumeration. For example, you could create an enumerated type `mdgModes`, with enumerated values `MANUAL_ADJUST` and `AUTO_ADJUST`. Specify the storage type as an unsigned integer.

```
Simulink.defineIntEnumType('mdgModes', ...
    {'MANUAL_ADJUST', 'AUTO_ADJUST'}, ...
    [18 28], ...
    'Description', 'Type definition of mdgModes.', ...
    'HeaderFile', 'Rte_Type.h', ...
    'DefaultValue', 'MANUAL_ADJUST', ...
    'AddClassNameToEnumNames', false, ...
    'StorageType', 'uint16' ...
);
```

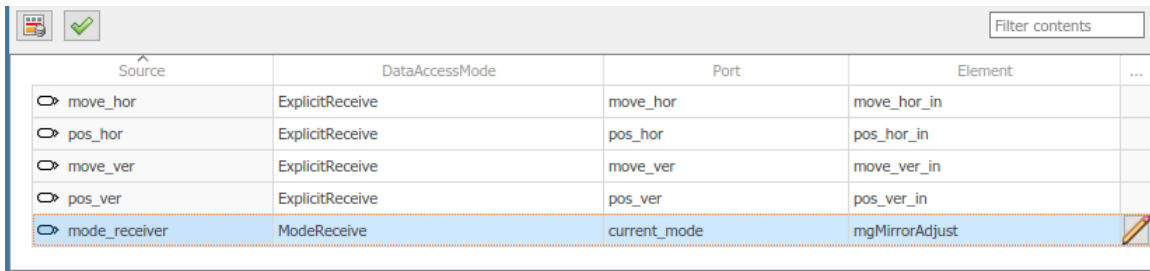


- 2 Apply the enumeration data type to a Simulink inport that represents an AUTOSAR mode-receiver port. In this Inport block dialog box, enumerated type `mdgModes` is specified as the inport data type.



- 3 To specify the mapping of the Simulink inport to the AUTOSAR mode-receiver port, use the Code Mappings editor (or equivalent AUTOSAR map functions).

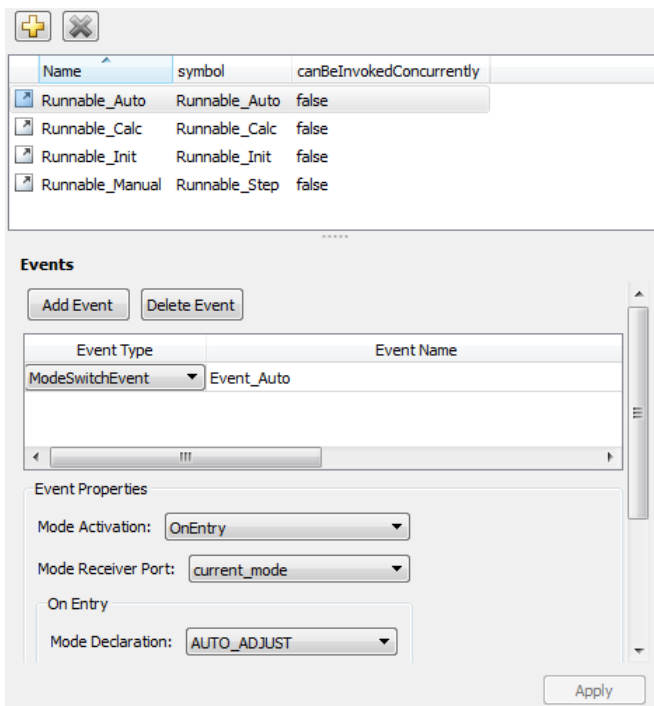
In the following example, in the **Imports** tab of the Code Mappings editor, Simulink inport `mode_receiver` is mapped to AUTOSAR mode-receiver port `current_mode` and AUTOSAR element `mgMirrorAdjust`.



Source	DataAccessMode	Port	Element	...
move_hor	ExplicitReceive	move_hor	move_hor_in	
pos_hor	ExplicitReceive	pos_hor	pos_hor_in	
move_ver	ExplicitReceive	move_ver	move_ver_in	
pos_ver	ExplicitReceive	pos_ver	pos_ver_in	
mode_receiver	ModeReceive	current_mode	mgMirrorAdjust	

To specify a mode-switch event to trigger an initialize runnable or exported runnable, general steps can include:

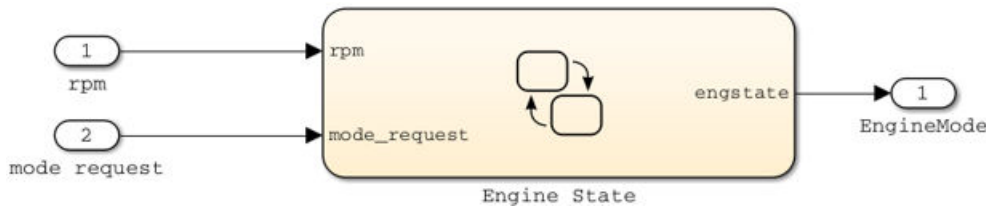
- 1 To edit, add, or remove AUTOSAR mode-switch interfaces and mode-receiver ports, use the AUTOSAR Dictionary (or equivalent AUTOSAR property functions).
- 2 In your model, choose or add a runnable that you want a mode-switch event to activate.
- 3 In the **Runnables** view of the AUTOSAR Dictionary, select the runnable that you want a mode-switch event to activate. Configure the event. In the following example, a mode-switch event is added for **Runnable_Auto**, and configured to activate on entry (versus on exit or on transition). It is mapped to a previously configured mode-receiver port and a mode declaration value that is valid for the selected port.



For more information, see “Configure AUTOSAR Mode-Switch Communication” on page 4-163.

Application Mode Manager

To model an application mode manager software component in Simulink, use an AUTOSAR mode sender port. Mode sender ports output a mode switch to connected mode user components. For example, here is an application mode manager, modeled in Simulink, that uses a mode sender port to output the current value of EngineMode.



You model the mode sender port as a model root output, which is mapped to an AUTOSAR mode sender port and a mode-switch (M-S) interface. The output data type is an enumeration class with an unsigned integer storage type, representing an AUTOSAR mode declaration group.

In Simulink, you can:

- Import AUTOSAR mode-switch communication elements from ARXML files.
 - The software imports ModeSwitchPoints, ModeSwitchInterfaces, and ModeDeclarationGroups.
 - For each AUTOSAR provider port that references an M-S interface, the importer creates a root output with ModeSend data access and with an AUTOSAR mode declaration group enumeration class.
 - The importer maps the model output to an AUTOSAR mode sender port with an M-S interface.
- Create AUTOSAR mode-switch communication elements.
 - Create a model root output, and set the output data type to an enumeration class that represents an AUTOSAR mode declaration group.
 - Create an AUTOSAR mode sender port with an associated M-S interface.
 - In the Code Mappings editor, set the output data access mode to ModeSend, and map the output to the AUTOSAR mode sender port.
- Generate ARXML files and C code for AUTOSAR mode sender ports and related AUTOSAR M-S communication elements.
 - The ARXML files include referenced ModeSwitchPoints, ModeSwitchInterfaces, and ModeDeclarationGroups.
 - The C code includes Rte_Switch API calls to communicate mode switches to other software components.

For more information, see “Configure AUTOSAR Mode-Switch Communication” on page 4-163.

Nonvolatile Data Interface

The AUTOSAR standard defines port-based nonvolatile (NV) data communication, in which an AUTOSAR software component reads and writes data to AUTOSAR nonvolatile components. To implement NV data communication, AUTOSAR software components define provide and require ports that send and receive NV data. For more information about modeling software component access to AUTOSAR nonvolatile memory, see “Model AUTOSAR Nonvolatile Memory” on page 2-40.

In Simulink, you can:

- Import AUTOSAR NV data interfaces and ports from ARXML files.
- Create AUTOSAR NV interfaces and ports, and map Simulink inports and outports to AUTOSAR NV ports.

You model AUTOSAR NV ports with Simulink inports and outports, in the same manner described in “Sender-Receiver Interface” on page 2-21.

- Generate C code and ARXML files for AUTOSAR NV data interfaces and ports.

For more information, see “Configure AUTOSAR Nonvolatile Data Communication” on page 4-169.

Parameter Interface

The AUTOSAR standard defines port-based parameters for parameter communication. AUTOSAR parameter communication relies on a parameter software component (`ParameterSwComponent`) and one or more atomic software components that require port-based access to parameter data. The `ParameterSwComponent` represents memory containing AUTOSAR parameters and provides parameter data to connected atomic software components.

In Simulink, you can model the receiver portion of AUTOSAR port-based parameter communication. In an AUTOSAR atomic software component, you create a parameter interface with data elements and a parameter receiver port.

For more information, see “Configure AUTOSAR Port Parameters for Communication with Parameter Component” on page 4-171.

Trigger Interface

The AUTOSAR standard defines external trigger event communication, in which an AUTOSAR software component or service signals an external trigger occurred event (`ExternalTriggerOccurredEvent`) to another component. The receiving component activates a runnable in response to the event.

In Simulink, you can model the receiver portion of AUTOSAR external trigger event communication. In a component that you want to react to an external trigger, you create a trigger interface, a trigger receiver port to receive an `ExternalTriggerOccurredEvent`, and a runnable that the event activates.

For more information, see “Configure Receiver for AUTOSAR External Trigger Event Communication” on page 4-175.

See Also

Related Examples

- “Configure AUTOSAR Sender-Receiver Communication” on page 4-96
- “Configure AUTOSAR Queued Sender-Receiver Communication” on page 4-112
- “Configure AUTOSAR Client-Server Communication” on page 4-142
- “Configure AUTOSAR Mode-Switch Communication” on page 4-163
- “Configure AUTOSAR Nonvolatile Data Communication” on page 4-169
- “Configure AUTOSAR Port Parameters for Communication with Parameter Component” on page 4-171
- “Configure Receiver for AUTOSAR External Trigger Event Communication” on page 4-175

More About

- “AUTOSAR Component Configuration” on page 4-3

Model AUTOSAR Component Behavior

In Simulink, you can model AUTOSAR component behavior, including behavior of runnables, events, and inter-runnable variables.

AUTOSAR Elements for Modeling Component Behavior

To model AUTOSAR component behavior, you model AUTOSAR elements that describe scheduling and resource sharing aspects of a component. The AUTOSAR elements that bear on component behavior include:

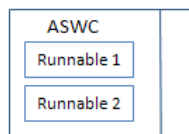
- Runnables and the events to which they respond
- Inter-runnable variables, used to communicate data between runnables in the same component
- Included data type sets, which provide component internal data types
- System constants, used to specify system-level constant values that are available for reference in component algorithms
- Per-instance memory, used to specify instance-specific global memory within a component
- Static and constant memory, for access to global data and parameter values within a component
- Shared and per-instance parameters, for access to component internal parameter data
- Port parameters, for port-based access to parameter data

This topic describes how to model the AUTOSAR elements that help you define component behavior.

Runnables

AUTOSAR software components contain runnables that are directly or indirectly scheduled by the underlying AUTOSAR operating system.

This figure shows an AUTOSAR software component with two runnables, `Runnable 1` and `Runnable 2`. RTEEvents, events generated by the AUTOSAR Runtime Environment (RTE), trigger each runnable. For example, `TimingEvent` is an RTEEvent that is generated periodically.



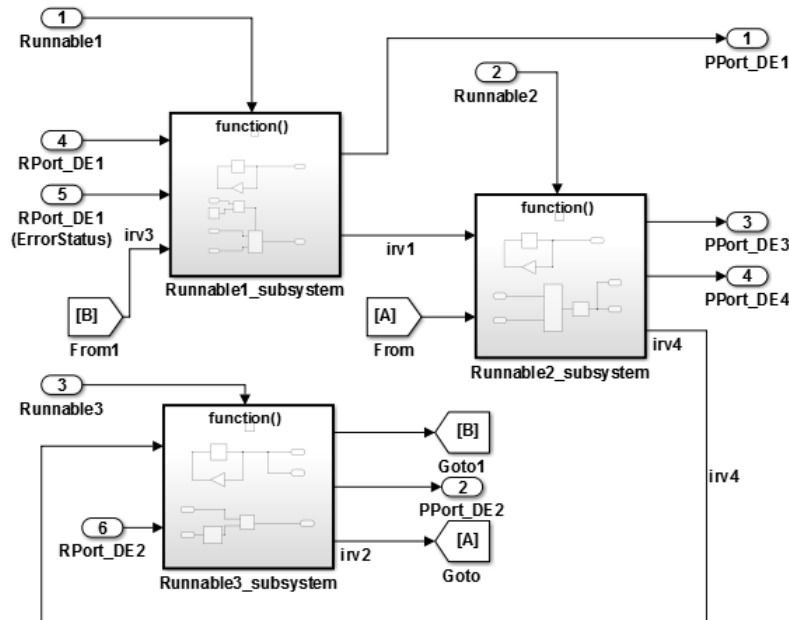
A component also can contain a single runnable, represented by a model, and can be single-rate or multirate.

Note The software generates an additional runnable for the initialization function regardless of the modeling pattern.

For more information, see “Configure AUTOSAR Runnables and Events” on page 4-178.

Inter-Runnable Variables

In AUTOSAR, *inter-runnable* variables are used to communicate data between runnables in the same component. You define these variables in a Simulink model by the signal lines that connect subsystems (runnables). For example, in the following figure, *irv1*, *irv2*, *irv3*, and *irv4* are inter-runnable variables.



You can specify the names and data access modes of the inter-runnable variables that you export.

Included Data Type Sets

In an AUTOSAR software component model, you can import and export an ARXML description of an AUTOSAR included data type set (IncludedDataSet). An IncludedDataSet is defined as part of the internal behavior of a software component. It contains references to AUTOSAR data type definitions that are internal to a component and not present in the component interface descriptions. The referenced internal data type definitions can be shared among multiple software components, as described in “Import and Reference Shared AUTOSAR Element Definitions” on page 3-29.

If you import ARXML files that contain an IncludedDataSet description into Simulink, the importer creates internal data types in the AUTOSAR component model and maps them to header file `Rte_Type.h`.

In an AUTOSAR component model, to configure internal data types for export in an ARXML IncludedDataSet description, map the internal data types to header file `Rte_Type.h`.
Building the component model:

- Exports an ARXML IncludedDataSet description for internal data types that are used in the model code.
- Generates `Rte_Type.h` header file entries for the internal data types.

For AUTOSAR IncludedDataSet export, Simulink supports these data types:

- Numeric
- Alias
- Bus
- Fixed-point
- Enumerated

Literal prefixes for enumeration literals are handled differently between imported and created `IncludedDataTypeSets`:

- If you import an `IncludedDataTypeSet` that defines a `LiteralPrefix` as a common prefix for enumeration literals, the importer preserves the `LiteralPrefix` for export and round trip of the `IncludedDataTypeSet`.
- If you configure internal data types in a component model for export in an AUTOSAR `IncludedDataTypeSet`, the exporter generates the data types in an `IncludedDataTypeSet` with an empty `LiteralPrefix`.

For more information, see “Configure Internal Data Types for AUTOSAR `IncludedDataTypeSets`” on page 4-199.

System Constants

AUTOSAR system constants (`SwSystemConstants`) specify system-level constant values that are available for reference in component algorithms. To add AUTOSAR system constants to your model, you can:

- Import them from ARXML files.
- Create them in Simulink by using `AUTOSAR.Parameter` objects that have **Storage class** set to `SystemConstant`.

You can then reference the AUTOSAR system constants in Simulink algorithms. For example, you can reference a system constant in a Gain block, or in a condition formula inside a variant subsystem or model reference.

When you reference an AUTOSAR system constant in your model:

- Exported ARXML files contain a corresponding `SwSystemConstant` and a corresponding AUTOSAR variation point proxy (`VariationPointProxy`) that references the `SwSystemConstant`. If you generate modular ARXML files, the `SwSystemConstant` is located in `modelName_datatype.arxml` and the `VariationPointProxy` is located in `modelName_component.arxml`.
- Generated C code uses the generated `VariationPointProxy` in places where the model uses the `SwSystemConstant`.

For an example of an AUTOSAR system constant that represents a conditional value associated with variant condition logic, see “Configure Variants for AUTOSAR Runnable Implementations” on page 4-220.

Per-Instance Memory

AUTOSAR supports per-instance memory, which allows you to specify instance-specific global memory within a software component. An AUTOSAR run-time environment generator allocates this memory and provides an API through which you access this memory.



Per-instance memory can be AUTOSAR-typed or C-typed. AUTOSAR-typed per-instance memory (`arTypedPerInstanceMemory`) is described using AUTOSAR data types rather than C types. When exported in ARXML files, `arTypedPerInstanceMemory` allows the use of calibration and measurement tools to monitor the global variable corresponding to per-instance memory.

AUTOSAR also allows you to use per-instance memory as a RAM mirror for data in nonvolatile RAM (NVRAM). You can access and use NVRAM in your AUTOSAR application. For more information about modeling software component access to AUTOSAR nonvolatile memory, see “Model AUTOSAR Nonvolatile Memory” on page 2-40.

To add AUTOSAR per-instance memory to your model, you can:

- Import per-instance memory definitions from ARXML files.
- Create model content that represents per-instance memory.


To model `arTypedPerInstanceMemory`, you can use block signals, discrete states, or data stores in your AUTOSAR model:


- To use block signals and discrete states, use the Code Mappings editor, **Signals/States** tab, to select a signal or state and map it to `arTypedPerInstanceMemory`. To view and modify AUTOSAR code and calibration attributes for the per-instance memory, click the  icon.
- To use data stores, use the Code Mappings editor, **Data Stores** tab, to select a data store and map it to `arTypedPerInstanceMemory`. To view and modify AUTOSAR code and calibration attributes for the per-instance memory, click the  icon.

For more information, see “Configure AUTOSAR Per-Instance Memory” on page 4-201.

Static and Constant Memory

AUTOSAR supports static memory (`StaticMemory`) and constant memory (`ConstantMemory`) data. Static memory corresponds to Simulink internal global signals. Constant memory corresponds to Simulink internal global parameters. In Simulink, you can import and export ARXML descriptions of AUTOSAR static and constant memory. When exported in ARXML files, static memory and constant memory allow the use of calibration and measurement tools to monitor the internal memory data.

To model AUTOSAR static memory in Simulink, use the Code Mappings editor, **Signals/States** or **Data Stores** tab. Select a signal, state, or data store and map it to `StaticMemory`. To view and modify AUTOSAR code and calibration attributes for the static memory, click the  icon.

To model AUTOSAR constant memory in Simulink, use the Code Mappings editor, **Parameters** tab, to select a parameter and map it to `ConstantMemory`. To view and modify AUTOSAR code and calibration attributes for the constant memory, click the  icon.


For more information, see “Configure AUTOSAR Static Memory” on page 4-206 and “Configure AUTOSAR Constant Memory” on page 4-210.

Shared and Per-Instance Parameters


AUTOSAR supports shared parameters (`SharedParameters`) and per-instance parameters (`PerInstanceParameters`) for use in software components that are potentially instantiated multiple times. Shared parameter values are shared among all instances of a component. Per-instance parameter values are unique and private to each component instance.

In Simulink, you can import and export ARXML descriptions of AUTOSAR shared and per-instance parameters. When exported in ARXML files, shared and per-instance parameters enable the use of calibration and measurement tools to monitor component parameters.

To model an AUTOSAR shared parameter in Simulink, configure a model workspace parameter that is not a model argument (that is, not unique to each instance of a multi-instance model). For example, in the Model Explorer view of the parameter, clear the **Argument** property. In the Code Mappings editor, **Parameters** tab, select the parameter and map it to parameter type `SharedParameter`. To

view and modify AUTOSAR code and calibration attributes for the shared parameter, click the  icon.

To model an AUTOSAR per-instance parameter in Simulink, configure a model workspace parameter that is a model argument (that is, unique to each instance of a multi-instance model). For example, in the Model Explorer view of the parameter, select the **Argument** property. In the Code Mappings editor, **Parameters** tab, select the parameter and map it to parameter `PerInstanceParameter`. To

view and modify AUTOSAR code and calibration attributes for the per-instance parameter, click the  icon.

For more information, see “Configure AUTOSAR Shared or Per-Instance Parameters” on page 4-212.

Port Parameters

The AUTOSAR standard defines port-based parameters for parameter communication. AUTOSAR parameter communication relies on a parameter software component (`ParameterSwComponent`) and one or more atomic software components that require port-based access to parameter data. The `ParameterSwComponent` represents memory containing AUTOSAR parameters and provides parameter data to connected atomic software components.

In Simulink, you can model the receiver side of AUTOSAR parameter communication. By importing ARXML descriptions or configuring a software component model, you can model:

- AUTOSAR parameter receiver component, which communicates with a `ParameterSwComponent` to receive parameter data.
- AUTOSAR parameter interface, which contains parameter data elements. The data elements map to parameter or lookup table objects in the model workspace.
- AUTOSAR parameter receiver port used to communicate with the `ParameterSwComponent`.

When you generate code for the AUTOSAR parameter receiver component:

- The exported ARXML files contain descriptions of the parameter receiver component, parameter interface, parameter data elements, and parameter receiver port.

- The generated C code contains AUTOSAR port parameter Rte function calls.

At run time, the software can access the parameter data elements as port-based parameters.

Because port parameter data is scoped to the model workspace and the AUTOSAR component:

- Different components can use the same parameter names without naming conflicts.
- An AUTOSAR composition can contain multiple instances of a parameter receiver component, each with instance-specific port parameter data values.

For more information, see “Configure AUTOSAR Port Parameters for Communication with Parameter Component” on page 4-171.

See Also

Data Store Memory

Related Examples

- “Configure AUTOSAR Runnables and Events” on page 4-178
- “Configure Variants for AUTOSAR Runnable Implementations” on page 4-220
- “Configure Internal Data Types for AUTOSAR IncludedDataTypeSets” on page 4-199
- “Configure AUTOSAR Per-Instance Memory” on page 4-201
- “Configure AUTOSAR Static Memory” on page 4-206
- “Configure AUTOSAR Constant Memory” on page 4-210
- “Configure AUTOSAR Shared or Per-Instance Parameters” on page 4-212
- “Configure AUTOSAR Port Parameters for Communication with Parameter Component” on page 4-171

More About

- “AUTOSAR Component Configuration” on page 4-3

Model AUTOSAR Variants

AUTOSAR software components use variants to enable or disable AUTOSAR interfaces or implementations in the execution path, based on defined conditions. Variation points in a component present a choice between two or more variants. Components can:

- Enable or disable an AUTOSAR port or runnable.
- Vary the implementation of an AUTOSAR runnable.
- Vary the array size of an AUTOSAR port.
- Specify predefined variants and system constant value sets for controlling variants in the component.

In Simulink, you can:

- Import and export AUTOSAR ports and runnables with variants.
- Model AUTOSAR variants.
 - To enable or disable an AUTOSAR port or runnable, use Variant Sink and Variant Source blocks.
 - To vary the implementation of an AUTOSAR runnable, use Variant Subsystem blocks.
 - To vary the array size of an AUTOSAR port, use Simulink symbolic dimensions.
- Resolve modeled variants by using predefined variants and system constant value sets imported from ARXML files.

AUTOSAR system constants serve as inputs to control component variation points. To model system constants, use `AUTOSAR.Parameter` data objects.

Variants for Ports and Runnables

AUTOSAR software components can use `VariationPoint` elements to enable or disable AUTOSAR elements, such as ports and runnables, based on defined conditions. In Simulink, you can:

- Import AUTOSAR ports and runnables with variation points.

The ARXML importer creates the required model elements, including Variant Sink and Variant Source blocks to propagate variant conditions and `AUTOSAR.Parameter` data objects to represent system constants with condition values.

- Model AUTOSAR elements with variation points.
 - To define variant condition logic and propagate variant conditions, use Variant Sink and Variant Source blocks.
 - To model AUTOSAR system constants and define condition values, use `AUTOSAR.Parameter` data objects with storage class `SystemConstant`.
- Run validation on the AUTOSAR configuration. The validation software verifies that variant conditions on Simulink blocks match the designed behavior from the imported ARXML files.
- Export AUTOSAR ports and runnables with variation points.

For more information, see “Configure Variants for AUTOSAR Elements” on page 4-217.

Variants for Runnable Implementations

To vary the implementation of an AUTOSAR runnable, AUTOSAR software components can specify variant condition logic inside a runnable. In Simulink, to model variant condition logic inside a runnable:

- Use Variant Subsystem blocks to define variant implementations and their associated variant condition logic.
- Use `AUTOSAR.Parameter` data objects to model AUTOSAR system constants and define condition values.

For more information, see “Configure Variants for AUTOSAR Runnable Implementations” on page 4-220.

Variants for Array Sizes

AUTOSAR software components can flexibly specify the dimensions of an AUTOSAR element, such as a port, by using a symbolic reference to a system constant. The system constant defines the array size of the port data type. The code generator supports models that include AUTOSAR elements with variant (symbolic) array sizes.

In Simulink, you can:

- Import AUTOSAR elements with variant array sizes.
 - The ARXML importer creates the required model elements, including `AUTOSAR.Parameter` data objects with storage class `SystemConstant`, to represent the array size values.
 - Each block that represents an AUTOSAR element with variant array sizes references `AUTOSAR.Parameter` data objects to define its dimensions.
- Model AUTOSAR elements with variant array sizes.
 - Create blocks that represent AUTOSAR elements.
 - To represent array size values, add `AUTOSAR.Parameter` data objects with storage class `SystemConstant`.
 - To specify an array size for an AUTOSAR element, reference an `AUTOSAR.Parameter` data object.
- Modify array size values in system constants between model simulations, without regenerating code for simulation.
- Generate C code and ARXML files with symbols corresponding to variant array sizes.

For more information, see “Configure Dimension Variants for AUTOSAR Array Sizes” on page 4-225.

Predefined Variants and System Constant Value Sets

To define the values that control variation points in an AUTOSAR software component, components use the following AUTOSAR elements:

- `SwSystemconst` — Defines a system constant that serves as an input to control a variation point.
- `SwSystemconstantValueSet` — Specifies a set of system constant values.

- **PredefinedVariant** — Describes a combination of system constant values, among potentially multiple valid combinations, to apply to an AUTOSAR software component.

Suppose that you have an ARXML specification of an AUTOSAR software component. If the ARXML files also define a **PredefinedVariant** or **SwSystemconstantValueSets** for controlling variation points in the component, you can resolve the variation points at model creation time. Specify a **PredefinedVariant** or **SwSystemconstantValueSets** with which the importer can initialize **SwSystemconst** data.

After model creation, you can run simulations and generate code based on the combination of variation point input values that you specified.

In Simulink, using the AUTOSAR property function **createSystemConstants**, you can redefine the **SwSystemconst** data that controls variation points without recreating the model. You can run simulations and generate code based on the revised combination of variation point input values.

Building the model exports previously imported **PredefinedVariants** and **SwSystemconstantValueSets** to ARXML files.

For more information, see “Control AUTOSAR Variants with Predefined Value Combinations” on page 4-227.

See Also

Related Examples

- “Configure Variants for AUTOSAR Elements” on page 4-217
- “Configure Variants for AUTOSAR Runnable Implementations” on page 4-220
- “Configure Dimension Variants for AUTOSAR Array Sizes” on page 4-225
- “Control AUTOSAR Variants with Predefined Value Combinations” on page 4-227

More About

- “AUTOSAR Component Configuration” on page 4-3

Model AUTOSAR Nonvolatile Memory

The AUTOSAR standard defines implicit and explicit mechanisms by which an AUTOSAR software component can read and write nonvolatile memory in an automotive system:

- Implicit access uses sender-receiver ports or data store memory blocks to access a copy of an AUTOSAR nonvolatile memory block in RAM.
- Explicit access uses client-server calls to directly access an AUTOSAR nonvolatile memory block.

Implicit Access to AUTOSAR Nonvolatile Memory

Implicit access to AUTOSAR nonvolatile memory uses a startup event to begin shadowing or mirroring a nonvolatile memory block in RAM. Using a RAM copy of nonvolatile memory can support faster access.

- 1 During ECU power-up, when a startup event occurs, a background task copies a memory block from nonvolatile memory space to RAM.
- 2 While the system runs, software components can access the nonvolatile data at RAM speed.
- 3 When a shutdown event occurs, before shutdown, a background task copies the shadowed or mirrored memory block back to nonvolatile memory space.

To model implicit read and write access to nonvolatile memory in an AUTOSAR component model, you configure either port-based nonvolatile (NV) data communication or an NVRAM mirror block.

In port-based NV data communication, an AUTOSAR software component reads and writes data to AUTOSAR nonvolatile components. To implement NV data communication, AUTOSAR software components define provide and require ports that send and receive NV data. In Simulink, you can:

- Import AUTOSAR NV data interfaces and ports from ARXML files.
- Create AUTOSAR NV interfaces and ports, and map Simulink inports and outports to AUTOSAR NV ports.

You model AUTOSAR NV ports with Simulink inports and outports, in the same manner described in “Sender-Receiver Interface” on page 2-21.

- Generate C code and ARXML files for AUTOSAR NV data interfaces and ports.

With port-based NV data communication, you can distribute or coordinate NV data access across software components. For example, multiple components can read the same NV data from a nonvolatile software component, while one component writes to it.

For more information, see “Configure AUTOSAR Nonvolatile Data Communication” on page 4-169.

To configure an NVRAM mirror block, an AUTOSAR software component maps a data store memory block to AUTOSAR typed per-instance memory (`ArTypedPerInstanceMemory`) and selects the option **NeedsNVRAMAccess**. This option indicates that the `ArTypedPerInstanceMemory` is a RAM mirror block and requires service from the NVRAM Manager (`NvM`) manager module. In Simulink, you can:

- Import AUTOSAR NVRAM mirror blocks from ARXML files.
- Create model content that configures data store memory blocks as AUTOSAR NVRAM mirror blocks.

- Generate C code and ARXML files for AUTOSAR NVRAM mirror blocks. An AUTOSAR run-time environment generator allocates the memory and provides an API through which the component accesses the memory.

For more information, see “Configure AUTOSAR Per-Instance Memory” on page 4-201.


Explicit Access to AUTOSAR Nonvolatile Memory

For the AUTOSAR Classic Platform, the AUTOSAR standard defines important services as part of Basic Software (BSW) that runs in the AUTOSAR Runtime Environment (RTE). Examples include services provided by the NVRAM Manager and the Diagnostic Event Manager. In the AUTOSAR RTE, AUTOSAR software components typically access BSW services using client-server communication.

Explicit access to AUTOSAR nonvolatile memory uses calls to the NVRAM Manager (NvM) service to directly access AUTOSAR nonvolatile memory space. Explicit access can be used in response to events, for example, air bag events, or at each time step, for example, for controllers that have no shutdown sequence.

To support system-level modeling of AUTOSAR components and services, AUTOSAR Blockset provides an AUTOSAR Basic Software block library. The library includes preconfigured Function Caller blocks for modeling component calls to NVM service interfaces, including NvMAdminCaller and NvMServiceCaller.

To implement client calls to AUTOSAR NVM service interfaces in your AUTOSAR software component, you drag and drop Basic Software blocks into an AUTOSAR model. Each block has prepopulated parameters, such as **Client port name** and **Operation**. If you modify the operation selection, the software updates the block inputs and outputs to correspond.

To configure the added blocks in the AUTOSAR software component, click the **Update** button  in the Code Mappings editor view of the model. The software creates AUTOSAR client-service interfaces, operations, and ports, and maps each Simulink function caller to an AUTOSAR client port and operation.

For more information, see “Configure Calls to AUTOSAR NVRAM Manager Service” on page 7-28.

To simulate an AUTOSAR component model that calls BSW services, create a containing composition, system, or harness model. In that containing model, provide a reference implementation of the NvM service operations called by the component.

The AUTOSAR Basic Software block library includes an NVRAM Service Component block. The block provides reference implementations NvM service operations. To support simulation of component calls to the NvM service, include the blocks in the containing model. You can insert the blocks in either of two ways:

- Automatically insert the blocks by creating a Simulink Test™ harness model.
- Manually insert the blocks into a containing composition, system, or harness model.

For more information, see “Configure AUTOSAR Basic Software Service Implementations for Simulation” on page 7-33 and “Simulate AUTOSAR Basic Software Services and Run-Time Environment” on page 7-36.

See Also

NvMAdminCaller | NvMServiceCaller | NVRAM Service Component

Related Examples

- “Configure AUTOSAR Nonvolatile Data Communication” on page 4-169
- “Configure AUTOSAR Per-Instance Memory” on page 4-201
- “Configure Calls to AUTOSAR NVRAM Manager Service” on page 7-28
- “Configure AUTOSAR Basic Software Service Implementations for Simulation” on page 7-33
- “Simulate AUTOSAR Basic Software Services and Run-Time Environment” on page 7-36

Model AUTOSAR Data Types

The AUTOSAR standard defines platform data types for use by AUTOSAR software components. In Simulink, you can model AUTOSAR data types used in elements such as data elements, operation arguments, calibration parameters, measurement variables, and inter-runnable variables. If you import an AUTOSAR component from ARXML files, Embedded Coder imports AUTOSAR data types and creates the required corresponding Simulink data types. During code generation, Embedded Coder exports ARXML descriptions for data types used in the component model and generates AUTOSAR data types in C code.

About AUTOSAR Data Types

AUTOSAR specifies data types that apply to:

- Data elements of a sender-receiver Interface
- Operation arguments of a client-server Interface
- Calibration parameters
- Measurement variables
- Inter-runnable variables

The data types fall into two categories:

- Platform (primitive) data types, which allow a direct mapping to C intrinsic types.
- Composite data types, which map to C arrays and structures.

To model AUTOSAR platform data types, use corresponding Simulink data types. You can configure the platform type names in the **XMLOptions** of the AUTOSAR Dictionary. For more information, see “AUTOSAR Platform Types” on page 4-47.

Simulink Data Type	AUTOSAR 3.x Platform Type	AUTOSAR 4.x Platform Type
boolean	Boolean	boolean
single	Float	float32
double	Double	float64
int8	SInt8	sint8
int16	SInt16	sint16
int32	SInt32	sint32
int64	SInt64	sint64
uint8	UInt8	uint8
uint16	UInt16	uint16
uint32	UInt32	uint32
uint64	UInt64	uint64

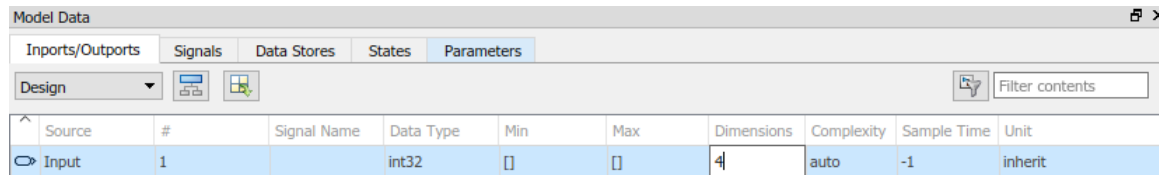
Not recommended starting in R2023a

AUTOSAR 3.x platform names will be removed in a future release.

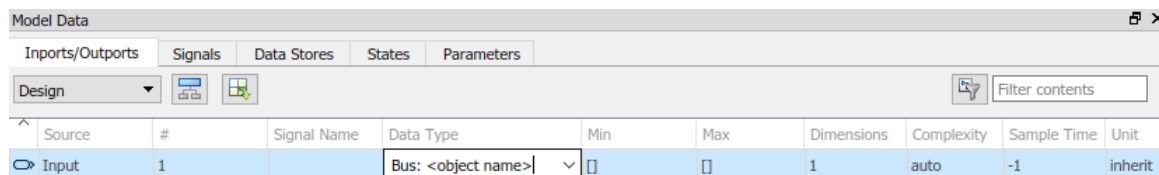
AUTOSAR composite data types are arrays and records, which are represented in Simulink by wide signals and bus objects, respectively. To configure a wide signal or bus object through Inport or

Output blocks, use the Model Data Editor. On the **Modeling** tab, click **Model Data Editor** and select the **Inports/Outports** tab. Select the **Design** view. From the list of inports and outports, select the source block to configure.

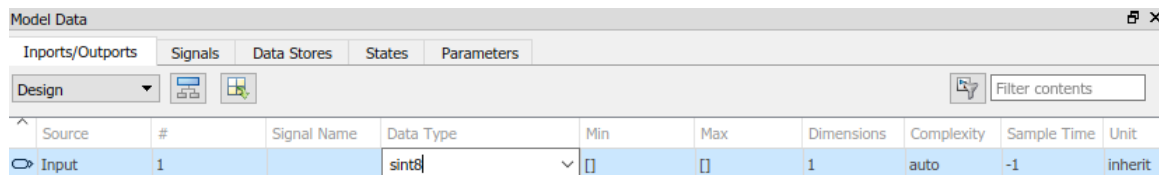
The following figure shows how to specify a wide signal, which corresponds to an AUTOSAR composite array.



The following figure shows how to specify a bus object, which corresponds to an AUTOSAR composite record.



To specify the data types of data elements and arguments of an operation prototype, use the drop-down list in the **Data Type** column. You can specify a Simulink built-in data type, such as `boolean`, `single`, or `int8`, or enter an (alias) expression for data type. For example, the following figure shows an alias `sint8`, corresponding to an AUTOSAR data type, in the **Data Type** column.



For more guidance in specifying the data type, you can use the **Data Type Assistant** on the **Signal Attributes** pane of the Inport or Output Block Parameters dialog box or in the Property Inspector.

Enumerated Data Types

AUTOSAR supports enumerated data types. For the import process, if there is a corresponding Simulink enumerated data type, the software uses the data type. The software checks that the two data types are consistent. However, if a corresponding Simulink data type is not found, the software automatically creates the enumerated data type using the `Simulink.defineIntEnumType` class. This automatic creation of data types is useful when you want to import a large quantity of enumerated data types.

Consider the following example:

```
<SHORT-NAME>BasicColors</SHORT-NAME>
  <COMPU-INTERNAL-TO-PHYS>
    <COMPU-SCALES>
      <COMPU-SCALE>
        <LOWER-LIMIT>0</LOWER-LIMIT>
        <UPPER-LIMIT>0</UPPER-LIMIT>
```



```
<COMPU-CONST>
<VT>Red</VT>
```

The software creates an enumerated data type using:

```
Simulink.defineIntEnumType( 'BasicColors', ...
    {'Red', 'Green', 'Blue'}, ...
    [0;1;2], ...
    'Description', 'Type definition of BasicColors.', ...
    'HeaderFile', 'Rte_Type.h', ...
    'AddClassNameToEnumNames', false);
```

Structure Parameters

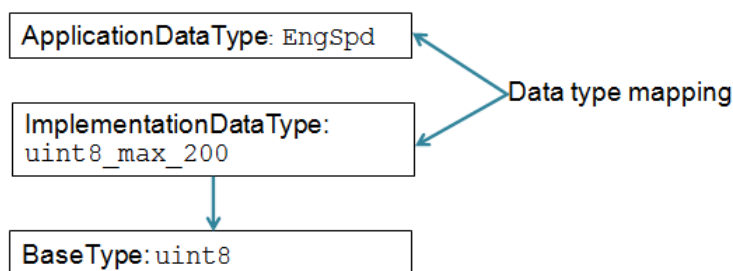
Before exporting an AUTOSAR software component, specify the data types of structure parameters to be `Simulink.Bus` objects. See “Control Field Data Types and Characteristics by Creating Parameter Object”. Otherwise, the software displays the following behavior:

- When you validate the AUTOSAR configuration, the software issues a warning.
- When you build the model, the software defines each data type to be an anonymous `struct` and generates a random, nondescriptive name for the data type.

When importing an AUTOSAR software component, if a parameter structure has a data type name that corresponds to an anonymous `struct`, the software sets the data type to `struct`. However, if the component has data elements that reference this anonymous `struct` data type, the software generates an error.

Data Types

The AUTOSAR standard defines an approach to AUTOSAR data types in which base data types are mapped to implementation data types and application data types. Application and implementation data types separate application-level physical attributes, such as real-world range of values, data structure, and physical semantics, from implementation-level attributes, such as stored-integer minimum and maximum and specification of a primitive type (for example, integer, Boolean, or real).



The software supports AUTOSAR data types in Simulink originated and round-trip workflows:

- For AUTOSAR components originated in Simulink, the software generates AUTOSAR application, implementation, and base types to preserve the information contained within Simulink data types.

In the AUTOSAR package structure created for Simulink originated components:

- You can specify separate packages to aggregate elements that relate to data types, including application data types, software base types, data type mapping sets, system constants, and units.

- Implementation data types are aggregated in the main data types package.

For more information, see “Configure AUTOSAR Packages” on page 4-84.

- For round-trip workflows involving AUTOSAR components originated outside MATLAB, the ARXML importer and exporter preserve data type information and mapping for each imported AUTOSAR data type.

For information about mapping value constraints between AUTOSAR application data types and Simulink data types, see “Application Data Type Physical Constraint Mapping” on page 2-48.

For AUTOSAR data types originated in Simulink, you can control some aspects of data type export. For example, you can control when application data types are generated, or specify the AUTOSAR package and short name exported for AUTOSAR data type mapping sets. For more information, see “Configure AUTOSAR Data Types Export” on page 4-244.

- “AUTOSAR Data Types in Simulink Originated Workflow” on page 2-46
- “AUTOSAR Data Types in Round-Trip Workflow” on page 2-47
- “Application Data Type Physical Constraint Mapping” on page 2-48

AUTOSAR Data Types in Simulink Originated Workflow

In the Simulink originated (bottom-up) workflow, you create a Simulink model and export the model as an AUTOSAR software component.

The software generates the application and implementation data types and base types to preserve the information contained within the Simulink data types:

- For Simulink data types, the software generates implementation data types.
- For each fixed-point type, in addition to the implementation data type, the software generates an application data type with the COMPU-METHOD-REF element to preserve scale and bias information. This application data type is mapped to the implementation data type.
- For each `Simulink.ValueType` object, the software generates an application data type reflecting the properties specified on the `Simulink.ValueType` object, including dimension, and minimum and maximum values. This application data type is mapped to an implementation data type.

Note The software does not support application data types for code generated from referenced models.

Simulink Data Type	AUTOSAR XML	
	Implementation Type	Application Type
Primitive (excluding fixed point), for example, <code>myInt16</code> Covers Boolean, integer, real <pre>myInt16 = Simulink.AliasType; myInt16.BaseType = 'int16';</pre>	<pre><IMPLEMENTATION-DATA-TYPE> <SHORT-NAME>myInt16</SHORT-NAME> <CATEGORY>VALUE</CATEGORY> ... </pre>	Not generated

Simulink Data Type	AUTOSAR XML	
	Implementation Type	Application Type
Primitive (fixed point), for example, myFixPt myFixPt = Simulink.NumericType; myFixPt.DataTypeMode = ...; myFixPt.IsAlias = true;	<IMPLEMENTATION-DATA-TYPE> <SHORT-NAME>myFixPt</SHORT-NAME> <CATEGORY>VALUE</CATEGORY> ...	<APPLICATION-PRIMITIVE-DATA-TYPE> <SHORT-NAME>myFixPt</SHORT-NAME> <COMPU-METHOD-REF>...
Enumeration, for example, myEnum Simulink.defineIntEnumType('myEnum',... {'Red','Green','Blue'},... [1;2;3],...);	<IMPLEMENTATION-DATA-TYPE> <SHORT-NAME>myEnum</SHORT-NAME> <CATEGORY>VALUE</CATEGORY> <COMPU-METHOD>...	Not generated
Record, for example, myRecord myRecord = Simulink.Bus;	<IMPLEMENTATION-DATA-TYPE> <SHORT-NAME>myRecord</SHORT-NAME> <CATEGORY>STRUCT</CATEGORY> ...	Not generated
Value types, for example, EngSpeed EngSpeed = Simulink.ValueType; EngSpeed.Min = 0; EngSpeed.Max = 65535; EngSpeed.DataType = 'uint32';	<IMPLEMENTATION-DATA-TYPE> <SHORT-NAME>EngSpeed</SHORT-NAME> <CATEGORY>VALUE</CATEGORY> ...	<APPLICATION-PRIMITIVE-DATA-TYPE> <SHORT-NAME>EngSpeed</SHORT-NAME> <COMPU-METHOD-REF>... <DATA-CONSTR-REF>... ...

AUTOSAR Data Types in Round-Trip Workflow

In the round-trip workflow, you first use the XML description generated by an AUTOSAR authoring tool to import on page 3-13 an AUTOSAR software component into a model. Later, you generate AUTOSAR C and XML code from the model.

If the data prototype references an application data type, the software stores application to implementation data type mapping within the model and uses the application data type name to define the Simulink data type.

For example, suppose that the authoring tool specifies an application data type:

```
AppLDT1
```

In this case, the software defines the following Simulink data type:

```
ImplLDT1
```

AUTOSAR XML		Simulink Data Type
Application Type	Implementation Type	
<APPLICATION-PRIMITIVE-DATA-TYPE> <SHORT-NAME>myFixPt</SHORT-NAME> <COMPU-METHOD-REF>...	<IMPLEMENTATION-DATA-TYPE> <SHORT-NAME>myInt</SHORT-NAME> ...	myFixPt = Simulink.NumericType; myFixPt.DataTypeMode = ...; myFixPt.IsAlias = true;

If the data prototype references an implementation data type, the software does not store mapping information and uses the implementation data type name to define the Simulink data type.

The software uses the application data types in simulations and the implementation data types for code generation. When you re-export the AUTOSAR software component, the software uses the stored information to provide the same mapping between the exported application and implementation data types.

Application Data Type Physical Constraint Mapping

In models configured for AUTOSAR, the software maps minimum and maximum values for Simulink data to the corresponding physical constraint values for AUTOSAR application data types.

- If you import ARXML files, the software imports `PhysConstr` values on `ApplicationDataTypes` in the ARXML files to `Min` and `Max` values on the corresponding Simulink data objects and root-level I/O signals.
- When you export ARXML files from a model, the software exports the `Min` and `Max` values specified on Simulink data objects and root-level I/O signals to the corresponding `ApplicationDataType PhysConstrs` in the ARXML files.
- Simulink data types with unspecified `Min` and `Max` correspond to AUTOSAR `ApplicationDataTypes` with full-range constraints. For example:
 - On import, if the `PhysConstr` values on an `ApplicationDataType` match the full lower and upper limits in the `InternalConstr` for the associated `ImplementationDataType`, the importer sets the Simulink `Min` and `Max` values to []. In those cases, Simulink implicitly enforces the default lower and upper limits based on the type.
 - On export, if the Simulink `Min` and `Max` values for a type are [], the software exports default lower and upper limit values for that type (for example, 0 and 1 for a boolean based type) to the ARXML `PhysConstr` description.

CompuMethod Categories for Data Types

AUTOSAR software components use computation methods (`CompuMethods`) to convert between the internal values and physical representation of AUTOSAR data. Common uses for `CompuMethods` are linear data scaling and calibration and measurement.

The `category` attribute of a `CompuMethod` represents a specialization of the `CompuMethod`, which can impose semantic constraints. The `CompuMethod` categories produced by the code generator include:

- `BITFIELD_TEXTTABLE` — Transform internal value into bitfield textual elements.
- `IDENTICAL` — Floating-point or integer function for which internal and physical values are identical and do not require conversion.
- `LINEAR` — Linear conversion of an internal value; for example, multiply the internal value with a factor, then add an offset.
- `RAT_FUNC` — Rational function; similar to linear conversion, but with conversion restrictions specific to rational functions.
- `SCALE_LINEAR_AND_TEXTTABLE` — Combination of `LINEAR` and `TEXTTABLE` scaling specifications.
- `TEXTTABLE` — Transform internal value into textual elements.

The ARXML exporter generates `CompuMethods` for every primitive application type, allowing calibration and measurement tools to monitor and interact with the application data. The following table shows the `CompuMethod` categories that the code generator produces for data types in a model that is configured for AUTOSAR.

Data Type	CompuMethod Category	CompuMethod on Application Type	CompuMethod on Implementation Type
Bitfield	BITFIELD_TEXTTABLE	Yes	Yes
Boolean	TEXTTABLE	Yes	Yes
Enumerated without storage type	TEXTTABLE	Yes	Yes
Enumerated with storage type	TEXTTABLE	Yes	No
Fixed-point	LINEAR RAT_FUNC (limited to reciprocal scaling) SCALE_LINEAR_AND_TEXTTABLE	Yes	No
Floating-point	IDENTICAL SCALE_LINEAR_AND_TEXTTABLE	Yes	No
Integer	IDENTICAL SCALE_LINEAR_AND_TEXTTABLE	Yes	No

For enumerated data types, the ARXML importer tool adheres to the AUTOSAR standard and sets the CompuMethod category TEXTTABLE to the following:

- 1 The value of the symbol attribute if it exists.
- 2 The value VT if it is a valid C identifier.
- 3 The value of the shortLabel.

For floating-point and integer data types that do not require conversion between internal and physical values, the exporter generates a generic CompuMethod with category IDENTICAL and short-name Identcl.

For information about configuring CompuMethods for code generation, see “Configure AUTOSAR CompuMethods” on page 4-236.

See Also

Related Examples

- “Organize Data into Structures in Generated Code” (Embedded Coder)
- “Configure AUTOSAR Data Types Export” on page 4-244
- “Automatic AUTOSAR Data Type Generation” on page 4-248
- “Configure AUTOSAR CompuMethods” on page 4-236

More About

- “AUTOSAR Data Types”

Model AUTOSAR Calibration Parameters and Lookup Tables

In Simulink, you can model AUTOSAR calibration parameters and lookup tables, which support run-time tuning of the AUTOSAR application with calibration and measurement tools.

AUTOSAR Calibration Parameters


A calibration parameter is a value in an Electronic Control Unit (ECU). You tune or modify these parameters using a calibration data management tool or an offline calibration tool.

The AUTOSAR standard specifies the following types of calibration parameters:

- Calibration parameters that belong to a *calibration component* (ParameterSwComponent), which AUTOSAR software components can access.
- Internal calibration parameters, which only one AUTOSAR software component defines and accesses.

To provide your Simulink model with access to calibration parameters, reference the calibration parameters in block parameters.

To map Simulink parameter objects in the model workspace to AUTOSAR calibration parameters, open the AUTOSAR code perspective and use the Code Mappings editor, **Parameters** tab. To view

and modify AUTOSAR code and calibration attributes for a selected parameter, click the  icon. For more information, see “Map Model Workspace Parameters to AUTOSAR Component Parameters” on page 4-54.

Calibration Parameters for STD_AXIS, FIX_AXIS, and COM_AXIS Lookup Tables

You can model standard axis (STD_AXIS), fix axis (FIX_AXIS), and common axis (COM_AXIS) lookup tables for AUTOSAR applications. AUTOSAR applications can use lookup tables in either or both of two ways:

- Implement fast search operations.
- Support tuning of the application with calibration and measurement tools.

A lookup table uses an array of data to map input values to output values, approximating a mathematical function. An n -dimensional lookup table can approximate an n -dimensional function. A COM_AXIS lookup table is one in which tunable breakpoints (axis points) are shared among multiple table axes.

The AUTOSAR standard defines calibration parameter categories for STD_AXIS, FIX_AXIS, and COM_AXIS lookup table data:

- CURVE, MAP, and CUBOID parameters represent 1-D, 2-D, and 3-D table data, respectively.
- COM_AXIS parameters represent axis data.

In Simulink, you can:

- Import ARXML files that contain AUTOSAR lookup tables in STD_AXIS, FIX_AXIS, and COM_AXIS configurations:

- For a lookup table in a STD_AXIS configuration, the importer creates a lookup table block and initializes it with a `Simulink.LookupTable` object.
- For a lookup table in a FIX_AXIS configuration, the importer creates a lookup table block and initializes the table values with a `Simulink.Parameter` object and the breakpoint values are initialized with values from fix axes parameters.
- For a lookup table in a COM_AXIS configuration, the importer creates a prelookup block initialized with a `Simulink.Breakpoint` object and an interpolation-using-prelookup block initialized with a `Simulink.LookupTable` object.
- The importer maps each created Simulink lookup table to AUTOSAR parameters with code and calibration attributes.
- If the ARXML files define input variables that measure lookup table inputs, the importer creates corresponding model content. If the input variables are global variables, the importer connects static global signals to lookup table block inputs. If the input variables are root-level inputs, the importer connects root-level inports to lookup table block inputs.
- Create STD_AXIS, FIX_AXIS, and COM_AXIS lookup tables and map them to AUTOSAR parameters. You map lookup table objects to AUTOSAR parameters by using the Code Mappings editor, **Parameters** tab.
 - To model an AUTOSAR lookup table in a STD_AXIS configuration, create an AUTOSAR Blockset Curve or Map block.

Open each lookup table block and configure it to generate a routine from the AUTOSAR 4.0 code replacement library (CRL). As you modify block settings, the block dialog box updates the name of the targeted AUTOSAR routine.

To store the data, create a single `Simulink.LookupTable` object in the model workspace. Use the object in the Curve or Map block.

Data appears in the generated C code as fields of a single structure. To control the characteristics of the structure type, such as its name, use the properties of the object.

- To model an AUTOSAR lookup table in a FIX_AXIS configuration, create an AUTOSAR Blockset Curve or Map block with **Breakpoints specification: Even spacing**.

Open each lookup table block and configure it to generate a routine from the AUTOSAR 4.0 code replacement library (CRL). As you modify block settings, the block dialog box updates the name of the targeted AUTOSAR routine.

To store the data, create a single `Simulink.Parameter` object in the model workspace. Use the object in the 1-D Lookup Table block.

Table data appears in the generated C code as a separate variable. The breakpoint values appear as constants.


- To model an AUTOSAR lookup table in a COM_AXIS configuration, create one or more AUTOSAR Blockset Prelookup blocks. Pair each Prelookup with an AUTOSAR Blockset Curve Using Prelookup or Map Using Prelookup block.

Open each lookup table block and configure it to generate a routine from the AUTOSAR 4.0 code replacement library (CRL). As you modify block settings, the block dialog box updates the name of the targeted AUTOSAR routine.

To store each set of table data, create a `Simulink.LookupTable` object in the model workspace. To store each breakpoint vector, create a `Simulink.Breakpoint` object in the

model workspace. Use each `Simulink.LookupTable` object in a Curve Using Prelookup or Map Using Prelookup block and each `Simulink.Breakpoint` object in a Prelookup block. You can reduce memory consumption by sharing breakpoint data between lookup tables.

Each set of table data appears in the generated C code as a separate variable. If the table size is tunable, each breakpoint vector appears as a structure with one field to store the breakpoint data and, optionally, one field to store the length of the vector. The second field enables you to tune the effective size of the table. If the table size is not tunable, each breakpoint vector appears as an array.

- Add AUTOSAR operating points to the lookup tables. Connect root level inports to Curve, Map, or Prelookup blocks. Alternatively, configure input signals to Curve, Map, or Prelookup blocks with static global memory.
- To map Simulink lookup table objects in the model workspace to AUTOSAR calibration parameters, open the AUTOSAR code perspective and use the Code Mappings editor, **Parameters** tab. To view and modify AUTOSAR code and calibration attributes for a selected parameter, click the  icon. For more information, see “Map Model Workspace Parameters to AUTOSAR Component Parameters” on page 4-54
- Configure the array layout for multidimensional lookup tables. In the Simulink Configuration Parameters dialog box, **Interface** pane, set **Array layout** to `Column-major` (the default) or `Row-major`. The array layout selection affects code generation, including C code and exported ARXML `SwRecordLayout` descriptions.

If you select row-major layout, go to the **Math and Data Types** pane and select the configuration option **Use algorithms optimized for row-major array layout**. The algorithm selection affects simulation and code generation.

- In the Configuration Parameters dialog box, **Interface** pane, select the AUTOSAR 4.0 code replacement library for C code generation.
- Generate ARXML and C code with `STD_AXIS`, `FIX_AXIS`, and `COM_AXIS` lookup table content.

The generated C code contains required `Ifl` and `Ifx` lookup function calls and `Rte` data access function calls.

The generated ARXML files contain information to support run-time calibration of the tunable lookup table parameters, including:

- Lookup table calibration parameters that reference the application data types — category `CURVE`, `MAP`, or `CUBOID` for table data, or category `COM_AXIS` for axis data.
- Application data types of category `CURVE`, `MAP`, `CUBOID`, and `COM_AXIS`, with the data calibration properties that you configured. The properties include **SwCalibrationAccess**, **DisplayFormat**, and **SwAddrMethod**.
- Software record layouts (`SwRecordLayouts`) referenced by the application data types of category `CURVE`, `MAP`, `CUBOID`, and `COM_AXIS`.

For more information, see “Configure Lookup Tables for AUTOSAR Calibration and Measurement” on page 4-273.

See Also

`Simulink.LookupTable` | `Simulink.Breakpoint` | 1-D Lookup Table | Curve | Curve Using Prelookup | Map | Map Using Prelookup | Prelookup | `getParameter` | `mapParameter`

Related Examples

- “Map Model Workspace Parameters to AUTOSAR Component Parameters” on page 4-54
- “Configure Lookup Tables for AUTOSAR Calibration and Measurement” on page 4-273

More About

- “Code Generation with AUTOSAR Code Replacement Library” on page 5-12

AUTOSAR Component Creation

- “Create AUTOSAR Software Component in Simulink” on page 3-2
- “Create and Configure AUTOSAR Software Component” on page 3-8
- “Import AUTOSAR XML Descriptions Into Simulink” on page 3-13
- “Import AUTOSAR Software Component with Multiple Runnables” on page 3-18
- “Import AUTOSAR Component to Simulink” on page 3-19
- “Import AUTOSAR Software Composition with Atomic Software Components (Classic Platform)” on page 3-24
- “Import AUTOSAR Software Component Updates” on page 3-25
- “Import and Reference Shared AUTOSAR Element Definitions” on page 3-29
- “Import AUTOSAR Package into Component Model” on page 3-31
- “AUTOSAR ARXML Importer” on page 3-35
- “Round-Trip Preservation of AUTOSAR XML File Structure and Element Information” on page 3-37
- “Limitations and Tips” on page 3-40

Create AUTOSAR Software Component in Simulink

To create a Simulink representation of an AUTOSAR software component, import an AUTOSAR XML component description into a new model or create a component in an existing model.

To create an AUTOSAR software component in an existing model, use one of these resources:

- AUTOSAR Component Quick Start — Creates a mapped AUTOSAR software component for your model and opens the model in the AUTOSAR code perspective.
- Simulink Start Page — Provides AUTOSAR Blockset model templates as a starting point for AUTOSAR software development.

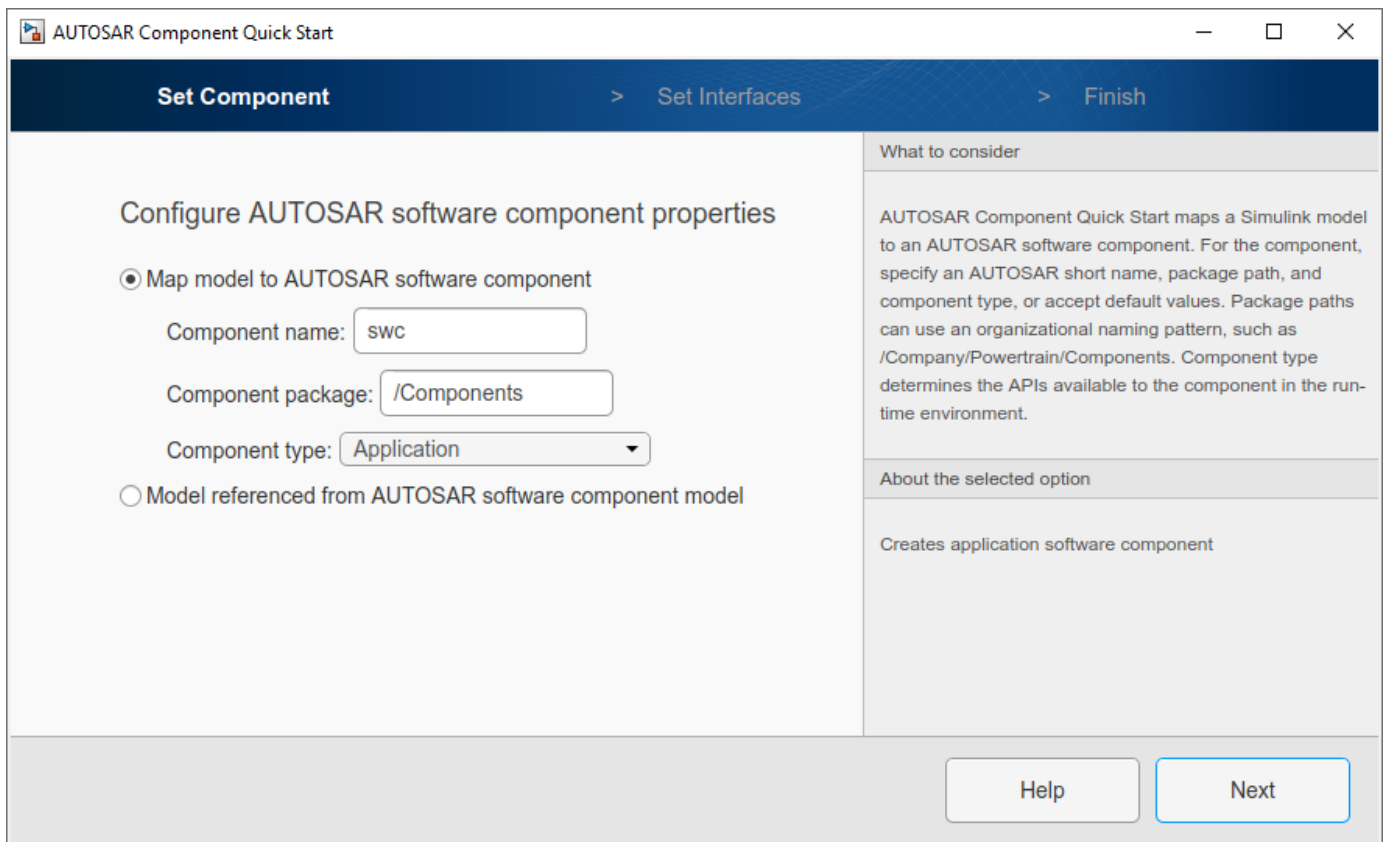
Alternatively, if you have Simulink Coder and Embedded Coder software, you can use the Embedded Coder Quick Start. To create an AUTOSAR software component for your model, open Embedded Coder Quick Start from the Embedded Coder **C Code** tab or the AUTOSAR Blockset **AUTOSAR** tab. As you work through the quick-start procedure, in the Output window, select output option **C code compliant with AUTOSAR** or **C++ code compliant with AUTOSAR Adaptive Platform**.

In this section...
“Create Mapped AUTOSAR Component with Quick Start” on page 3-2
“Create Mapped AUTOSAR Component with Simulink Start Page” on page 3-5

Create Mapped AUTOSAR Component with Quick Start

To create a mapped AUTOSAR software component using the AUTOSAR Component Quick Start:

- 1** Open a Simulink component model for which an AUTOSAR software component is not mapped. This example uses AUTOSAR example model `swc`. For adaptive component creation, you can use AUTOSAR example model `LaneGuidance`.
- 2** In the model window:
 - a** Open the Configuration Parameters dialog box, **Code Generation** pane, and set the system target file to either `autosar.tlc` or `autosar_adaptive.tlc`. Click **OK**.
 - b** On the **Apps** tab, click **AUTOSAR Component Designer**. Because the model is unmapped, the AUTOSAR Component Quick Start opens.
- 3** To configure the model for AUTOSAR software component development, work through the quick-start procedure.



In the **Set Component** pane:

- For a Classic Platform software component, specify an AUTOSAR short name, package path, and component type, or accept default values.

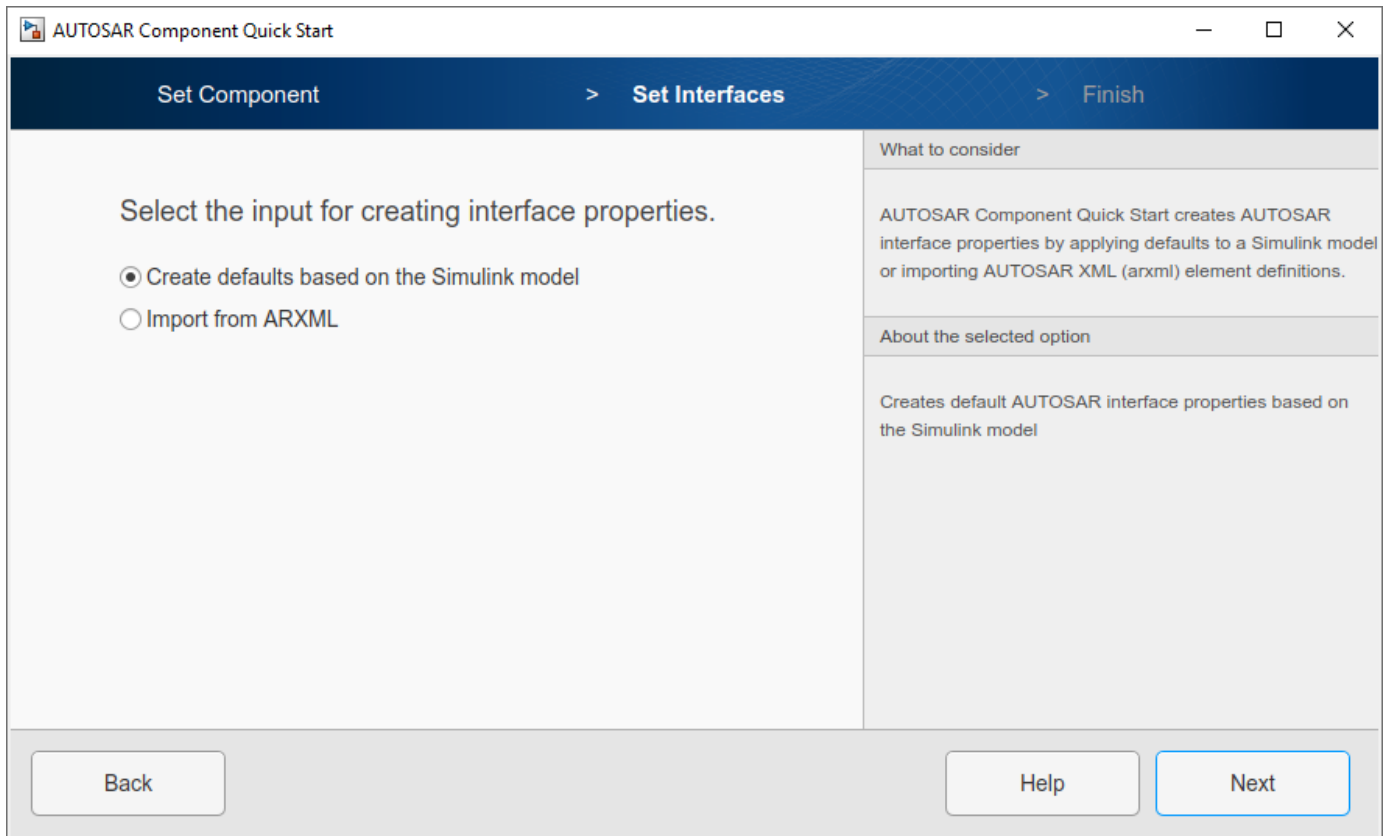
Component types include `Application`, `ComplexDeviceDriver`, `EcuAbstraction`, `SensorActuator`, and `ServiceProxy`. The most common types are `Application` and `SensorActuator`. For more information, see “Import AUTOSAR Software Composition with Atomic Software Components (Classic Platform)” on page 3-24.

- For an Adaptive Platform software component, specify an AUTOSAR short name and package path.

For the Classic Platform, you can also map a submodel referenced from an AUTOSAR software component model. For more information, see “Map Calibration Data for Submodels Referenced from AUTOSAR Component Models” on page 4-65.

Click **Next**.

- 4 If you are creating a Classic Platform software component, a **Set Interfaces** pane opens.



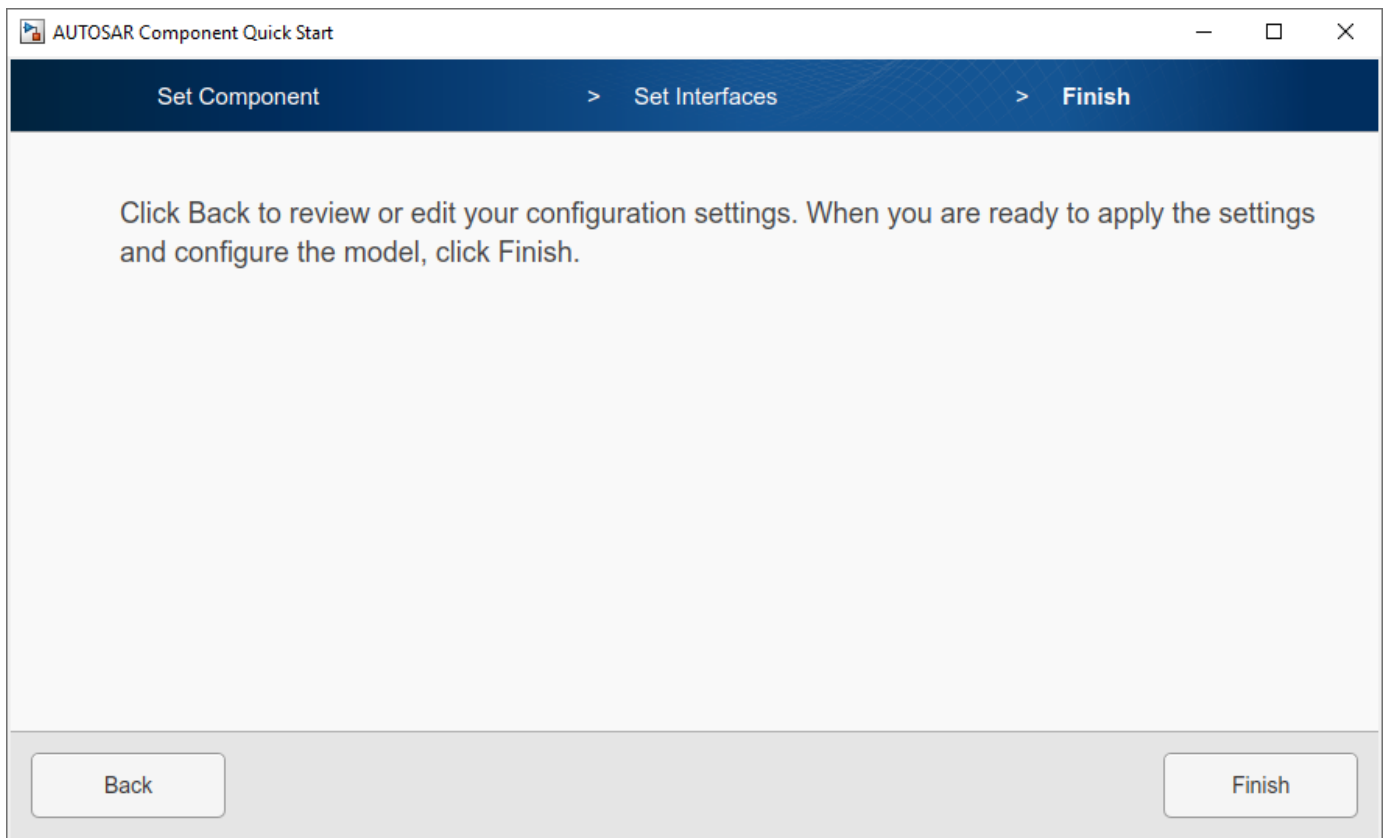
In the **Set Interfaces** pane, select an option for creating component interface properties.

- If you select **Create defaults based on the Simulink model**, at the conclusion of the quick-start procedure, the software creates component interface properties by applying AUTOSAR defaults to the model.
- If you select **Import from ARXML**, an **ARXML Files** field opens. Specify one or more AUTOSAR XML files containing packages of shared AUTOSAR element definitions. For example, you can specify data type related definitions that are common to many components. For more information, see “Import and Reference Shared AUTOSAR Element Definitions” on page 3-29 and the examples “Import AUTOSAR Package into Component Model” on page 3-31 and “Import AUTOSAR Package into Adaptive Component Model” on page 6-17.



Click **Next**.

- 5 The **Finish** pane opens.



When you click **Finish**, your model opens in the AUTOSAR code perspective. To continue configuring the component model, see “AUTOSAR Component Configuration” on page 4-3.

Create Mapped AUTOSAR Component with Simulink Start Page

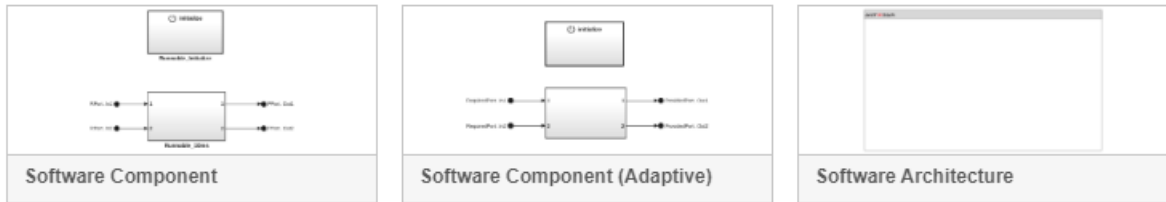
The Simulink Start Page provides AUTOSAR Blockset model templates as a starting point for AUTOSAR software component development. You can select either a Classic Platform or Adaptive Platform component template and click **Create Model**. (If you have System Composer software, you can also select an architecture model template. For more information, see “Create AUTOSAR Architecture Models” on page 8-2.)

The created model is preconfigured with AUTOSAR system target file and other code generation settings, but is not yet mapped to an AUTOSAR software component. After you examine and refine the template model, use the AUTOSAR Component Quick Start (or potentially the Embedded Coder Quick Start) to map the model to an AUTOSAR software component. For example:

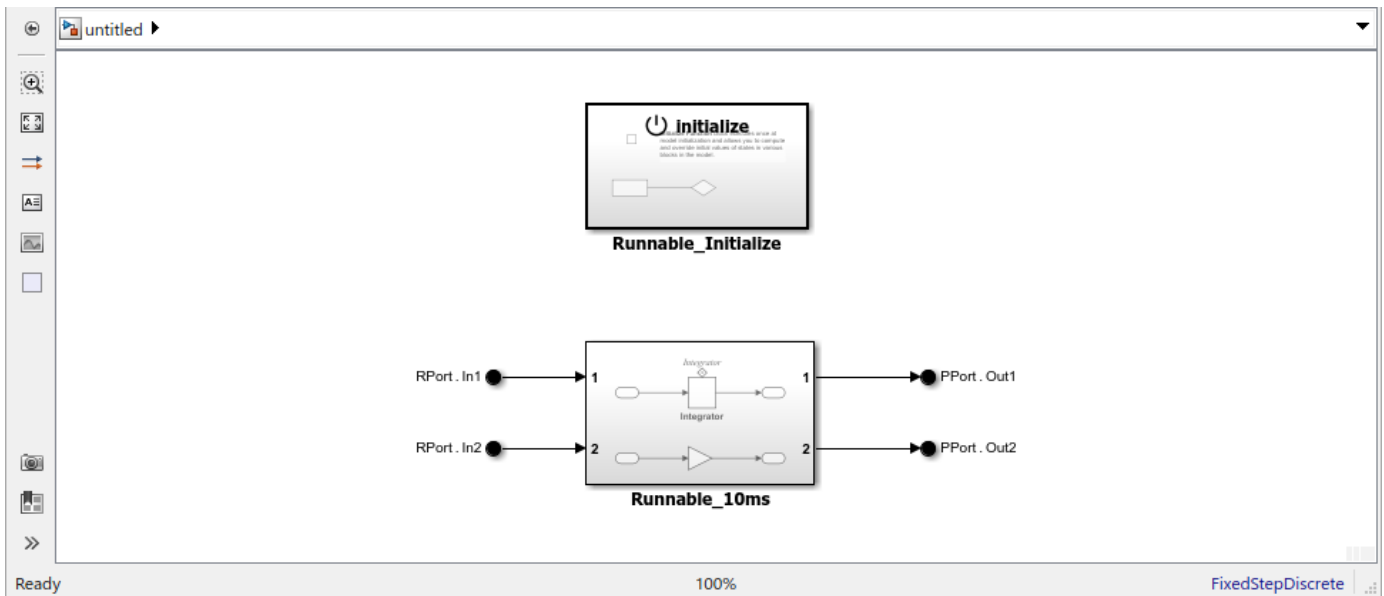
- 1 Open the Simulink Start Page. For example, enter the MATLAB command `simulink` or open a new model from the MATLAB or Simulink toolstrip.

The Start Page opens. On the **New** tab, scroll down to **AUTOSAR Blockset** and expand the product row.

▼ AUTOSAR Blockset

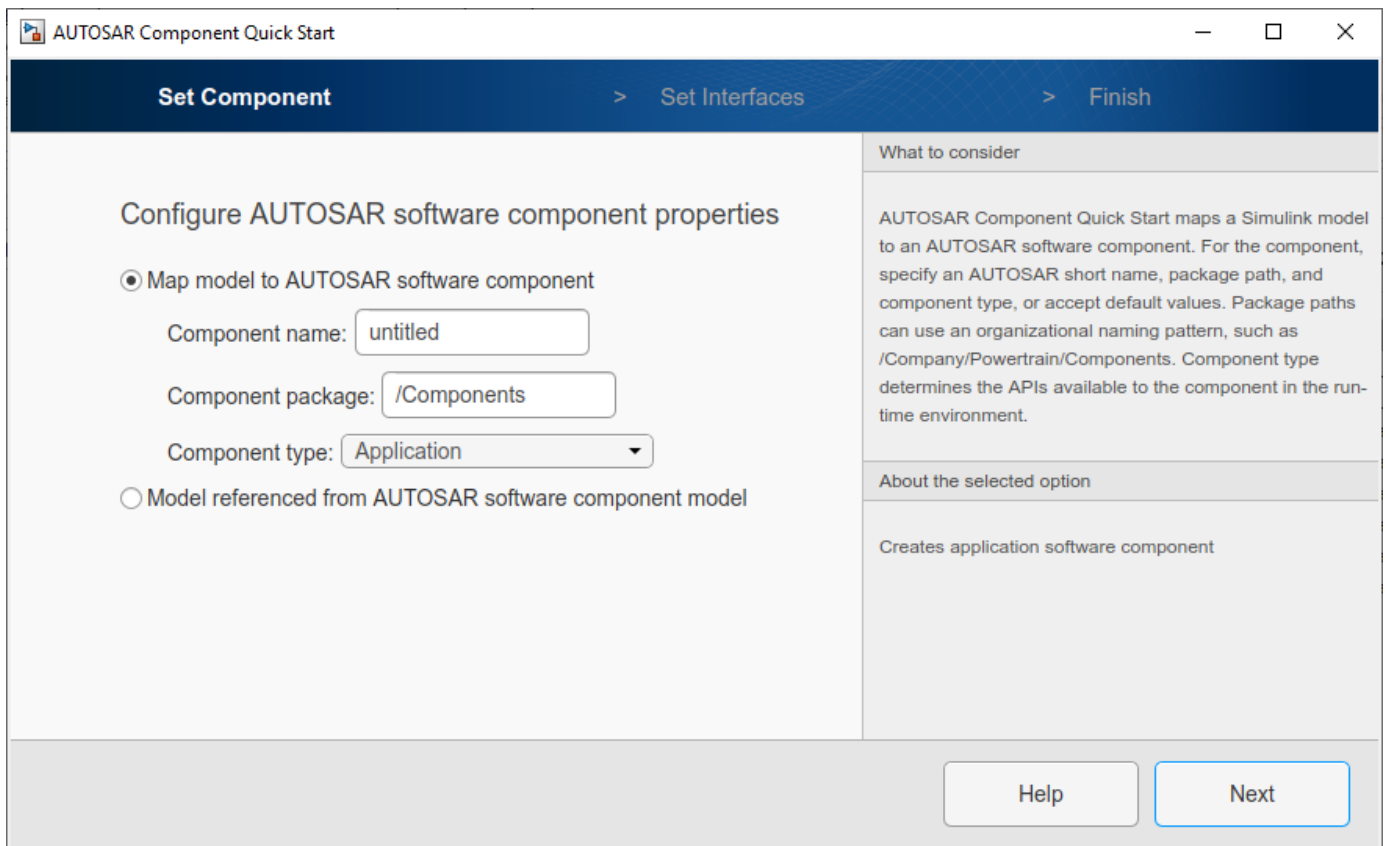


- Place your cursor over the template you want to use and click **Create Model**. A model based on the template opens. (The Simulink Start Page closes.)



In this example, the created model is a starting point for developing a software component for the AUTOSAR Classic Platform.

- Explore the model and refine the configuration according to your requirements. Optionally, you can develop the component behavior. To map the model to an AUTOSAR software component, use the AUTOSAR Component Quick Start. On the **Apps** tab, click **AUTOSAR Component Designer**. Because the model is unmapped, the AUTOSAR Component Quick Start opens.



- 4 Work through the quick-start procedure. If necessary, refer to “Create Mapped AUTOSAR Component with Quick Start” on page 3-2. When you click **Finish**, your model opens in the AUTOSAR code perspective. To continue configuring the component model, see “AUTOSAR Component Configuration” on page 4-3

See Also

Related Examples

- “AUTOSAR Component Configuration” on page 4-3
- “Create and Configure AUTOSAR Software Component” on page 3-8
- “Create and Configure AUTOSAR Adaptive Software Component” on page 6-6

Create and Configure AUTOSAR Software Component

Create an AUTOSAR software component model from an algorithm model.

AUTOSAR Blockset software supports AUTomotive Open System ARchitecture (AUTOSAR), an open and standardized automotive software architecture. Automobile manufacturers, suppliers, and tool developers jointly develop AUTOSAR components. To develop AUTOSAR components in Simulink, follow this general workflow:

- 1 Create a Simulink representation of an AUTOSAR component.
- 2 Develop the component by refining the AUTOSAR configuration and creating algorithmic model content.
- 3 Generate ARXML descriptions and algorithmic C code for testing in Simulink or integration into an AUTOSAR run-time environment. (AUTOSAR code generation requires Simulink Coder and Embedded Coder.)

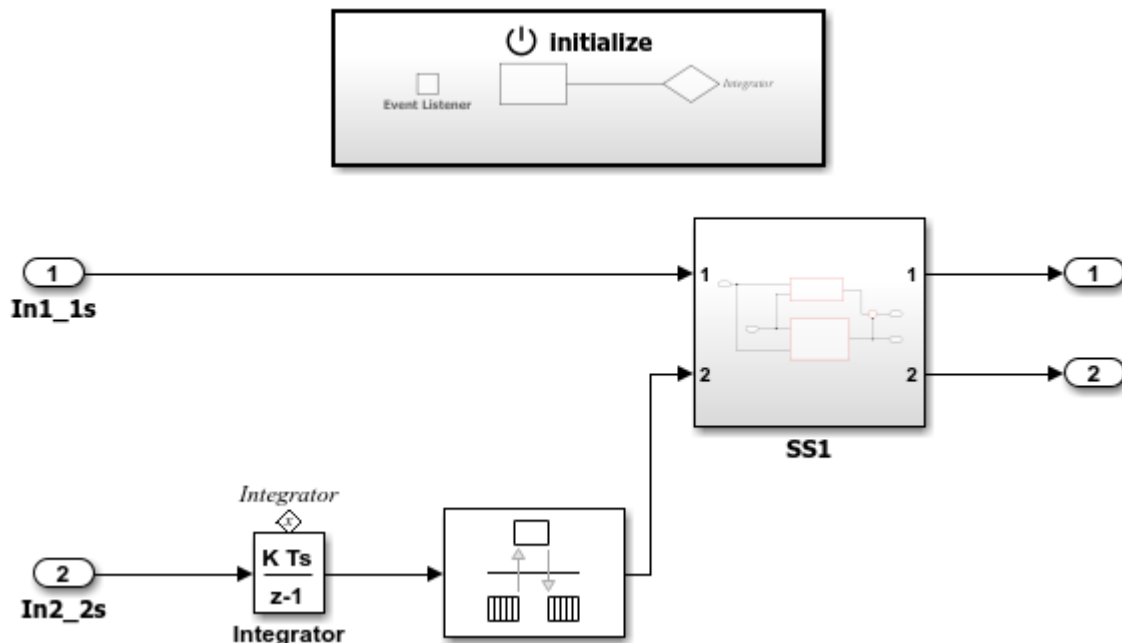
Create AUTOSAR Software Component in Simulink

To create an initial Simulink representation of an AUTOSAR software component, you take one of these actions:

- Create an AUTOSAR software component using an existing Simulink model.
- Import an AUTOSAR software component description from ARXML files into a new Simulink model. (See example “Import AUTOSAR Component to Simulink” on page 3-19.)

To create an AUTOSAR software component using an existing model, first open a Simulink component model for which an AUTOSAR software component is not mapped. This example uses AUTOSAR example model swc.

```
open_system('swc');
```



In the model window, on the **Modeling** tab, select **Model Settings**. In the Configuration Parameters dialog box, **Code Generation** pane, set the system target file to `autosar.tlc`. Click **OK**.

To configure the model as a mapped AUTOSAR software component, use the AUTOSAR Component Quick Start. On the **Apps** tab, click **AUTOSAR Component Designer**. The AUTOSAR Component Quick Start opens.

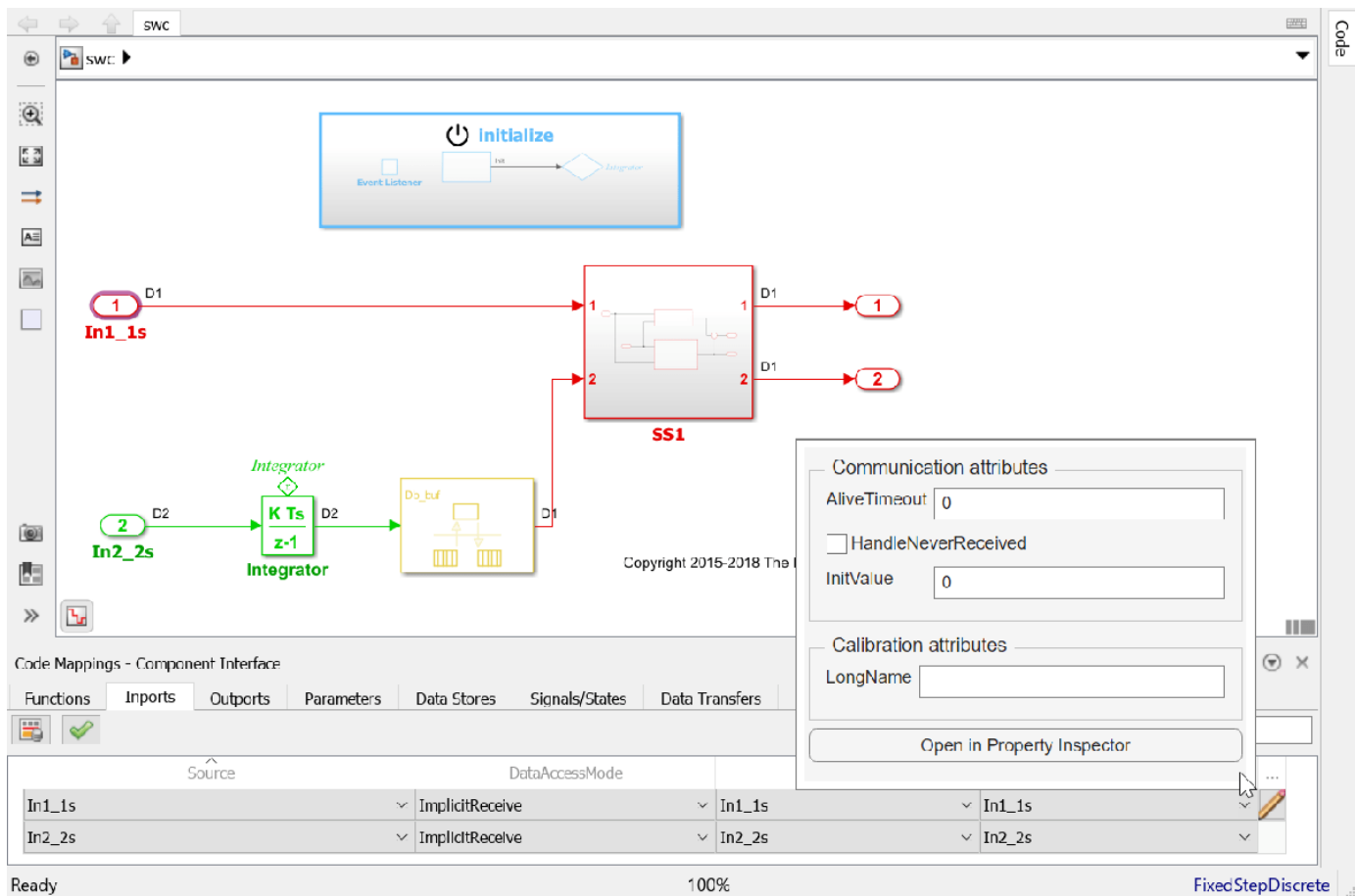
The screenshot shows the 'AUTOSAR Component Quick Start' dialog box with the 'Set Component' pane active. The breadcrumb navigation shows 'AUTOSAR Platform > Set Component > Set Interfaces > Finish'. The main area is titled 'Configure AUTOSAR software component properties'. There are two radio button options: 'Map model to AUTOSAR software component' (selected) and 'Model referenced from AUTOSAR software component model'. Under the selected option, there are three input fields: 'Component name' with the value 'swc', 'Component package' with the value '/Components', and 'Component type' with a dropdown menu showing 'Application'. To the right, there is a 'What to consider' section with explanatory text and an 'About the selected option' section stating 'Creates application software component'. At the bottom, there are 'Back', 'Help', and 'Next' buttons.

To configure the model for AUTOSAR software component development, work through the quick-start procedure. This example accepts default settings for the options in the Quick Start **AUTOSAR Platform**, **Set Component**, and **Set Interfaces** panes.

In the **Finish** pane, when you click **Finish**, your model opens in the AUTOSAR code perspective.

Configure AUTOSAR Software Component in Simulink

The AUTOSAR code perspective displays your model, and directly below the model, the Code Mappings editor.

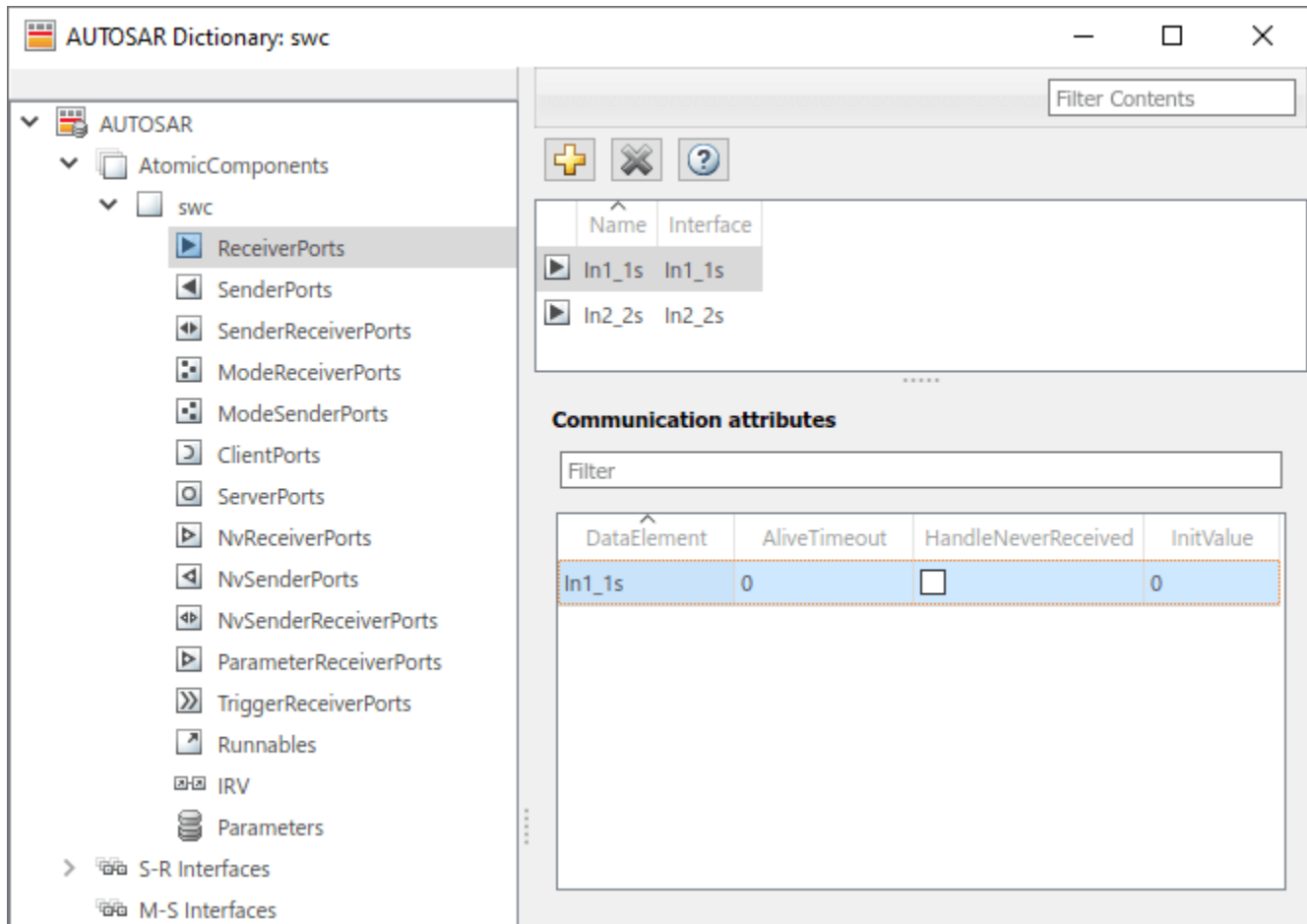


Next you use the Code Mappings editor and the AUTOSAR Dictionary to further develop the AUTOSAR component.

The Code Mappings editor displays entry-point functions, model inports, outports, parameters, and other Simulink elements relevant to your AUTOSAR platform. Use the editor to map Simulink model elements to AUTOSAR component elements from a Simulink model perspective. AUTOSAR component elements are defined in the AUTOSAR standard, and include runnable entities, ports, and inter-runnable variables (IRVs).

Open each Code Mapping tab and examine the mapped model elements. To modify the AUTOSAR mapping for an element, select an element and modify its associated properties. When you select an element, it is highlighted in the model. To view additional code and communication attributes for the element, click the edit icon.

To configure the AUTOSAR properties of the mapped AUTOSAR software component, open the AUTOSAR Dictionary. In the Code Mappings editor, click the AUTOSAR Dictionary button, which is the leftmost icon. The AUTOSAR Dictionary opens in the AUTOSAR view that corresponds to the Simulink element that you last selected and mapped in the Code Mappings editor. If you selected and mapped a Simulink inport, the dictionary opens in ReceiverPorts view and displays the AUTOSAR port to which you mapped the inport.



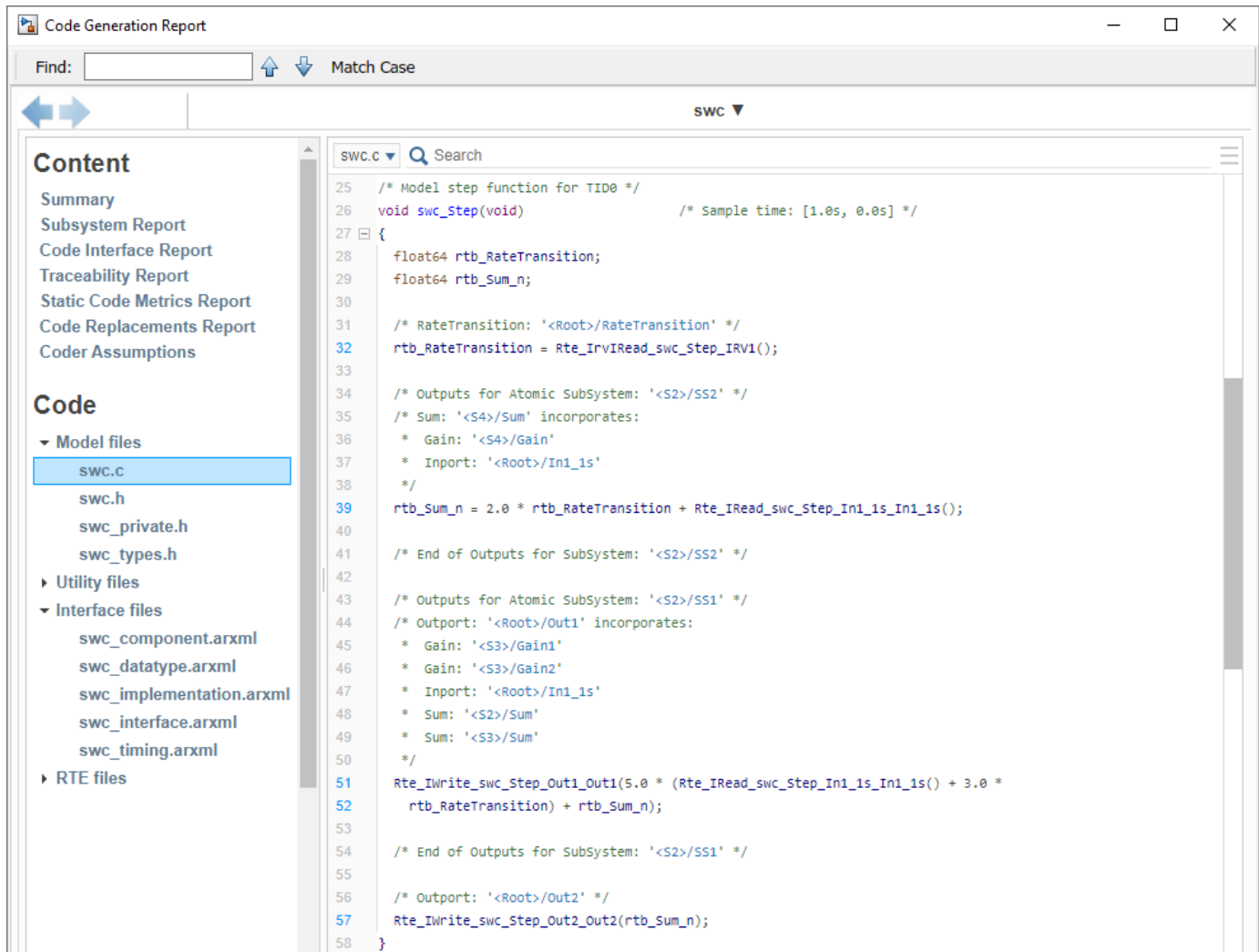
The AUTOSAR Dictionary displays the mapped AUTOSAR component and its elements, communication interfaces, computation methods, software address methods, and XML options. Use the dictionary to configure AUTOSAR elements and properties from an AUTOSAR component perspective.

Open each node and examine its AUTOSAR elements. To modify an AUTOSAR element, select an element and modify its associated properties. AUTOSAR XML and AUTOSAR-compliant C code generated from the model reflect your modifications.

Generate C Code and ARXML Descriptions (Embedded Coder)

If you have Simulink Coder and Embedded Coder software, you can build the AUTOSAR model. Building the AUTOSAR model generates AUTOSAR-compliant C code and exports AUTOSAR XML (ARXML) descriptions. In the model window, press **Ctrl+B** or, on the **AUTOSAR** tab, click **Generate Code**.

When the build completes, a code generation report opens. Examine the report. Verify that your Code Mappings editor and AUTOSAR Dictionary changes are reflected in the C code and ARXML descriptions. For example, use the **Find** field to search for the names of the Simulink model elements and AUTOSAR component elements that you modified.



Related Links

- “AUTOSAR Component Configuration” on page 4-3
- “Code Generation” (Classic Platform)
- “AUTOSAR Blockset”

Import AUTOSAR XML Descriptions Into Simulink

In Simulink, you can import AUTOSAR software components, compositions, or packages of shared elements from AUTOSAR XML (ARXML) files. You use the ARXML importer, which is implemented as an `arxml.importer` object. For more information, see “AUTOSAR ARXML Importer” on page 3-35.

To import ARXML software description files into Simulink, first call the `arxml.importer` function. In the function argument, specify one or more ARXML files that describe software components, compositions, or packages of shared elements. For example:

```
ar = arxml.importer('ThrottlePositionControlComposition.arxml')
```

The function:

- Parses the ARXML files to identify AUTOSAR software descriptions.
- If you entered the function call without a terminating semicolon (;), lists the AUTOSAR content of the specified ARXML file or files.
- Returns a handle to an ARXML importer object. In subsequent function calls, the object represents the parsed AUTOSAR software descriptions in the ARXML files.

For more information, see “Create ARXML Importer Object” on page 3-14.

Next, based on what you want to import, call an `arxml.importer` create or update function. For example:

- To import an ARXML software component and create an AUTOSAR model, call `createComponentAsModel`. For more information, see “Import Software Component and Create Model” on page 3-14.
- To import an ARXML software composition and create AUTOSAR models, call `createCompositionAsModel`. For more information, see “Import Software Composition and Create Models” on page 3-15.
- To update an existing AUTOSAR model with external ARXML file changes, call `updateModel`. For more information, see “Import Component or Composition External Updates Into Model” on page 3-16.
- To update an existing AUTOSAR component model with packages of shared element definitions, call `updateAUTOSARProperties`. For more information, see “Import Shared Element Packages into Component Model” on page 3-16.

To help support the round trip of AUTOSAR elements between an AUTOSAR authoring tool (AAT) and the Simulink model-based design environment, ARXML import preserves imported AUTOSAR XML file structure, elements, and element universal unique identifiers (UUIDs) for ARXML export. For more information, see “Round-Trip Preservation of AUTOSAR XML File Structure and Element Information” on page 3-37.

After you import an AUTOSAR software component or composition into Simulink, you can develop the behavior and configuration of the component or composition model. To refine the component configuration, see “AUTOSAR Component Configuration” on page 4-3.

To configure ARXML export options, see “Configure AUTOSAR XML Options” on page 4-43.

In this section...

“Create ARXML Importer Object” on page 3-14

In this section...

“Import Software Component and Create Model” on page 3-14

“Import Software Composition and Create Models” on page 3-15

“Import Component or Composition External Updates Into Model” on page 3-16

“Import Shared Element Packages into Component Model” on page 3-16

Create ARXML Importer Object

Before you import ARXML descriptions into Simulink, call the `arxml.importer` function. Specify the names of one or more ARXML files that contain descriptions of AUTOSAR software components, compositions, or shared elements. The function parses the descriptions in the specified ARXML files and creates an importer object that represents the parsed information. Subsequent calls to `createComponentAsModel` or other ARXML importer functions must specify the importer object.

For example, open the `autosar_sw_c` model and build it.

```
openExample('autosar_sw_c');
```

The below call specifies a main AUTOSAR software component file, `autosar_sw_c_component.arxml`, and related dependent files containing data type, implementation, and interface information that completes the software component description.

```
ar = arxml.importer({'autosar_sw_c_component.arxml', 'autosar_sw_datatype.arxml', ...
                  'autosar_sw_implementation.arxml', 'autosar_sw_interface.arxml'})
```

This call specifies an ARXML file, `ThrottlePositionControlComposition.arxml`, which describes an AUTOSAR software composition and its aggregated AUTOSAR components.

```
ar = arxml.importer('ThrottlePositionControlComposition.arxml')
```

If you enter the `arxml.importer` function call without a terminating semicolon (;), the importer lists the AUTOSAR content of the specified XML file or files. The information includes paths to software components in the AUTOSAR package structure, which you use in the next step.

In this example, the path to software composition `ThrottlePositionControlComposition` is `/Company/Components/ThrottlePositionControlComposition`. The path to software component `Controller` is `/Company/Components/Controller`.

```
ar =
The file "path/ThrottlePositionControlComposition.arxml" contains:
 1 Composition-Software-Component-Type:
   '/Company/Components/ThrottlePositionControlComposition'
 2 Application-Software-Component-Type:
   '/Company/Components/Controller'
   '/Company/Components/ThrottlePositionMonitor'
 3 Sensor-Actuator-Software-Component-Type:
   '/Company/Components/AccelerationPedalPositionSensor'
   '/Company/Components/ThrottlePositionActuator'
   '/Company/Components/ThrottlePositionSensor'
```

Import Software Component and Create Model

To import a parsed atomic software component into a Simulink model, call the `createComponentAsModel` function. Specify the ARXML importer object and the component to create as a model. The parsed ARXML files must specify all dependencies for the component.

The following example creates a Simulink representation of an AUTOSAR atomic software component.


```

ar = arxml.importer('ThrottlePositionControlComposition.arxml');
names = getComponentNames(ar)

names = 5x1 cell
    {'/Company/Components/Controller'          }
    {'/Company/Components/ThrottlePositionMonitor' }
    {'/Company/Components/AccelerationPedalPositionSensor' }
    {'/Company/Components/ThrottlePositionActuator' }
    {'/Company/Components/ThrottlePositionSensor' }

createComponentAsModel(ar, '/Company/Components/Controller', ...
    'ModelPeriodicRunnablesAs', 'AtomicSubsystem');

```

The `'ModelPeriodicRunnablesAs'` argument controls whether the importer models AUTOSAR periodic runnables as atomic subsystems with periodic rates (the default) or function-call subsystems with periodic rates. Specify `AtomicSubsystem` unless your design requires use of function-call subsystems. For more information, see “Import AUTOSAR Software Component with Multiple Runnables” on page 3-18.

To import Simulink data objects for AUTOSAR data into a Simulink data dictionary, you can set the `'DataDictionary'` argument on the model creation. If the specified dictionary does not already exist, the importer creates it.

To explicitly designate an AUTOSAR runnable as the initialization runnable in a component, use the `'InitializationRunnable'` argument on the model creation.

For more information, see the `createComponentAsModel` reference page and the example “Import AUTOSAR Component to Simulink” on page 3-19.

Import Software Composition and Create Models

To import a parsed atomic software composition into a Simulink model, call the `createCompositionAsModel` function. Specify the ARXML importer object and the composition to create as a model. The parsed ARXML files must specify all dependencies for the composition.

The following example creates a Simulink representation of an AUTOSAR software composition.

```

ar = arxml.importer('ThrottlePositionControlComposition.arxml');
names = getComponentNames(ar, 'Composition')

names = 1x1 cell array
    {'/Company/Components/ThrottlePositionControlComposition'}

createCompositionAsModel(ar, '/Company/Components/ThrottlePositionControlComposition');

Creating model 'ThrottlePositionSensor' for component 1 of 5:
  /Company/Components/ThrottlePositionSensor
Creating model 'ThrottlePositionMonitor' for component 2 of 5:
  /Company/Components/ThrottlePositionMonitor
Creating model 'Controller' for component 3 of 5:
  /Company/Components/Controller
Creating model 'AccelerationPedalPositionSensor' for component 4 of 5:
  /Company/Components/AccelerationPedalPositionSensor
Creating model 'ThrottlePositionActuator' for component 5 of 5:
  /Company/Components/ThrottlePositionActuator
Creating model 'ThrottlePositionControlComposition' for composition 1 of 1:
  /Company/Components/ThrottlePositionControlComposition

```

To include existing Simulink atomic software component models in the composition model, use the `'ComponentModels'` argument on the composition model creation.

For more information, see the `createCompositionAsModel` reference page and the example “Import AUTOSAR Composition to Simulink” on page 7-2.

For compositions containing more than 20 software components, sharing AUTOSAR properties among components can significantly improve performance and reduce duplication for composition

workflows. To configure a composition import to store AUTOSAR properties for component sharing, use the 'DataDictionary' and 'ShareAUTOSARProperties' arguments. For more information, see the `createCompositionAsModel` reference page and the example “Import AUTOSAR Composition and Share AUTOSAR Dictionary”.

Import Component or Composition External Updates Into Model

After you import a parsed atomic software component or composition into a Simulink model, the ARXML description of the component or composition might continue to evolve in a different AUTOSAR authoring environment. To update your AUTOSAR component or composition model with external changes, call the `updateModel` function. Specify an ARXML importer object representing the ARXML changes and the existing AUTOSAR model to update.

The following example updates an existing AUTOSAR component model named `Controller` with changes from the file `ThrottlePositionControlComposition_updated.arxml`.

```
% Create and open AUTOSAR controller component model
ar = arxml.importer('ThrottlePositionControlComposition.arxml');
createComponentAsModel(ar, '/Company/Components/Controller', ...
    'ModelPeriodicRunnablesAs', 'AtomicSubsystem');

% Update AUTOSAR controller component model (model must be open)
ar2 = arxml.importer('ThrottlePositionControlComposition_updated.arxml');
updateModel(ar2, 'Controller');
```

For more information, see the `updateModel` reference page, “Import AUTOSAR Software Component Updates” on page 3-25, and the update section of the example “Import AUTOSAR Component to Simulink” on page 3-19.

Import Shared Element Packages into Component Model

After you create an AUTOSAR software component model, either by starting in Simulink or importing ARXML component descriptions, you can update the AUTOSAR properties of the component model with predefined elements and properties that are shared among components. To update the component AUTOSAR properties with packages of shared element definitions, call the `updateAUTOSARProperties` function. Specify an ARXML importer object representing the ARXML shared element definitions and the existing AUTOSAR model to update.

The following example updates an AUTOSAR component model with element definitions from the file `SwAddrMethods.arxml`.

```
modelName = 'autosar_swc';
openExample(modelName);
ar = arxml.importer('SwAddrMethods.arxml');
updateAUTOSARProperties(ar, modelName);
```

For more information, see the `updateAUTOSARProperties` reference page, “Import and Reference Shared AUTOSAR Element Definitions” on page 3-29, and the example “Import AUTOSAR Package into Component Model” on page 3-31.

See Also

`arxml.importer`

Related Examples

- “Import AUTOSAR Component to Simulink” on page 3-19

- “Import AUTOSAR Composition to Simulink” on page 7-2
- “Import AUTOSAR Software Component Updates” on page 3-25
- “Import and Reference Shared AUTOSAR Element Definitions” on page 3-29
- “Import AUTOSAR Package into Component Model” on page 3-31
- “Configure AUTOSAR XML Options” on page 4-43
- “Round-Trip Preservation of AUTOSAR XML File Structure and Element Information” on page 3-37

More About

- “AUTOSAR ARXML Importer” on page 3-35
- “Workflows for AUTOSAR” on page 1-13

Import AUTOSAR Software Component with Multiple Runnables

The AUTOSAR ARXML importer functions `createComponentAsModel` and `createCompositionAsModel` and the AUTOSAR architecture function `importFromARXML` can import AUTOSAR software components with multiple runnable entities into a new Simulink model. Use the `'ModelPeriodicRunnablesAs'` argument on the model creation to specify whether the importer models AUTOSAR periodic runnables as atomic subsystems with periodic rates (the default) or function-call subsystems with periodic rates.

If you set `'ModelPeriodicRunnablesAs'` to the default value, `'AtomicSubsystem'`, the importer creates rate-based models. If the ARXML code contains periodic runnables, the importer adds rate-based model content, including atomic subsystems and data transfer lines with rate transitions, and maps them to corresponding periodic runnables and IRVs imported from the AUTOSAR software component.

If you set `'ModelPeriodicRunnablesAs'` to `'FunctionCallSubsystem'`, the importer creates function-call-based models. The importer adds function-call subsystem or function blocks and signal lines and maps them to corresponding runnables and IRVs imported from the AUTOSAR software component.

Set `'ModelPeriodicRunnablesAs'` to `'AtomicSubsystem'` unless your design requires use of function-call subsystems. The following call directs the importer to import a multi-runnable AUTOSAR software component from an ARXML file and map it into a new rate-based model.

```
openExample('autosarblockset/ImportAUTOSARComponentToSimulinkExample',...  
'supportingfile','ThrottlePositionControlComposition.arxml');  
ar = arxml.importer('ThrottlePositionControlComposition.arxml')  
createComponentAsModel(ar,'/Company/Components/Controller',...  
'ModelPeriodicRunnablesAs','AtomicSubsystem')
```

For more information, see “Model AUTOSAR Software Components” on page 2-3.

See Also

`createComponentAsModel` | `createCompositionAsModel` | `importFromARXML`

Related Examples

- “Import AUTOSAR Component to Simulink” on page 3-19
- “Import AUTOSAR Composition to Simulink” on page 7-2
- “Import AUTOSAR Composition into Architecture Model” on page 8-63

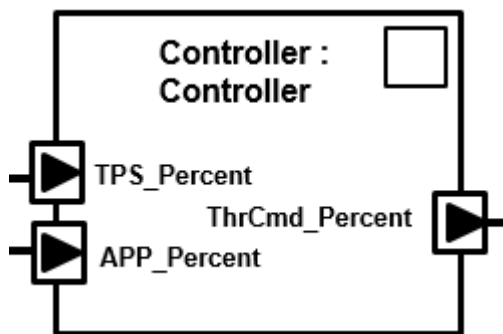
Import AUTOSAR Component to Simulink

Create Simulink® model from XML description of AUTOSAR software component.

Import AUTOSAR Component from ARXML File to Simulink

Here is an AUTOSAR application software component that implements a controller in an automotive throttle position control system. The controller component takes input values from an accelerator pedal position (APP) sensor and a throttle position sensor (TPS). The controller translates the values into input values for a throttle actuator.

The component was created in an AUTOSAR authoring tool and exported to the file `ThrottlePositionControlComposition.arxml`.



Use the MATLAB function `createComponentAsModel` to import the AUTOSAR XML (ARXML) description and create an initial Simulink representation of the AUTOSAR component. First, parse the ARXML description file and list the components it contains.

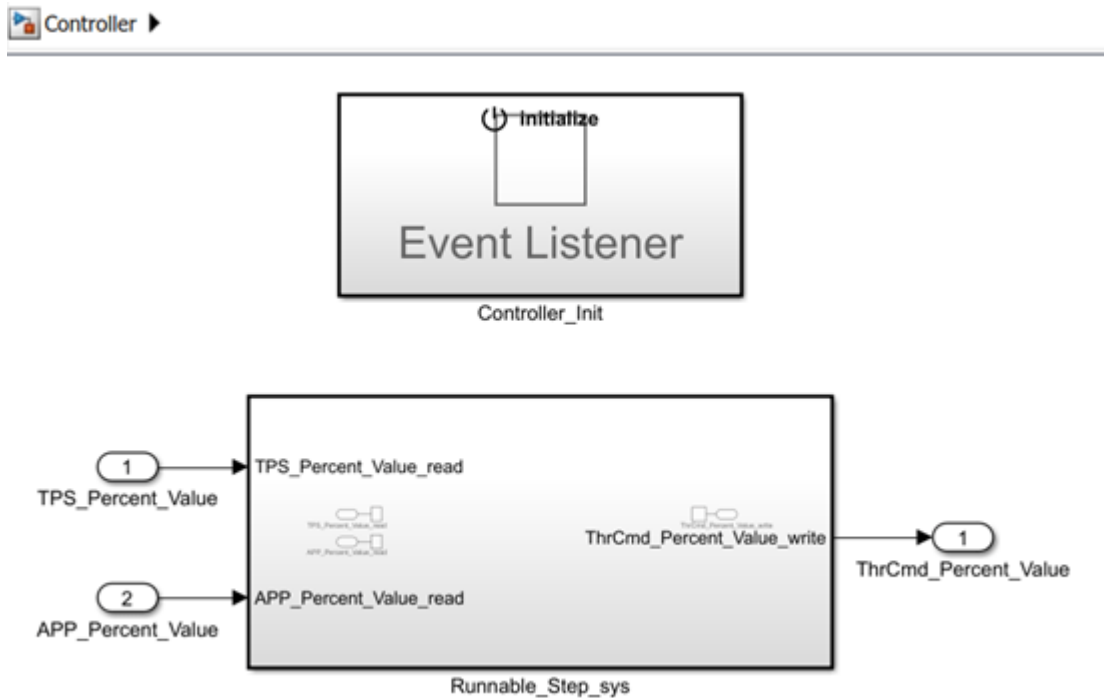
```
ar = arxml.importer('ThrottlePositionControlComposition.arxml');
names = getComponentNames(ar)

names = 5x1 cell
    {'/Company/Components/Controller'           }
    {'/Company/Components/ThrottlePositionMonitor' }
    {'/Company/Components/AccelerationPedalPositionSensor'}
    {'/Company/Components/ThrottlePositionActuator' }
    {'/Company/Components/ThrottlePositionSensor' }
```

For the `Controller` software component, use `createComponentAsModel` to create a Simulink representation.

```
createComponentAsModel(ar, '/Company/Components/Controller', ...
    'ModelPeriodicRunnablesAs', 'AtomicSubsystem');
```

The function call creates a component model that represents an AUTOSAR application software component. An atomic subsystem represents an AUTOSAR periodic runnable, and an Initialize Function block represents an AUTOSAR initialize runnable. Simulink inports and outports represent AUTOSAR ports.



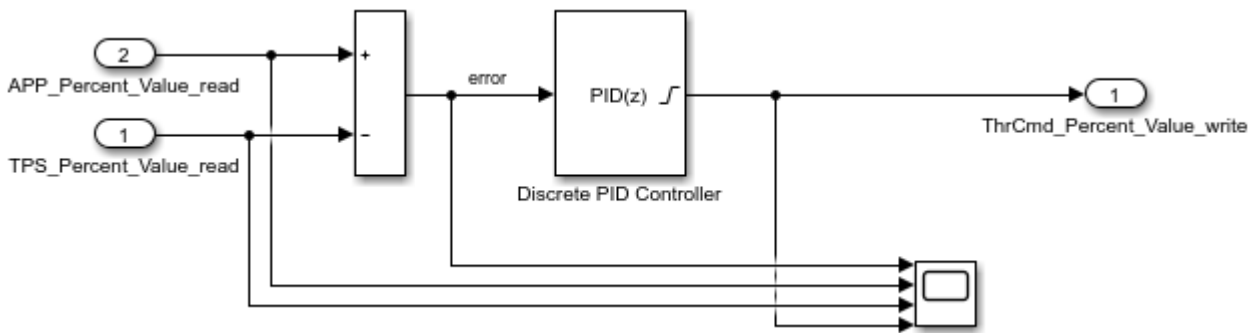
Develop AUTOSAR Component Algorithm, Simulate, and Generate Code

After creating an initial Simulink representation of the AUTOSAR component, you develop the component. You refine the AUTOSAR configuration and create algorithmic model content.

For example, the **Runnable_Step_sys** subsystem in the **Controller** component model contains an initial stub implementation of the controller behavior.



Here is a possible implementation of the throttle position controller behavior. (To explore this implementation, see the model `autosar_sw_controller`, which is provided with the example “Design and Simulate AUTOSAR Components and Generate Code” on page 4-77.) The component takes as inputs an APP sensor percent value from a pedal position sensor and a TPS percent value from a throttle position sensor. Based on these values, the controller calculates the *error*. The error is the difference between where the operator wants the throttle, based on the pedal sensor, and the current throttle position. In this implementation, a Discrete PID Controller block uses the error value to calculate a throttle command percent value to provide to a throttle actuator. A scope displays the error value and the Discrete PID Controller block output value over time.



As you develop AUTOSAR components, you can:

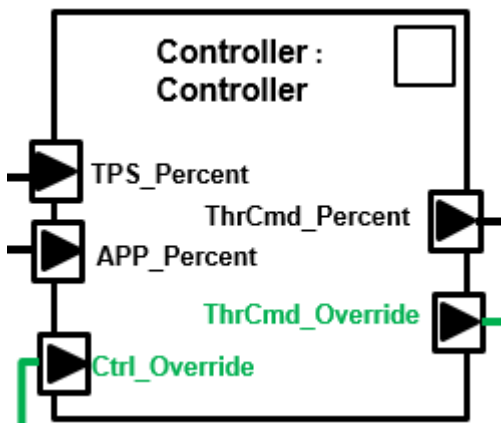
- Simulate the component model individually or in a containing composition or test harness.
- Generate ARXML component description files and algorithmic C code for testing in Simulink or integration into an AUTOSAR run-time environment. (AUTOSAR code generation requires Simulink Coder and Embedded Coder.)

For more information on developing, simulating, and building AUTOSAR components, see example “Design and Simulate AUTOSAR Components and Generate Code” on page 4-77.

Update AUTOSAR Component Model with Architectural Changes from Authoring Tool

Suppose that, after you imported the AUTOSAR software component into Simulink and began developing algorithms, architectural changes were made to the component in the AUTOSAR authoring tool.

Here is the revised component. The changes add a control override receive port and a throttle command override provide port. In the AUTOSAR authoring tool, the revised component is exported to the file `ThrottlePositionControlComposition_updated.arxml`.



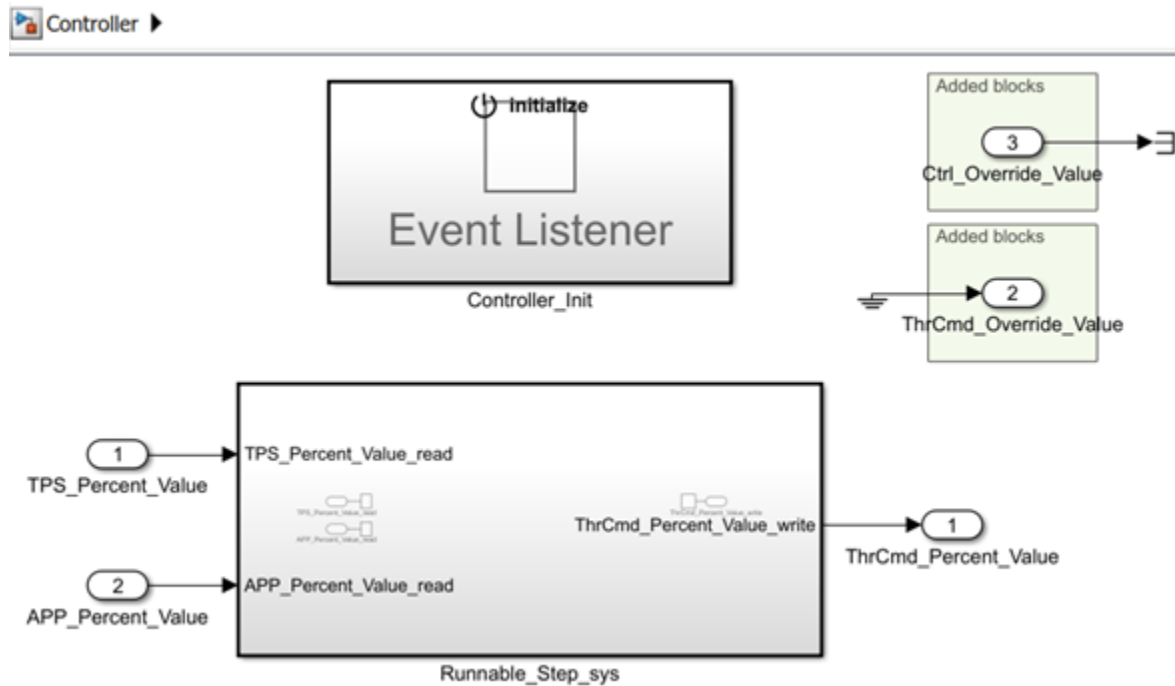
Use the MATLAB function `updateModel` to import the architectural revisions from the ARXML file. The function updates the AUTOSAR component model with the changes and reports the results.

```
ar2 = arxml.importer('ThrottlePositionControlComposition_updated.arxml');
updateModel(ar2, 'Controller');
```

```

### Updating model Controller
### Saving original model as Controller_backup.slx
### Creating HTML report Controller_update_report.html
    
```

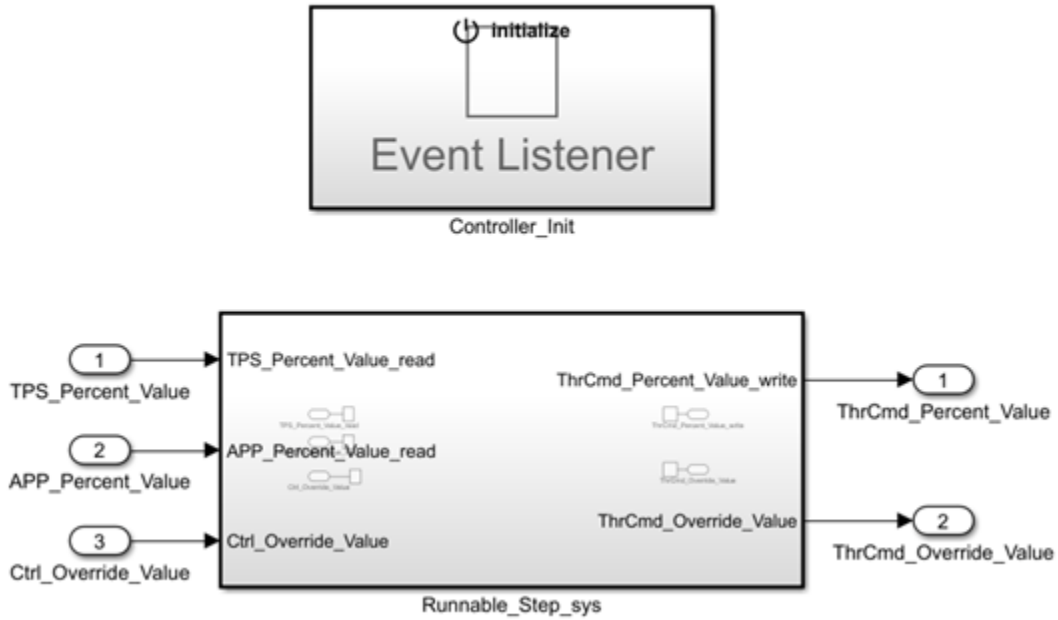
After the update, in the component model, highlighting indicates where changes occurred.



The function also generates and displays an HTML AUTOSAR update report. The report lists changes that the update made to Simulink and AUTOSAR elements in the component model. In the report, you can click hyperlinks to navigate from change descriptions to model changes.

Connect the added blocks, update the inputs and outputs inside the subsystem, and update the model diagram. For example:

Controller ▶



Related Links

- [createComponentAsModel](#)
- [updateModel](#)
- [“Component Creation”](#)
- [“Import AUTOSAR Software Component Updates”](#) on page 3-25
- [“Design and Simulate AUTOSAR Components and Generate Code”](#) on page 4-77

Import AUTOSAR Software Composition with Atomic Software Components (Classic Platform)

You can import an AUTOSAR software composition from ARXML files into a new Simulink model. AUTOSAR compositions aggregate AUTOSAR software components and potentially other compositions. Use the `arxml.importer.createCompositionAsModel` to import a composition.

The following types of AUTOSAR atomic software components, if found in the ARXML description of a composition, are imported and represented as component models.

- Application component
- Sensor-actuator component
- Complex device driver component
- ECU abstraction component
- Service proxy component

Application and sensor-actuator components are frequently imported, created, and modeled in Simulink. For complex device driver, ECU abstraction, or service proxy components that you import from compositions, you can model only the application side of their behavior in Simulink. For example, a complex device driver component can access Runtime Environment (RTE) device driver interfaces as an application-level component. But you cannot model the corresponding Basic Software (BSW) device drivers in Simulink.

See Also

`createCompositionAsModel`

Related Examples

- “Import AUTOSAR Composition to Simulink” on page 7-2

Import AUTOSAR Software Component Updates

After you create a Simulink model that represents an AUTOSAR software component or composition, the ARXML description of the component or composition can change independently. Using `arxml.importer.updateModel`, you can import the modified ARXML description and update the model to reflect the changes. The update generates an HTML report that details automatic updates applied to the model, and additional manual changes that you must perform.

In this section...

“Update Model with AUTOSAR Software Component Changes” on page 3-25

“AUTOSAR Update Report Section Examples” on page 3-26

Update Model with AUTOSAR Software Component Changes

To update a model with AUTOSAR software component changes described in ARXML files:

- 1 Open a model for which you previously imported or exported ARXML files. This example uses the example ARXML file `ThrottlePositionControlComposition.arxml` to create a Controller model.

```
openExample('autosarblockset/ImportAUTOSARComponentToSimulinkExample',...
'supportingfile','ThrottlePositionControlComposition.arxml');
% Create and open AUTOSAR controller component model
ar = arxml.importer('ThrottlePositionControlComposition.arxml');
createComponentAsModel(ar,'/Company/Components/Controller',...
'ModelPeriodicRunnablesAs','AtomicSubsystem');
```

- 2 Issue MATLAB commands to import ARXML descriptions into the model and update the model with changes.

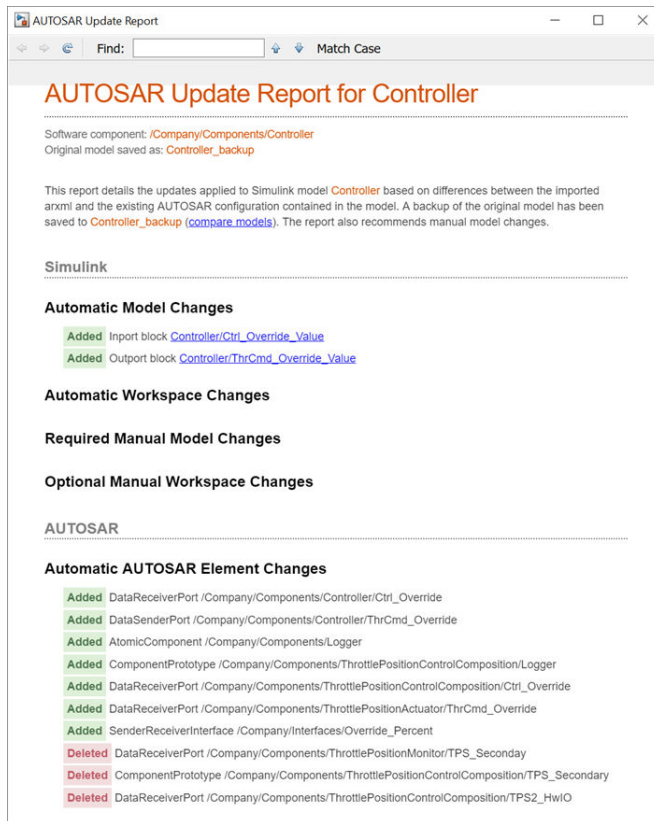
Note The imported ARXML descriptions must contain the AUTOSAR software component mapped by the model.

For example, the following commands update the Controller model with changes from ARXML file `ThrottlePositionControlComposition_updated.arxml`.

```
openExample('autosarblockset/ImportAUTOSARComponentToSimulinkExample',...
'supportingfile','ThrottlePositionControlComposition_updated.arxml');
% Update AUTOSAR controller component model
ar2 = arxml.importer('ThrottlePositionControlComposition_updated.arxml');
updateModel(ar2,'Controller');

### Updating model Controller
### Saving original model as Controller_backup.slx
### Creating HTML report Controller_update_report.html
```

The AUTOSAR Update Report opens.



- 3 Examine the report.
 - a Verify that the ARXML importer has updated the model content and configuration based on the ARXML changes.
 - b Optionally, click **compare models** to compare the original model with the updated model. Tabular and graphical views of the differences open. You can click a changed element in the tabular view to navigate to a graphical view of the change.
 - c Optionally, use the **Find** field to search for a term. You can quickly navigate to specific elements or other strings of interest.
- 4 If the report lists required manual model changes, such as deleting a Simulink block, perform the required changes.

If you make a required change to the model, further configuration could be required to pass validation. To see if more manual model changes are required, repeat the update procedure, rerunning the `updateModel` function with the same ARXML files.

For live-script update examples, see “Import AUTOSAR Component to Simulink” on page 3-19 and “Import AUTOSAR Composition to Simulink” on page 7-2.

AUTOSAR Update Report Section Examples

An ARXML update operation generates an AUTOSAR Update Report in HTML format. The report displays change information in sections:

- “Automatic Model Changes” on page 3-27

- “Automatic Workspace Changes” on page 3-27
- “Required Manual Model Changes” on page 3-27
- “Automatic AUTOSAR Element Changes” on page 3-28

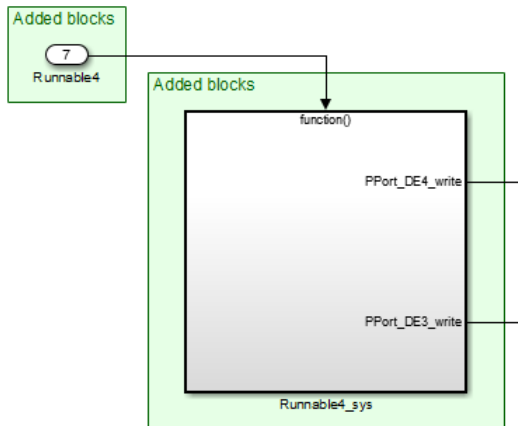
Automatic Model Changes

The AUTOSAR Update Report section **Automatic Model Changes** lists Simulink block additions, block property updates, and model parameter updates made by the importer. For example:

Automatic Model Changes

Updated	OutDataTypeStr of Output mySWC/Runnable3/TicToc_iv from Inherit: auto to int8
Updated	PortDimensions of Output mySWC/Runnable3/TicToc_iv from -1 to 1
Updated	SignalType of Output mySWC/Runnable3/TicToc_iv from auto to real
Updated	SamplingMode of Output mySWC/Runnable3/TicToc_iv from auto to Sample based
Added	SubSystem block mySWC/Runnable4_sys
Added	TriggerPort block mySWC/Runnable4_sys/function
Added	Inport block mySWC/Runnable4
Added	Outport block mySWC/Runnable4_sys/PPort_DE4_write
Added	Outport block mySWC/Runnable4_sys/PPort_DE3_write

In the updated model, green highlighting identifies added blocks.



Automatic Workspace Changes

The AUTOSAR Update Report section **Automatic Workspace Changes** lists Simulink data object additions and property updates made by the importer. For example:

Automatic Workspace Changes

Added	AUTOSAR.Parameter INC2
Updated	Value of AUTOSAR.Parameter INC from 1 to 2
Updated	Data Type of AUTOSAR.Parameter INC from UInt8 to uint8

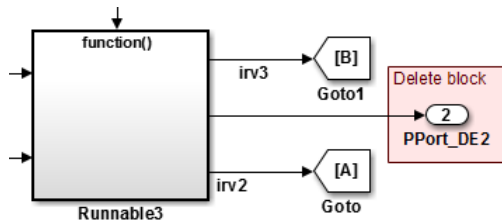
Required Manual Model Changes

The AUTOSAR Update Report section **Required Manual Model Changes** lists model changes, such as block deletions, that are required. For example:

Required Manual Model Changes

Delete Port `mySWC/PPort_DE2` from mySWC

In the updated model, red highlighting identifies the block to delete.



Automatic AUTOSAR Element Changes

The AUTOSAR Update Report section **Automatic AUTOSAR Element Changes** lists AUTOSAR element additions and property updates made by the importer. For example:

Automatic AUTOSAR Element Changes

Added	Runnable /pkg/swc/ASWC/IB/Runnable4
Added	TimingEvent /pkg/swc/ASWC/IB/Event
Added	InvData /pkg/swc/ASWC/IB/IRV5
Added	InvData /pkg/swc/ASWC/IB/IRV6
Added	ConstantSpecification /pkg/dt/Ground/INC2
Added	DataConstr /pkg/dt/DataConstrs/UInt8
Added	SwBaseType /pkg/dt/SwBaseTypes/uint8
Updated	SwCalibrationAccess of ParameterData /pkg/swc/ASWC/IB/INC from ReadOnly to ReadWrite
Updated	Value of LiteralReal /pkg/dt/Ground/INC/INC from 1.0 to 2.0
Added	DataConstr reference /pkg/dt/DataConstrs/UInt8 to /pkg/dt/UInt8
Added	SwBaseType reference /pkg/dt/SwBaseTypes/uint8 to /pkg/dt/UInt8
Updated	InternalBehaviorQualifiedName of AUTOSAR XmlOptions from /pkg/swc/ASWC_ib to /pkg/swc/IB
Updated	InternalDataConstraintExport of AUTOSAR XmlOptions from false to true

See Also

updateModel

Related Examples

- “Import AUTOSAR XML Descriptions Into Simulink” on page 3-13
- “Import AUTOSAR Component to Simulink” on page 3-19
- “Import AUTOSAR Composition to Simulink” on page 7-2
- “Configure AUTOSAR Code Generation” on page 5-7

More About

- “Workflows for AUTOSAR” on page 1-13

Import and Reference Shared AUTOSAR Element Definitions

When developing an AUTOSAR software component in Simulink, you can import AUTOSAR element definitions that are common to many components. For example, multiple product lines and teams might share elements such as interfaces, data types, and software address methods (SwAddrMethods). Benefits of sharing AUTOSAR element definitions include lower risk of definition conflicts and easier code integration.

After you create an AUTOSAR component model, you import the definitions from AUTOSAR XML (ARXML) files that contain packages of shared AUTOSAR elements. By default, the imported definitions are read-only, which prevents changes, but you can also import them as read/write. You can then reference the imported elements in your component model.

When you import an element definition, its dependencies are also imported. For example, importing a CompuMethod definition also imports Unit and PhysicalDimension definitions. Importing an ImplementationDataType also imports a SwBaseType definition.

If you import AUTOSAR numeric or enumeration data types, you can use the createNumericType and createEnumeration functions to create corresponding Simulink data type objects.

When you build the model, exported ARXML code contains references to the shared elements. Their definitions remain in the element description ARXML files from which you imported them. The element description files are exported with their names, file structure, and content preserved.

To set up and reuse AUTOSAR element definitions:

- 1 Create one or more ARXML files containing definitions of AUTOSAR elements for components to share. Elements that are supported for reference use in Simulink include:
 - CompuMethod, Unit, and Dimension
 - ImplementationDataType and SwBaseType
 - ApplicationDataType
 - SwSystemConst, SwSystemConstValueSet, and PredefinedVariant
 - SwRecordLayout
 - SwAddrMethod
 - ClientServerInterface, SenderReceiverInterface, ModeSwitchInterface, NvDataInterface, ParameterInterface, and TriggerInterface.
- 2 For each component model that shares a set of definitions, use the `arxml.importer` function `updateAUTOSARProperties` to add the element definitions to the model. This example shows how to import definitions from the example shared descriptions file `SwAddrMethods.arxml` into the example model `autosar_swc`.

```
openExample('autosarblockset/ReuseAUTOSARElementsInComponentModelExample');
modelName = 'autosar_swc';
open_system(modelName);
ar = arxml.importer('SwAddrMethods.arxml');
updateAUTOSARProperties(ar,modelName);
```

Optionally, using property-value pairs, you can specify subsets of elements to import. For more information, see `updateAUTOSARProperties`.

The importer generates an HTML report that details the updates applied to the model.

AUTOSAR

Automatic AUTOSAR Element Changes

Added	Package /Company/Powertrain/SwAddrMethods
Added	SwAddrMethod /Company/Powertrain/SwAddrMethods/CODE
Added	SwAddrMethod /Company/Powertrain/SwAddrMethods/CALIB
Added	SwAddrMethod /Company/Powertrain/SwAddrMethods/CONST
Added	SwAddrMethod /Company/Powertrain/SwAddrMethods/VAR_NO_INIT
Added	SwAddrMethod /Company/Powertrain/SwAddrMethods/VAR_INIT
Added	SwAddrMethod /Company/Powertrain/SwAddrMethods/VAR_POWER_ON_CLEARED
Added	SwAddrMethod /Company/Powertrain/SwAddrMethods/VAR_CLEARED

- 3 Your model can reference the imported elements in various ways. For example, you can select imported SwAddrMethod values for AUTOSAR data to group the data for calibration and measurement. See the example “Import AUTOSAR Package into Component Model” on page 3-31.
- 4 Generate model code. The exported ARXML code contains references to the imported elements. The element description files from which you imported definitions are exported with their names, file structure, and content preserved.

[-] Interface files

[SwAddrMethods.arxml](#)
[autosar_swk_component.arxml](#)
[autosar_swk_datatype.arxml](#)
[autosar_swk_implementation.arxml](#)
[autosar_swk_interface.arxml](#)
[autosar_swk_timing.arxml](#)

See Also

updateAUTOSARProperties

Related Examples

- “Import AUTOSAR Package into Component Model” on page 3-31
- “Import AUTOSAR XML Descriptions Into Simulink” on page 3-13

Import AUTOSAR Package into Component Model

Import and reference shared ARXML element definitions.

Add AUTOSAR Element Definitions to Model

When developing an AUTOSAR software component in Simulink, you can import AUTOSAR element definitions that are common to many components. After you create an AUTOSAR component model, you import the definitions from AUTOSAR XML (ARXML) files that contain packages of shared AUTOSAR elements. To help implement the component behavior, you want to reference predefined elements such as interfaces, data types, and software address methods (SwAddrMethods).

Suppose that you are developing an AUTOSAR software component model. You want to import predefined SwAddrMethod elements that are shared by multiple product lines and teams. This example uses AUTOSAR importer function `updateAUTOSARProperties` to import definitions from shared descriptions file `SwAddrMethods.arxml` into example model `autosar_swc`.

```
modelName = 'autosar_swc';
open_system(modelName);
ar = arxml.importer('SwAddrMethods.arxml');
updateAUTOSARProperties(ar,modelName);

### Updating model autosar_swc
### Saving original model as autosar_swc_backup.slx
### Creating HTML report autosar_swc_update_report.html
```

The function copies the contents of the specified ARXML files to the AUTOSAR Dictionary of the specified model and generates an HTML report listing the element additions.

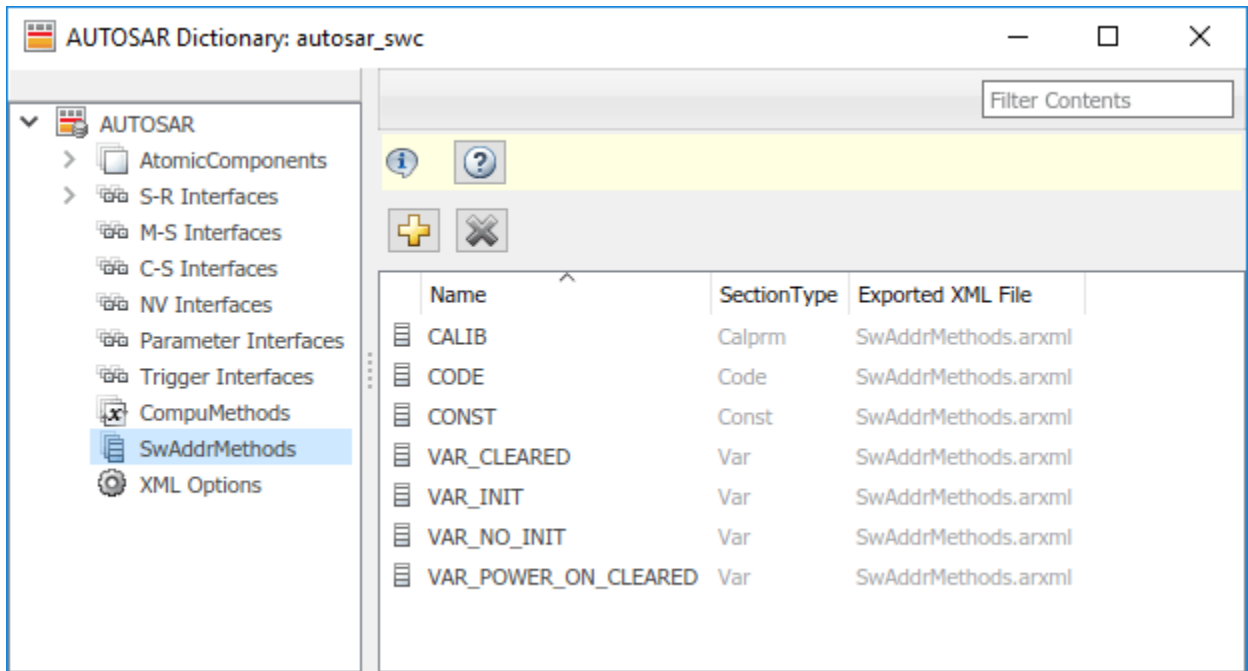
AUTOSAR

Automatic AUTOSAR Element Changes

Added	Package /Company/Powertrain/SwAddrMethods
Added	SwAddrMethod /Company/Powertrain/SwAddrMethods/CALIB
Added	SwAddrMethod /Company/Powertrain/SwAddrMethods/VAR_INIT
Added	SwAddrMethod /Company/Powertrain/SwAddrMethods/CONST
Added	SwAddrMethod /Company/Powertrain/SwAddrMethods/VAR_NO_INIT
Added	SwAddrMethod /Company/Powertrain/SwAddrMethods/VAR_POWER_ON_CLEARED
Added	SwAddrMethod /Company/Powertrain/SwAddrMethods/CODE
Added	SwAddrMethod /Company/Powertrain/SwAddrMethods/VAR_CLEARED

You can view the added elements as elements in the AUTOSAR Dictionary. By default, the elements are imported as read-only.

```
set_param(modelName,'SimulationCommand','update'); % Update diagram
autosar_ui_launch(modelName); % Open AUTOSAR Dictionary
```



Reference and Configure Imported AUTOSAR Elements

After importing the AUTOSAR elements to the software component model, you can reference and configure them in the same manner as any AUTOSAR Dictionary element. For example, use the AUTOSAR code perspective to apply imported SwAddrMethod definition CODE to a model entry-point function.

```
% Map step runnable function to SwAddrMethod CODE
slMap = autosar.api.getSimulinkMapping(modelName);
mapFunction(slMap, 'Periodic:D1', 'Runnable_1s', 'SwAddrMethod', 'CODE');
```

The screenshot displays the Simulink environment for an AUTOSAR Atomic Software Component (ASWC). The main workspace shows a block diagram titled "AUTOSAR Atomic Software Component (ASWC) with Periodic Runnables Modeled Using Multiple Rates". The diagram includes an "initialize" block, a "Runnable_Initialize" block, and two periodic runnables: "Runnable_1s" and "Runnable_2s". "Runnable_1s" contains a "SS1" block, and "Runnable_2s" contains an "Integrator" block. Data flows are indicated by red and green arrows between these components.

Below the workspace is the "Code Mappings - AUTOSAR SW Component" window. It features tabs for "Functions", "Imports", "Outputs", "Parameters", "Data Stores", "Signals/States", "Data Transfers", and "Function Callers". The "Functions" tab is active, showing a table with the following data:

Source	Runnable
fx Periodic:D1 [Sample Time: 1s]	Runnable_1s
fx Periodic:D2 [Sample Time: 2s]	Runnable_2s
fx Initialize	Runnable_Init

A context menu is open over the table, showing options for "SwAddrMethod" (set to "CODE") and "Internal Data SwAddrMethod" (set to "<None>").

Generate AUTOSAR C Code and XML Descriptions (Embedded Coder)

If you have Simulink Coder and Embedded Coder software, you can generate AUTOSAR-compliant C code and export ARXML descriptions from the model. To build the model, enter the command `slbuild(modelName);`.

Building the model generates an HTML code generation report. The C code contains a software address method CODE section.

```
/* SwAddrMethod CODE for Runnable */
#define ASWC_START_SEC_CODE
#include "ASWC_MemMap.h"

void Runnable_1s(void)           /* Sample time: [1.0s, 0.0s] */
{
}

#define ASWC_STOP_SEC_CODE
#include "ASWC_MemMap.h"
```

The ARXML descriptions define and reference SwAddrMethod CODE.

```
<RUNNABLE-ENTITY UUID="aee47805-d92a-5904-e77f-21f87869fd95">
  <SHORT-NAME>Runnable_1s</SHORT-NAME>
  <MINIMUM-START-INTERVAL>0</MINIMUM-START-INTERVAL>
  <SW-ADDR-METHOD-REF DEST="SW-ADDR-METHOD">/Company/Powertrain/SwAddrMethods/CODE</SW-ADDR-METHOD-REF>
  <CAN-BE-INVOKED-CONCURRENTLY>false</CAN-BE-INVOKED-CONCURRENTLY>
</RUNNABLE-ENTITY>
```

Export preserves the file structure and content of the shared descriptions file SwAddrMethods.arxml from which you added SwAddrMethod definitions.

[-] Interface files

[SwAddrMethods.arxml](#)
[autosar_swc_component.arxml](#)
[autosar_swc_datatype.arxml](#)
[autosar_swc_implementation.arxml](#)
[autosar_swc_interface.arxml](#)
[autosar_swc_timing.arxml](#)

Related Links

- [updateAUTOSARProperties](#)
- “Import and Reference Shared AUTOSAR Element Definitions” on page 3-29
- “Import AUTOSAR XML Descriptions Into Simulink” on page 3-13

AUTOSAR ARXML Importer

The AUTOSAR ARXML importer imports AUTOSAR description files produced by an AUTOSAR authoring tool (AAT) into a Simulink model. The importer first parses ARXML code that describes AUTOSAR software components, compositions, or packages of predefined elements for component sharing. Then, based on commands that you issue, the importer imports a subset of the elements and objects in the ARXML description into Simulink. The subset consists of AUTOSAR elements relevant for Simulink model-based design of an automotive application. For example, for an imported component, the subset includes AUTOSAR ports, interfaces, data types, aspects of internal behavior, and packages.

For imported software components, the importer creates an initial Simulink representation of each component, with an initial, default mapping of Simulink model elements to AUTOSAR component elements. The initial representation provides a starting point for further AUTOSAR configuration and model-based design.

As part of the import operation, the importer validates the XML in the imported ARXML files. If XML validation fails for a file, the importer displays errors. For example:

```
Error
The IsService attribute is undefined for interface /mtest_pkg/mtest_if/In1
in file hArxmlFileErrorMissingIsService_SR_3p2.arxml:48.
Specify the IsService attribute to be either true or false
```

In this example message, the file name is a hyperlink, and you can click the hyperlink to see the location of the error in the ARXML file.

To help support the round trip of AUTOSAR elements between an AAT and the Simulink model-based design environment, Embedded Coder:

- Preserves imported AUTOSAR XML file structure, elements, and element universal unique identifiers (UUIDs) for ARXML export. For more information, see “Round-Trip Preservation of AUTOSAR XML File Structure and Element Information” on page 3-37.
- Provides the ability to update an AUTOSAR model based on changes found in imported ARXML files. For more information, see “Import AUTOSAR Software Component Updates” on page 3-25.

The AUTOSAR ARXML importer is implemented as an `arxml.importer` object. For a complete list of functions, see the `arxml.importer` object reference page.

See Also

Related Examples

- “Import AUTOSAR XML Descriptions Into Simulink” on page 3-13
- “Import AUTOSAR Component to Simulink” on page 3-19
- “Import AUTOSAR Composition to Simulink” on page 7-2
- “Import AUTOSAR Software Component Updates” on page 3-25
- “Import and Reference Shared AUTOSAR Element Definitions” on page 3-29
- “Import AUTOSAR Package into Component Model” on page 3-31
- “Configure AUTOSAR XML Options” on page 4-43
- “Import AUTOSAR Adaptive Software Descriptions” on page 6-12

- “Import AUTOSAR Adaptive Components to Simulink” on page 6-13
- “Import AUTOSAR Package into Adaptive Component Model” on page 6-17
- “Configure AUTOSAR Adaptive XML Options” on page 6-33

More About

- “Workflows for AUTOSAR” on page 1-13
- “Round-Trip Preservation of AUTOSAR XML File Structure and Element Information” on page 3-37

Round-Trip Preservation of AUTOSAR XML File Structure and Element Information

To support the round trip of AUTOSAR elements between an AUTOSAR authoring tool (AAT) and Simulink®, ARXML import preserves imported AUTOSAR XML file structure and content for ARXML export. When you import ARXML files for an AUTOSAR component into Simulink®, the importer preserves:

- AUTOSAR XML file structure. You can compare the ARXML files that you import with the corresponding ARXML files that you export.
- AUTOSAR element information, including properties, references, and packages. The importer preserves relationships between elements.

After import, you can view and configure AUTOSAR software component elements and properties in the AUTOSAR Dictionary. Use the AUTOSAR Dictionary to configure AUTOSAR elements. The properties that you modify are reflected in exported ARXML descriptions and potentially in generated AUTOSAR-compliant C or C++ code. For more information, see “Configure AUTOSAR Elements and Properties” on page 4-8 or “Configure AUTOSAR Adaptive Elements and Properties” on page 6-21.

AUTOSAR elements that you create in Simulink® export to one or more *modelName*.arxml* files, which are separate from the imported XML files. You control the file packaging of new elements by configuring XML options in the AUTOSAR Dictionary. For example, you can set XML option **Exported XML file packaging** to *Single file* or *Modular*. For more information, see “Configure AUTOSAR XML Options” on page 4-43 or “Configure AUTOSAR Adaptive XML Options” on page 6-33.

When you export ARXML files from a Simulink® model, the code generator preserves the imported XML file structure, element information, and UUIDs, while applying your modifications. The exported files include updated versions of the same ARXML files that you imported, and one or more *modelName*.arxml* files, based on whether you set **Exported XML file packaging** to *Single file* or *Modular*. The *modelName*.arxml* files include:

- Implementation descriptions.
- If you added AUTOSAR interface or data-related elements in Simulink®, interface and data descriptions.

For the Adaptive Platform, manifests for AUTOSAR executables and service instances are also included.

Suppose that, in a working folder, you create a Simulink® model named `Controller.slx` from example ARXML file `ThrottlePositionController.arxml`.

```
ar = arxml.importer('ThrottlePositionController.arxml');
createComponentAsModel(ar, '/Company/Components/Controller', ...
    'ModelPeriodicRunnablesAs', 'AtomicSubsystem');
```

In the created model, add an AUTOSAR software address method (SwAddrMethod) named CODE and reference it from an AUTOSAR runnable function.

```
arProps = autosar.api.getAUTOSARProperties('Controller');
addPackageableElement(arProps, 'SwAddrMethod', ...
    '/AUTOSAR_Platform/SwAddrMethods', 'CODE', 'SectionType', 'Code')
```

```
slMap = autosar.api.getSimulinkMapping('Controller');
mapFunction(slMap, 'StepFunction', 'Runnable_Step', 'SwAddrMethod', 'CODE')
```

The function name value 'StepFunction' is obsolete and will be removed in a future release. For valid function name values, use `autosar.api.getSimulinkMapping(modelName).find("Functions")`.
The function name value 'StepFunction' is obsolete and will be removed in a future release. For valid function name values, use `autosar.api.getSimulinkMapping(modelName).find("Functions")`.

Display the SwAddrMethod CODE path and step function mapping information.

```
swAddrMethodPath = find(arProps,[], 'SwAddrMethod', 'PathType', 'FullyQualified', ...
    'SectionType', 'Code')
```

```
swAddrMethodPath = 1x1 cell array
    {'/AUTOSAR_Platform/SwAddrMethods/CODE'}
```

```
[arRunnableName,arRunnableSwAddrMethod] = getFunction(slMap, 'StepFunction')
```

The function name value 'StepFunction' is obsolete and will be removed in a future release. For valid function name values, use `autosar.api.getSimulinkMapping(modelName).find("Functions")`.

```
arRunnableName =
'Runnable_Step'
```

```
arRunnableSwAddrMethod =
'CODE'
```

You can view the modifications in the AUTOSAR Dictionary, **SwAddrMethods** view, and the Code Mappings editor, **Functions** tab.

Build the model, for example, by using the command `slbuild('Controller')`. If the model has **Exported XML file packaging** set to `Modular`, the build exports these ARXML files:

- `ThrottlePositionController.arxml` — Updated version of the ARXML file from which the model was created. To track changes, you can compare earlier versions of an ARXML file with the most recent exported version.
- `Controller_implementation.arxml` — Component implementation information (always generated).
- `Controller_datatype.arxml` — Data-related information that reflects your SwAddrMethod changes to the component model. In the file, AUTOSAR package `/AUTOSAR_Platform/SwAddrMethods` contains SwAddrMethod CODE.

See Also

Related Examples

- “Configure AUTOSAR XML Options” on page 4-43
- “Configure AUTOSAR Adaptive XML Options” on page 6-33

More About

- “Import AUTOSAR XML Descriptions Into Simulink” on page 3-13
- “Configure AUTOSAR Elements and Properties” on page 4-8
- “Configure AUTOSAR Adaptive Elements and Properties” on page 6-21

Limitations and Tips

The following limitations apply to AUTOSAR component creation.

In this section...
“Cannot Save Importer Objects in MAT-Files” on page 3-40
“ApplicationRecordDataType and ImplementationDataType Element Names Must Match” on page 3-40

Cannot Save Importer Objects in MAT-Files

If you try to save an `arxml.importer` object in a MAT-file, you lose the AUTOSAR information. If you reload the MAT-file, then the object is null (`handle = -1`), because of the Java® objects that compose the `arxml.importer` object.

ApplicationRecordDataType and ImplementationDataType Element Names Must Match

The element name of an imported `ApplicationRecordDataType` must match the element name of the corresponding `ImplementationDataType`. For example, if an imported `ApplicationRecordDataType` has element `PVAL_1` and the corresponding `ImplementationDataType` has element `IPVAL_1`, the software flags the mismatch and instructs you to rename the elements to match.

AUTOSAR Component Development

- “AUTOSAR Component Configuration” on page 4-3
- “Configure AUTOSAR Elements and Properties” on page 4-8
- “Map AUTOSAR Elements for Code Generation” on page 4-50
- “Map Calibration Data for Submodels Referenced from AUTOSAR Component Models” on page 4-65
- “Incrementally Update AUTOSAR Mapping After Model Changes” on page 4-74
- “Design and Simulate AUTOSAR Components and Generate Code” on page 4-77
- “Configure AUTOSAR Packages” on page 4-84
- “Configure AUTOSAR Package for Component, Interface, CompuMethod, or SwAddrMethod” on page 4-94
- “Configure AUTOSAR Sender-Receiver Communication” on page 4-96
- “Configure AUTOSAR Queued Sender-Receiver Communication” on page 4-112
- “Configure AUTOSAR Ports By Using Simulink Bus Ports” on page 4-138
- “Configure AUTOSAR Client-Server Communication” on page 4-142
- “Configure AUTOSAR Mode-Switch Communication” on page 4-163
- “Configure AUTOSAR Nonvolatile Data Communication” on page 4-169
- “Configure AUTOSAR Port Parameters for Communication with Parameter Component” on page 4-171
- “Configure Receiver for AUTOSAR External Trigger Event Communication” on page 4-175
- “Configure AUTOSAR Runnables and Events” on page 4-178
- “Configure AUTOSAR Runnable Execution Order” on page 4-181
- “Configure AUTOSAR Initialize, Reset, or Terminate Runnables” on page 4-187
- “Add Top-Level Asynchronous Trigger to Periodic Rate-Based System” on page 4-193
- “Configure AUTOSAR Initialization Runnable (R4.1)” on page 4-196
- “Configure Disabled Mode for AUTOSAR Runnable Event” on page 4-198
- “Configure Internal Data Types for AUTOSAR IncludedDataTypeSets” on page 4-199
- “Configure AUTOSAR Per-Instance Memory” on page 4-201
- “Configure AUTOSAR Static Memory” on page 4-206
- “Configure AUTOSAR Constant Memory” on page 4-210
- “Configure AUTOSAR Shared or Per-Instance Parameters” on page 4-212
- “Configure Variants for AUTOSAR Elements” on page 4-217
- “Configure Variants for AUTOSAR Runnable Implementations” on page 4-220
- “Export Variation Points for AUTOSAR Calibration Data” on page 4-223
- “Configure Dimension Variants for AUTOSAR Array Sizes” on page 4-225
- “Control AUTOSAR Variants with Predefined Value Combinations” on page 4-227
- “Configure Postbuild Variant Conditions for AUTOSAR Software Components” on page 4-229

- “Configure Variant Parameter Values for AUTOSAR Elements” on page 4-232
- “Configure AUTOSAR CompuMethods” on page 4-236
- “Configure AUTOSAR Data Types Export” on page 4-244
- “Automatic AUTOSAR Data Type Generation” on page 4-248
- “Configure Parameters and Signals for AUTOSAR Calibration and Measurement” on page 4-250
- “Configure Subcomponent Data for AUTOSAR Calibration and Measurement” on page 4-255
- “Configure AUTOSAR Data for Calibration and Measurement” on page 4-262
- “Configure Lookup Tables for AUTOSAR Calibration and Measurement” on page 4-273
- “Configure and Map AUTOSAR Component Programmatically” on page 4-293
- “AUTOSAR Property and Map Function Examples” on page 4-299
- “Limitations and Tips” on page 4-321

AUTOSAR Component Configuration

After you create an AUTOSAR software component model in Simulink, use the Code Mappings editor and the AUTOSAR Dictionary to further develop the AUTOSAR component. The Code Mappings editor and the AUTOSAR Dictionary provide mapping and properties views of the component model, which can be used separately and together to configure the AUTOSAR component:

- Code Mappings editor — Using a tabbed table format, displays entry-point functions, inports, outports, and other Simulink elements relevant to your AUTOSAR platform. Use this view to map model elements to AUTOSAR component elements from a Simulink model perspective.
- AUTOSAR Dictionary — Using a tree format, displays a mapped AUTOSAR component and its elements, communication interfaces, computation methods, software address methods, and XML options. Use this view to configure AUTOSAR elements from an AUTOSAR component perspective.

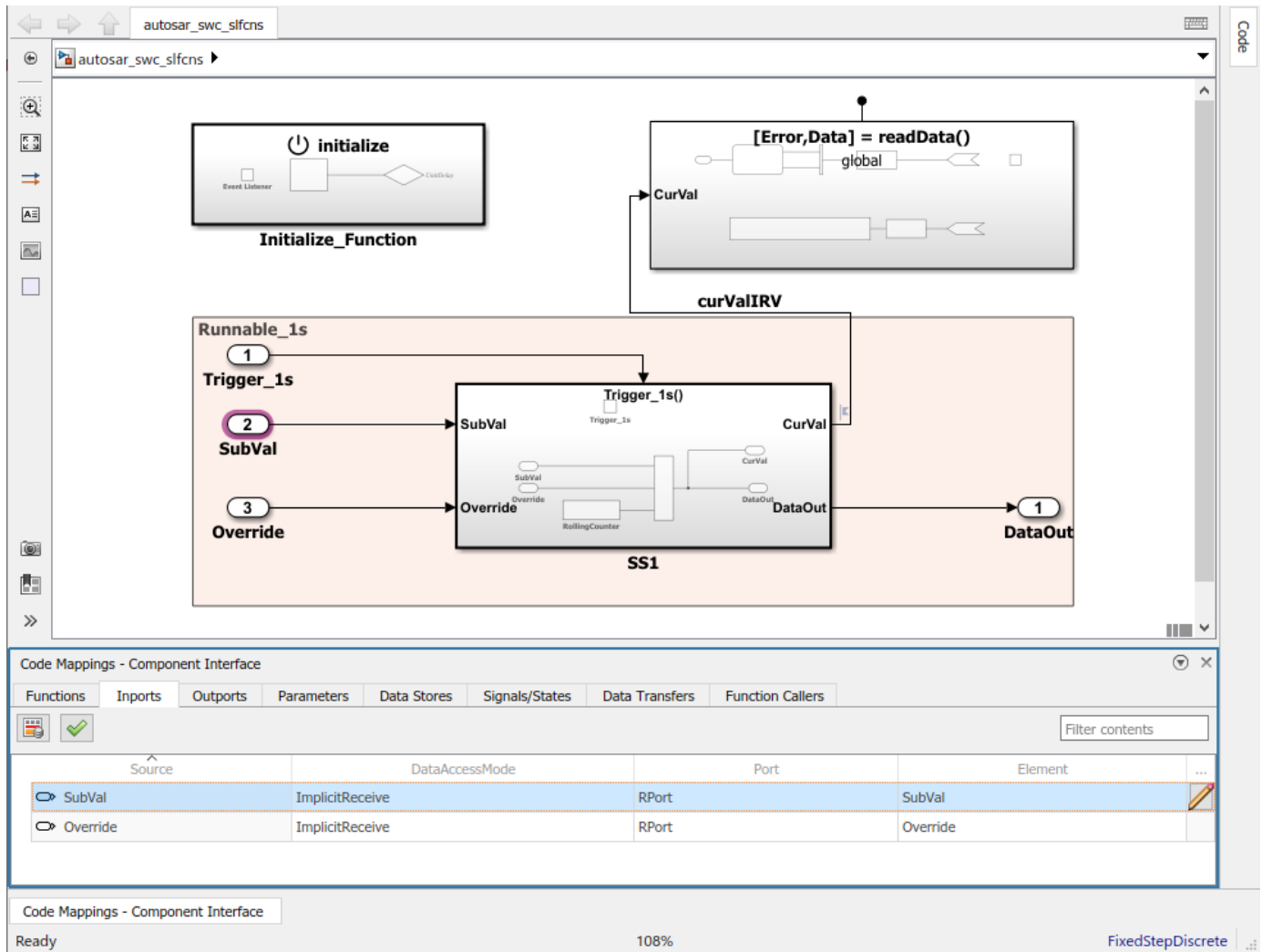
Alternatively, you can configure AUTOSAR mapping and properties programmatically. See “Configure and Map AUTOSAR Component Programmatically” on page 4-293.

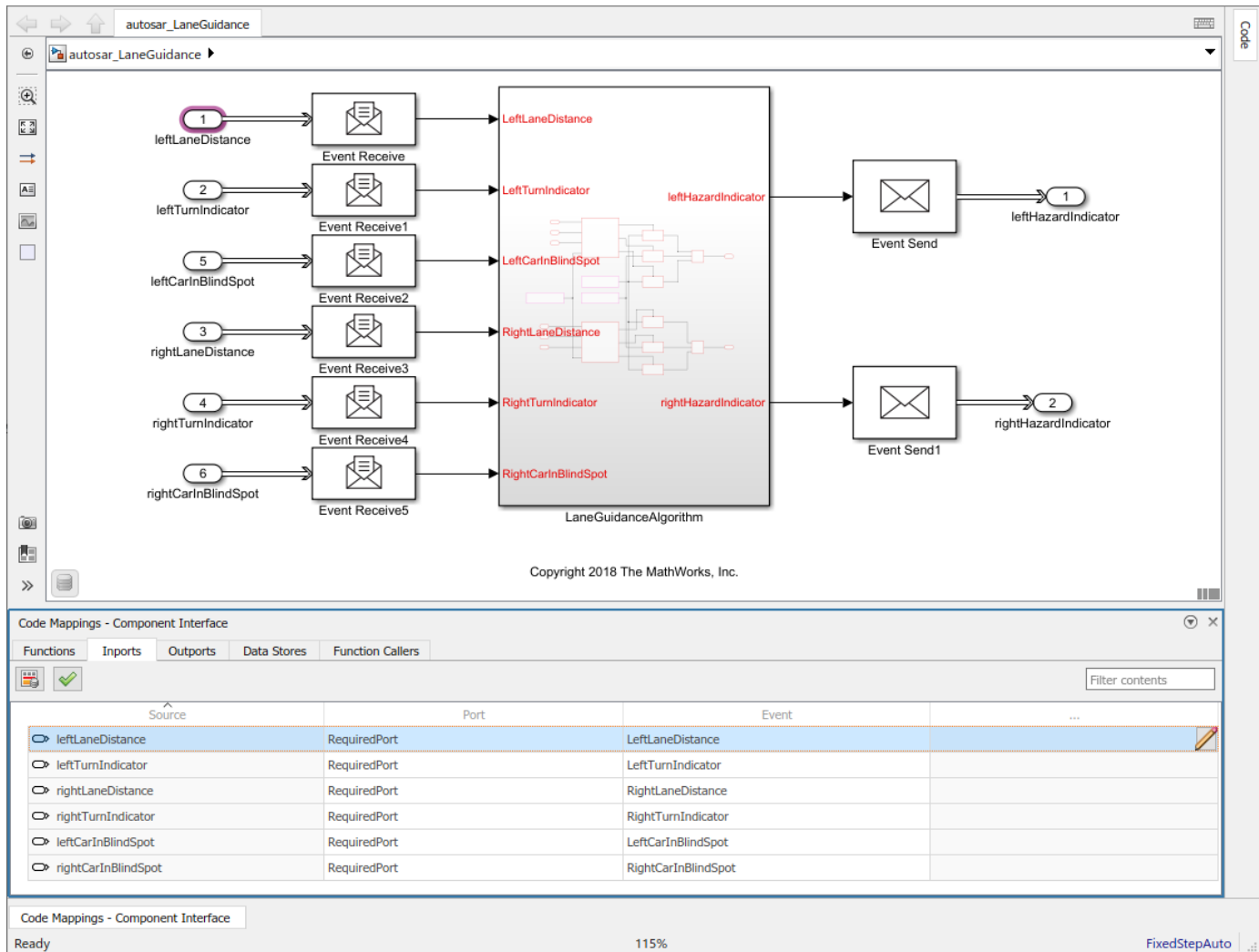
In a model for which an AUTOSAR system target file (`autosar.tlc` or `autosar_adaptive.tlc`) is selected, create or open a mapped view of the AUTOSAR model. In the model window, do one of the following:

- From the **Apps** tab, open the AUTOSAR Component Designer app.
- Click the perspective control in the lower-right corner and select **Code**.

If the model has not yet been mapped to an AUTOSAR software component, the AUTOSAR Component Quick Start opens. To configure the model for AUTOSAR component development, work through the quick-start procedure and click **Finish**. For more information, see “Create Mapped AUTOSAR Component with Quick Start” on page 3-2.


The model opens in the AUTOSAR Code perspective. This perspective displays the model and directly below the model, the Code Mappings editor. Here are the perspectives for the Classic and Adaptive Platforms.

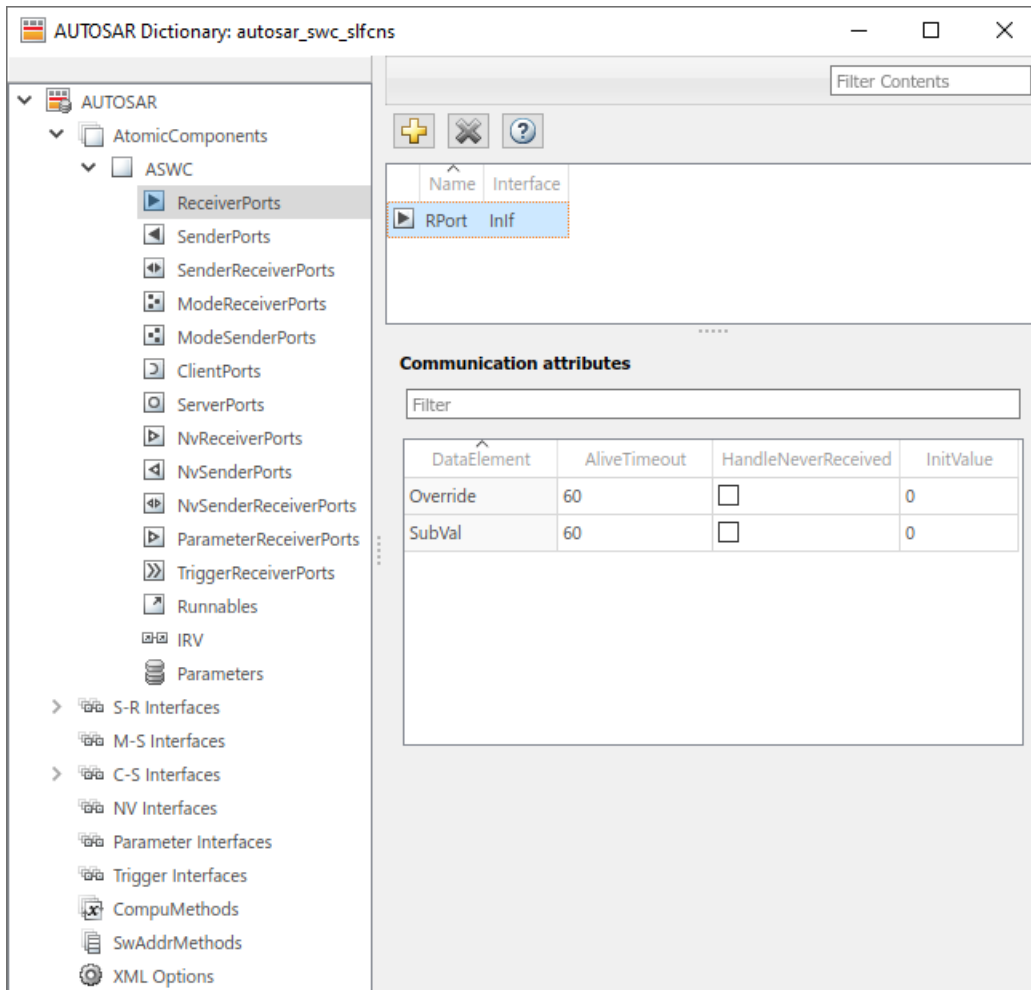







The Code Mappings editor provides in-canvas access to AUTOSAR mapping information, with batch editing, element filtering, easy navigation to model elements and AUTOSAR properties, and model element traceability. To view and modify additional AUTOSAR attributes for an element, select the

element and click the  icon.

To open an AUTOSAR properties view of the component model, either click the **AUTOSAR Dictionary** button  in the Code Mappings editor or, on the **AUTOSAR** tab, select **Code Interface > AUTOSAR Dictionary**. The AUTOSAR Dictionary opens.



As you progressively configure the model representation of the AUTOSAR component, you can:

- Freely switch between the Simulink and AUTOSAR perspectives, by selecting **AUTOSAR** tab menu entries or by clicking buttons.
- Use the **Filter contents** field (where available) to selectively display some elements, while omitting others, in the current view.
- In the Code Mappings editor, click the **Update** button  to update the Simulink to AUTOSAR mapping of the model with changes to Simulink entry-point functions, data transfers, and function callers.
- In the Code Mappings editor, click the **Validate** button  to validate the AUTOSAR component configuration.
- In the Code Mappings editor, with an element selected, click the **Edit** icon  to view and modify additional AUTOSAR attributes for the element.

See Also

Related Examples

- “Map AUTOSAR Elements for Code Generation” on page 4-50
- “Configure AUTOSAR Elements and Properties” on page 4-8
- “Map AUTOSAR Adaptive Elements for Code Generation” on page 6-37
- “Configure AUTOSAR Adaptive Elements and Properties” on page 6-21
- “Configure and Map AUTOSAR Component Programmatically” on page 4-293

Configure AUTOSAR Elements and Properties

In Simulink, you can use the AUTOSAR Dictionary and the Code Mappings editor separately or together to graphically configure an AUTOSAR software component and map Simulink model elements to AUTOSAR component elements. For more information, see “AUTOSAR Component Configuration” on page 4-3.

Use the AUTOSAR Dictionary to configure AUTOSAR elements from an AUTOSAR perspective. Using a tree format, the AUTOSAR Dictionary displays a mapped AUTOSAR component and its elements, communication interfaces, computation methods, software address methods, and XML options. Use the tree to select AUTOSAR elements and configure their properties. The properties that you modify are reflected in exported ARXML descriptions and potentially in generated AUTOSAR-compliant C code.


AUTOSAR Elements Configuration Workflow

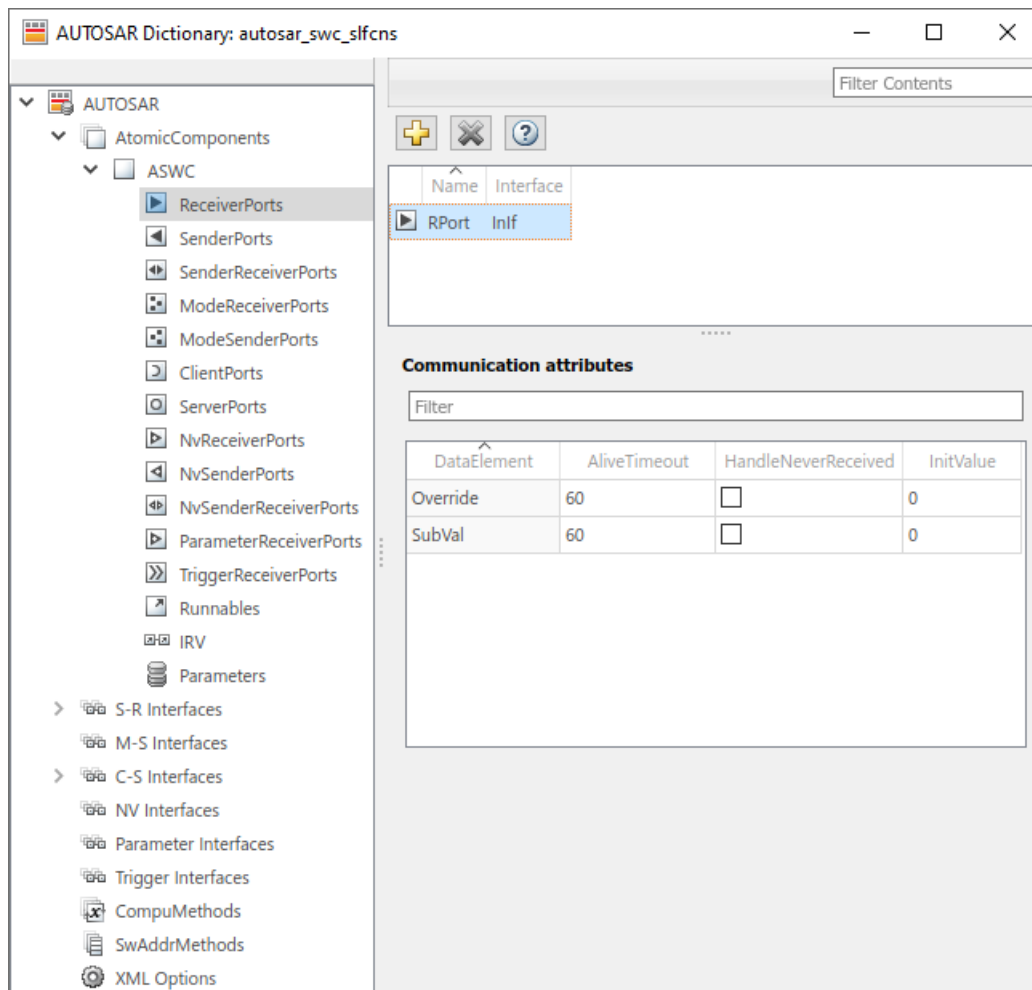
To configure AUTOSAR component elements for the Classic Platform in Simulink:


- 1 Open a model for which AUTOSAR system target file `autosar.tlc` is selected.
- 2 Create or open a mapped view of the AUTOSAR model. In the model window, do one of the following:
 - From the **Apps** tab, open the AUTOSAR Component Designer app.
 - Click the perspective control in the lower-right corner and select **Code**.

If the model has not yet been mapped to an AUTOSAR software component, the AUTOSAR Component Quick Start opens. Work through the quick-start procedure and click **Finish**. For more information, see “Create Mapped AUTOSAR Component with Quick Start” on page 3-2.

The model opens in the AUTOSAR Code perspective. This perspective displays the model and directly below the model, the Code Mappings editor.

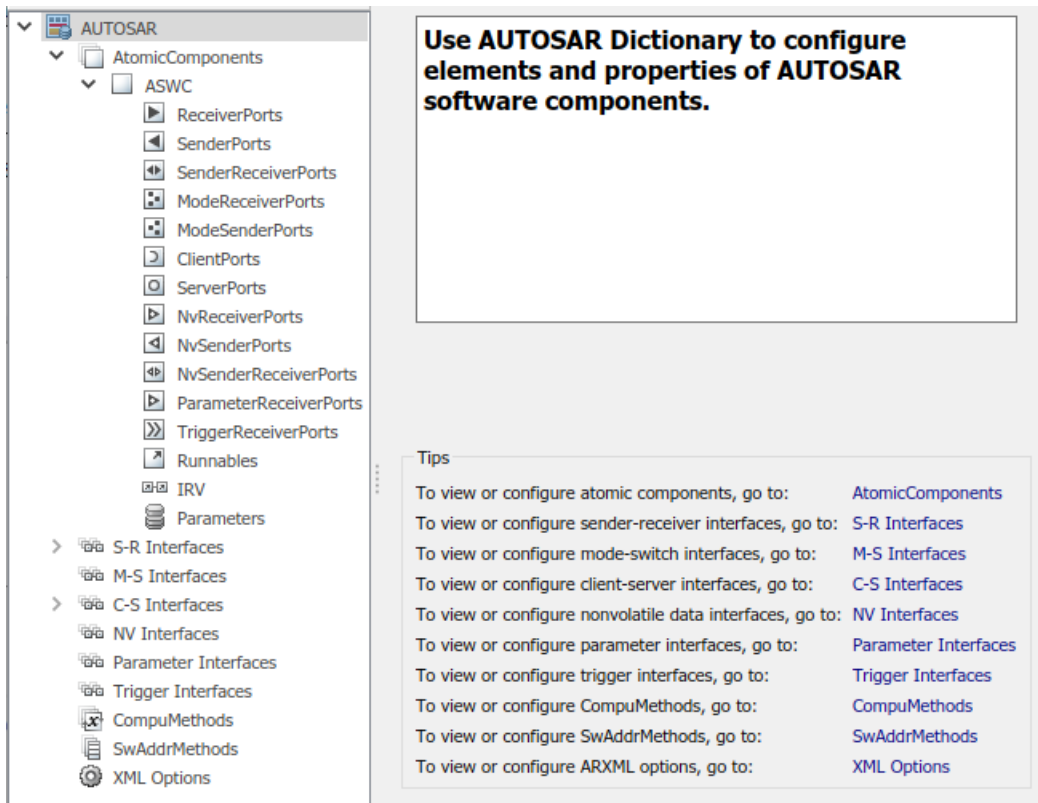
- 3 Open the AUTOSAR Dictionary. Either click the **AUTOSAR Dictionary** button  in the Code Mappings editor or, on the **AUTOSAR** tab, select **Code Interface > AUTOSAR Dictionary**.




- 4 To configure AUTOSAR elements and properties, navigate the AUTOSAR Dictionary tree. You can add elements, remove elements, or select elements to view and modify their properties. Use the **Filter Contents** field (where available) to selectively display some elements, while omitting others, in the current view.
- 5 After configuring AUTOSAR elements and properties, open the Code Mappings editor. Use the Code Mapping tabs to map Simulink elements to new or modified AUTOSAR elements.
- 6 Click the **Validate** button  to validate the AUTOSAR component configuration. If errors are reported, address them and then retry validation.

Configure AUTOSAR Atomic Software Components

AUTOSAR atomic software components contain AUTOSAR elements defined in the AUTOSAR standard, such as ports, runnables, inter-runnable variables (IRVs), and parameters. In the AUTOSAR Dictionary, component elements appear in a tree format under the component that owns them. To access component elements and their properties, you expand the component name.

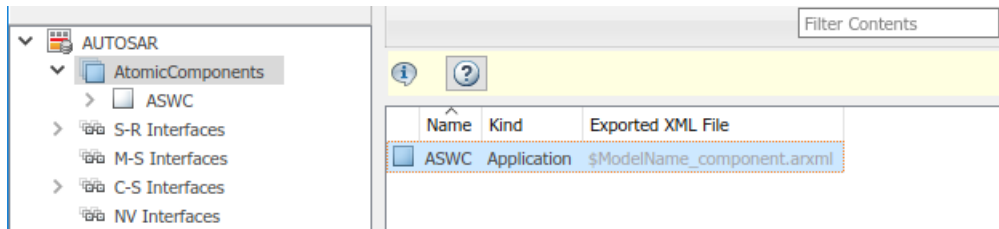


To configure AUTOSAR atomic software component elements and properties:

- 1 Open a model for which a mapped AUTOSAR software component has been created. For more information, see “Component Creation”.
- 2 From the **Apps** tab, open the AUTOSAR Component Designer app.
- 3 Open the AUTOSAR Dictionary. Either click the **AUTOSAR Dictionary** button  in the Code Mappings editor or, on the **AUTOSAR** tab, select **Code Interface > AUTOSAR Dictionary**.
- 4 In the leftmost pane of the AUTOSAR Dictionary, under **AUTOSAR**, select **AtomicComponents**.

The atomic components view in the AUTOSAR Dictionary displays atomic components and their types. You can:

- Select an AUTOSAR component and select a menu value for its kind (that is, its atomic software component type):
 - **Application** for application component
 - **ComplexDeviceDriver** for complex device driver component
 - **EcuAbstraction** for ECU abstraction component
 - **SensorActuator** for sensor or actuator component
 - **ServiceProxy** for service proxy component
- Rename a component by editing its name text.

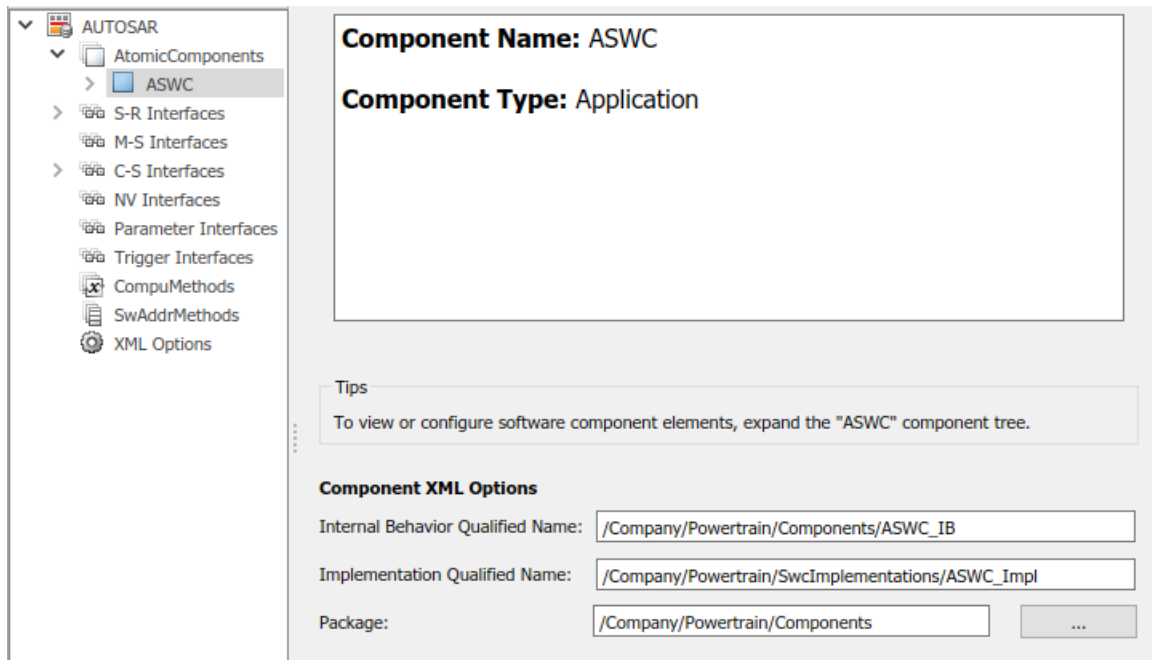


- 5 In the leftmost pane of the AUTOSAR Dictionary, expand **AtomicComponents** and select an AUTOSAR component.

The component view in the AUTOSAR Dictionary displays the name and type of the selected component, and component options for ARXML file export. You can:

- Modify the internal behavior qualified name to be generated for the component. Specify an AUTOSAR package path and a name.
- Modify the implementation qualified name to be generated for the component. Specify an AUTOSAR package path and a name.
- Modify the AUTOSAR package to be generated for the component. To specify the AUTOSAR package path, you can do either of the following:
 - Enter a package path in the **Package** parameter field. Package paths can use an organizational naming pattern, such as /CompanyName/Powertrain.
 - Click the button to the right of the **Package** field to open the AUTOSAR Package Browser. Use the browser to navigate to an existing package or create a new package. When you select a package in the browser and click **Apply**, the component **Package** parameter value is updated with your selection. For more information about the AUTOSAR Package Browser, see “Configure AUTOSAR Package for Component, Interface, CompuMethod, or SwAddrMethod” on page 4-94.

For more information about component XML options, see “Configure AUTOSAR Packages” on page 4-84.



Configure AUTOSAR Ports

An AUTOSAR software component contains communication ports defined in the AUTOSAR standard, including sender-receiver (S-R), client-server (C-S), mode-switch (M-S), nonvolatile (NV) data, trigger, and parameter interfaces. In the AUTOSAR Dictionary, communication ports appear in a tree format under the component that owns them and under a port type name. To access port elements and their properties, you expand the component name and expand the port type name.

- “Sender-Receiver Ports” on page 4-12
- “Mode-Switch Ports” on page 4-14
- “Client-Server Ports” on page 4-16
- “Nonvolatile Data Ports” on page 4-17
- “Parameter Receiver Ports” on page 4-19
- “Trigger Receiver Ports” on page 4-20



Sender-Receiver Ports

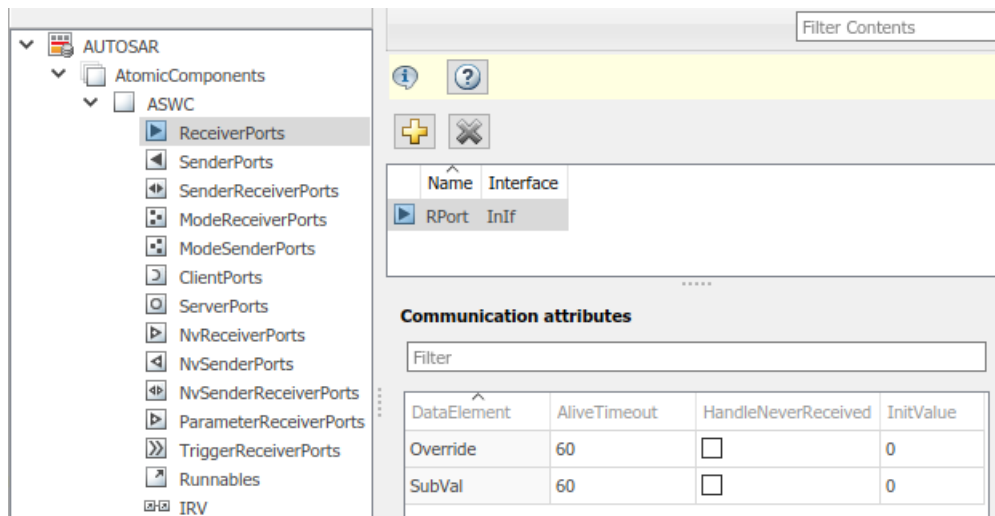
The AUTOSAR Dictionary views of sender and receiver ports support modeling AUTOSAR sender-receiver (S-R) communication in Simulink. You use the AUTOSAR Dictionary to configure AUTOSAR S-R ports, S-R interfaces, and S-R data elements in your model. For more information, see “Configure AUTOSAR Sender-Receiver Communication” on page 4-96 and “Configure AUTOSAR Queued Sender-Receiver Communication” on page 4-112.

To configure AUTOSAR S-R port elements and properties, open a model for which a mapped AUTOSAR software component has been created and open the AUTOSAR Dictionary.

- 1 In the leftmost pane of the AUTOSAR Dictionary, expand the component name and select **ReceiverPorts**.



The receiver ports view in the AUTOSAR Dictionary lists receiver ports and their properties. You can:

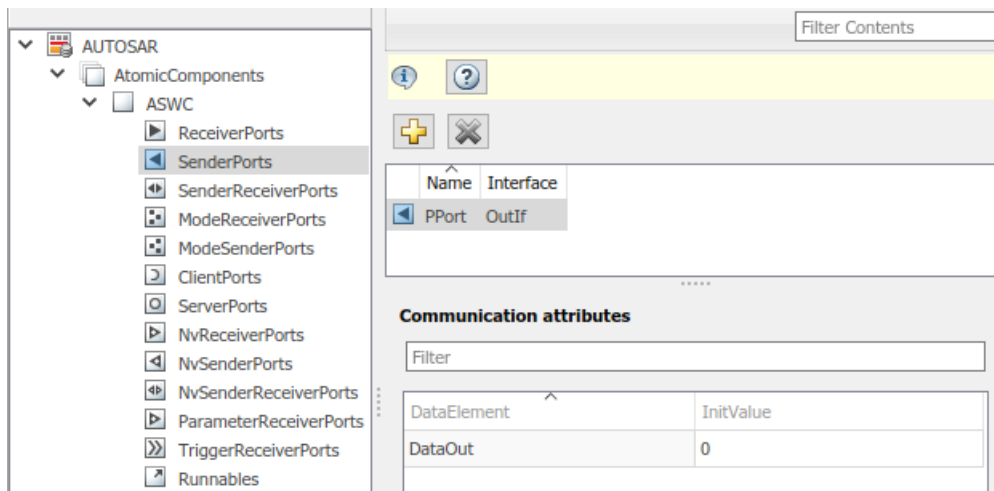
- Select an AUTOSAR receiver port, and view and optionally reselect its associated S-R interface.
- Rename a receiver port by editing its name text.
- When you select a receiver port, the AUTOSAR Dictionary displays additional port communication specification (ComSpec) attributes. For nonqueued receiver ports, you can modify ComSpec attributes `AliveTimeout`, `HandleNeverReceived`, and `InitValue`. For queued receiver ports, you can modify ComSpec attribute `QueueLength`. For more information, see “Configure AUTOSAR Sender-Receiver Port ComSpecs” on page 4-108.
- To add a receiver port, click the **Add** button  and use the Add Ports dialog box. Specify a port name and associate it with an existing S-R interface.
- To remove a receiver port, select the port and then click the **Delete** button .



- 2 In the leftmost pane of the AUTOSAR Dictionary, select **SenderPorts**.



The sender ports view in the AUTOSAR Dictionary lists sender ports and their properties. You can:

- Select an AUTOSAR sender port, and view and optionally reselect its associated S-R interface.
- Rename a sender port by editing its name text.
- When you select a sender port, the AUTOSAR Dictionary displays additional port communication specification (ComSpec) attributes. For nonqueued sender ports, you can modify ComSpec attribute `InitValue`. For more information, see “Configure AUTOSAR Sender-Receiver Port ComSpecs” on page 4-108.
- To add a sender port, click the **Add** button  and use the Add Ports dialog box. Specify a port name and associate it with an existing S-R interface.
- To remove a sender port, select the port and then click the **Delete** button .

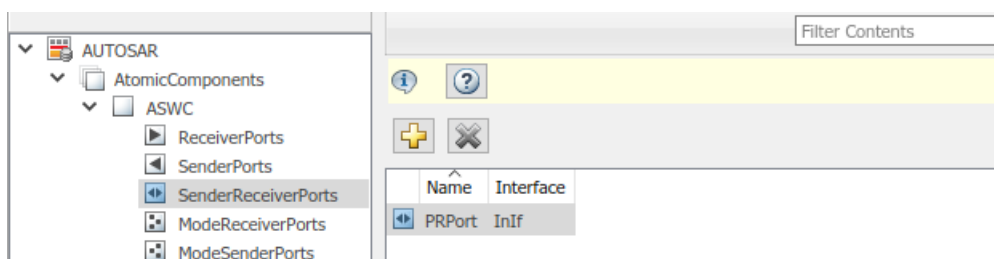


- 3 In the leftmost pane of the AUTOSAR Dictionary, select **SenderReceiverPorts**.

The sender-receiver ports view in the AUTOSAR Dictionary lists sender-receiver ports and their properties. You can:

- Select an AUTOSAR sender-receiver port, and view and optionally reselect its associated S-R interface.
- Rename a sender-receiver port by editing its name text.
- To add a sender-receiver port, click the **Add** button  and use the Add Ports dialog box. Specify a port name and associate it with an existing S-R interface.
- To remove a sender-receiver port, select the port and then click the **Delete** button .

Note AUTOSAR sender-receiver ports require AUTOSAR schema version 4.1 or higher. To select a schema version for the model, go to **AUTOSAR Code Generation Options** (Embedded Coder) in the Configuration Parameters dialog box.





Mode-Switch Ports

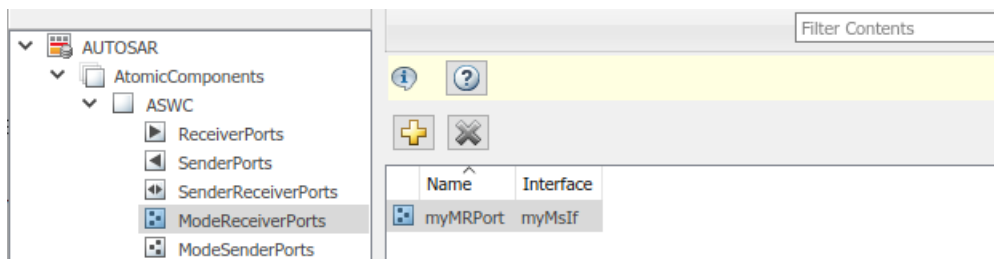
The AUTOSAR Dictionary views of mode sender and receiver ports support modeling AUTOSAR mode-switch (M-S) communication in Simulink. You use the AUTOSAR Dictionary to configure AUTOSAR M-S ports and M-S interfaces in your model. For more information, see “Configure AUTOSAR Mode-Switch Communication” on page 4-163.

To configure AUTOSAR M-S port elements and properties, open a model for which a mapped AUTOSAR software component has been created and open the AUTOSAR Dictionary.

- 1 In the leftmost pane of the AUTOSAR Dictionary, expand the component name and select **ModeReceiverPorts**.



The mode receiver ports view in the AUTOSAR Dictionary lists mode receiver ports and their properties. You can:

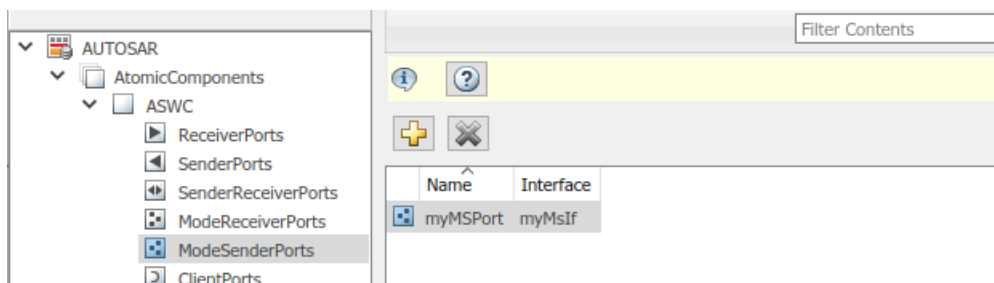
- Select an AUTOSAR mode receiver port, and view and optionally reselect its associated M-S interface.
- Rename a mode receiver port by editing its name text.
- To add a mode receiver port, click the **Add** button  and use the Add Ports dialog box. Specify a port name and associate it with an existing M-S interface. If an M-S interface does not exist in the component, you must create one before adding the port.
- To remove a mode receiver port, select the port and then click the **Delete** button .



- 2 In the leftmost pane of the AUTOSAR Dictionary, select **ModeSenderPorts**.

The mode sender ports view in the AUTOSAR Dictionary lists mode sender ports and their properties. You can:

- Select an AUTOSAR mode sender port, and view and optionally reselect its associated M-S interface.
- Rename a mode sender port by editing its name text.
- To add a mode sender port, click the **Add** button  and use the Add Ports dialog box. Specify a port name and associate it with an existing M-S interface. If an M-S interface does not exist in the component, you must create one before adding the port.
- To remove a mode sender port, select the port and then click the **Delete** button .





Client-Server Ports

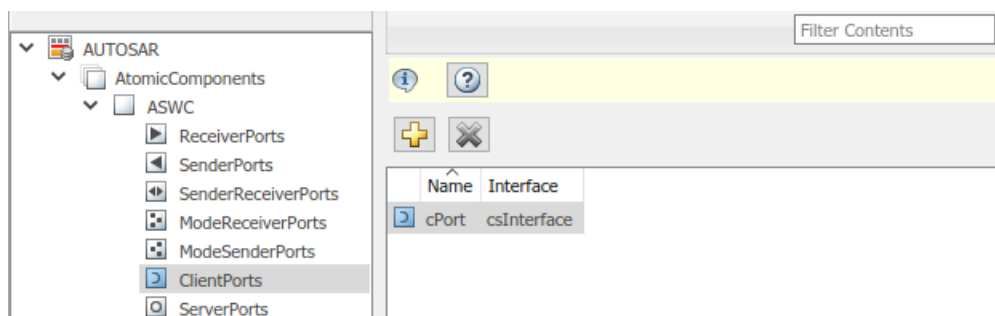
The AUTOSAR Dictionary views of client and server ports support modeling AUTOSAR client-server (C-S) communication in Simulink. You use the AUTOSAR Dictionary to configure AUTOSAR C-S ports, C-S interfaces, and C-S operations in your model. For more information, see “Configure AUTOSAR Client-Server Communication” on page 4-142.

To configure AUTOSAR C-S port elements and properties, open a model for which a mapped AUTOSAR software component has been created and open the AUTOSAR Dictionary.

- 1 In the leftmost pane of the AUTOSAR Dictionary, expand the component name and select **ClientPorts**.



The client ports view in the AUTOSAR Dictionary lists client ports and their properties. You can:

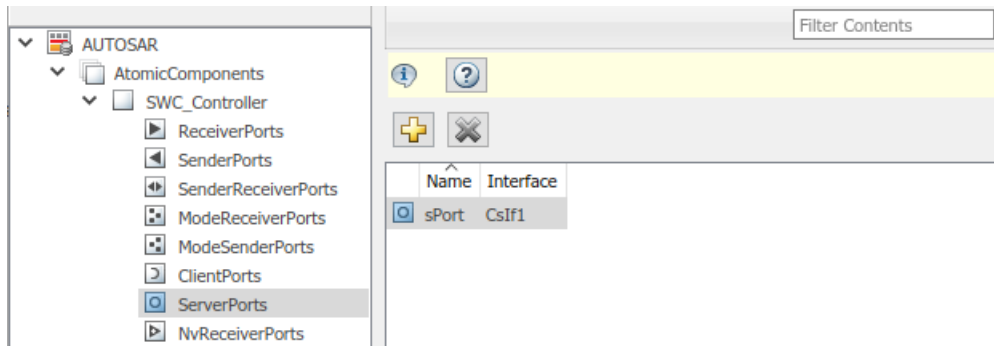
- Select an AUTOSAR client port, and view and optionally reselect its associated C-S interface.
- Rename a client port by editing its name text.
- To add a client port, click the **Add** button  and use the Add Ports dialog box. Specify a port name and associate it with an existing C-S interface. If a C-S interface does not exist in the component, you must create one before adding the port.
- To remove a client port, select the port and then click the **Delete** button .



- 2 In the leftmost pane of the AUTOSAR Dictionary, select **ServerPorts**.

The server ports view in the AUTOSAR Dictionary lists server ports and their properties. You can:

- Select an AUTOSAR server port, and view and optionally reselect its associated C-S interface.
- Rename a server port by editing its name text.
- To add a server port, click the **Add** button  and use the Add Ports dialog box. Specify a port name and associate it with an existing C-S interface. If a C-S interface does not exist in the component, you must create one before adding the port.
- To remove a server port, select the port and then click the **Delete** button .





Nonvolatile Data Ports

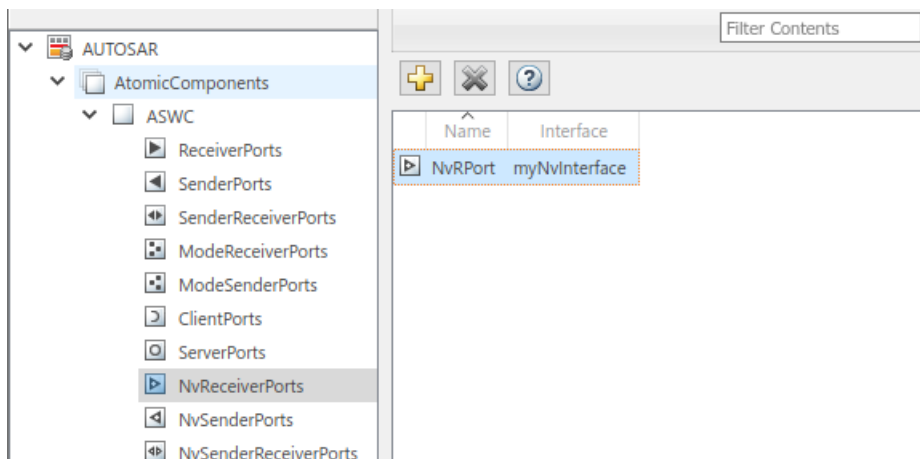
The AUTOSAR Dictionary views of nonvolatile (NV) sender and receiver ports support modeling AUTOSAR NV data communication in Simulink. You use the AUTOSAR Dictionary to configure AUTOSAR NV ports, NV interfaces, and NV data elements in your model. For more information, see “Configure AUTOSAR Nonvolatile Data Communication” on page 4-169.

To configure AUTOSAR NV port elements and properties, open a model for which a mapped AUTOSAR software component has been created and open the AUTOSAR Dictionary.

- 1 In the leftmost pane of the AUTOSAR Dictionary, expand the component name and select **NvReceiverPorts**.



The NV receiver ports view in the AUTOSAR Dictionary lists NV receiver ports and their properties. You can:

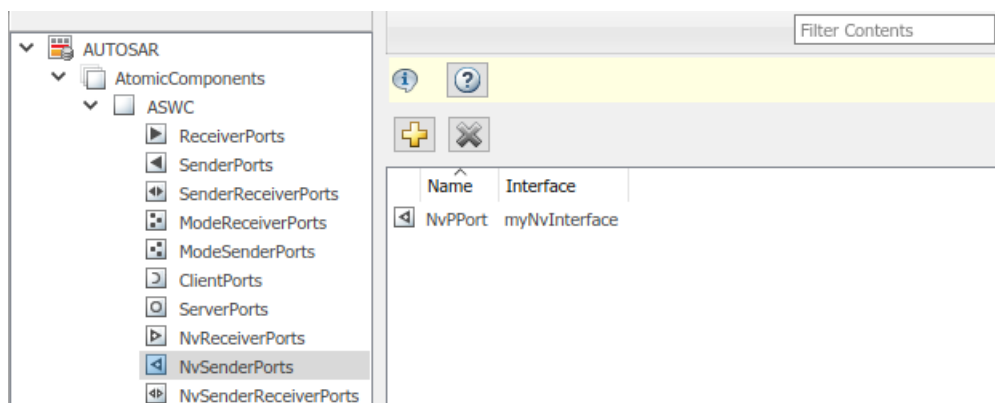
- Select an AUTOSAR NV receiver port, and view and optionally reselect its associated NV data interface.
- Rename an NV receiver port by editing its name text.
- To add an NV receiver port, click the **Add** button  and use the Add Ports dialog box. Specify a port name and associate it with an existing NV interface.
- To remove an NV receiver port, select the port and then click the **Delete** button .



2 In the leftmost pane of the AUTOSAR Dictionary, select **NvSenderPorts**.



The NV sender ports view in the AUTOSAR Dictionary lists NV sender ports and their properties. You can:

- Select an AUTOSAR NV sender port, and view and optionally reselect its associated NV data interface.
- Rename an NV sender port by editing its name text.
- To add an NV sender port, click the **Add** button  and use the Add Ports dialog box. Specify a port name and associate it with an existing NV interface.
- To remove an NV sender port, select the port and then click the **Delete** button .

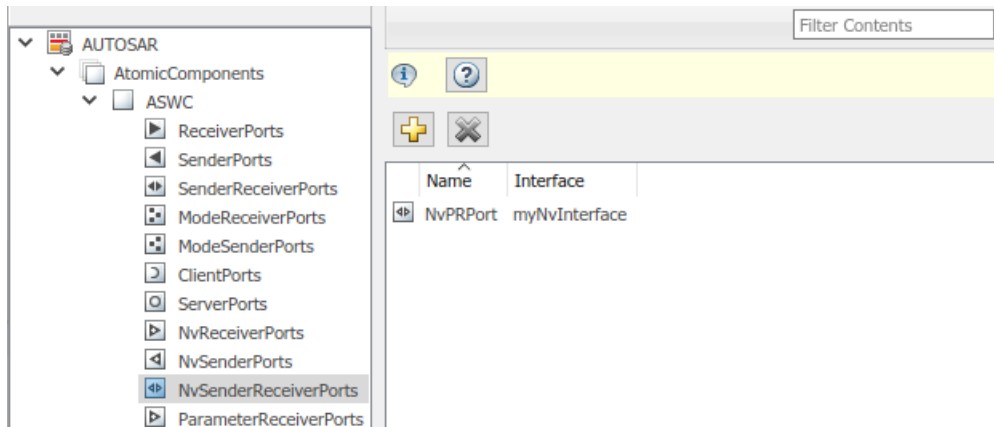


3 In the leftmost pane of the AUTOSAR Dictionary, select **NvSenderReceiverPorts**.

The NV sender-receiver ports view in the AUTOSAR Dictionary lists NV sender-receiver ports and their properties. You can:

- Select an AUTOSAR NV sender-receiver port, and view and optionally reselect its associated NV data interface.
- Rename an NV sender-receiver port by editing its name text.
- To add an NV sender-receiver port, click the **Add** button  and use the Add Ports dialog box. Specify a port name and associate it with an existing NV interface.
- To remove an NV sender-receiver port, select the port and then click the **Delete** button .

Note AUTOSAR NV sender-receiver ports require AUTOSAR schema version 4.1 or higher. To select a schema version for the model, go to **AUTOSAR Code Generation Options** (Embedded Coder) in the Configuration Parameters dialog box.





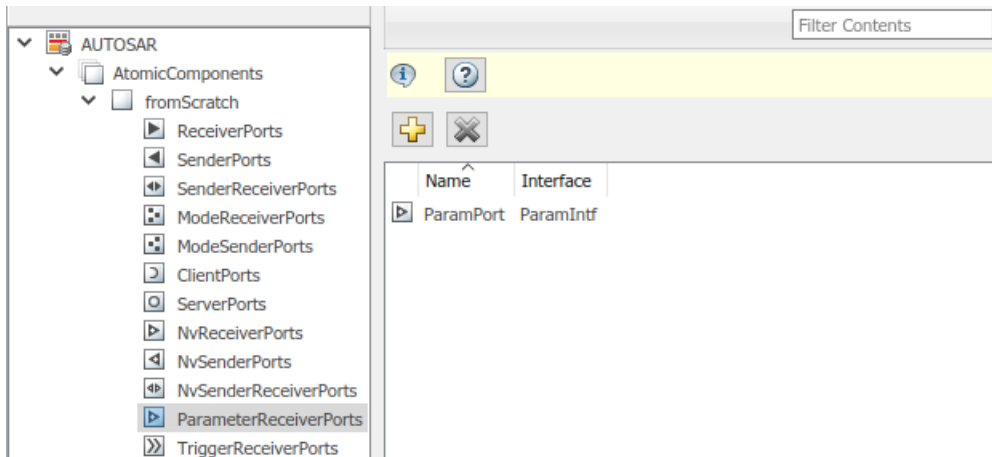
Parameter Receiver Ports

The AUTOSAR Dictionary view of parameter receiver ports supports modeling the receiver side of AUTOSAR parameter communication in Simulink. You use the AUTOSAR Dictionary to configure AUTOSAR parameter receiver ports, parameter interfaces, and parameter data elements in your model. For more information, see “Configure AUTOSAR Port Parameters for Communication with Parameter Component” on page 4-171.

To configure AUTOSAR parameter receiver port elements and properties, open a model for which a mapped AUTOSAR software component has been created and open the AUTOSAR Dictionary. In the leftmost pane of the AUTOSAR Dictionary, expand the component name and select **ParameterReceiverPorts**.

The parameter receiver ports view in the AUTOSAR Dictionary lists parameter receiver ports and their properties. You can:

- Select an AUTOSAR parameter receiver port, and view and optionally reselect its associated parameter interface.
- Rename a parameter receiver port by editing its name text.
- To add a parameter receiver port, click the **Add** button  and use the Add Ports dialog box. Specify a port name and associate it with an existing parameter interface.
- To remove a parameter receiver port, select the port and then click the **Delete** button .





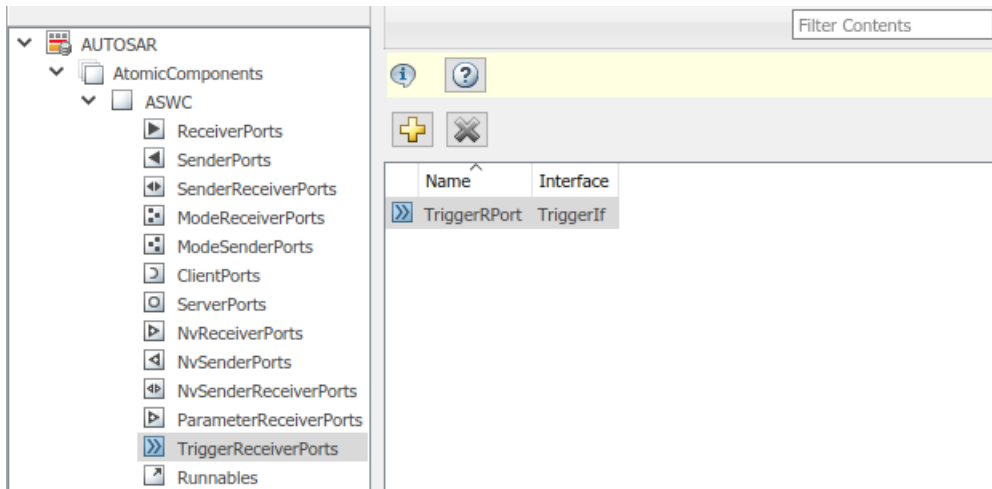
Trigger Receiver Ports

The AUTOSAR Dictionary view of trigger receiver ports supports modeling the receiver side of AUTOSAR trigger communication in Simulink. You use the AUTOSAR Dictionary to configure AUTOSAR trigger receiver ports, trigger interfaces, and triggers in your model. For more information, see “Configure Receiver for AUTOSAR External Trigger Event Communication” on page 4-175.

To configure AUTOSAR trigger receiver port elements and properties, open a model for which a mapped AUTOSAR software component has been created and open the AUTOSAR Dictionary. In the leftmost pane of the AUTOSAR Dictionary, expand the component name and select **TriggerReceiverPorts**.

The trigger receiver ports view in the AUTOSAR Dictionary lists trigger receiver ports and their properties. You can:

- Select an AUTOSAR trigger receiver port, and view and optionally reselect its associated trigger interface.
- Rename a trigger receiver port by editing its name text.
- To add a trigger receiver port, click the **Add** button  and use the Add Ports dialog box. Specify a port name and associate it with an existing trigger interface.
- To remove a trigger receiver port, select the port and then click the **Delete** button .



Configure AUTOSAR Runnables

The **Runnables** view in the AUTOSAR Dictionary supports modeling AUTOSAR runnable entities (runnables) and events, which implement aspects of internal AUTOSAR component behavior, in Simulink. You use the AUTOSAR Dictionary to configure AUTOSAR runnables and associated events that activate them. For more information, see “Configure AUTOSAR Runnables and Events” on page 4-178.

In the AUTOSAR Dictionary, runnables appear in a tree format under the component that owns them. To access runnable and event elements and their properties, you expand the component name.

To configure AUTOSAR runnable and event elements and properties, open a model for which a mapped AUTOSAR software component has been created and open the AUTOSAR Dictionary. In the leftmost pane of the AUTOSAR Dictionary, expand the component name and select **Runnables**.

The runnables view in the AUTOSAR Dictionary lists runnables for the AUTOSAR component. You can:



- Rename an AUTOSAR runnable by editing its name text.
- Modify the symbol name for a runnable. The specified AUTOSAR runnable symbol-name is exported in ARXML and C code. For example, if you change the symbol-name of `Runnable1` from `Runnable1` to `test_symbol`, the symbol-name `test_symbol` appears in the exported ARXML and C code. Here is a sample of the exported ARXML descriptions:

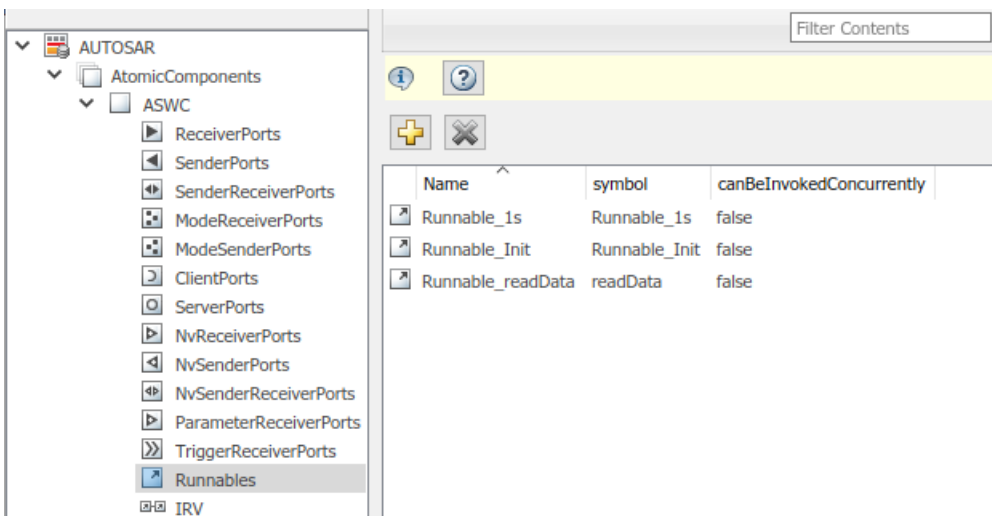
```
<RUNNABLE-ENTITY UUID="...">
  <SHORT-NAME>Runnable1</SHORT-NAME>
  ...
  <SYMBOL>test_symbol</SYMBOL>
  ...
</RUNNABLE-ENTITY>
```

Here is a sample of the generated C code:

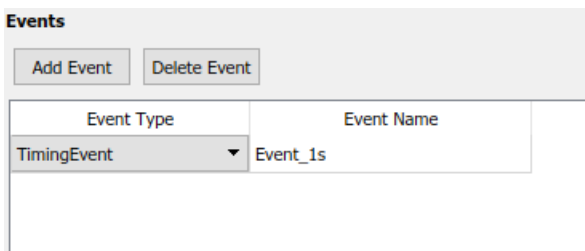
```
/* Model step function for TID1 */
void test_symbol(void)          /* Explicit Task: Runnable1 */
{
  ...
}
```

Note For an AUTOSAR server runnable — that is, a runnable with an `OperationInvokedEvent` — the **symbol** name must match the Simulink server function name.

- For an AUTOSAR server runnable, set the runnable property `canBeInvokedConcurrently` to designate whether to enforce concurrency constraints. For nonserver runnables, leave `canBeInvokedConcurrently` set to `false`. For more information, see “Concurrency Constraints for AUTOSAR Server Runnables” on page 4-159.
- To add a runnable, click the **Add** button .
- To remove a runnable, select the runnable and then click the **Delete** button .



Select a runnable to see its list of associated events. The **Events** pane lists each AUTOSAR event with its type — `TimingEvent`, `DataReceivedEvent`, `ModeSwitchEvent`, `OperationInvokedEvent`, `InitEvent`, `DataReceiveErrorEvent`, or `ExternalTriggerOccurredEvent` — and name. You can rename an AUTOSAR event by editing its name text. You can use the buttons **Add Event** and **Delete Event** to add or delete events from a runnable.



If you select an event of type `DataReceivedEvent`, the runnable is activated by a `DataReceivedEvent`. Select the event name to display its **Trigger** property. Select a trigger for the event from the list of available trigger ports.

Events

Add Event Delete Event

Event Type	Event Name
DataReceivedEvent ▼	Event

Event Properties

Trigger RPort.SubVal ▼

If you select an event of type `DataReceiveErrorEvent`, the runnable is activated by a `DataReceiveErrorEvent`. Select the event name to display its **Trigger** property. Select a trigger for the event from the list of available trigger ports. (For more information on using a `DataReceiveErrorEvent`, see “Configure AUTOSAR Receiver Port for `DataReceiveErrorEvent`” on page 4-106.)

Events

Add Event Delete Event

Event Type	Event Name
DataReceiveErrorEvent ▼	DRE_Evt

Event Properties

Trigger RPort.Override ▼

If you select an event of type `ModeSwitchEvent`, the **Mode Activation** and **Mode Receiver Port** properties are displayed. Select a mode receiver port for the event from the list of configured mode-receiver ports. Select a mode activation value for the event from the list of values (`OnEntry`, `OnExit`, or `OnTransition`). Based on the value you select, one or two **Mode Declaration** drop-down lists appear. Select a mode (or two modes) for the event, among those declared by the mode declaration group associated with the Simulink inport that models the AUTOSAR mode-receiver port. (For more information on using a `ModeSwitchEvent`, see “Configure AUTOSAR Mode-Switch Communication” on page 4-163.)

Events

Add Event Delete Event

Event Type	Event Name
ModeSwitchEvent	Event_Run

Event Properties

Mode Activation: OnTransition

Mode Receiver Port: MRPort

Transition From

Mode Declaration: Sleep

Transition To

Mode Declaration: Run

If you select an event of type `OperationInvokedEvent`, the runnable becomes an AUTOSAR server runnable. Select the event name to display its **Trigger** property. Select a trigger for the event from the list of available server port and operation combinations. The **Operation Signature** is displayed below the **Trigger** property. (For more information on using an `OperationInvokedEvent`, see “Configure AUTOSAR Client-Server Communication” on page 4-142.)

Events

Add Event Delete Event

Event Type	Event Name
OperationInvokedEvent	Event_readData

Event Properties

Trigger: SrvPort.readData

Operation Signature:
Error readData(Out Data)

If you select an event of type `InitEvent`, you can rename the event by editing its name text. (For more information on using an `InitEvent`, see “Configure AUTOSAR Initialization Runnable (R4.1)” on page 4-196.)

Note AUTOSAR `InitEvents` require AUTOSAR schema version 4.1 or higher. To select a schema version for the model, go to **AUTOSAR Code Generation Options** (Embedded Coder) in the Configuration Parameters dialog box.

The screenshot shows a window titled "Events" with two buttons: "Add Event" and "Delete Event". Below the buttons is a table with two columns: "Event Type" and "Event Name". The "Event Type" column has a dropdown menu with "InitEvent" selected. The "Event Name" column contains the text "Event".

Event Type	Event Name
InitEvent	Event

If you select an event of type `ExternalTriggerOccurredEvent`, the runnable is activated when an AUTOSAR software component or service signals an external trigger event. Select the event name to display its **Trigger** property. Select a trigger for the event from the list of available trigger receiver port and trigger combinations. (For more information on using an `ExternalTriggerOccurredEvent`, see “Configure Receiver for AUTOSAR External Trigger Event Communication” on page 4-175.)

The screenshot shows a window titled "Events" with two buttons: "Add Event" and "Delete Event". Below the buttons is a table with two columns: "Event Type" and "Event Name". The "Event Type" column has a dropdown menu with "ExternalTriggerOccurredEvent" selected. The "Event Name" column contains the text "Event_Trigger". Below the table is a section titled "Event Properties" with a "Trigger" dropdown menu set to "TriggerRPort.Trigger1".

Event Type	Event Name
ExternalTriggerOccurredEvent	Event_Trigger

Event Properties

Trigger: TriggerRPort.Trigger1

Configure AUTOSAR Inter-Runnable Variables



The **IRV** view in the AUTOSAR Dictionary supports modeling AUTOSAR inter-runnable variables (IRVs), which connect runnables and implement aspects of internal AUTOSAR component behavior, in Simulink. You use the AUTOSAR Dictionary to create AUTOSAR IRVs and configure IRV data properties. For more information, see “Configure AUTOSAR Data for Calibration and Measurement” on page 4-262.

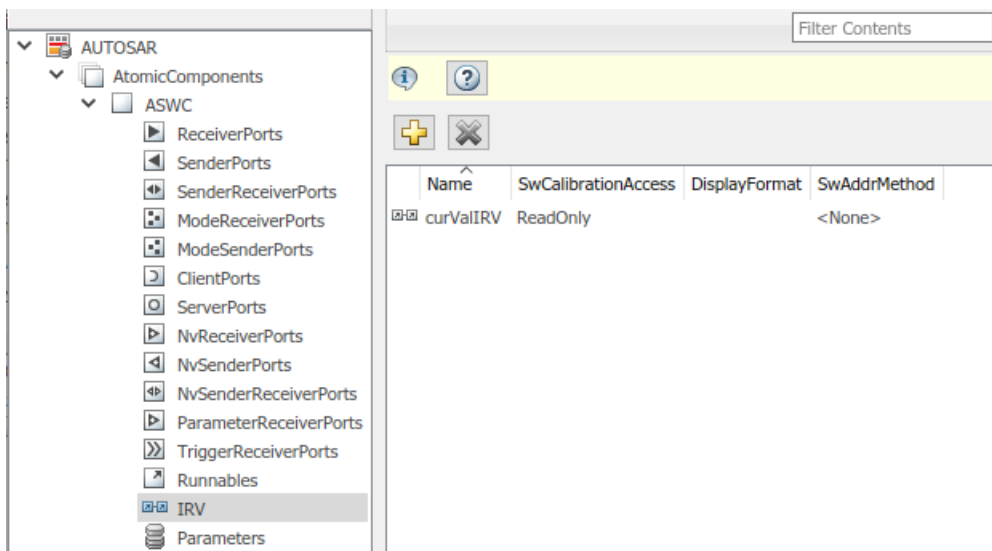
In the AUTOSAR Dictionary, IRVs appear in a tree format under the component that owns them. To access IRV elements and their properties, you expand the component name.

To configure AUTOSAR IRV elements and properties, open a model for which a mapped AUTOSAR software component has been created and open the AUTOSAR Dictionary. In the leftmost pane of the AUTOSAR Dictionary, expand the component name and select **IRV**.

The IRV view in the AUTOSAR Dictionary lists IRVs for the AUTOSAR component. You can:

- Rename an AUTOSAR IRV by editing its name text.
- Specify the level of calibration and measurement tool access to IRV data. Select an IRV and set its **SwCalibrationAccess** value to `ReadOnly`, `ReadWrite`, or `NotAccessible`.

- Optionally specify the format to be used by calibration and measurement tools to display the IRV data. In the **DisplayFormat** field, enter an ANSI® C printf format specifier string. For example, %2.1d specifies a signed decimal number, with a minimum width of 2 characters and a maximum precision of 1 digit, producing a displayed value such as 12.2. For more information about constructing a format specifier string, see “Configure DisplayFormat” on page 4-264.
- Optionally specify a software address method for the IRV data. Select or enter a value for **SwAddrMethod**. AUTOSAR software components use SwAddrMethods to group data in memory for access by calibration and measurement tools. For more information, see “Configure AUTOSAR SwAddrMethods” on page 4-42.
- To add an IRV, click the **Add** button .
- To remove an IRV, select the IRV and then click the **Delete** button .



Configure AUTOSAR Parameters



The **Parameters** view in the AUTOSAR Dictionary supports modeling AUTOSAR internal calibration parameters, for use with AUTOSAR integrated and distributed lookups, in Simulink. You use the AUTOSAR Dictionary to create AUTOSAR internal parameters and configure parameter data properties. For port-based calibration parameters, you create “Parameter Interfaces” on page 4-36.

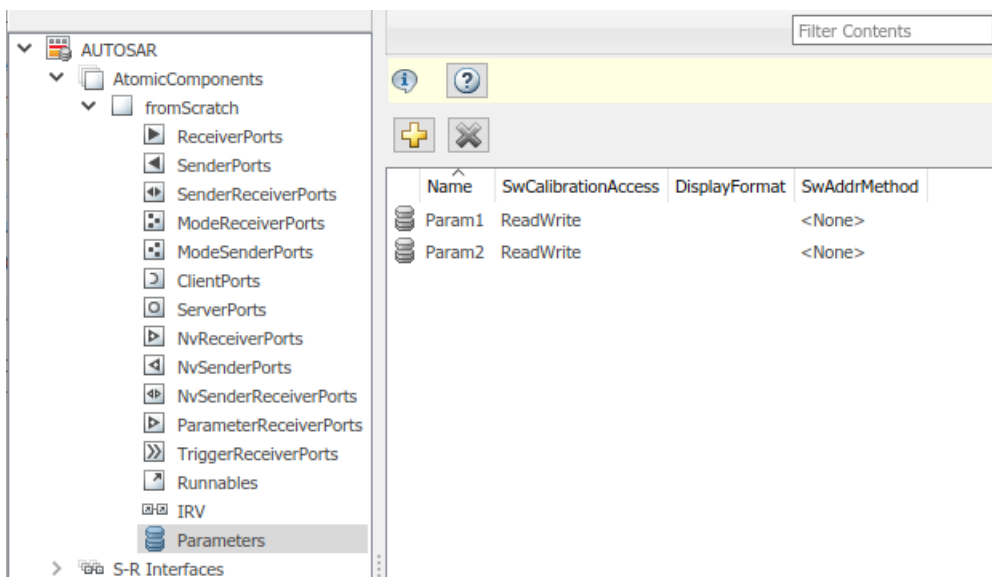
In the AUTOSAR Dictionary, internal parameters appear in a tree format under the component that owns them. To access parameter elements and their properties, you expand the component name.

To configure AUTOSAR parameter elements and properties, open a model for which a mapped AUTOSAR software component has been created and open the AUTOSAR Dictionary. In the leftmost pane of the AUTOSAR Dictionary, expand the component name and select **Parameters**.

The parameters view in the AUTOSAR Dictionary lists internal parameters for the AUTOSAR component. You can:

- Rename an AUTOSAR parameter by editing its name text.

- Specify the level of calibration and measurement tool access to parameters. Select a parameter and set its **SwCalibrationAccess** value to `ReadOnly`, `ReadWrite`, or `NotAccessible`.
- Optionally specify the format to be used by calibration and measurement tools to display the parameter data. In the **DisplayFormat** field, enter an ANSI C `printf` format specifier string. For example, `%2.1d` specifies a signed decimal number, with a minimum width of 2 characters and a maximum precision of 1 digit, producing a displayed value such as 12.2. For more information about constructing a format specifier string, see “Configure DisplayFormat” on page 4-264.
- Optionally specify a software address method for the parameter data. Select or enter a value for **SwAddrMethod**. AUTOSAR software components use `SwAddrMethods` to group data in memory for access by calibration and measurement tools. For more information, see “Configure AUTOSAR SwAddrMethods” on page 4-42.
- To add an internal parameter, click the **Add** button .
- To remove an internal parameter, select the parameter and then click the **Delete** button .



Configure AUTOSAR Communication Interfaces

An AUTOSAR software component uses communication interfaces defined in the AUTOSAR standard, including sender-receiver (S-R), client-server (C-S), mode-switch (M-S), nonvolatile (NV) data, trigger, and parameter interfaces. In the AUTOSAR Dictionary, communication interfaces appear in a tree format under the interface type name. To access interface elements and their properties, you expand the interface type name.

- “Sender-Receiver Interfaces” on page 4-28
- “Mode-Switch Interfaces” on page 4-29
- “Client-Server Interfaces” on page 4-31
- “Nonvolatile Data Interfaces” on page 4-34
- “Parameter Interfaces” on page 4-36
- “Trigger Interfaces” on page 4-38



Sender-Receiver Interfaces

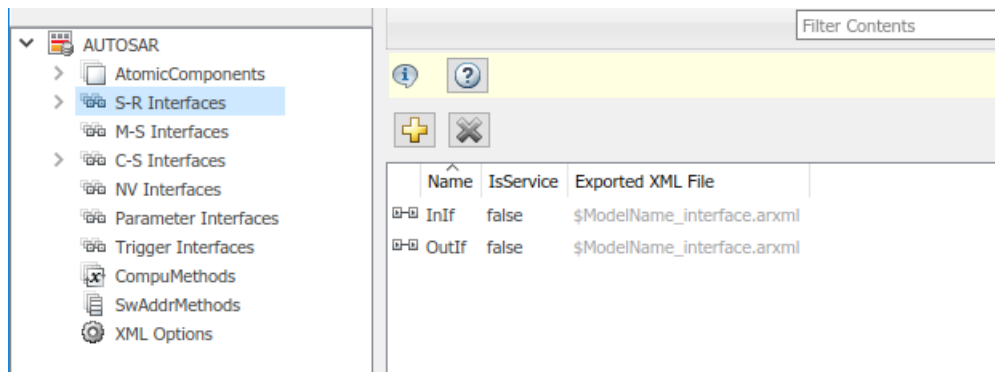
The **S-R Interfaces** view in the AUTOSAR Dictionary supports modeling AUTOSAR sender-receiver (S-R) communication in Simulink. You use the AUTOSAR Dictionary to configure AUTOSAR S-R ports, S-R interfaces, and S-R data elements in your model. For more information, see “Configure AUTOSAR Sender-Receiver Communication” on page 4-96 and “Configure AUTOSAR Queued Sender-Receiver Communication” on page 4-112.

To configure AUTOSAR S-R interface elements and properties, open a model for which a mapped AUTOSAR software component has been created and open the AUTOSAR Dictionary.

- 1 In the leftmost pane of the AUTOSAR Dictionary, select **S-R Interfaces**.

The S-R interfaces view in the AUTOSAR Dictionary lists AUTOSAR sender-receiver interfaces and their properties. You can:

- Select an S-R interface and then select a menu value to specify whether or not it is a service.
- Rename an S-R interface by editing its name text.
- To add an S-R interface, click the **Add** button  and use the Add Interfaces dialog box. Specify an interface name, the number of data elements it contains, whether the interface is a service, and the path of the Interface package.
- To remove an S-R interface, select the interface and then click the **Delete** button .

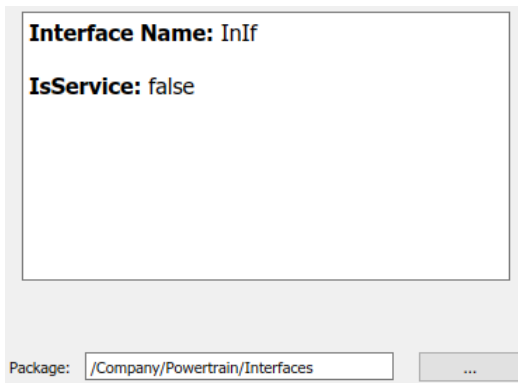


- 2 In the leftmost pane of the AUTOSAR Dictionary, expand **S-R Interfaces** and select an S-R interface from the list.

The S-R interface view in the AUTOSAR Dictionary displays the name of the selected S-R interface, whether or not it is a service, and the AUTOSAR package to be generated for the interface.



To modify the AUTOSAR package for the interface, you can do either of the following:

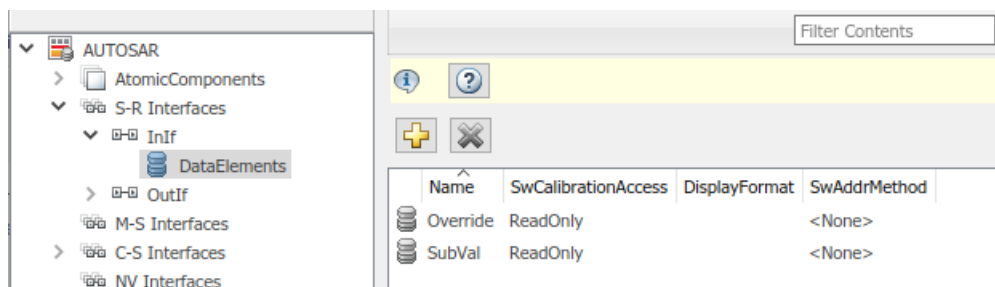
- Enter a package path in the **Package** parameter field.
- Click the button to the right of the **Package** field to open the AUTOSAR Package Browser. Use the browser to navigate to an existing package or create a new package. When you select a package in the browser and click **Apply**, the interface **Package** parameter value is updated with your selection. For more information about the AUTOSAR Package Browser, see “Configure AUTOSAR Package for Component, Interface, CompuMethod, or SwAddrMethod” on page 4-94.



- 3 In the leftmost pane of the AUTOSAR Dictionary, expand the selected interface and select **DataElements**.

The data elements view in the AUTOSAR Dictionary lists AUTOSAR sender-receiver interface data elements and their properties. You can:

- Select an S-R interface data element and edit its name value.
- Specify the level of calibration and measurement tool access to S-R interface data elements. Select a data element and set its **SwCalibrationAccess** value to `ReadOnly`, `ReadWrite`, or `NotAccessible`.
- Optionally specify the format to be used by calibration and measurement tools to display the data element. In the **DisplayFormat** field, enter an ANSI C `printf` format specifier string. For example, `%2.1d` specifies a signed decimal number, with a minimum width of 2 characters and a maximum precision of 1 digit, producing a displayed value such as 12.2. For more information about constructing a format specifier string, see “Configure DisplayFormat” on page 4-264.
- Optionally specify a software address method for the data element. Select or enter a value for **SwAddrMethod**. AUTOSAR software components use `SwAddrMethods` to group data in memory for access by calibration and measurement tools. For more information, see “Configure AUTOSAR SwAddrMethods” on page 4-42.
- To add a data element, click the **Add** button .
- To remove a data element, select the data element and then click the **Delete** button .



Mode-Switch Interfaces



The **M-S Interfaces** view in the AUTOSAR Dictionary supports modeling AUTOSAR mode-switch (M-S) communication in Simulink. You use the AUTOSAR Dictionary to configure AUTOSAR M-S ports

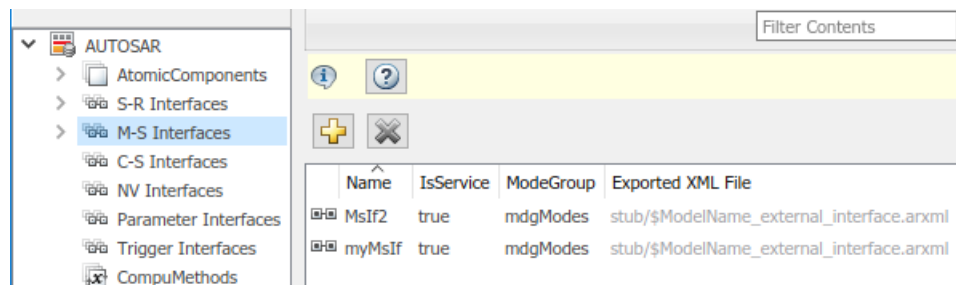
and M-S interfaces in your model. For more information, see “Configure AUTOSAR Mode-Switch Communication” on page 4-163.

To configure AUTOSAR M-S interface elements and properties, open a model for which a mapped AUTOSAR software component has been created and open the AUTOSAR Dictionary.

- 1 In the leftmost pane of the AUTOSAR Dictionary, select **M-S Interfaces**.

The M-S interfaces view in the AUTOSAR Dictionary lists AUTOSAR mode-switch interfaces and their properties. You can:

- Select an M-S interface, specify whether or not it is a service, and modify the name of its associated mode group.
- The **IsService** property defaults to `true`. The `true` setting assumes that the M-S interface participates in run-time mode management, for example, performed by the Basic Software Mode Manager.
- A mode group contains mode values, declared in Simulink using enumeration. For more information, see “Configure AUTOSAR Mode-Switch Communication” on page 4-163.
- Rename an M-S interface by editing its name text.
- To add an M-S interface, click the **Add** button  and use the Add Interfaces dialog box. Specify an interface name, the name of a mode group, whether the interface is a service, and the path of the Interface package.
- To remove an M-S interface, select the interface and then click the **Delete** button .

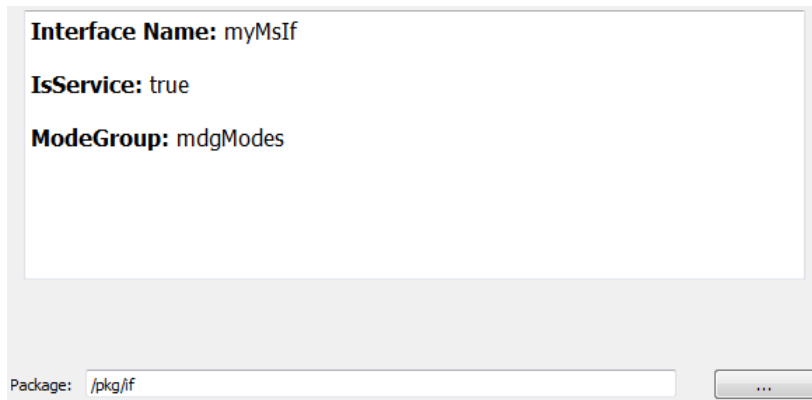


- 2 In the leftmost pane of the AUTOSAR Dictionary, expand **M-S Interfaces** and select an M-S interface from the list.

The M-S interface view in the AUTOSAR Dictionary displays the name of the selected M-S interface, whether or not it is a service, its associated mode group, and the AUTOSAR package for the interface.

To modify the AUTOSAR package for the interface, you can do either of the following:

- Enter a package path in the **Package** parameter field.
- Click the button to the right of the **Package** field to open the AUTOSAR Package Browser. Use the browser to navigate to an existing package or create a new package. When you select a package in the browser and click **Apply**, the interface **Package** parameter value is updated with your selection. For more information about the AUTOSAR Package Browser, see “Configure AUTOSAR Package for Component, Interface, CompuMethod, or SwAddrMethod” on page 4-94.





Client-Server Interfaces

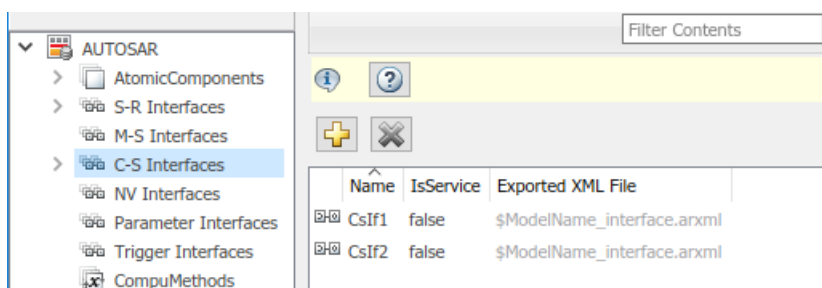
The **C-S Interfaces** view in the AUTOSAR Dictionary supports modeling AUTOSAR client-server (C-S) communication in Simulink. You use the AUTOSAR Dictionary to configure AUTOSAR C-S ports, C-S interfaces, and C-S operations in your model. For more information, see “Configure AUTOSAR Client-Server Communication” on page 4-142.

To configure AUTOSAR C-S interface elements and properties, open a model for which a mapped AUTOSAR software component has been created and open the AUTOSAR Dictionary.

- 1 In the leftmost pane of the AUTOSAR Dictionary, select **C-S Interfaces**.

The C-S interfaces view in the AUTOSAR Dictionary lists AUTOSAR client-server interfaces and their properties. You can:

- Select a C-S interface and then select a menu value to specify whether or not it is a service.
- Rename a C-S interface by editing its name text.
- To add a C-S interface, click the **Add** button  and use the Add Interfaces dialog box. Specify an interface name, the number of associated operations it contains, whether the interface is a service, and the path of the Interface package.
- To remove a C-S interface, select the interface and then click the **Delete** button .

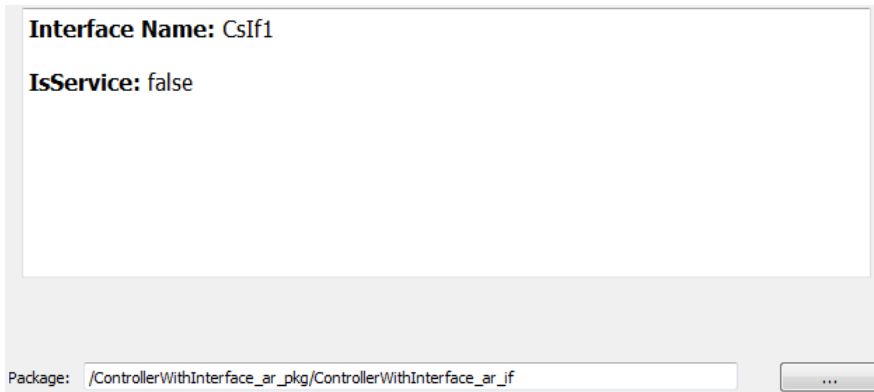


- 2 In the leftmost pane of the AUTOSAR Dictionary, expand **C-S Interfaces** and select a C-S interface from the list.

The C-S interface view in the AUTOSAR Dictionary displays the name of the selected C-S interface, whether or not it is a service, and the AUTOSAR package for the interface.



To modify the AUTOSAR package for the interface, you can do either of the following:

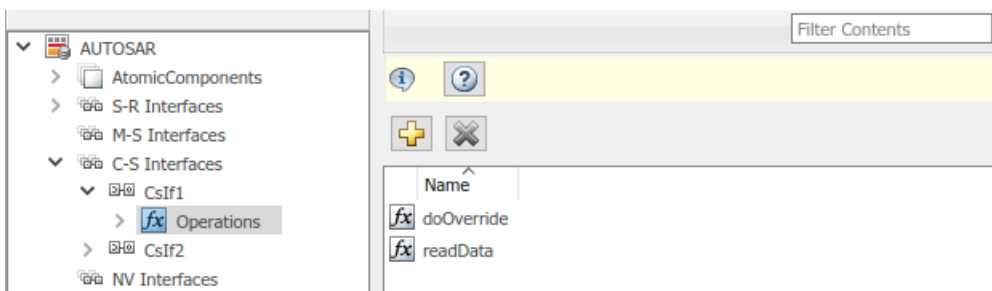
- Enter a package path in the **Package** parameter field.
- Click the button to the right of the **Package** field to open the AUTOSAR Package Browser. Use the browser to navigate to an existing package or create a new package. When you select a package in the browser and click **Apply**, the interface **Package** parameter value is updated with your selection. For more information about the AUTOSAR Package Browser, see “Configure AUTOSAR Package for Component, Interface, CompuMethod, or SwAddrMethod” on page 4-94.



- 3 In the leftmost pane of the AUTOSAR Dictionary, expand the selected interface and select **Operations**.

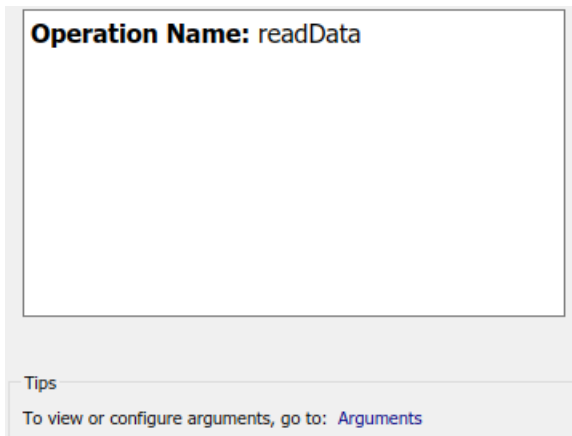
The operations view in the AUTOSAR Dictionary lists AUTOSAR client-server interface operations. You can:

- Select a C-S interface operation and edit its name value.
- To add an operation, click the **Add** button  and use the Add Operation dialog box. In the dialog box, specify an operation name and an associated Simulink function. To create operation arguments from a Simulink function, select the associated Simulink function among those present in the configuration. If you are creating an operation without arguments, select **None**.
- To remove an operation, select the operation and then click the **Delete** button .





- 4 In the leftmost pane of the AUTOSAR Dictionary, expand **Operations** and select an operation from the list.

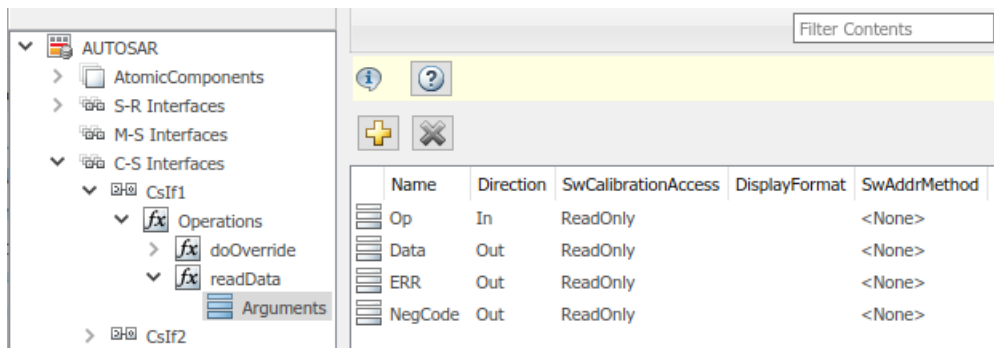
The operations view in the AUTOSAR Dictionary displays the name of the selected C-S operation.



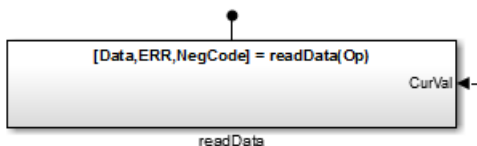
- 5 In the leftmost pane of the AUTOSAR Dictionary, expand the selected operation and select **Arguments**.

The arguments view in the AUTOSAR Dictionary lists AUTOSAR client-server operation arguments and their properties. You can:

- Select a C-S operation argument and edit its name value.
- Specify the direction of the C-S operation argument. Set its **Direction** value to In, Out, InOut, or Error. Select Error if the operation argument returns application error status. For more information, see “Configure AUTOSAR Client-Server Error Handling” on page 4-156.
- Specify the level of calibration and measurement tool access to C-S operation arguments. Select an argument and set its **SwCalibrationAccess** value to ReadOnly, ReadWrite, or NotAccessible.
- Optionally specify the format to be used by calibration and measurement tools to display the argument. In the **DisplayFormat** field, enter an ANSI C `printf` format specifier string. For example, `%2.1d` specifies a signed decimal number, with a minimum width of 2 characters and a maximum precision of 1 digit, producing a displayed value such as 12.2. For more information about constructing a format specifier string, see “Configure DisplayFormat” on page 4-264.
- Optionally specify a software address method for the argument. Select or enter a value for **SwAddrMethod**. AUTOSAR software components use SwAddrMethods to group data in memory for access by calibration and measurement tools. For more information, see “Configure AUTOSAR SwAddrMethods” on page 4-42.
- To add an argument, click the **Add** button .
- To remove an argument, select the argument and then click the **Delete** button .



The displayed server operation arguments were created from the following Simulink Function block.





Nonvolatile Data Interfaces

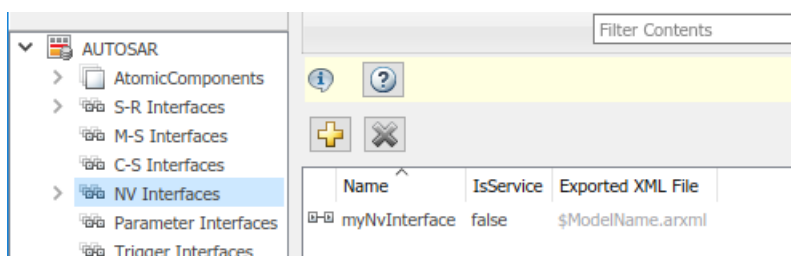
The **NV Interfaces** view in the AUTOSAR Dictionary supports modeling AUTOSAR nonvolatile (NV) data communication in Simulink. You use the AUTOSAR Dictionary to configure AUTOSAR NV ports, NV interfaces, and NV data elements in your model. For more information, see “Configure AUTOSAR Nonvolatile Data Communication” on page 4-169.

To configure AUTOSAR NV interface elements and properties, open a model for which a mapped AUTOSAR software component has been created and open the AUTOSAR Dictionary.

- 1 In the leftmost pane of the AUTOSAR Dictionary, select **NV Interfaces**.

The NV interfaces view in the AUTOSAR Dictionary lists AUTOSAR NV data interfaces and their properties. You can:

- Select an NV interface and then select a menu value to specify whether or not it is a service.
- Rename an NV interface by editing its name text.
- To add an NV interface, click the **Add** button  and use the Add Interfaces dialog box. Specify an interface name, the number of associated data elements it contains, whether the interface is a service, and the path of the Interface package.
- To remove an NV interface, select the interface and then click the **Delete** button .



- 2 In the leftmost pane of the AUTOSAR Dictionary, expand **NV Interfaces** and select an NV interface from the list.

The NV interface view in the AUTOSAR Dictionary displays the name of the selected NV data interface, whether or not it is a service, and the AUTOSAR package to be generated for the interface.

To modify the AUTOSAR package for the interface, you can do either of the following:

- Enter a package path in the **Package** parameter field.
- Click the button to the right of the **Package** field to open the AUTOSAR Package Browser. Use the browser to navigate to an existing package or create a new package. When you select a package in the browser and click **Apply**, the interface **Package** parameter value is updated with your selection. For more information about the AUTOSAR Package Browser, see “Configure AUTOSAR Package for Component, Interface, CompuMethod, or SwAddrMethod” on page 4-94.

The screenshot shows a configuration window for an NV interface. It contains the following text:

Interface Name: myNvInterface



IsService: false

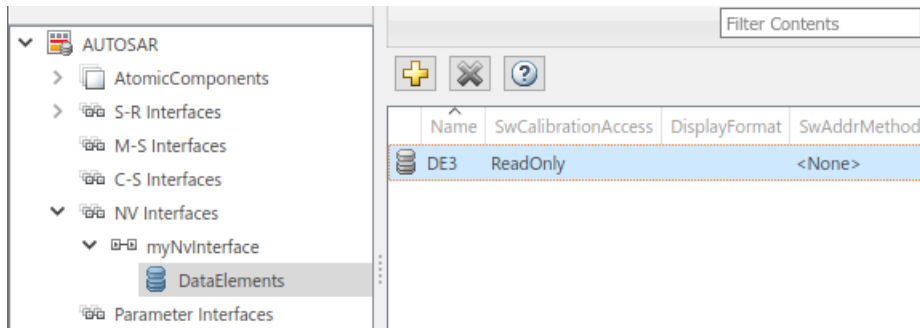
At the bottom, there is a text field labeled "Package:" containing the value "/pkg/iff" and a button with three dots to its right.

- 3 In the leftmost pane of the AUTOSAR Dictionary, expand the selected interface and select **DataElements**.

The data elements view in the AUTOSAR Dictionary lists AUTOSAR NV interface data elements and their properties. You can:

- Select an NV interface data element and edit its name value.
- Specify the level of calibration and measurement tool access to the NV interface data elements. Select a data element and set its **SwCalibrationAccess** value to **ReadOnly**, **ReadWrite**, or **NotAccessible**.
- Optionally specify the format to be used by calibration and measurement tools to display the data element. In the **DisplayFormat** field, enter an ANSI C `printf` format specifier string. For example, `%2.1d` specifies a signed decimal number, with a minimum width of 2 characters and a maximum precision of 1 digit, producing a displayed value such as 12.2. For more information about constructing a format specifier string, see “Configure DisplayFormat” on page 4-264.
- Optionally specify a software address method for the data element. Select or enter a value for **SwAddrMethod**. AUTOSAR software components use **SwAddrMethods** to group data in memory for access by calibration and measurement tools. For more information, see “Configure AUTOSAR SwAddrMethods” on page 4-42.

- To add a data element, click the **Add** button .
- To remove a data element, select the data element and then click the **Delete** button .





Parameter Interfaces

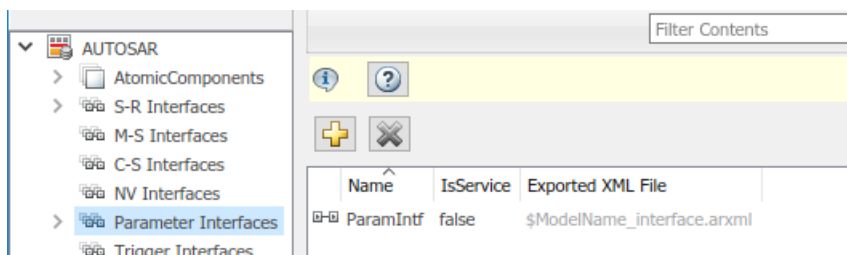
The **Parameter Interfaces** view in the AUTOSAR Dictionary supports modeling the receiver side of AUTOSAR parameter communication in Simulink. You use the AUTOSAR Dictionary to configure AUTOSAR parameter receiver ports, parameter interfaces, and parameter data elements in your model. For more information, see “Configure AUTOSAR Port Parameters for Communication with Parameter Component” on page 4-171.

To configure AUTOSAR parameter interface elements and properties, open a model for which a mapped AUTOSAR software component has been created and open the AUTOSAR Dictionary.

- 1 In the leftmost pane of the AUTOSAR Dictionary, select **Parameter Interfaces**.

The parameter interfaces view in the AUTOSAR Dictionary lists AUTOSAR parameter interfaces and their properties. You can:

- Select a parameter interface and then select a menu value to specify whether or not it is a service.
- Rename a parameter interface by editing its name text.
- To add a parameter interface, click the **Add** button  and use the Add Interfaces dialog box. Specify an interface name, the number of associated data elements it contains, whether the interface is a service, and the path of the Interface package.
- To remove a parameter interface, select the interface and then click the **Delete** button .



- 2 In the leftmost pane of the AUTOSAR Dictionary, expand **Parameter Interfaces** and select a parameter interface from the list.

The parameter interface view in the AUTOSAR Dictionary displays the name of the selected parameter interface, whether or not it is a service, and the AUTOSAR package to generate for the interface.



To modify the AUTOSAR package for the interface, you can do either of the following:

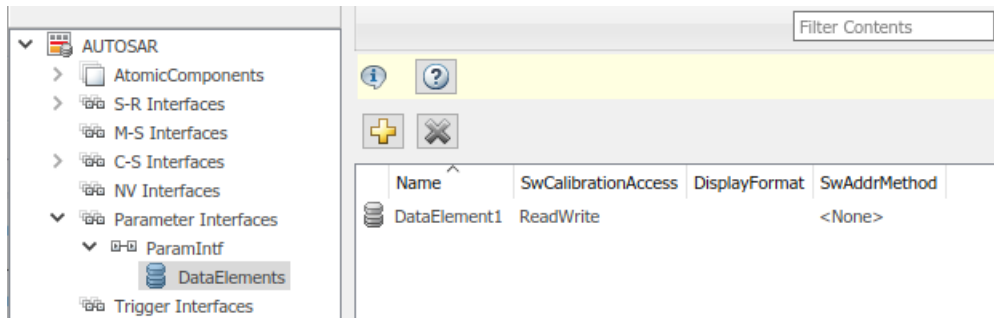
- Enter a package path in the **Package** parameter field.
- Click the button to the right of the **Package** field to open the AUTOSAR Package Browser. Use the browser to navigate to an existing package or create a new package. When you select a package in the browser and click **Apply**, the interface **Package** parameter value is updated with your selection. For more information about the AUTOSAR Package Browser, see “Configure AUTOSAR Package for Component, Interface, CompuMethod, or SwAddrMethod” on page 4-94.



- 3 In the leftmost pane of the AUTOSAR Dictionary, expand the selected interface and select **DataElements**.

The data elements view in the AUTOSAR Dictionary lists AUTOSAR parameter interface data elements and their properties. You can:

- Select a parameter interface data element and edit its name value.
- Specify the level of calibration and measurement tool access to parameter interface data elements. Select a data element and set its **SwCalibrationAccess** value to **ReadOnly**, **ReadWrite**, or **NotAccessible**.
- Optionally specify the format to be used by calibration and measurement tools to display the data element. In the **DisplayFormat** field, enter an ANSI C `printf` format specifier string. For example, `%2.1d` specifies a signed decimal number, with a minimum width of 2 characters and a maximum precision of 1 digit, producing a displayed value such as 12.2. For more information about constructing a format specifier string, see “Configure DisplayFormat” on page 4-264.
- Optionally specify a software address method for the data element. Select or enter a value for **SwAddrMethod**. AUTOSAR software components use **SwAddrMethods** to group data in memory for access by calibration and measurement tools. For more information, see “Configure AUTOSAR SwAddrMethods” on page 4-42.
- To add a data element, click the **Add** button .
- To remove a data element, select the data element and then click the **Delete** button .





Trigger Interfaces

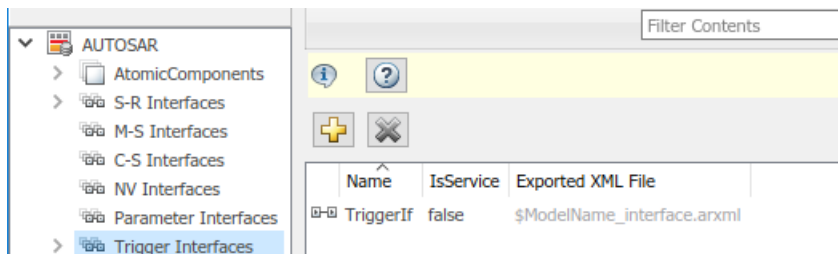
The **Trigger Interfaces** view in the AUTOSAR Dictionary supports modeling the receiver side of AUTOSAR trigger communication in Simulink. You use the AUTOSAR Dictionary to configure AUTOSAR trigger receiver ports, trigger interfaces, and triggers in your model. For more information, see “Configure Receiver for AUTOSAR External Trigger Event Communication” on page 4-175.

To configure AUTOSAR trigger interface elements and properties, open a model for which a mapped AUTOSAR software component has been created and open the AUTOSAR Dictionary.

- 1 In the leftmost pane of the AUTOSAR Dictionary, select **Trigger Interfaces**.

The trigger interfaces view in the AUTOSAR Dictionary lists AUTOSAR trigger interfaces and their properties. You can:

- Select a trigger interface and then select a menu value to specify whether or not it is a service.
- Rename a trigger interface by editing its name text.
- To add a trigger interface, click the **Add** button  and use the Add Interfaces dialog box. Specify an interface name, the number of associated triggers it contains, whether the interface is a service, and the path of the Interface package.
- To remove a trigger interface, select the interface and then click the **Delete** button .

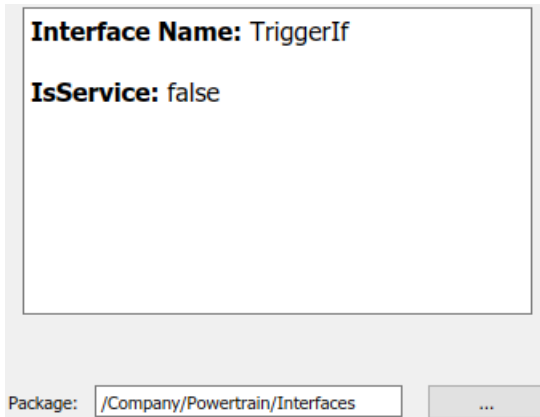


- 2 In the leftmost pane of the AUTOSAR Dictionary, expand **Trigger Interfaces** and select a trigger interface from the list.

The trigger interface view in the AUTOSAR Dictionary displays the name of the selected trigger interface, whether or not it is a service, and the AUTOSAR package to be generated for the interface.

To modify the AUTOSAR package for the interface, you can do either of the following:

- Enter a package path in the **Package** parameter field.
- Click the button to the right of the **Package** field to open the AUTOSAR Package Browser. Use the browser to navigate to an existing package or create a new package. When you select a package in the browser and click **Apply**, the interface **Package** parameter value is updated with your selection. For more information about the AUTOSAR Package Browser, see “Configure AUTOSAR Package for Component, Interface, CompuMethod, or SwAddrMethod” on page 4-94.



- 3 In the leftmost pane of the AUTOSAR Dictionary, expand the selected interface and select **Triggers**.



The triggers view in the AUTOSAR Dictionary lists AUTOSAR triggers and their properties. You can:

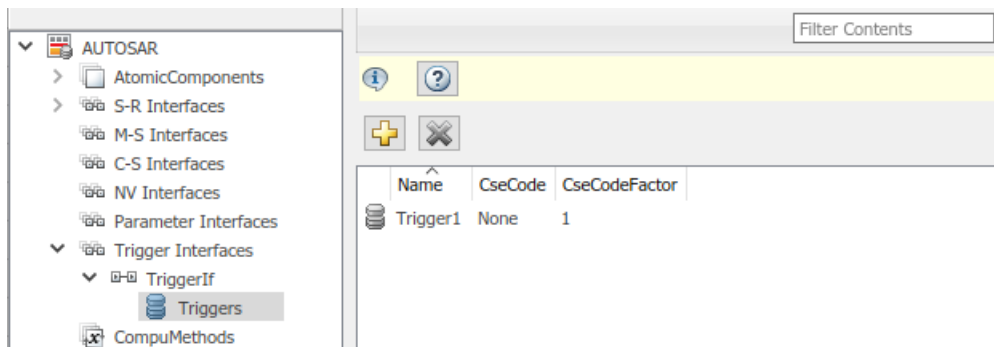
- Select a trigger and edit its name value.
- If the trigger is periodic, you can use **CseCode** and **CseCodeFactor** to specify a period for the trigger. (Otherwise, leave the period unspecified.)
 - To specify the time base of the period, select a value from the **CseCode** menu. The values are based on ASAM codes for scaling unit (CSE).
 - To specify the scaling factor for the period, enter an integer value in the **CseCodeFactor** field.

For example, to specify a period of 15 milliseconds, set **CseCode** to CSE3 (1 millisecond) and set **CseCodeFactor** to 15.

CseCode	Time Base
None	Unspecified (trigger is not periodic)
CSE0	1 μ sec (microsecond)
CSE1	10 μ sec
CSE2	100 μ sec
CSE3	1 msec (millisecond)
CSE4	10 msec
CSE5	100 msec

CseCode	Time Base
CSE6	1 second
CSE7	10 seconds
CSE8	1 minute
CSE9	1 hour
CSE10	1 day
CSE20	1 fs (femtosecond)
CSE21	10 fs
CSE22	100 fs
CSE23	1 ps (picosecond)
CSE24	10 ps
CSE25	100 ps
CSE26	1 ns (nanosecond)
CSE27	10 ns
CSE28	100 ns
CSE100	Angular degrees
CSE101	Revolutions (1 = 360 degrees)
CSE102	Cycle (1 = 720 degrees)
CSE997	Computing cycle
CSE998	When frame available
CSE999	Always when there is a new value
CSE1000	Nondeterministic (no fixed scaling)

- To add a trigger, click the **Add** button .
- To remove a trigger, select the trigger and then click the **Delete** button .





Configure AUTOSAR Computation Methods

The **CompuMethods** view in the AUTOSAR Dictionary supports modeling AUTOSAR computation methods (CompuMethods), which specify conversions between internal values and physical representation of AUTOSAR data, in Simulink. You use the AUTOSAR Dictionary to create and

configure AUTOSAR CompuMethods. For more information, see “Configure AUTOSAR CompuMethods” on page 4-236.

To configure AUTOSAR CompuMethod elements and properties, open a model for which a mapped AUTOSAR software component has been created and open the AUTOSAR Dictionary. Select **CompuMethods**.

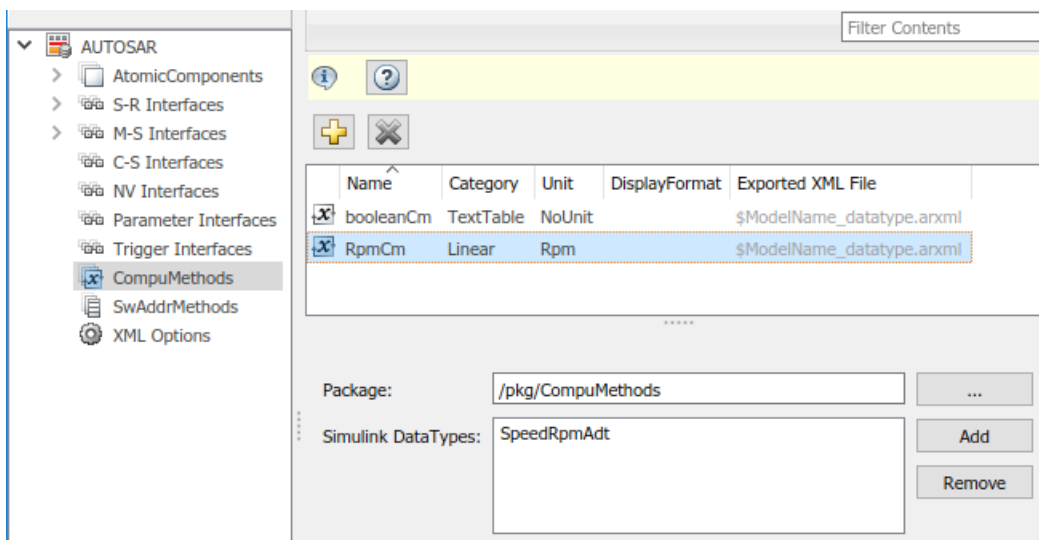
The CompuMethods view in the AUTOSAR Dictionary displays CompuMethods and their properties. You can:

- Select a CompuMethod and modify properties, such as name, category, unit, display format for calibration and measurement, AUTOSAR package to be generated for the CompuMethod, and a list of Simulink data types that reference the CompuMethod. For property descriptions, see “Configure AUTOSAR CompuMethod Properties” on page 4-236.
- To add a CompuMethod, click the **Add** button  and use the Add CompuMethod dialog box, which is described below.
- To remove a CompuMethod, select the CompuMethod and then click the **Delete** button .

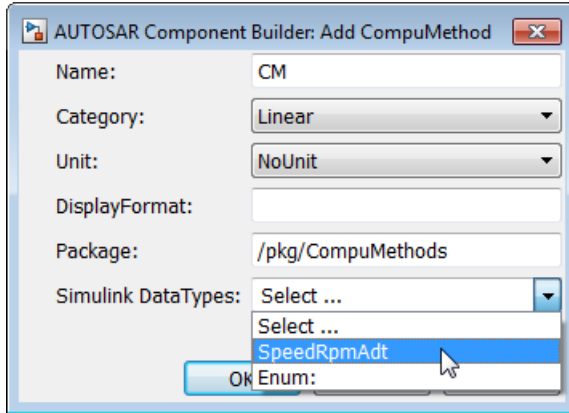
To modify the AUTOSAR package for a CompuMethod, you can do either of the following:

- Enter a package path in the **Package** parameter field.
- Click the button to the right of the **Package** field to open the AUTOSAR Package Browser. Use the browser to navigate to an existing package or create a new package. When you select a package in the browser and click **Apply**, the CompuMethod **Package** parameter value is updated with your selection. For more information about the AUTOSAR Package Browser, see “Configure AUTOSAR Package for Component, Interface, CompuMethod, or SwAddrMethod” on page 4-94.

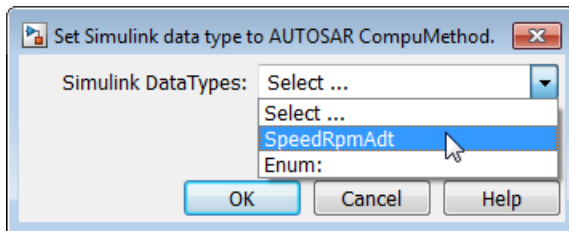
To associate a CompuMethod with a Simulink data type used in the model, select a CompuMethod and click the **Add** button to the right of **Simulink DataTypes**. This action opens a dialog box with a list of available data types. To add a data type to the **Simulink DataTypes** list, select the data type and click **OK**. To remove a data type from the **Simulink DataTypes** list, select the data type and click **Remove**.



The Add CompuMethod dialog box lets you create a CompuMethod and specify its initial properties, such as name, category, unit, display format for calibration and measurement, AUTOSAR package to be generated for the CompuMethod, and a Simulink data type that references the CompuMethod.



Clicking the **Add** button to the right of **Simulink DataTypes** opens the Set Simulink data type to AUTOSAR CompuMethod dialog box. This dialog box lets you select a Simulink data type to add to **Simulink DataTypes**, the list of Simulink data types that reference a CompuMethod. In the list of available data types, select a `Simulink.NumericType` or `Simulink.AliasType`, or enter the name of a Simulink enumerated type.



Configure AUTOSAR SwAddrMethods

The **SwAddrMethods** view in the AUTOSAR Dictionary supports modeling AUTOSAR software address methods (SwAddrMethods). AUTOSAR software components use SwAddrMethods to group data and function definitions in memory, primarily for efficiency, performance, and data access by run-time calibration tools. In the AUTOSAR Dictionary, you can view or create AUTOSAR SwAddrMethods and then assign SwAddrMethods to data and functions that you want to group together. For more information, see “Configure SwAddrMethod” on page 4-267.



To configure AUTOSAR SwAddrMethod elements and properties, open a model for which a mapped AUTOSAR software component has been created and open the AUTOSAR Dictionary. Select **SwAddrMethods**.

The SwAddrMethods view in the AUTOSAR Dictionary displays SwAddrMethods and their properties. You can:

- Select a SwAddrMethod and modify properties, such as name, section type, and AUTOSAR package to be generated for the SwAddrMethod.

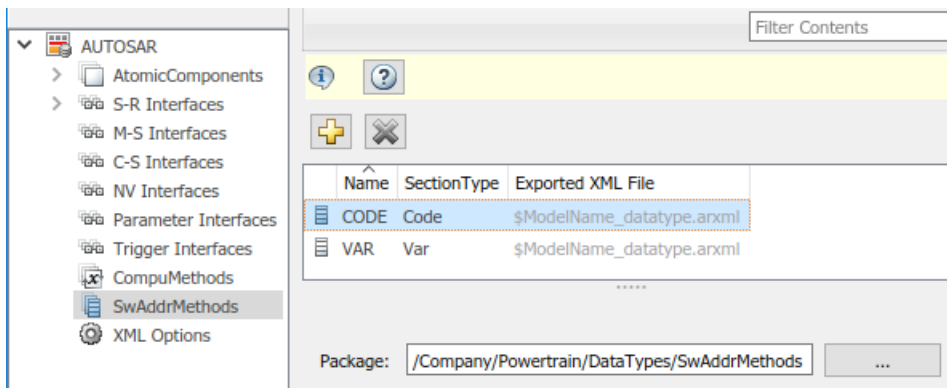
To modify the section type, select a value from the **SectionType** drop-down list. The listed values correspond to SwAddrMethod section types listed in the AUTOSAR standard.

SectionType Value	SwAddrMethod Section Type
CalibrationVariables	CALIBRATION-VARIABLES
Calprm	CALPRM
Code	CODE
ConfigData	CONFIG-DATA
Const	CONST
ExcludeFromFlash	EXCLUDE-FROM-FLASH
Var	VAR

- To add a SwAddrMethod, click the **Add** button  and use the Add SwAddrMethod dialog box. Specify a SwAddrMethod name, a section type, and the path of the SwAddrMethod package.
- To remove a SwAddrMethod, select the SwAddrMethod and then click the **Delete** button .

To modify the AUTOSAR package for a SwAddrMethod, you can do either of the following:

- Enter a package path in the **Package** parameter field.
- Click the button to the right of the **Package** field to open the AUTOSAR Package Browser. Use the browser to navigate to an existing package or create a new package. When you select a package in the browser and click **Apply**, the SwAddrMethod **Package** parameter value is updated with your selection. For more information about the AUTOSAR Package Browser, see “Configure AUTOSAR Package for Component, Interface, CompuMethod, or SwAddrMethod” on page 4-94.



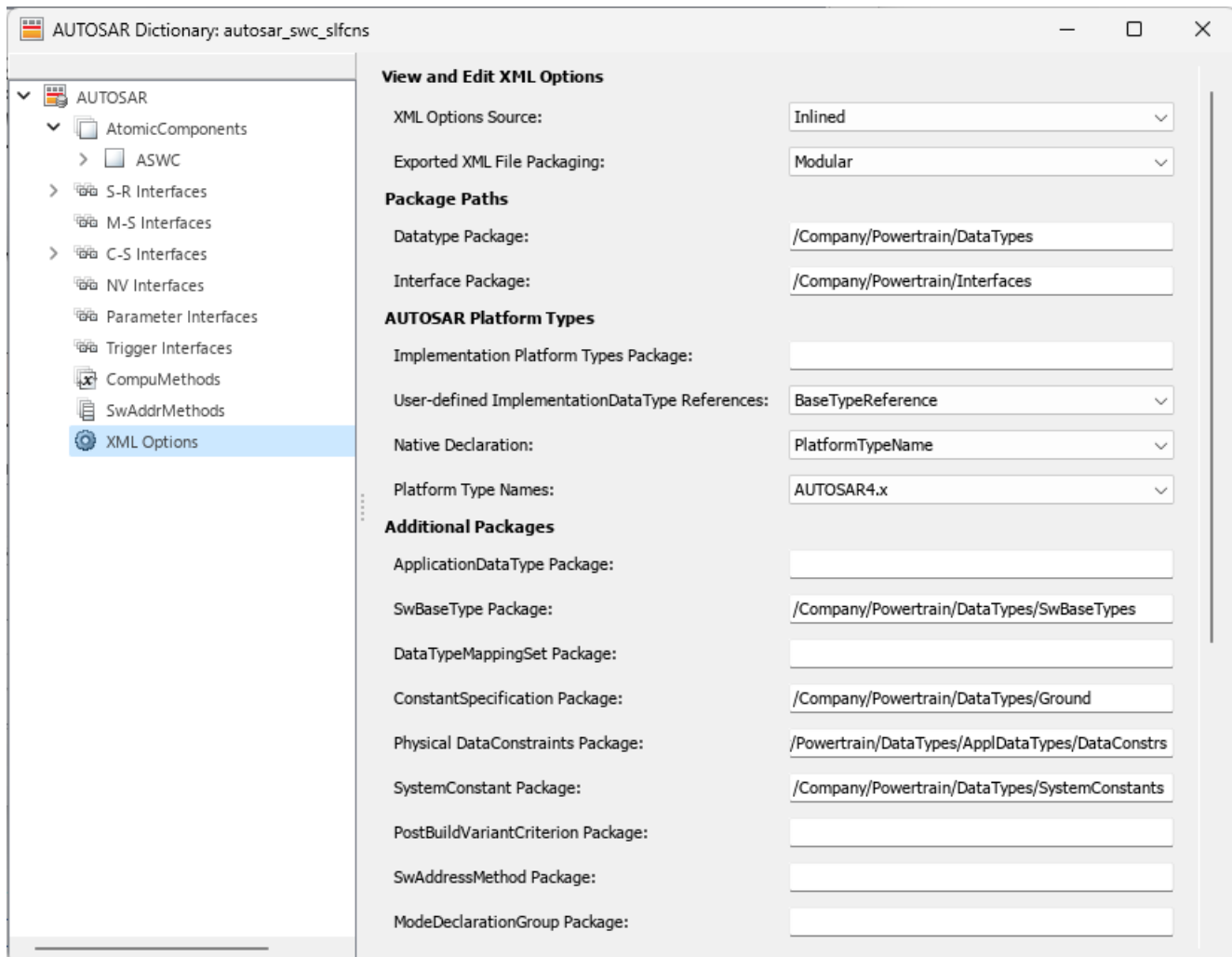
Configure AUTOSAR XML Options

To configure AUTOSAR XML options for ARXML file export, open a model for which a mapped AUTOSAR software component has been created and open the AUTOSAR Dictionary. Select **XML Options**.

The XML options view in the AUTOSAR Dictionary displays XML export parameters and their values. You can configure:

- XML options source (for components in architecture modeling)
- XML file packaging for AUTOSAR elements created in Simulink
- AUTOSAR package paths

- AUTOSAR platform types
- Aspects of exported AUTOSAR XML content



- “XML Options Source” on page 4-44
- “Exported XML File Packaging” on page 4-45
- “AUTOSAR Package Paths” on page 4-46
- “AUTOSAR Platform Types” on page 4-47
- “Additional XML Options” on page 4-48

XML Options Source

The XML options view displays the parameter **XML Options Source**. If the current component model is contained in an AUTOSAR architecture model, this parameter indicates which XML options to use in model builds. Specify `Inherit` from `AUTOSAR architecture model` to use shared architecture model XML option settings, which promote consistency across the model hierarchy. Specify `Inlined` in this model to override the shared settings with the component model local XML option settings.

If the current component model is not contained in an AUTOSAR architecture model, the **XML Options Source** parameter has no effect.

Alternatively, you can programmatically configure the XML options source by calling the AUTOSAR `set` function. For property `XmlOptionsSource`, specify either `Inlined` or `Inherit`. For example:

```
arProps = autosar.api.getAUTOSARProperties(hModel);
set(arProps, 'XmlOptions', 'XmlOptionsSource', 'Inlined');
```

For more information about architecture model XML options, see “Generate and Package AUTOSAR Composition XML Descriptions and Component Code” on page 8-43.

Exported XML File Packaging

In the XML options view, you can specify the granularity of XML file packaging for AUTOSAR elements created in Simulink. (Imported AUTOSAR XML files retain their file structure, as described in “Round-Trip Preservation of AUTOSAR XML File Structure and Element Information” on page 3-37.) Select one of the following values for **Exported XML file packaging**.

- **Single file** — Exports XML into a single file, `modelName.arxml`.
- **Modular** — Exports XML into multiple files, named according to the type of information contained.

Exported File Name	By Default Contains...
<code>modelName_component.arxml</code>	<p>Software components, including:</p> <ul style="list-style-type: none"> • Ports • Events • Runnables • Inter-runnable variables (IRVs) • Included data type sets • Component-scoped parameters and variables <p>This is the main ARXML file exported for the Simulink model. In addition to software components, the component file contains packageable elements that the exporter does not move to data type, implementation, interface, or timing files based on AUTOSAR element category.</p>

Exported File Name	By Default Contains...
<code>modelname_datatype.arxml</code>	Data types and related elements, including: <ul style="list-style-type: none"> • Application data types • Software base types • Data type mapping sets • Constant specifications • Physical data constraints • System constants • Software address methods • Mode declaration groups • Computation methods • Units and unit groups • Software record layouts • Internal data constraints
<code>modelname_implementation.arxml</code>	Software component implementation, including code descriptors.
<code>modelname_interface.arxml</code>	Interfaces, including S-R, C-S, M-S, NV, parameter, and trigger interfaces. The interfaces include type-specific elements, such as S-R data elements, C-S operations, port-based parameters, or triggers.
<code>modelname_timing.arxml</code>	Timing model, including runnable execution order constraints.

Alternatively, you can programmatically configure exported XML file packaging by calling the AUTOSAR `set` function. For property `ArxmlFilePackaging`, specify either `SingleFile` or `Modular`. For example:

```
arProps = autosar.api.getAUTOSARProperties(hModel);
set(arProps, 'XmlOptions', 'ArxmlFilePackaging', 'SingleFile');
```

For more information, see “Generate AUTOSAR C and XML Files” on page 5-9.

AUTOSAR Package Paths

In the XML options view, you can configure AUTOSAR packages (AR-PACKAGES), which contain groups of AUTOSAR elements and reside in a hierarchical AR-PACKAGE structure. (The AR-PACKAGE structure for a component is logically distinct from the ARXML file partitioning selected with the XML option **Exported XML file packaging** or imported from AUTOSAR XML files.) For more information about AUTOSAR packages, see “Configure AUTOSAR Packages” on page 4-84.

Inspect and modify the AUTOSAR package paths:

- Grouped under the heading **Package Paths**.

The screenshot shows a dialog box titled "Package Paths". It contains two rows of configuration options:

- Datatype Package:** The text input field contains the path `/Company/Powertrain/DataTypes`.
- Interface Package:** The text input field contains the path `/Company/Powertrain/Interfaces`.

- Grouped under the heading **Additional Packages**.

Additional Packages	
ApplicationDataType Package:	<input type="text" value="/Company/Powertrain/AplDataTypes"/>
SwBaseType Package:	<input type="text" value="/Company/Powertrain/SwBaseTypes"/>
DataTypeMappingSet Package:	<input type="text" value="/Company/Powertrain/DataTypeMappings"/>
ConstantSpecification Package:	<input type="text" value="/Company/Powertrain/Constants"/>
Physical DataConstraints Package:	<input type="text" value="/Company/Powertrain/DataConstrs"/>
SystemConstant Package:	<input type="text" value="/Company/Powertrain/SystemConstants"/>
PostBuildVariantCriterion Package:	<input type="text"/>
SwAddressMethod Package:	<input type="text"/>
ModeDeclarationGroup Package:	<input type="text"/>
CompuMethod Package:	<input type="text" value="/Company/Powertrain/CompuMethods"/>
Unit Package:	<input type="text" value="/Company/Powertrain/Units"/>
SwRecordLayout Package:	<input type="text"/>
Internal DataConstraints Package:	<input type="text" value="/Company/Powertrain/DataTypes/DataConstrs"/>

Alternatively, you can programmatically configure an AUTOSAR package path by calling the AUTOSAR set function. Specify a package property name and a package path. For example:

```
arProps = autosar.api.getAUTOSARProperties(hModel);
set(arProps, 'XmlOptions', 'ApplicationDataTypePackage', '/Company/Powertrain/DataTypes/AplDataTypes');
```

For more information about AUTOSAR package property names and defaults, see “Configure AUTOSAR Packages and Paths” on page 4-85.

AUTOSAR Platform Types

In the XML options view, you can configure aspects of the AUTOSAR platform type packaging and naming behavior.

AUTOSAR Platform Types	
Implementation Platform Types Package:	<input type="text"/>
User-defined ImplementationDataType References:	<input type="text" value="BaseTypeReference"/>
Native Declaration:	<input type="text" value="PlatformTypeName"/>
Platform Type Names:	<input type="text" value="AUTOSAR4.x"/>

You can:

- Specify the top-level package name for AUTOSAR implementation platform types and base types by entering the package name in the **Implementation Platform Types Package** field.

Implementation platform types are grouped as an `ImplementationDataTypes` subpackage. Base types are grouped as a `BaseTypes` subpackage.

For Modular ARXML export, the top-level package and its content is exported to the `stub/modelname_platformtypes.arxml` file.

- Specify the implementation data type reference behavior. Select either of the following values for **User-defined Implementation Types References**:

- **PlatformTypeReference** — User-defined implementation data types reference an AUTOSAR implementation data type (CATEGORY is set to TYPE_REFERENCE in the ARXML).
- **BaseTypeReference** — User-defined implementation data types reference an SW base type (CATEGORY is set to VALUE in the ARXML).
- Control whether the native declaration inherits the AUTOSAR platform type name or uses a C integral type name. Select either of the following values for **Native Declaration**:
 - **PlatformTypeName** — The native declaration inherits the AUTOSAR platform type name.
 - **CIntegralTypeName** — The native declaration uses a C integral type name according to the hardware configuration specified in the model settings.
- Control the schema version of the platform type names in the exported ARXML. Select one of the following values for **Platform Type Names**:
 - **AUTOSAR4.x**— Platform type names comply with the AUTOSAR 4.x platform type naming specification.
 - **AUTOSAR3.x**— Platform type names comply with the AUTOSAR 3.x platform type naming specification.

Not recommended starting in R2023a

Support for AUTOSAR 3.x platform names will be removed in a future release.

Alternatively, you can programmatically configure AUTOSAR platform type XML options by calling the AUTOSAR `set` function. Specify a property name and value. The valid property names are `PlatformDataTypePackage`, `UsePlatformTypeReferences`, `NativeDeclaration`, and `PlatformTypeNames`. For example:

```
arProps = autosar.api.getAUTOSARProperties(hModel);

set(arProps, 'XmlOptions', 'PlatformDataTypePackage', '/AUTOSAR_PlatformTypes');
set(arProps, 'XmlOptions', 'UsePlatformTypeReferences', 'PlatformTypeReference');
set(arProps, 'XmlOptions', 'NativeDeclaration', 'PlatformTypeName');
set(arProps, 'XmlOptions', 'PlatformTypeNames', 'AUTOSAR4.x');
```

Additional XML Options

In the XML options view, under the heading **Additional Options**, you can configure aspects of exported AUTOSAR XML content.

Additional Options	
ImplementationDataType Reference:	Allowed
SwCalibrationAccess DefaultValue:	ReadWrite
CompuMethod Direction:	InternalToPhys
Internal DataConstraints Export:	<input checked="" type="checkbox"/>

You can:

- Optionally override the default behavior for generating AUTOSAR application data types in ARXML code. To force generation of an application data type for each AUTOSAR data type, change the value of **ImplementationDataType Reference** from `Allowed` to `NotAllowed`. For more information, see “Control Application Data Type Generation” on page 4-244.
- Control the default value of the **SwCalibrationAccess** property of generated AUTOSAR measurement variables, calibration parameters, and signal and parameter data objects. Select one of the following values for **SwCalibrationAccess DefaultValue**:

- **ReadOnly** — Read access only.
- **ReadWrite** (default) — Read and write access.
- **NotAccessible** — Not accessible with calibration and measurement tools.

For more information, see “Configure SwCalibrationAccess” on page 4-262.

- Control the direction of CompuMethod conversion for linear-function CompuMethods. Select one of the following values for **CompuMethod Direction**:
 - **InternalToPhys** (default) — Generate CompuMethod sections for conversion of internal values into their physical representations.
 - **PhysToInternal** — Generate CompuMethod sections for conversion of physical values into their internal representations.
 - **Bidirectional** — Generate CompuMethod sections for both internal-to-physical and physical-to-internal conversion directions.

For more information, see “Configure CompuMethod Direction for Linear Functions” on page 4-238.

- Optionally override the default behavior for generating internal data constraint information for AUTOSAR implementation data types in ARXML code. To force export of internal data constraints for implementation data types, select the option **Internal DataConstraints Export**. For more information, see “Configure AUTOSAR Internal Data Constraints Export” on page 4-246.

Alternatively, you can programmatically configure the additional XML options by calling the AUTOSAR `set` function. Specify a property name and value. The valid property names are `ImplementationTypeReference`, `SwCalibrationAccessDefault`, `CompuMethodDirection`, and `InternalDataConstraintExport`. For example:

```
arProps = autosar.api.getAUTOSARProperties(hModel);
set(arProps, 'XmlOptions', 'ImplementationTypeReference', 'NotAllowed');
set(arProps, 'XmlOptions', 'SwCalibrationAccessDefault', 'ReadOnly');
set(arProps, 'XmlOptions', 'CompuMethodDirection', 'PhysToInternal');
set(arProps, 'XmlOptions', 'InternalDataConstraintExport', true);
```

See Also

Related Examples

- “Map AUTOSAR Elements for Code Generation” on page 4-50
- “Configure and Map AUTOSAR Component Programmatically” on page 4-293
- “Configure AUTOSAR Code Generation” on page 5-7

More About

- “AUTOSAR Component Configuration” on page 4-3

Map AUTOSAR Elements for Code Generation

In Simulink, you can use the Code Mappings editor and the AUTOSAR Dictionary separately or together to graphically configure an AUTOSAR software component and map Simulink model elements to AUTOSAR component elements. For more information, see “AUTOSAR Component Configuration” on page 4-3.

Use the Code Mappings editor to map Simulink model elements to AUTOSAR component elements from a Simulink model perspective. The editor display consists of several tabbed tables, including **Functions**, **Inports**, and **Outports**. Use the tables to select Simulink elements and map them to corresponding AUTOSAR elements. The mappings that you configure are reflected in generated AUTOSAR-compliant C code and exported ARXML descriptions.

The Code Mappings editor also provides mapping for submodels referenced from AUTOSAR software component models. For more information, see “Map Calibration Data for Submodels Referenced from AUTOSAR Component Models” on page 4-65.

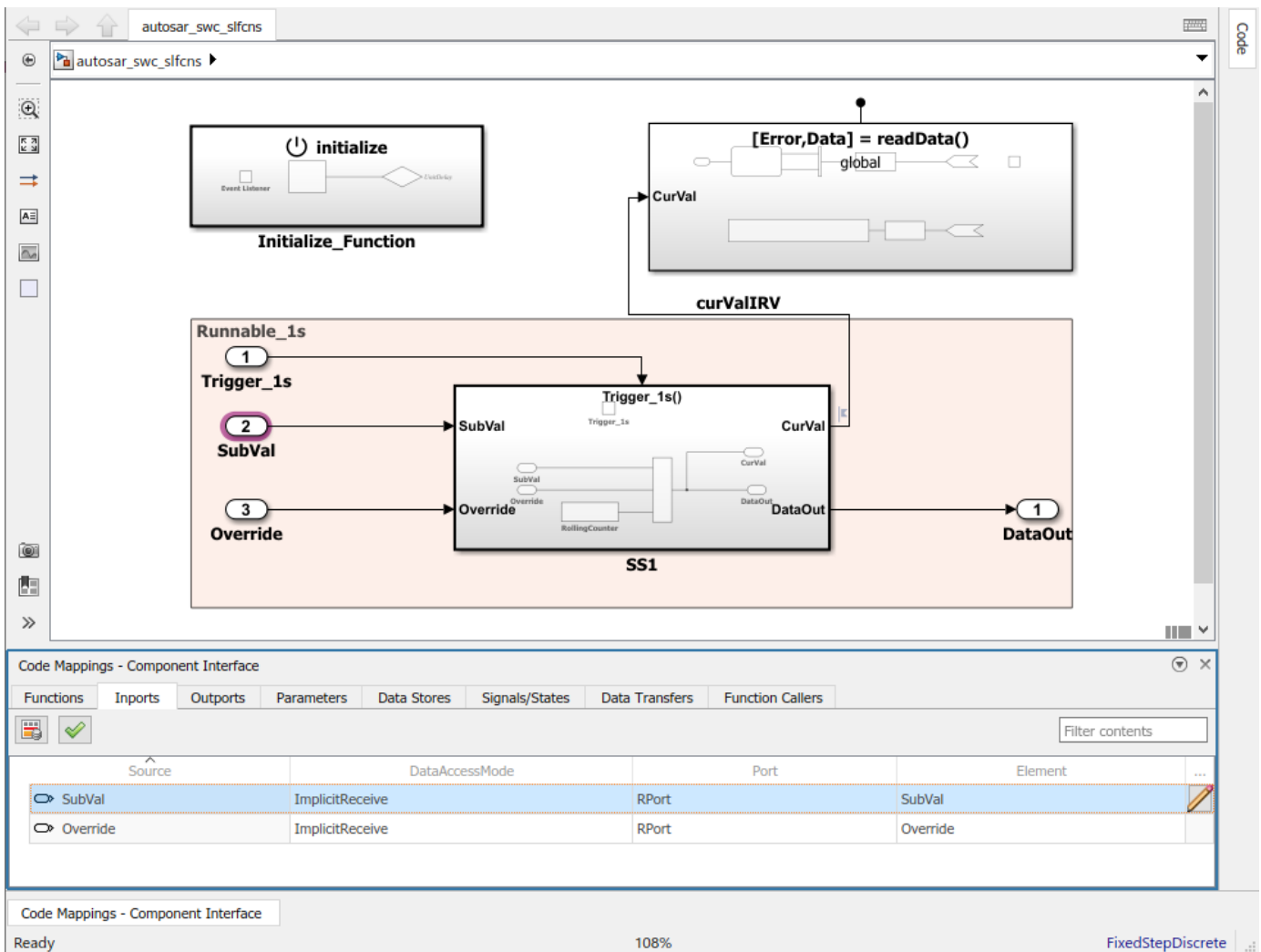
Simulink to AUTOSAR Mapping Workflow

To map Simulink model elements to AUTOSAR software component elements:

- 1 Open a model for which AUTOSAR system target file `autosar.tlc` is selected.
- 2 Create or open a mapped view of the AUTOSAR model. In the model window, do one of the following:
 - From the **Apps** tab, open the AUTOSAR Component Designer app.
 - Click the perspective control in the lower-right corner and select **Code**.

If the model has not yet been mapped to an AUTOSAR software component, the AUTOSAR Component Quick Start opens. To configure the model for AUTOSAR component development, work through the quick-start procedure and click **Finish**. For more information, see “Create Mapped AUTOSAR Component with Quick Start” on page 3-2.

The model opens in the AUTOSAR Code perspective. This perspective displays the model and directly below the model, the Code Mappings editor.




The Code Mappings editor provides in-canvas access to AUTOSAR mapping information, with batch editing, element filtering, easy navigation to model elements and AUTOSAR properties, and model element traceability. To view and modify additional AUTOSAR attributes for an

element, select the element and click the  icon.

3 Navigate the Code Mappings editor tabs to perform these actions:

- Map a Simulink entry-point function to an AUTOSAR runnable.
- Map a Simulink inport or outport to an AUTOSAR receiver or sender port and a sender-receiver data element, with a specific data access mode.
- Map a Simulink model workspace parameter to an AUTOSAR component parameter.
- Map a Simulink data store to an AUTOSAR variable.
- Map a Simulink block signal or state to an AUTOSAR variable.
- Map a Simulink data transfer line to an AUTOSAR inter-runnable variable (IRV).
- Map a Simulink function caller to an AUTOSAR client port and a client-server operation.


Use the **Filter contents** field (where available) to selectively display some elements, while omitting others, in the current view.

- 4 After mapping model elements, click the **Validate** button  to validate the AUTOSAR component configuration. If errors are reported, address them and then retry validation.

Map Entry-Point Functions to AUTOSAR Runnables

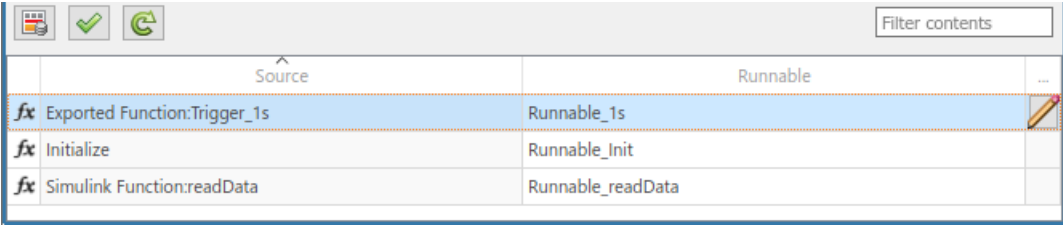
The **Functions** tab of the Code Mappings editor supports modeling AUTOSAR runnable entities (runnables) in Simulink. After using the AUTOSAR Dictionary to create AUTOSAR runnables and AUTOSAR events, which implement aspects of internal behavior in an AUTOSAR component, open the Code Mappings editor. Use the **Functions** tab to map Simulink entry-point functions to AUTOSAR runnables.




For more information, see “Configure AUTOSAR Runnables and Events” on page 4-178.

The **Functions** tab of the Code Mappings editor maps each Simulink entry-point function to an AUTOSAR runnable. Click the **Update** button  to load or update Simulink entry-point functions in the model.

In the **Functions** tab, you can:


- Map a Simulink entry-point function by selecting the entry-point function and then selecting a menu value for an AUTOSAR runnable, among those listed for the AUTOSAR component.



Source	Runnable
 Exported Function:Trigger_1s	Runnable_1s
 Initialize	Runnable_Init
 Simulink Function:readData	Runnable_readData

- Specify software address methods (SwAddrMethods) for the runnable function code and internal data. If you specify SwAddrMethod names, code generation uses the names to group runnable function and data definitions in memory sections. For more information, see “Configure SwAddrMethod” on page 4-267.

The SwAddrMethod names must be defined in the model. For example, the example model `autosar_swc_counter` defines SwAddrMethods named `CODE` and `VAR`.

To specify SwAddrMethods for a runnable, select the corresponding entry-point function and click the  icon. The dialog displays the code attributes **SwAddrMethod** and **Internal Data SwAddrMethod** for the selected function. Select SwAddrMethod names among the valid values listed for each property.

To create additional SwAddrMethod names in the component, use the AUTOSAR Dictionary, SwAddrMethods view. For more information, see “Configure AUTOSAR SwAddrMethods” on page 4-42.

Note Code generation for runnable internal data `SwAddrMethods` requires setting the model configuration option **Code Generation > Interface > Generate separate internal data per entry-point function** (GroupInternalDataByFunction) to on.

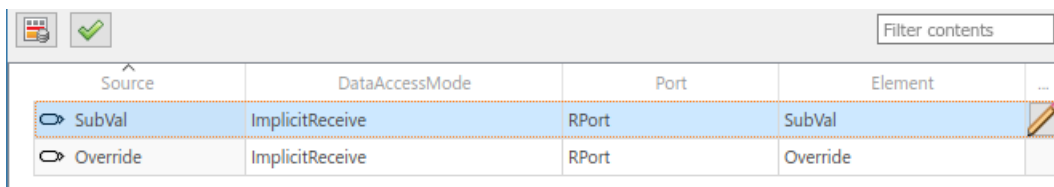
Map Inports and Outports to AUTOSAR Sender-Receiver Ports and Data Elements

The **Inports** and **Outports** tabs of the Code Mappings editor support modeling AUTOSAR sender-receiver (S-R) communication in Simulink. After using the AUTOSAR Dictionary to create AUTOSAR S-R ports, S-R interfaces, and S-R data elements in your model, open the Code Mappings editor. Use the **Inports** and **Outports** tabs to map Simulink root inports and outports to AUTOSAR receiver and sender ports and AUTOSAR S-R data elements.


For more information, see “Configure AUTOSAR Sender-Receiver Communication” on page 4-96 and “Configure AUTOSAR Queued Sender-Receiver Communication” on page 4-112.

The **Inports** tab of the Code Mappings editor maps each Simulink root inport to an AUTOSAR receiver port and an S-R interface data element. In the **Inports** tab, you can:

- Map a Simulink inport by selecting the inport and then selecting menu values for an AUTOSAR port and an AUTOSAR element, among those listed for the AUTOSAR component.
- Select an AUTOSAR data access mode for the port: `ImplicitReceive`, `ExplicitReceive`, `ExplicitReceiveByVal`, `QueuedExplicitReceive`, `ErrorStatus`, `IsUpdated`, `EndToEndRead`, or `ModeReceive`.



Source	DataAccessMode	Port	Element
SubVal	ImplicitReceive	RPort	SubVal
Override	ImplicitReceive	RPort	Override

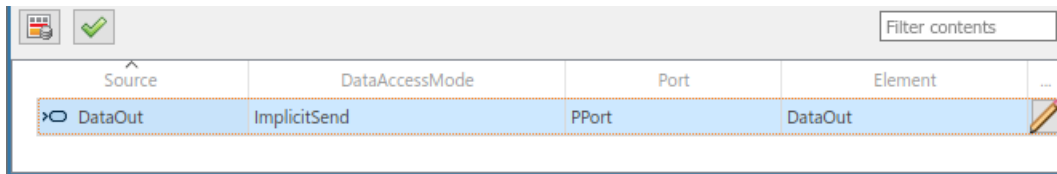
To view additional port communication specification (ComSpec) attributes, select an inport and click the  icon.


- For AUTOSAR nonqueued receiver ports, you can modify ComSpec attributes `AliveTimeout`, `HandleNeverReceived`, and `InitValue`.
- For queued receiver ports, you can modify ComSpec attribute `QueueLength`.

For more information, see “Configure AUTOSAR Sender-Receiver Port ComSpecs” on page 4-108.

The **Outports** tab of the Code Mappings editor maps each Simulink root outport to an AUTOSAR sender port and an S-R interface data element. In the **Outports** tab, you can:

- Map a Simulink outport by selecting the outport and then selecting menu values for an AUTOSAR port and an AUTOSAR element.
- Select an AUTOSAR data access mode for the port: `ImplicitSend`, `ImplicitSendByRef`, `ExplicitSend`, `QueuedExplicitSend`, `EndToEndWrite`, or `ModeSend`.



To view additional port communication specification (ComSpec) attributes, select an outpost that maps to an AUTOSAR nonqueued sender port and click the  icon. In the dialog, you can modify the ComSpec attribute `InitValue`. For more information, see “Configure AUTOSAR Sender-Receiver Port ComSpecs” on page 4-108.

Map Model Workspace Parameters to AUTOSAR Component Parameters

On the **Parameters** tab of the Code Mappings editor, you can map Simulink model workspace parameters to AUTOSAR parameters for AUTOSAR run-time calibration. Examples of model workspace parameters you can map include:

- Simulink parameter objects
- Simulink lookup table objects
- Simulink breakpoint objects

By mapping lookup table and breakpoint objects to AUTOSAR calibration parameters, you can model AUTOSAR parameters for integrated and distributed lookups. For more information, see “Configure Lookup Tables for AUTOSAR Calibration and Measurement” on page 4-273.

After creating model workspace parameters, for example, using Model Explorer, open the Code Mappings editor and select the **Parameters** tab. Select Simulink model workspace parameters and map them to:

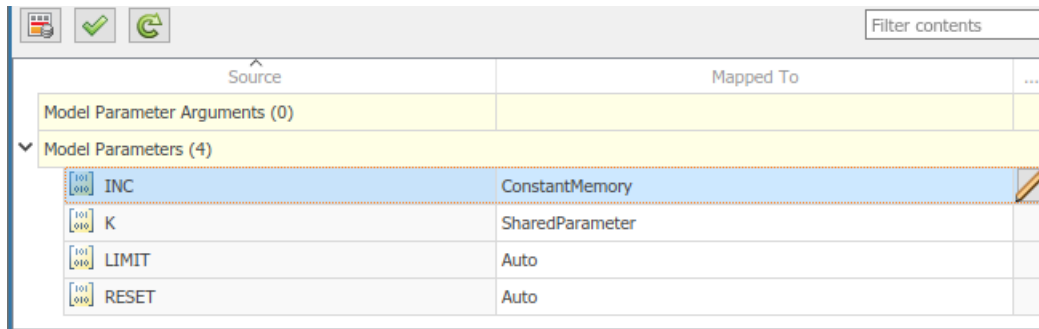
- AUTOSAR component internal parameters, such as constant memory, shared parameters, or per-instance parameters.
- AUTOSAR port-based parameters, used by parameter receiver components for port-based access to parameter data.

For more information, see “Configure AUTOSAR Constant Memory” on page 4-210, “Configure AUTOSAR Shared or Per-Instance Parameters” on page 4-212, and “Configure AUTOSAR Port Parameters for Communication with Parameter Component” on page 4-171.

The **Parameters** tab lists each Simulink model workspace parameter that you can map to an AUTOSAR parameter. On the **Parameters** tab:

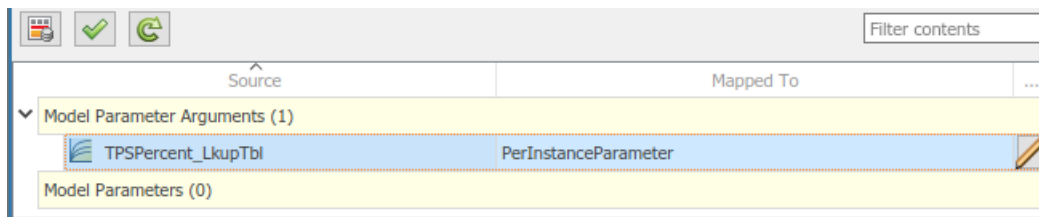
- If a Simulink model workspace parameter is not configured as a model argument (that is, not unique to each instance of a multi-instance model), you can map the parameter by selecting it and then selecting a menu value for an AUTOSAR parameter type. For this workflow, the valid parameter types are `ConstantMemory`, `SharedParameter`, or `Auto`. To accept software mapping defaults, specify `Auto`.


For example, here is the **Parameters** tab for example model `autosar_sw_counter`.



- If a Simulink model workspace parameter is configured as a model argument (that is, unique to each instance of a multi-instance model), map the parameter by selecting it and then selecting a menu value for an AUTOSAR parameter type. For this workflow, the valid parameter types are PerInstanceParameter, PortParameter, or Auto. To accept software mapping defaults, specify Auto.

For example, here is the **Parameters** tab for example model `autosar_sw_c_throttle_sensor`. Example model `autosar_composition` contains two instances of `autosar_sw_c_throttle_sensor`.



- If you select a parameter type other than Auto, you can click the  icon to view and modify other code and calibration attributes for the parameter.

Attribute	Purpose
Const (ConstantMemory only)	Select or clear the option to indicate whether to include C type qualifier <code>const</code> in generated code for the AUTOSAR parameter. For more information, see “Specify C Type Qualifiers for AUTOSAR Static and Constant Memory” on page 4-61.
Volatile (ConstantMemory only)	Select or clear the option to indicate whether to include C type qualifier <code>volatile</code> in generated code for the AUTOSAR parameter. For more information, see “Specify C Type Qualifiers for AUTOSAR Static and Constant Memory” on page 4-61.
AdditionalNativeTypeQualifier (ConstantMemory only)	Specify an AUTOSAR additional native type qualifier to include in generated code for the AUTOSAR parameter. For example, <code>my_qualifier</code> . For more information, see “Specify C Type Qualifiers for AUTOSAR Static and Constant Memory” on page 4-61.

Attribute	Purpose
SwAddrMethod	Select a SwAddrMethod name from the names listed as valid for the AUTOSAR parameter. For example, the model <code>autosar_sw_counter</code> defines VAR. Code generation uses the SwAddrMethod name to group AUTOSAR parameters in a memory section for access by calibration and measurement tools. For more information, see “Configure SwAddrMethod” on page 4-267.
SwCalibrationAccess	Specify how calibration and measurement tools can access the AUTOSAR parameter. Valid access values include <code>ReadOnly</code> , <code>ReadWrite</code> , and <code>NotAccessible</code> . For more information, see “Configure SwCalibrationAccess” on page 4-262.
DisplayFormat	Specify a display format for the AUTOSAR parameter. For example, <code>%5.1f</code> . AUTOSAR display format specifications control the width and precision display for calibration and measurement data. For more information, see “Configure DisplayFormat” on page 4-264.
Port (PortParameter only)	Select the name of a parameter receiver port configured in the AUTOSAR Dictionary.
DataElement (PortParameter only)	Select the name of a parameter interface data element configured in the AUTOSAR Dictionary.
LongName	Specify a description for the parameter.

Map Data Stores to AUTOSAR Variables

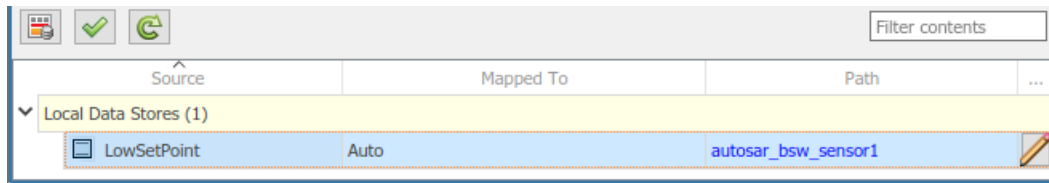
On the **Data Stores** tab of the Code Mappings editor, you can map Simulink data store memory blocks to AUTOSAR variables for AUTOSAR run-time calibration. After creating data store memory blocks in your model, open the Code Mappings editor and select the **Data Stores** tab. Select data stores and map them to AUTOSAR variables, such as AUTOSAR-typed per-instance memory or AUTOSAR static memory.


For more information, see “Configure AUTOSAR Per-Instance Memory” on page 4-201 and “Configure AUTOSAR Static Memory” on page 4-206.

The **Data Stores** tab lists each data store that you can map to an AUTOSAR variable. You can:

- Map a Simulink data store by selecting the data store, and then selecting a menu value for an AUTOSAR variable type: `ArTypedPerInstanceMemory`, `StaticMemory`, or `Auto`. To accept software mapping defaults, specify `Auto`.

For example, here is the **Local Data Stores** tab for example model `autosar_bsw_sensor1`.



- If you select a variable type other than Auto, you can click the  icon to view and modify other code and calibration attributes for the variable.

Attribute	Purpose
ShortName	Specify a short name for the AUTOSAR variable. For example, <code>dsmsig</code> . If unspecified, ARXML export generates a short name.
Volatile (StaticMemory only)	Select or clear the option to indicate whether to include C type qualifier <code>volatile</code> in generated code for the AUTOSAR variable. For more information, see “Specify C Type Qualifiers for AUTOSAR Static and Constant Memory” on page 4-61.
AdditionalNativeTypeQualifier (StaticMemory only)	Specify an AUTOSAR additional native type qualifier to include in generated code for the AUTOSAR variable. For example, <code>my_qualifier</code> . For more information, see “Specify C Type Qualifiers for AUTOSAR Static and Constant Memory” on page 4-61.
NeedsNVRAMAccess (ArTypedPerInstanceMemory only)	Select or clear the option to indicate whether the AUTOSAR variable needs access to nonvolatile RAM on a processor. To configure the per-instance memory to be a mirror block for a specific NVRAM block, select the option.
SwAddrMethod	Select a <code>SwAddrMethod</code> name from the names listed as valid for the AUTOSAR variable. Code generation uses the <code>SwAddrMethod</code> name to group AUTOSAR variables in a memory section for access by calibration and measurement tools. For more information, see “Configure <code>SwAddrMethod</code> ” on page 4-267.
RestoreAtStart (ArTypedPerInstanceMemory only)	Select or clear the option to indicate if the state should be read out at startup.
StoreAtShutdown (ArTypedPerInstanceMemory only)	Select or clear the option to indicate if the state written away at shutdown.
SwCalibrationAccess	Specify how calibration and measurement tools can access the AUTOSAR variable. Valid access values include <code>ReadOnly</code> , <code>ReadWrite</code> , and <code>NotAccessible</code> . For more information, see “Configure <code>SwCalibrationAccess</code> ” on page 4-262.

Attribute	Purpose
DisplayFormat	Specify a display format for the AUTOSAR variable. For example, %5.1f. AUTOSAR display format specifications control the width and precision display for calibration and measurement data. For more information, see “Configure DisplayFormat” on page 4-264.
LongName	Specify a description for the variable.


Map Block Signals and States to AUTOSAR Variables

On the **Signals/States** tab of the Code Mappings editor, you can:

- Map Simulink block signals and states to AUTOSAR variables for AUTOSAR run-time calibration.
- Selectively add or remove block signals from AUTOSAR component signal mapping.

In the Code Mappings editor, Simulink block states that correspond to state owner blocks are available for mapping.

To make Simulink block signals available for mapping, use a Code Mappings editor button or a model cue:

- In the model canvas, select one or more signals. Open the Code Mappings editor, **Signals/States** tab, and click the **Add** button .
- In the model canvas, select a signal. Place your cursor over the displayed ellipsis and select model cue **Add selected signals to code mappings**.

Alternatively, call MATLAB function `addSignal`.

After selectively adding block signals to AUTOSAR component signal mapping, open the Code Mappings editor and select the **Signals/States** tab. Select block signals and states and map them to AUTOSAR variables, such as AUTOSAR-typed per-instance memory or AUTOSAR static memory.

For more information, see “Configure AUTOSAR Per-Instance Memory” on page 4-201 and “Configure AUTOSAR Static Memory” on page 4-206.

The **Signals/States** tab, **Signals** node, lists each Simulink block signal that you can map to an AUTOSAR variable. You can map a Simulink block signal by selecting the signal and then selecting a menu value for an AUTOSAR variable type: `ArTypedPerInstanceMemory`, `StaticMemory`, or `Auto`. To accept software mapping defaults, specify `Auto`.

For example, here is the **Signals/States** tab for example model `autosar_sw_counter`.

Source	Mapped To	Path	...
Signals (3)			
equal_to_count	StaticMemory	autosar_swc_counter	
sum_out	ArTypedPerInstanceMemory	autosar_swc_counter	
switch_out	Auto	autosar_swc_counter	
States (1)			
X	StaticMemory	autosar_swc_counter	


The **Signals/States** tab, **States** node, lists each configurable Simulink block state that you can map to an AUTOSAR variable. You can map a Simulink block state by selecting the state and then selecting a menu value for an AUTOSAR variable type: `ArTypedPerInstanceMemory`, `StaticMemory`, or `Auto`. To accept software mapping defaults, specify `Auto`.

If you map a signal or state to a variable type other than `Auto`, you can click the icon to view and modify other code and calibration attributes for the variable.

Attribute	Purpose
ShortName	Specify a short name for the AUTOSAR variable. For example, <code>SM_equal_to_count</code> . If unspecified, ARXML export generates a short name. <ul style="list-style-type: none"> For signals, the auto-generated short name can differ from the signal name. For states, the auto-generated short name is based on the state name if one exists. If the state is unnamed, the generated name can differ from the block name.
Volatile (StaticMemory only)	Select or clear the option to indicate whether to include C type qualifier <code>volatile</code> in generated code for the AUTOSAR variable. For more information, see “Specify C Type Qualifiers for AUTOSAR Static and Constant Memory” on page 4-61.
AdditionalNativeTypeQualifier (StaticMemory only)	Specify an AUTOSAR additional native type qualifier to include in generated code for the AUTOSAR variable. For example, <code>my_qualifier</code> . For more information, see “Specify C Type Qualifiers for AUTOSAR Static and Constant Memory” on page 4-61.

Attribute	Purpose
SwAddrMethod	Select a SwAddrMethod name from the names listed as valid for the AUTOSAR variable. For example, the model <code>autosar_sw_counter</code> defines VAR. Code generation uses the SwAddrMethod name to group AUTOSAR variables in a memory section for access by calibration and measurement tools. For more information, see “Configure SwAddrMethod” on page 4-267.
SwCalibrationAccess	Specify how calibration and measurement tools can access the AUTOSAR variable. Valid access values include <code>ReadOnly</code> , <code>ReadWrite</code> , and <code>NotAccessible</code> . For more information, see “Configure SwCalibrationAccess” on page 4-262.
DisplayFormat	Specify a display format for the AUTOSAR variable. For example, <code>%5.1f</code> . AUTOSAR display format specifications control the width and precision display for calibration and measurement data. For more information, see “Configure DisplayFormat” on page 4-264.
LongName	Specify a description for the AUTOSAR variable.

To remove Simulink block signals from AUTOSAR component signal mapping, use a Code Mappings editor button or a model cue:


- In the model canvas or on the **Signals/States** tab, select one or more signals. On the **Signals/States** tab, click the **Remove** button .
- In the model canvas, select a signal. Place your cursor over the displayed ellipsis and select model cue **Remove selected signals from code mappings**.

Alternatively, call MATLAB function `removeSignal`.

Map Data Transfers to AUTOSAR Inter-Runnable Variables

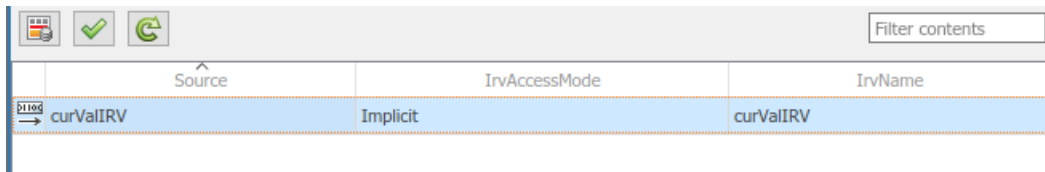
The **Data Transfers** tab of the Code Mappings editor supports modeling AUTOSAR inter-runnable variables (IRVs) in Simulink. After using the AUTOSAR Dictionary to create AUTOSAR IRVs, which connect runnables and implement aspects of internal behavior in an AUTOSAR component, open the Code Mappings editor. Use the **Data Transfers** tab to map Simulink data transfer lines to AUTOSAR IRVs.

For more information, see “Model AUTOSAR Component Behavior” on page 2-31. For illustrations of how IRVs are used with rate-based and function-call-based runnables, see the example models in “Model AUTOSAR Software Components” on page 2-3.

The **Data Transfers** tab of the Code Mappings editor maps each Simulink data transfer line to an AUTOSAR IRV. Click the **Update** button  to load or update Simulink data transfers in your model.

In the **Data Transfers** tab, you can map a Simulink data transfer line by selecting the signal name and then selecting menu values for an IRV access mode (**Implicit** or **Explicit**) and an AUTOSAR IRV name, among those listed for the AUTOSAR component.

For example, here is the **Data Transfers** tab for example model `autosar_swc_slfcns`.




Source	IrvAccessMode	IrvName
curValIRV	Implicit	curValIRV

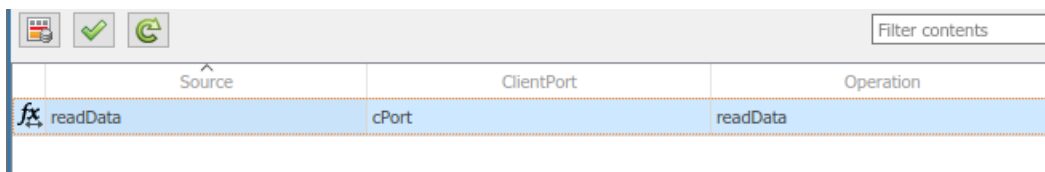
Map Function Callers to AUTOSAR Client-Server Ports and Operations

The **Function Callers** tab of the Code Mappings editor supports modeling the client side of AUTOSAR client-server (C-S) communication in Simulink. After using the AUTOSAR Dictionary to create AUTOSAR client ports, C-S interfaces, and C-S operations in your model, open the Code Mappings editor. Use the **Function Callers** tab to map Simulink function callers to AUTOSAR client ports and AUTOSAR C-S operations.

For more information, see “Configure AUTOSAR Client-Server Communication” on page 4-142.

The **Function Callers** tab of the Code Mappings editor maps each Simulink function caller to an AUTOSAR client port and an AUTOSAR C-S interface operation. Click the **Update** button  to load or update Simulink function callers in the model.

In the **Function Callers** tab, you can map a Simulink function caller by selecting the function caller name and then selecting menu values for an AUTOSAR client port and an AUTOSAR operation, among those listed for the AUTOSAR component.



Source	ClientPort	Operation
readData	cPort	readData

Specify C Type Qualifiers for AUTOSAR Static and Constant Memory

For an AUTOSAR component, you can configure C type qualifiers to customize generated AUTOSAR-compliant C code for AUTOSAR static memory and AUTOSAR constant memory. For example, you can apply C type qualifiers such as `const` or `volatile` to control compiler optimizations.

In an AUTOSAR model, use the Code Mappings editor to configure C type qualifiers for model signals, states, data stores, and parameters that are mapped to AUTOSAR `StaticMemory` or AUTOSAR `ConstantMemory`. Building the model exports type qualifiers to ARXML files and generates AUTOSAR-compliant C code that uses the type qualifiers.

For example, in the Code Mappings editor, **Signals/States** tab, suppose that you map a signal to

`StaticMemory`. Select the signal and click the  icon to display additional code attributes.

ShortName: SM_equal_to_count

Volatile

AdditionalNativeTypeQualifier: my_qualifier

SwAddrMethod: VAR

Calibration attributes

SwCalibrationAccess: ReadOnly

DisplayFormat:

LongName:

Open in Property Inspector

If you select the `Volatile` attribute and specify `AdditionalNativeTypeQualifier` to be `my_qualifier`:

- Exported ARXML files define the `AdditionalNativeTypeQualifier`:

```
<ADDITIONAL-NATIVE-TYPE-QUALIFIER>volatile my_qualifier</ADDITIONAL-NATIVE-TYPE-QUALIFIER>
```

- Generated C code uses the C type qualifiers, for example:

```
/* Static Memory for Internal Data */
volatile my_qualifier boolean SM_equal_to_count;
```

For more information, see “Map Block Signals and States to AUTOSAR Variables” on page 4-58, “Map Data Stores to AUTOSAR Variables” on page 4-56, and “Map Model Workspace Parameters to AUTOSAR Component Parameters” on page 4-54.

Specify Default Data Packaging for AUTOSAR Internal Variables

AUTOSAR Blockset provides functions to control the default data packaging used for internal variables in the generated code for an AUTOSAR component model.

For models configured with a single instance of an AUTOSAR software component, you can specify that internal data store, signal, and state data are packaged:

- With or without packing it in a structure
- With private or public visibility

For AUTOSAR software components that are instantiated multiple times and configured to generate reentrant, reusable code, you can specify to package internal variables as determined by Simulink or to use C-typed per-instance memory.

The functions `getInternalDataPackaging` and `setInternalDataPackaging` return and set the default data packaging setting used for internal data stores, signals, and states in the generated code for an AUTOSAR component model.

Valid setting values are:

- For single-instance models:

- **Default** — Accept the default internal data packaging provided by the software. Use `Default` for submodels referenced from AUTOSAR component models.
- **PrivateGlobal** — Package internal variable data without a `struct` and make it private (visible only to `model.c`).
- **PrivateStructure** — Package internal variable data in a `struct` and make it private (visible only to `model.c`).
- **PublicGlobal** — Package internal variable data without a `struct` and make it public (extern declaration in `model.h`).
- **PublicStructure** — Package internal variable data in a `struct` and make it public (extern declaration in `model.h`).
- For multi-instance models:
 - **Default** — Accept the default internal data packaging provided by the software. Use `Default` for submodels referenced from AUTOSAR component models.
 - **CTypedPerInstanceMemory** — Package internal variable data for each instance of an AUTOSAR software component to use C-typed per-instance memory in a `struct` and make it public (declaration in `model.h`).

This example modifies the default data packaging setting used for internal variables in the generated code for an AUTOSAR component model. First, it returns the current internal data packaging setting for the model. Then it sets the internal data packaging such that the code generator packages the internal variable data in a `struct` and makes it private.

```
hModel = 'autosar_sw';
openExample(hModel);
slMap = autosar.api.getSimulinkMapping(hModel);
pkgSetting1 = getInternalDataPackaging(slMap)
setInternalDataPackaging(slMap, 'PrivateStructure')
pkgSetting2 = getInternalDataPackaging(slMap)

pkgSetting1 =
    'Default'

pkgSetting2 =
    'PrivateStructure'
```

If the data packaging is set to `PrivateGlobal` or `PrivateStructure`, building the model generates header file `model_private.h`, even when model configuration parameter **File packaging format** is set to `Compact`.

If the model configuration option **Generate separate internal data per entry-point function** is set for the AUTOSAR model, task-based internal data grouping overrides the AUTOSAR internal data packaging setting. However, the AUTOSAR setting determines the public or private visibility of the generated internal data groups.

For more information, see the `getInternalDataPackaging` and `setInternalDataPackaging` reference pages.

See Also

Related Examples

- “Map Calibration Data for Submodels Referenced from AUTOSAR Component Models” on page 4-65

- “Configure AUTOSAR Elements and Properties” on page 4-8
- “Configure and Map AUTOSAR Component Programmatically” on page 4-293

More About

- “AUTOSAR Component Configuration” on page 4-3
- “Code Generation”
- “Multi-Instance Components” on page 2-8

Map Calibration Data for Submodels Referenced from AUTOSAR Component Models

In a Simulink implementation of an AUTOSAR design, model references allow you to organize and manage large or numerous AUTOSAR components hierarchically. You can define an algorithm in a submodel and reference it repeatedly. Referenced models compile independently from the models that use them, which allows modular development, reuse and sharing of algorithms across multiple components and designs, and incremental code generation.

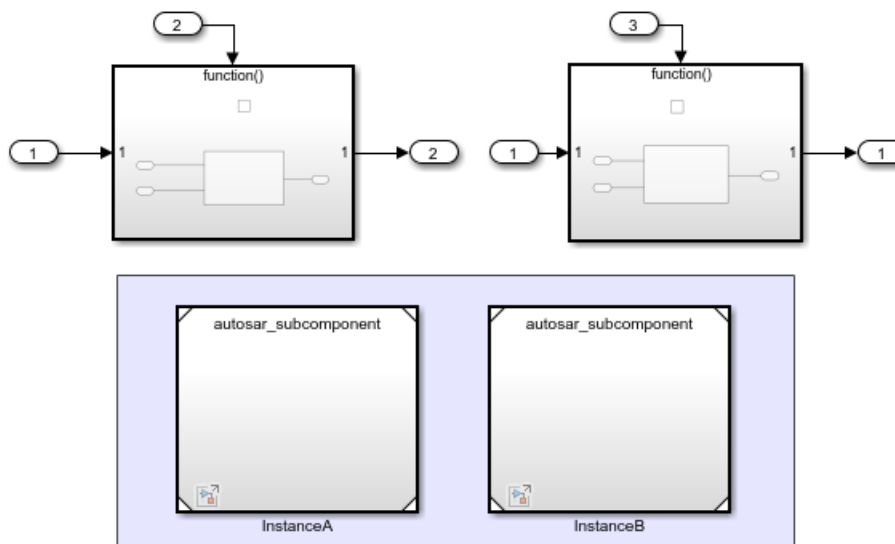
For any model in an AUTOSAR model reference hierarchy, you can configure the model data for run-time calibration. In submodels referenced from AUTOSAR software component models, you can use the Code Mappings editor or equivalent functions to map parameters, data stores, signals, and states. Submodel mapped internal data can be used in memory sections, and is available for software-in-the-loop (SIL) and processor-in-the-loop (PIL) testing from the top model or calibration in the AUTOSAR run-time environment.

Submodel Data Mapping Workflow

To map Simulink submodel elements to AUTOSAR software component elements:

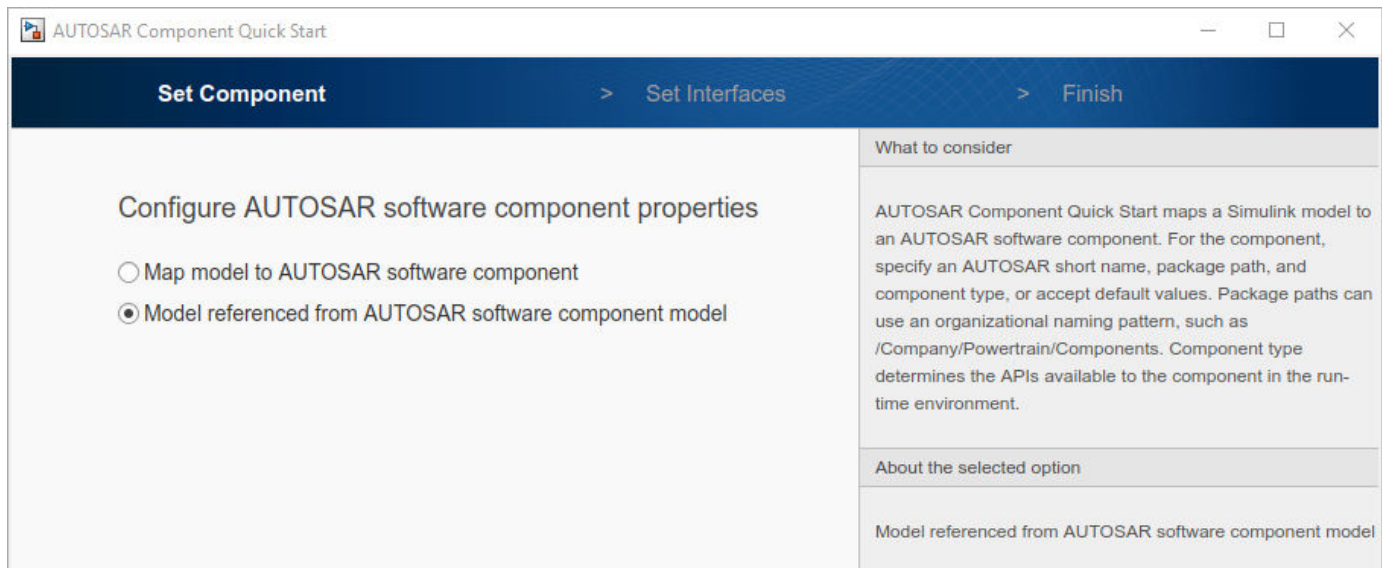
- Configure the submodel as a model referenced from an AUTOSAR software component model. Use the AUTOSAR Component Quick Start or the AUTOSAR function `autosar.api.create`.
- In the AUTOSAR Code perspective, use the Code Mappings editor to configure the submodel internal data.
- To generate C code and AUTOSAR XML (ARXML) files that support run-time calibration of the submodel internal data, open and build the component model that references the submodel.

For this example, select a model that is referenced from an AUTOSAR software component model. This example uses `autosar_subcomponent`, which is referenced twice in the AUTOSAR component model `autosar_component`. These models are associated with the example script “Configure Subcomponent Data for AUTOSAR Calibration and Measurement” on page 4-255.




Open the submodel standalone, that is, in a separate model window. In the model window, from the **Apps** tab, open the AUTOSAR Component Designer app. If the submodel is mapped, it opens in the AUTOSAR Code perspective.

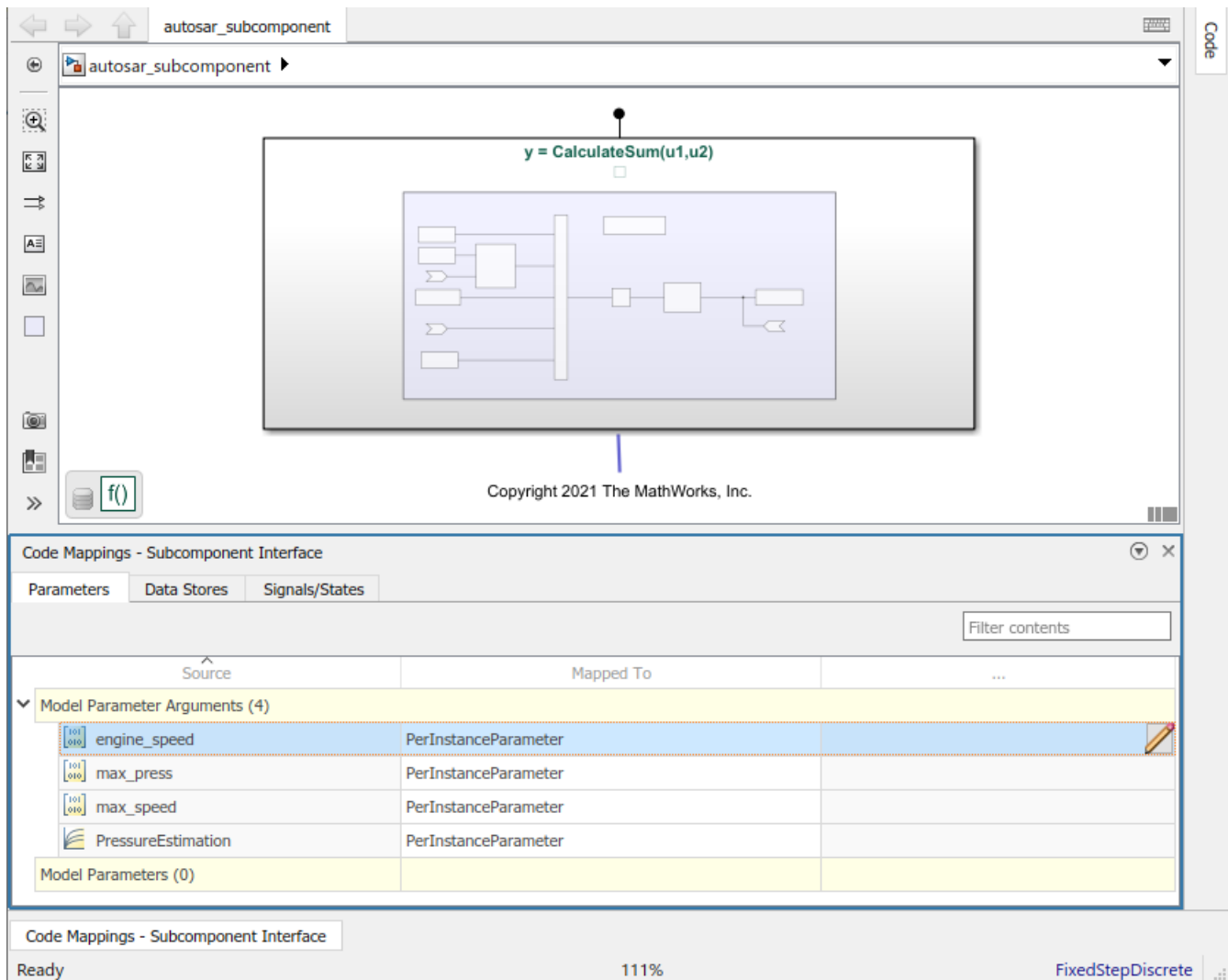
If the submodel is unmapped, the AUTOSAR Component Quick Start opens. Work through the quick-start procedure. In the **Set Component** pane, select **Model referenced from AUTOSAR software component model**



When you complete the quick-start procedure and click **Finish**, the submodel opens in the AUTOSAR Code perspective.

In the AUTOSAR Code perspective, use the Code Mappings editor to:

- Map individual parameters to `PerInstanceParameters`.
- If block signals that must be mapped to AUTOSAR variables are not displayed in the Code Mappings editor, select the signals and add them to the mapping table.
- Map individual signals, states, and data stores to `ArTypedPerInstanceMemorys`.
- After setting the **Mapped To** property for a parameter, signal, state, or data store, click the  icon to view and modify other AUTOSAR code and calibration attributes.



The screenshot displays the MATLAB/Simulink environment. The top window, titled 'autosar_subcomponent', shows a block diagram of a subcomponent model. The model is represented by a large blue rectangle with a smaller blue rectangle inside it. The text 'y = CalculateSum(u1,u2)' is visible above the inner rectangle. Below the diagram, the text 'Copyright 2021 The MathWorks, Inc.' is displayed. The bottom window, titled 'Code Mappings - Subcomponent Interface', shows a table with columns 'Source' and 'Mapped To'. The table lists four 'Model Parameter Arguments' and one 'Model Parameters' entry.

Source	Mapped To	...
Model Parameter Arguments (4)		
engine_speed	PerInstanceParameter	
max_press	PerInstanceParameter	
max_speed	PerInstanceParameter	
PressureEstimation	PerInstanceParameter	
Model Parameters (0)		

If you have Simulink Coder and Embedded Coder software, you can build the component model that references the submodel. When you build, the exported ARXML files and generated C code support run-time calibration of the submodel internal data. The ARXML files exported for the top model include descriptions of the submodel parameters, signals, states, and data stores, as well as software address methods used in the submodel. The generated C code references the submodel internal data. The model build also generates macros that provide access to the submodel data for SIL and PIL testing and calibration in the AUTOSAR run-time environment. For more information, see “Generate Submodel Data Macros for Verification and Deployment” on page 4-73.

To programmatically configure a submodel as a model referenced from an AUTOSAR software component model, call the AUTOSAR function `autosar.api.create` and specify the name-value pair argument 'ReferencedFromComponentModel', `true`. For example:

```
hModel = 'autosar_subcomponent';
open_system(hModel);
autosar.api.create(hModel, 'default', 'ReferencedFromComponentModel', true);
```

To programmatically add shared definitions of software address methods to use with the submodel, call the AUTOSAR importer function `updateModel` and specify the name of an AUTOSAR XML (ARXML) file containing the shared definitions. For example:

```
ar = arxml.importer('SwAddrMethods.arxml');
updateModel(ar,hModel);
```

Map Submodel Parameters to AUTOSAR Component Parameters

On the **Parameters** tab of the Code Mappings editor, you can map Simulink submodel parameters to AUTOSAR per-instance parameters for AUTOSAR run-time calibration. Examples of model workspace parameters you can map include:

- Simulink parameter objects
- Simulink lookup table objects
- Simulink breakpoint objects

By mapping lookup table and breakpoint objects to AUTOSAR internal calibration parameters, you can model AUTOSAR parameters for integrated and distributed lookups. For more information, see “Configure Lookup Tables for AUTOSAR Calibration and Measurement” on page 4-273.

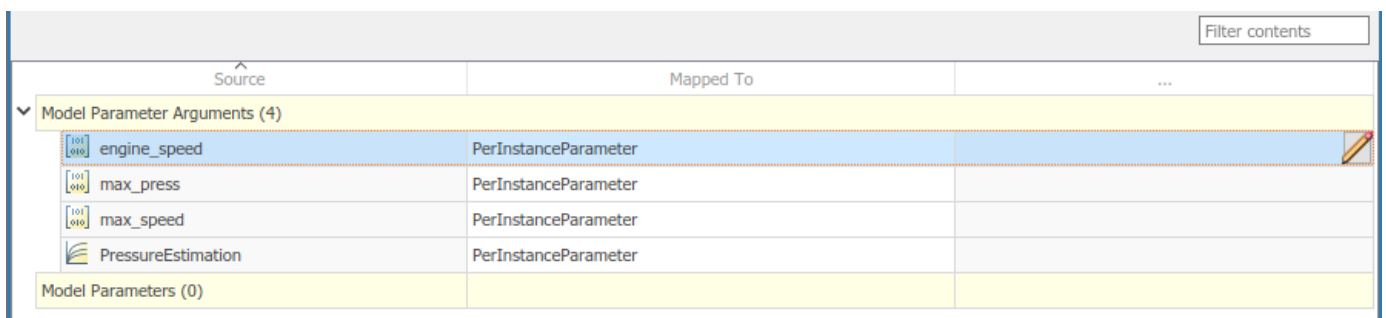
After creating model workspace parameters in your model, for example, using Model Explorer, open the Code Mappings editor and select the **Parameters** tab. Select Simulink model workspace parameters and map them to AUTOSAR component per-instance parameters.


For more information, see “Configure Model Workspace Parameters as AUTOSAR Per-Instance Parameters” on page 4-213.


The **Parameters** tab lists each Simulink model workspace parameter that you can map to an AUTOSAR parameter. You can:

- Map a parameter by selecting it and then selecting a menu value for an AUTOSAR parameter type: `PerInstanceParameter` or `Auto`. To accept software mapping defaults, specify `Auto`.

For example, here is the **Parameters** tab for submodel `autosar_subcomponent`. AUTOSAR software component model `autosar_component` contains two instances of `autosar_subcomponent`.



Source	Mapped To	...
Model Parameter Arguments (4)		
engine_speed	PerInstanceParameter	
max_press	PerInstanceParameter	
max_speed	PerInstanceParameter	
PressureEstimation	PerInstanceParameter	
Model Parameters (0)		

- If you select parameter type `PerInstanceParameter`, click the  icon to view and modify other AUTOSAR code and calibration attributes for the parameter.

Attribute	Purpose
SwAddrMethod	Select a SwAddrMethod name from the names listed as valid for the AUTOSAR parameter. For example, the submodel <code>autosar_subcomponent</code> defines <code>CALIB_32</code> . Code generation uses the SwAddrMethod name to group AUTOSAR parameters in a memory section for access by calibration and measurement tools. For more information, see “Configure SwAddrMethod” on page 4-267.
SwCalibrationAccess	Specify how calibration and measurement tools can access the AUTOSAR parameter. Valid access values include <code>ReadOnly</code> , <code>ReadWrite</code> , and <code>NotAccessible</code> . For more information, see “Configure SwCalibrationAccess” on page 4-262.
DisplayFormat	Specify a display format for the AUTOSAR parameter. For example, <code>%5.1f</code> . AUTOSAR display format specifications control the width and precision display for calibration and measurement data. For more information, see “Configure DisplayFormat” on page 4-264.
LongName	Specify a description for the AUTOSAR parameter.

Map Submodel Data Stores to AUTOSAR Variables

On the **Data Stores** tab of the Code Mappings editor, you can map Simulink submodel data store memory blocks to AUTOSAR-typed per-instance memory elements for AUTOSAR run-time calibration.


After creating data store memory blocks in your model, open the Code Mappings editor and select the **Data Stores** tab. Select data stores and map them to AUTOSAR-typed per-instance memory elements. For more information, see “Configure AUTOSAR Per-Instance Memory” on page 4-201.

The **Data Stores** tab lists each Simulink data store that you can map to an AUTOSAR variable. You can:

- Map a data store by selecting the data store, and then selecting a menu value for an AUTOSAR variable type: `ArTypedPerInstanceMemory` or `Auto`. To accept software mapping defaults, specify `Auto`.

For example, here is the **Local Data Stores** tab for submodel `autosar_subcomponent`. AUTOSAR software component model `autosar_component` contains two instances of `autosar_subcomponent`.

Source	Mapped To	Path	...
Local Data Stores (1)			
DSM_local	ArTypedPerInstanceMemory	autosar_subcomponent/Simulink Function	

- If you select variable type `ArTypedPerInstanceMemory`, click the  icon to view and modify other AUTOSAR code and calibration attributes for the variable.


Attribute	Purpose
ShortName	Specify a short name for the AUTOSAR variable. For example, <code>dsmsig</code> . If unspecified, ARXML export generates a short name.
NeedsNVRAMAccess	Select or clear the option to indicate whether the AUTOSAR variable needs access to nonvolatile RAM on a processor. To configure the per-instance memory to be a mirror block for a specific NVRAM block, select the option.
SwAddrMethod	Select a <code>SwAddrMethod</code> name from the names listed as valid for the AUTOSAR variable. For example, the submodel <code>autosar_subcomponent</code> defines <code>VAR_INIT_32</code> . Code generation uses the <code>SwAddrMethod</code> name to group AUTOSAR variables in a memory section for access by calibration and measurement tools. For more information, see “Configure <code>SwAddrMethod</code> ” on page 4-267.
RestoreAtStart	Select or clear the option to indicate if the state should be read out at startup.
StoreAtShutdown	Select or clear the option to indicate if the state written away at shutdown.
SwCalibrationAccess	Specify how calibration and measurement tools can access the AUTOSAR variable. Valid access values include <code>ReadOnly</code> , <code>ReadWrite</code> , and <code>NotAccessible</code> . For more information, see “Configure <code>SwCalibrationAccess</code> ” on page 4-262.
DisplayFormat	Specify a display format for the AUTOSAR variable. For example, <code>%5d</code> . AUTOSAR display format specifications control the width and precision display for calibration and measurement data. For more information, see “Configure <code>DisplayFormat</code> ” on page 4-264.
LongName	Specify a description for the AUTOSAR variable.

Map Submodel Signals and States to AUTOSAR Variables

On the **Signals/States** tab of the Code Mappings editor, you can map Simulink submodel block signals and states to AUTOSAR-typed per-instance memory elements for AUTOSAR run-time calibration.

In the Code Mappings editor, Simulink block states that correspond to state owner blocks are available for mapping.

To make Simulink block signals available for mapping, use a Code Mappings editor button or a model cue:

- In the model canvas, select one or more signals. Open the Code Mappings editor, **Signals/States** tab, and click the **Add** button .
- In the model canvas, select a signal. Place your cursor over the displayed ellipsis and select model cue **Add selected signals to code mappings**.

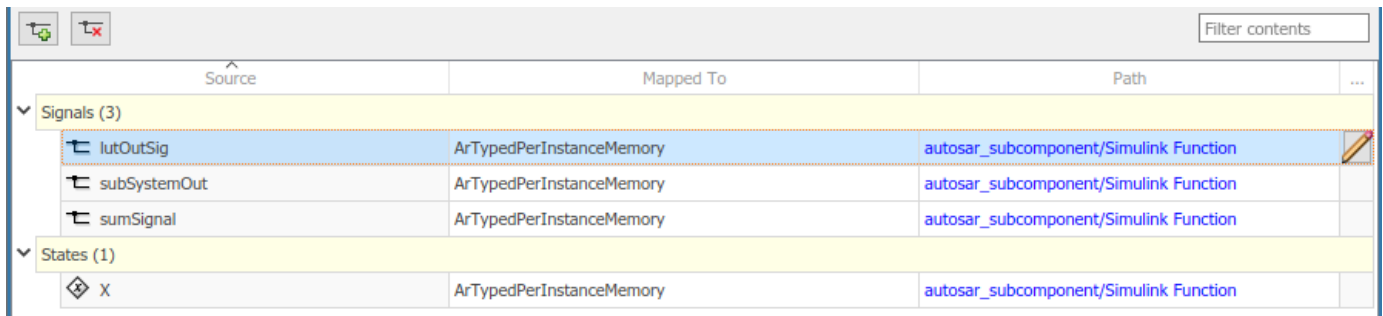
Alternatively, call MATLAB function `addSignal`.


After selectively adding block signals to AUTOSAR signal mapping, open the Code Mappings editor and select the **Signals/States** tab. Select block signals and states and map them to AUTOSAR-typed per-instance memory elements. For more information, see “Configure AUTOSAR Per-Instance Memory” on page 4-201.


The **Signals/States** tab lists each Simulink block signal and state that you can map to an AUTOSAR variable. You can:

- Map a Simulink signal or state by selecting the signal or state, and then selecting a menu value for an AUTOSAR variable type: `ArTypedPerInstanceMemory` or `Auto`. To accept software mapping defaults, specify `Auto`.

For example, here is the **Signals/States** tab for submodel `autosar_subcomponent`. AUTOSAR software component model `autosar_component` contains two instances of `autosar_subcomponent`.




Source	Mapped To	Path	...
▼ Signals (3)			
lutOutSig	ArTypedPerInstanceMemory	autosar_subcomponent/Simulink Function	
subSystemOut	ArTypedPerInstanceMemory	autosar_subcomponent/Simulink Function	
sumSignal	ArTypedPerInstanceMemory	autosar_subcomponent/Simulink Function	
▼ States (1)			
X	ArTypedPerInstanceMemory	autosar_subcomponent/Simulink Function	

- If you select variable type `ArTypedPerInstanceMemory`, click the  icon to view and modify other AUTOSAR code and calibration attributes for the variable.

Attribute	Purpose
ShortName	Specify a short name for the AUTOSAR variable. For example, <code>lutsig</code> . If unspecified, ARXML export generates a short name. <ul style="list-style-type: none"> For signals, the auto-generated short name can differ from the signal name. For states, the auto-generated short name is based on the state name if one exists. If the state is unnamed, the generated name can differ from the block name.
SwAddrMethod	Select a <code>SwAddrMethod</code> name from the names listed as valid for the AUTOSAR variable. For example, the submodel <code>autosar_subcomponent</code> defines <code>VAR_INIT_32</code> . Code generation uses the <code>SwAddrMethod</code> name to group AUTOSAR variables in a memory section for access by calibration and measurement tools. For more information, see “Configure <code>SwAddrMethod</code> ” on page 4-267.
SwCalibrationAccess	Specify how calibration and measurement tools can access the AUTOSAR variable. Valid access values include <code>ReadOnly</code> , <code>ReadWrite</code> , and <code>NotAccessible</code> . For more information, see “Configure <code>SwCalibrationAccess</code> ” on page 4-262.
DisplayFormat	Specify a display format for the AUTOSAR variable. For example, <code>%ld</code> . AUTOSAR display format specifications control the width and precision display for calibration and measurement data. For more information, see “Configure <code>DisplayFormat</code> ” on page 4-264.
LongName	Specify a description for the AUTOSAR variable.

To remove Simulink block signals from AUTOSAR signal mapping, use a Code Mappings editor button or a model cue:

- In the model canvas or on the **Signals/States** tab, select one or more signals. On the **Signals/States** tab, click the **Remove** button .
- In the model canvas, select a signal. Place your cursor over the displayed ellipsis and select model cue **Remove selected signals from code mappings**.

Alternatively, call MATLAB function `removeSignal`.

Generate Submodel Data Macros for Verification and Deployment

When you build an AUTOSAR software component model that references a submodel, the exported ARXML files and generated C code support run-time calibration of the submodel internal data.

- The ARXML files exported for the top model include descriptions of the submodel parameters, signals, states, and data stores, as well as software address methods used in the submodel.
- The generated C code references the submodel internal data.

The model build also generates macros that provide access to the submodel mapped internal data for SIL and PIL testing from the referencing component model, and calibration in the AUTOSAR run-time environment. If an AUTOSAR component model build encompasses a referenced model with mapped internal data, the generated submodel header file references these macros:

- `INCLUDE_RTE_HEADER` — Flag indicating whether to include an RTE component header.
- `RTE_COMPONENT_HEADER` — Name of a header file containing definitions for submodel internal parameters, signals, states, and data stores.

For example, if you build AUTOSAR software component model `autosar_component`, which contains two instances of `autosar_subcomponent`, the generated file `autosar_subcomponent.h` contains this code.

```
#ifndef INCLUDE_RTE_HEADER
#include RTE_COMPONENT_HEADER
#endif
```

If you run top-model SIL or PIL testing from an AUTOSAR software component model that references a mapped submodel, the SIL or PIL model build automatically picks up the submodel internal data definitions.

When you integrate the generated code into an AUTOSAR run-time environment, you must configure the `INCLUDE_RTE_HEADER` and `RTE_COMPONENT_HEADER` macros to include the submodel internal data definitions.

See Also

`autosar.api.create` | **Code Mappings Editor**

Related Examples

- “Configure Subcomponent Data for AUTOSAR Calibration and Measurement” on page 4-255
- “Configure Model Workspace Parameters as AUTOSAR Per-Instance Parameters” on page 4-213
- “Configure AUTOSAR Per-Instance Memory” on page 4-201
- “Map AUTOSAR Elements for Code Generation” on page 4-50
- “Integrate Generated Code for Multi-Instance Software Components” on page 5-31

More About

- “AUTOSAR Component Configuration” on page 4-3

Incrementally Update AUTOSAR Mapping After Model Changes

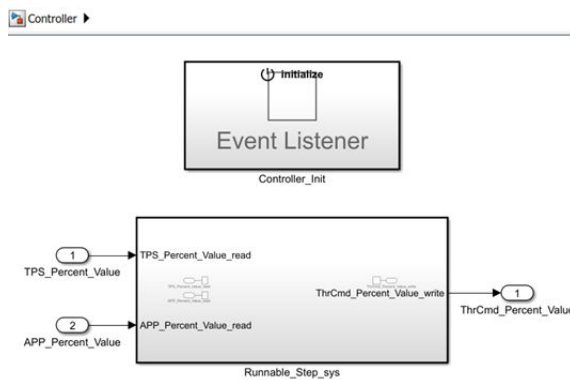
While developing an AUTOSAR software component model, you can use function `autosar.api.create` to incrementally configure and map Simulink elements as you add them to your model. When used with a mapped AUTOSAR model, `autosar.api.create` does not recreate or replace the current Simulink to AUTOSAR mapping. Instead, the function updates the mapping to reflect your model changes. The function:

- Preserves current model configuration and mapping.
- Finds and maps unmapped model elements.
- Updates the AUTOSAR Dictionary for deleted model elements.

In this example, you add inports and outports to a mapped AUTOSAR software component model. Then you use `autosar.api.create` to create and map corresponding AUTOSAR elements with default naming and properties. After the incremental update, you can edit the default naming and properties as you require.

- 1 Open a mapped AUTOSAR software component model. For this example, create a model named `Controller` from an ARXML file, `ThrottlePositionControlComposition.arxml`.

```
openExample('autosarblockset/ImportAUTOSARComponentToSimulinkExample',...
'supportingfile','ThrottlePositionControlComposition.arxml');
ar = arxml.importer('ThrottlePositionControlComposition.arxml');
createComponentAsModel(ar,'/Company/Components/Controller',...
'ModelPeriodicRunnablesAs','AtomicSubsystem');
```

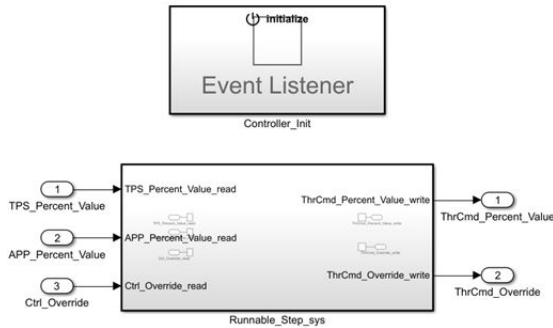


In the Code Mappings editor, **Inports** and **Outports** tabs, here is the initial Simulink to AUTOSAR mapping of Simulink inports and outports in the model.

Source	DataAccessMode	Port	Element
TPS_Percent_Value	ImplicitReceive	TPS_Percent	Value
APP_Percent_Value	ImplicitReceive	APP_Percent	Value

Source	DataAccessMode	Port	Element
ThrCmd_Percent_Value	ImplicitSend	ThrCmd_Percent	Value

- 2 Add an inport and an outport to subsystem block `Runnable_Step_sts`, and a corresponding inport and outport inside the subsystem. For example, inside the subsystem, add inport `Ctrl_Override_read` and outport `ThrCommand_Override_write`. At the top level, add inport `Ctrl_Override` and outport `ThrCommand_Override`. Connect the inports and outports.



- 3 To configure and map the added inports and outports, call the `autosar.api.create` function. Use either of these forms.

```
autosar.api.create('Controller','incremental');
autosar.api.create('Controller');
```

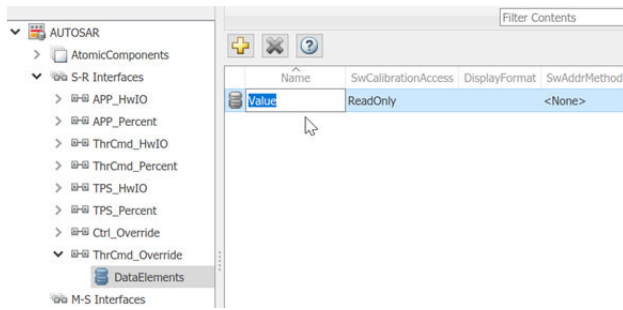
For more information about function syntax and behavior, see `autosar.api.create`.

- 4 In the Code Mappings editor, **Inports** and **Outports** tabs, here is the updated Simulink to AUTOSAR mapping of Simulink inports and outports in the model. Notice that the added inport and outport are each mapped to an AUTOSAR port and data element, which the function created in the AUTOSAR Dictionary. The function also created the S-R interfaces that own each data element.

Source	DataAccessMode	Port	Element
TPS_Percent_Value	ImplicitReceive	TPS_Percent	Value
APP_Percent_Value	ImplicitReceive	APP_Percent	Value
Ctrl_Override	ImplicitReceive	Ctrl_Override	Ctrl_Override

Source	DataAccessMode	Port	Element
ThrCmd_Percent_Value	ImplicitSend	ThrCmd_Percent	Value
ThrCmd_Override	ImplicitSend	ThrCmd_Override	ThrCmd_Override

- 5 The function provided default naming and properties for the AUTOSAR ports, S-R interfaces, and data elements created in the AUTOSAR Dictionary. You can edit the naming and properties to correspond with peer elements or match your design requirements. For example, you can rename the created data elements to `Value` to match the other S-R interface data elements in the model.



See Also

`autosar.api.create`

More About

- “AUTOSAR Component Configuration” on page 4-3

Design and Simulate AUTOSAR Components and Generate Code

Develop AUTOSAR components by implementing behavior algorithms, simulating components and compositions, and generating component code.

Begin with Simulink Representation of AUTOSAR Components

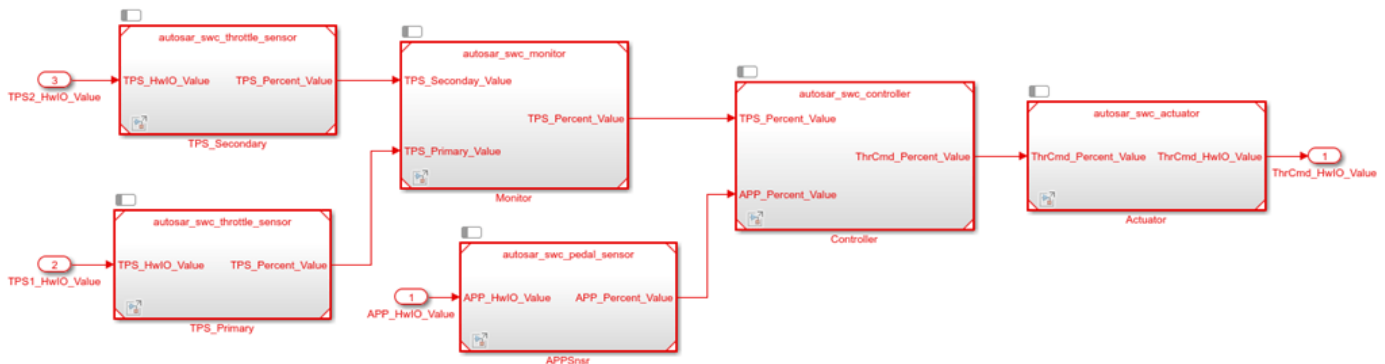
To develop AUTOSAR components in Simulink®, you first create a Simulink representation of an AUTOSAR software component. AUTOSAR component creation can start from an ARXML component description or an existing Simulink design.

- To import an AUTOSAR software component description from ARXML files and create an initial Simulink model representation, see example “Import AUTOSAR Component to Simulink” on page 3-19 or example “Import AUTOSAR Composition to Simulink” on page 7-2.
- To create an initial model representation of an AUTOSAR software component in Simulink, see “Create AUTOSAR Software Component in Simulink” on page 3-2.

This example uses a Simulink representation of an AUTOSAR software composition named `autosar_composition`, which models a throttle position control system. The composition contains six interconnected AUTOSAR software components -- four sensor/actuator components and two application components.

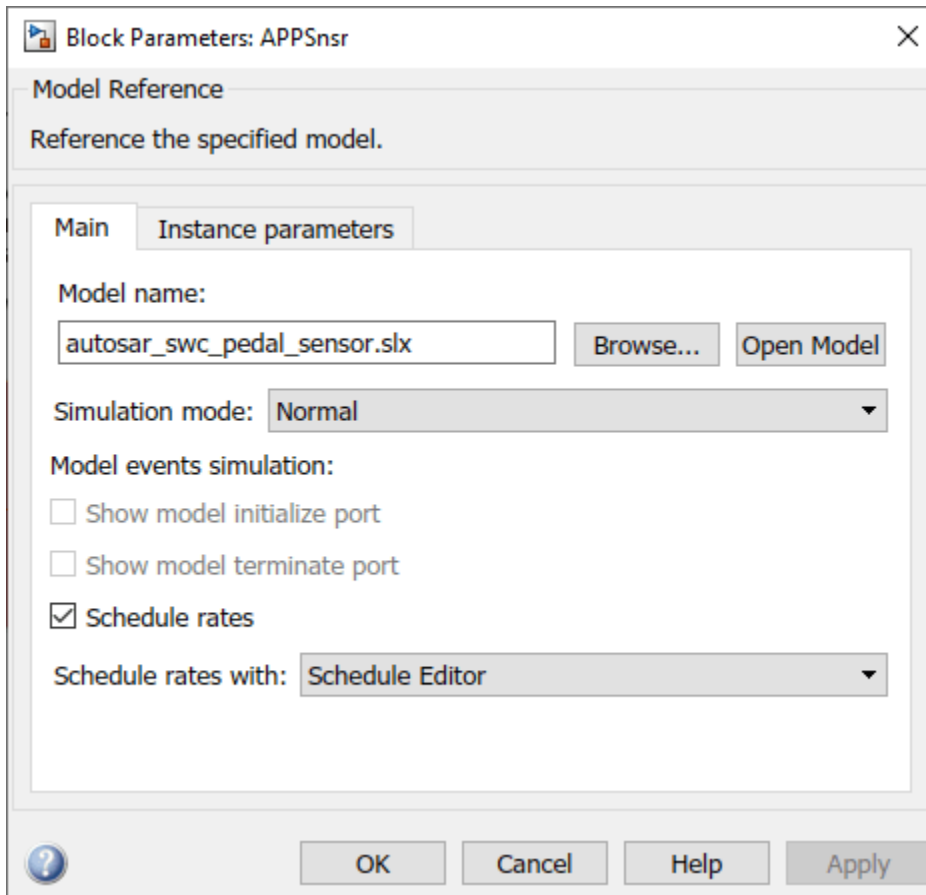
Open the composition model `autosar_composition`.

```
open_system('autosar_composition');
```



Signal lines between component models represent AUTOSAR assembly connectors. Signal lines between component models and data inports and outports represent AUTOSAR delegation connectors.

In a composition model, component models can be rate-based, function-call based, or a mix of both. This composition contains rate-based component models. In each component model, atomic subsystems model AUTOSAR periodic runnables. To allow rate-based runnable tasks to be scheduled on the same basis as exported functions, the component models use the Model block option **Schedule rates**. This option displays model periodic event ports for rate-based models.



Functional Overview of Throttle Position Control Composition

The objective of the composition model `autosar_composition` is to control an automotive throttle based on input from an accelerator pedal and feedback from the throttle. Inside the composition, a controller component takes input values from an accelerator pedal position (APP) sensor and two throttle position sensors (TPSs). The controller then translates the values into input values for a throttle actuator. The throttle actuator generates a hardware command that adjusts the throttle position.

The composition model has root inports for an accelerator pedal sensor and two throttle sensors, and a root outport for a command to throttle hardware. The composition requires sensor input values to arrive already normalized to analog/digital converter (ADC) range. The composition components are three sensors, one monitor, one controller, and one actuator.

- Sensor component model `autosar_swc_pedal_sensor` takes an APP sensor HWIO value from a composition inport and converts it to an APP sensor percent value.
- Primary and secondary instances of sensor component model `autosar_swc_throttle_sensor` take TPS HWIO values from composition inports and convert them to TPS percent values.
- Application component model `autosar_swc_monitor` decides which TPS signal to pass through to the controller.
- Application component model `autosar_swc_controller` takes the APP sensor percent value from the pedal sensor and the TPS percent value provided by the TPS monitor. Based on these

values, the controller calculates a throttle command percent value to provide to the throttle actuator.

- Actuator component model `autosar_swc_actuator` takes the throttle command percent value provided by the controller and converts it to a throttle command HWIO value.

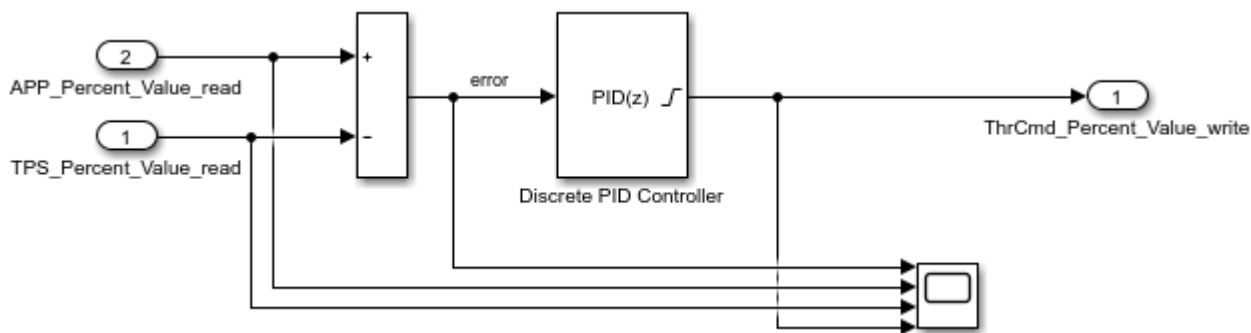
Develop AUTOSAR Component Algorithms

After creating initial Simulink representations of one or more AUTOSAR software components, you develop the components by refining the AUTOSAR configuration and creating algorithmic model content.

To develop AUTOSAR component algorithms, open each component and provide Simulink content that implements the component behavior. For example, consider the `autosar_swc_controller` component model in the `autosar_composition` model. When first imported or created in Simulink, the initial representation of the `autosar_swc_controller` component likely contained an initial stub implementation of the controller behavior.



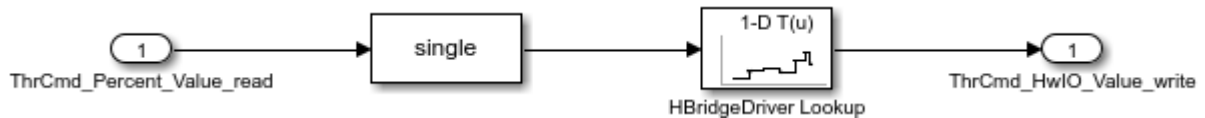
Component model `autosar_swc_controller` provides this implementation of the throttle position controller behavior. The component takes as inputs an APP sensor percent value from a pedal position sensor and a TPS percent value provided by a throttle position sensor monitor. Based on these values, the controller calculates the *error*, which is the difference between where the automobile driver wants the throttle, based on the pedal sensor, and the current throttle position. A Discrete PID Controller block uses the error value to calculate a throttle command percent value to provide to a throttle actuator. A scope displays the error value and the Discrete PID Controller block output value over time.



The sensor and actuator component models in the `autosar_composition` model use lookup tables to implement their value conversions. For example, consider the `autosar_swc_actuator` component model. When first imported or created in Simulink, the initial representation of the `autosar_swc_actuator` component likely contained an initial stub implementation of the actuator behavior.



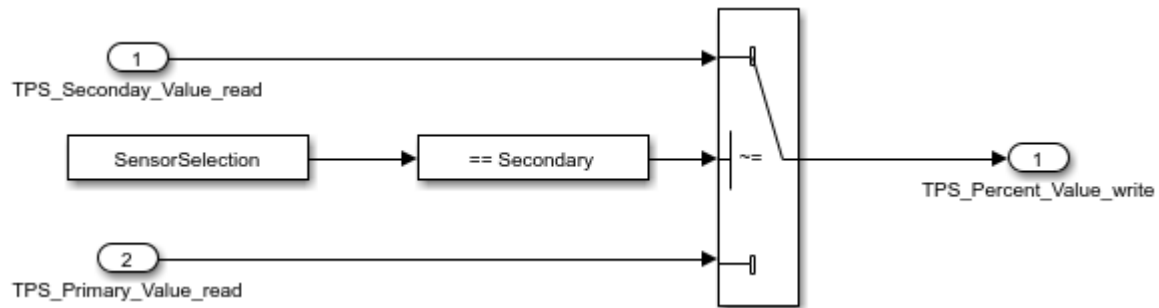
Component model `autosar_swc_actuator` provides this implementation of the throttle position actuator behavior. The component takes the throttle command percent value provided by the controller and converts it to a throttle command HWIO value. A hardware bridge command lookup table generates the output value.



The monitor component model in the `autosar_composition` model implements logic for selecting which TPS signal to provide to the controller component. When first imported or created in Simulink, the initial representation of the `autosar_swc_monitor` component likely contained an initial stub implementation of the monitor behavior.



Component model `autosar_swc_monitor` provides this implementation of the throttle position monitor behavior. The component takes TPS percent values from primary and secondary throttle position sensors and decides which TPS signal to pass through to the controller. A Switch block determines which value is passed through, based on sensor selection logic.



Simulate AUTOSAR Components and Composition

As you develop AUTOSAR components, you can simulate component models individually or as a group in a containing composition.

Simulate the implemented Controller component model.

```
open_system('autosar_swc_controller');
simOutComponent = sim('autosar_swc_controller');
close_system('autosar_swc_controller');
```

Simulate the `autosar_composition` model.

```
simOutComposition = sim('autosar_composition');
```

Generate AUTOSAR Component Code (Embedded Coder)

As you develop each AUTOSAR component, if you have Simulink Coder and Embedded Coder software, you can generate ARXML component description files and algorithmic C code for testing in Simulink or integration into an AUTOSAR run-time environment.

For example, to build the implemented `autosar_swc_controller` component model, open the model. Press **Ctrl+B** or enter the MATLAB command `slbuild('autosar_swc_controller')`.

The model build exports ARXML descriptions, generates AUTOSAR-compliant C code, and opens an HTML code generation report describing the generated files. In the report, you can examine the generated files and click hyperlinks to navigate between generated code and source blocks in the component model.

Alternatives for AUTOSAR System-Level Simulation

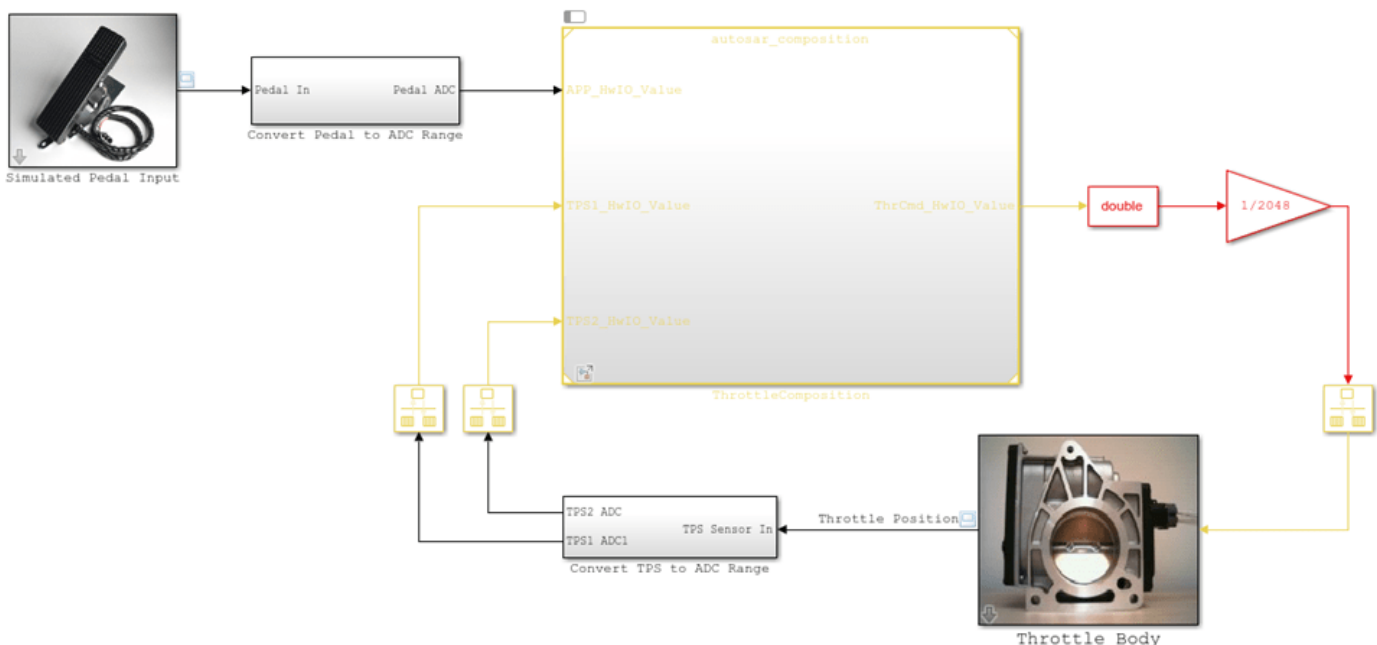
After you develop AUTOSAR components and compositions, you can test groups of components that belong together in a system-level simulation. You can:

- Combine components in a composition for simulation.
- Create a test harness with components, a scheduler, a plant model, and potentially Basic Software service components and callers. Use the test harness to perform an open-loop or closed-loop system simulation.

For an example of open-loop simulation that uses Simulink Test, see “Testing AUTOSAR Compositions” (Simulink Test). The example performs back-to-back testing for an AUTOSAR composition model.

For an example of a closed-loop simulation, open example model `autosar_system`. This model provides a system-level test harness for the AUTOSAR composition model `autosar_composition`.

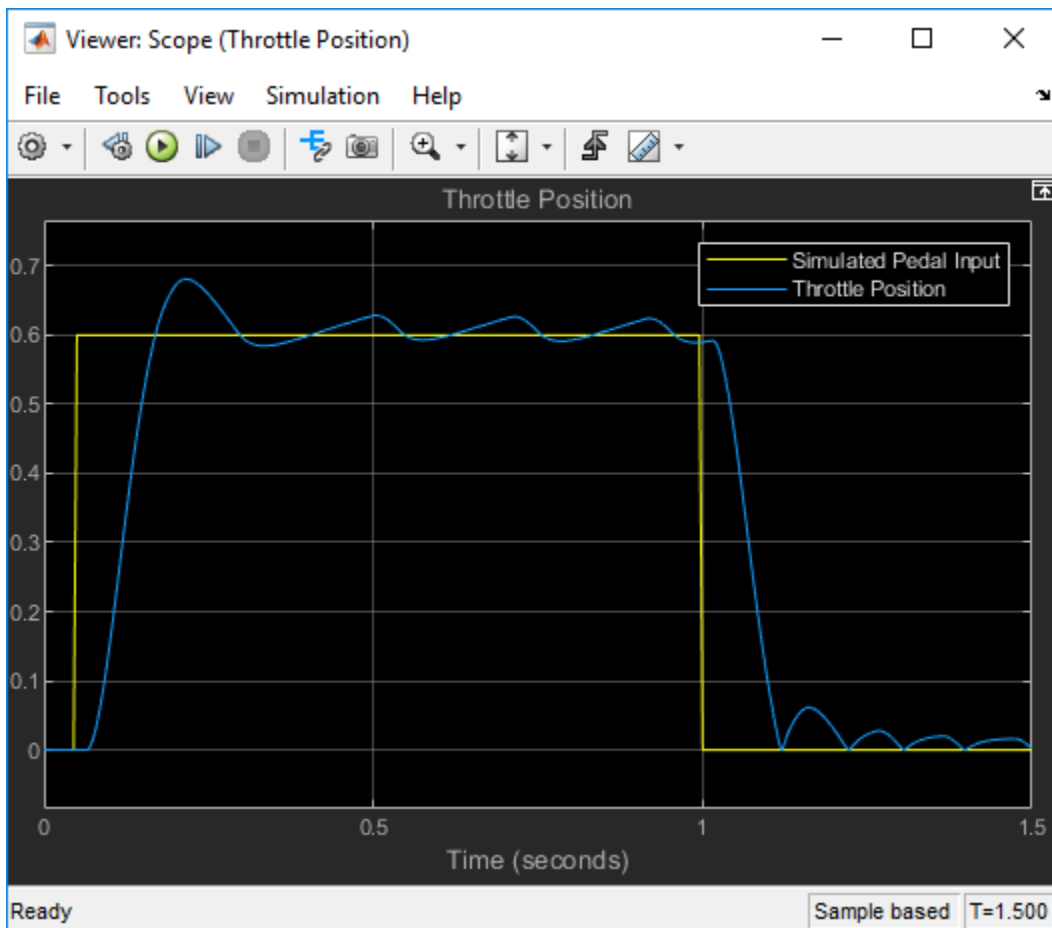
```
open_system('autosar_system');
```



The objective of the system-level model `autosar_system` is performing system-level simulation of the plant and controller portions of the automotive throttle position control system. The system-level model combines the composition model `autosar_composition` with block representations of the physical accelerator pedal and throttle devices in a closed-loop system. The model takes output values from the pedal and throttle device blocks, converts the values to analog/digital converter (ADC) range, and provides the values as inputs to the composition. The system model also takes the throttle command HWIO value generated by the composition and converts it to an acceptable input value for the throttle device block. A system-level throttle position scope displays the accelerator pedal sensor input value against the throttle position sensor input value over time.

If you simulate the system-level model, the throttle position scope indicates how well the throttle-position control algorithms in the throttle composition model are tracking the accelerator pedal input. You can modify the system to improve the composition behavior. For example, you can modify component algorithms to bring the accelerator pedal and throttle position values closer in alignment or you can change a sensor source.

```
simOutSystem = sim('autosar_system');
```



Related Links

- “Import AUTOSAR Component to Simulink” on page 3-19
- “Import AUTOSAR Composition to Simulink” on page 7-2

- “Create AUTOSAR Software Component in Simulink” on page 3-2
- “Component Development”
- “Testing AUTOSAR Compositions” (Simulink Test)

Configure AUTOSAR Packages

In Simulink, you can modify the hierarchical AUTOSAR package structure, as defined by the AUTOSAR standard, that Embedded Coder exports to ARXML code.

AR-PACKAGE Structure

The AUTOSAR standard defines AUTOSAR packages (AR-PACKAGEs). AR-PACKAGEs contain groups of AUTOSAR elements and reside in a hierarchical AR-PACKAGE structure. In an AUTOSAR authoring tool (AAT) or in Simulink, you can configure an AR-PACKAGE structure to:

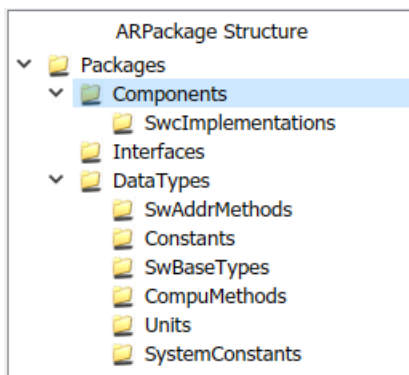
- Conform to an organizational or standardized AR-PACKAGE structure.
- Establish a namespace for elements in a package.
- Provide a basis for relative references to elements.

The ARXML importer imports AR-PACKAGEs, their elements, and their paths into Simulink. The model configuration preserves the packages for subsequent export to ARXML code. In general, the software preserves AUTOSAR packages across round-trips between an AAT and Simulink.

If your AUTOSAR component originated in Simulink, at component creation, the AUTOSAR component builder creates an initial default AR-PACKAGE structure, containing the following packages.

- Software components
- Data types
- Port interfaces
- Implementation

For example, suppose that you start with a simple Simulink model, such as `rtwdemo_counter`. Rename it to `mySWC`. Configure the model for AUTOSAR code generation. (For example, open the Embedded Coder Quick Start and select AUTOSAR code generation.) When you build the model, its initial AR-PACKAGE structure resembles the following figure.



After component creation, you can use the **XML Options** view in the AUTOSAR Dictionary to specify additional AR-PACKAGEs. (See “Configure AUTOSAR XML Options” on page 4-43 or “Configure AUTOSAR Adaptive XML Options” on page 6-33.) Each AR-PACKAGE represents an AUTOSAR element category. During code generation, the ARXML exporter generates a package if any elements

of its category exist in the model. For each package, you specify a path, which defines its location in the AR-PACKAGE structure.

Using XML options, you can configure AUTOSAR packages for the following categories of AUTOSAR elements:

- Application data types
- Software base types
- Data type mapping sets
- Constants and values
- Physical data constraints (referenced by application data types or data prototypes)
- System constants
- Software address methods
- Mode declaration groups
- Computation methods
- Units and unit groups
- Software record layouts (for application data types of category CURVE, MAP, CUBOID, or COM_AXIS)
- Internal data constraints (referenced by implementation data types)

Note

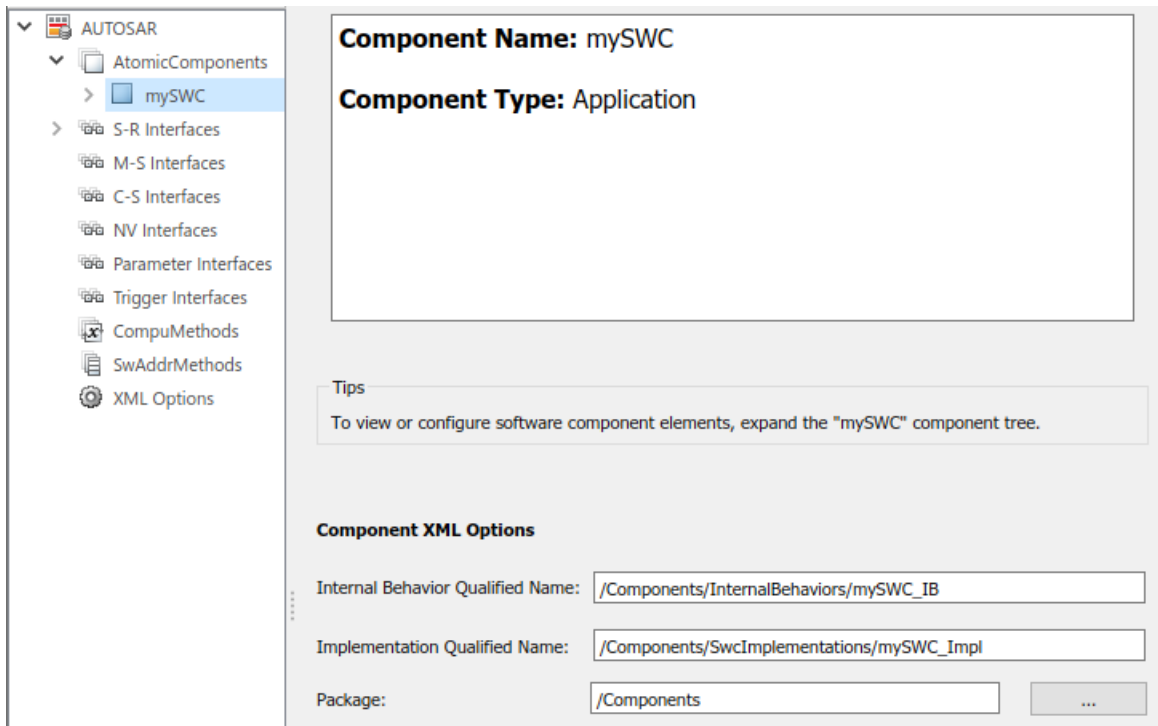
- For packages that you define in XML options, the ARXML exporter generates a package only if the model contains an element of the package category. For example, the exporter generates a software address method package only if the model contains a software address method element.
 - You can specify separate packages for the listed elements, for example, application data types. Implementation data types are aggregated in the main data types package.
-

The AR-PACKAGE structure is logically distinct from the single-file or modular-file partitioning that you can select for ARXML export, using the XML option **Exported XML file packaging**. For more information about AUTOSAR package export, see “AR-PACKAGE Location in Exported ARXML Files” on page 4-91.

Configure AUTOSAR Packages and Paths

If you import an AR-PACKAGE structure into Simulink, the ARXML importer preserves package-element relationships and package paths defined in the ARXML code. Also, the importer populates packaging properties in the component and **XML Options** views in the AUTOSAR Dictionary. If the ARXML code does not assign AUTOSAR elements to packages based on category, the importer uses heuristics to determine an optimal category association for a package. However, a maximum of one package can be associated with a category.

Suppose that you start with a non-AUTOSAR Simulink model and configure the model for AUTOSAR code generation. (For example, open the Embedded Coder Quick Start and select AUTOSAR code generation.) The software creates an initial default AR-PACKAGE structure. After component creation, the component view in the AUTOSAR Dictionary displays **Component XML Options**, including package paths for the component, internal behavior, and implementation.



The **XML Options** view displays paths for AUTOSAR data type and interface packages, and additional packages.

View and Edit XML Options

XML Options Source:

Exported XML File Packaging:

Package Paths

Datatype Package:

Interface Package:

AUTOSAR Platform Types

Implementation Platform Types Package:

User-defined ImplementationDataType References:

Native Declaration:

Platform Type Names:

Additional Packages

ApplicationDataType Package:

SwBaseType Package:

DataTypeMappingSet Package:

ConstantSpecification Package:

Physical DataConstraints Package:

SystemConstant Package:

PostBuildVariantCriterion Package:

SwAddressMethod Package:

ModeDeclarationGroup Package:

Using the **Additional Packages** subpane, you can populate path fields for additional packages or leave them empty. If you leave a package field empty, and if the model contains packageable elements of that category, the ARXML exporter uses internal rules to calculate the package path. The application of internal rules is backward-compatible with earlier releases. The following table lists the XML option packaging properties with their rule-based default package paths.

Property Name	Package Paths Based on Internal Rules
InternalBehaviorQualifiedName	/Components/InternalBehaviors/ <i>modelName_IB</i>
ImplementationQualifiedName	/Components/SwcImplementations/ <i>modelName_Impl</i>
ComponentQualifiedName	/Components/ <i>modelName</i> (The dialog box displays the component path without the short name.)
DataTypePackage	/DataTypes (Data type packaging is affected by the property settings for AUTOSAR Platform Types. See “AUTOSAR Platform Types” on page 4-47 for more information.)
InterfacePackage	/Interfaces
ApplicationDataTypePackage	<i>DataTypePackage</i> /ApplDataTypes

Property Name	Package Paths Based on Internal Rules
SwBaseTypePackage	<i>DataTypePackage/SwBaseTypes</i> (SW base type packaging is affected by the property settings for AUTOSAR Platform Types. See “AUTOSAR Platform Types” on page 4-47 for more information.)
DataTypeMappingPackage	<i>DataTypePackage/DataTypeMappings</i>
ConstantSpecificationPackage	<i>DataTypePackage/Constants</i>
DataConstraintPackage	<i>ApplicationDataTypePackage/DataConstrs</i>
SystemConstantPackage	<i>DataTypePackage/SystemConstants</i>
SwAddressMethodPackage	<i>DataTypePackage/SwAddrMethods</i>
ModeDeclarationGroupPackage	<i>DataTypePackage/ModeDeclarationGroups</i>
CompuMethodPackage	<i>DataTypePackage/CompuMethods</i>
UnitPackage	<i>DataTypePackage/Units</i>
SwRecordLayoutPackage	<i>DataTypePackage/SwRecordLayouts</i>
InternalDataConstraintPackage	<i>DataTypePackage/DataConstrs</i>

To set a packaging property in the MATLAB Command Window or in a script, use an AUTOSAR property `set` function call similar to the following:

```
hModel = 'autosar_sw_counter';
openExample(hModel);
arProps=autosar.api.getAUTOSARProperties(hModel);
set(arProps,'XmlOptions','ApplicationDataTypePackage','/Company/Powertrain/DataTypes/ADTs');
get(arProps,'XmlOptions','ApplicationDataTypePackage')
```

For a sample script, see “Configure AUTOSAR XML Export” on page 4-319.

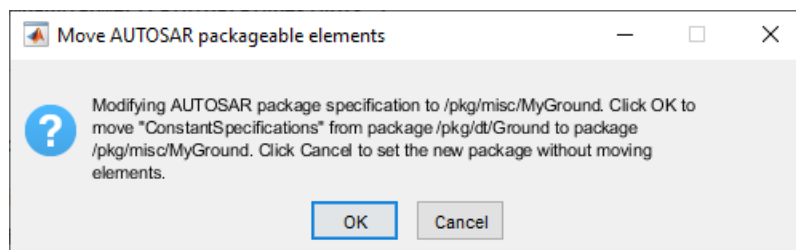
For an example of configuring and exporting AUTOSAR packages, see “Export AUTOSAR Packages” on page 4-89.

Control AUTOSAR Elements Affected by Package Path Modifications

If you modify an AUTOSAR package path, and if packageable elements of that category are affected, you can:

- Move the elements from the existing package to the new package.
- Set the new package path without moving the elements.

If you modify a package path in the AUTOSAR Dictionary, and if packageable elements of that category are affected, a dialog box opens. For example, if you modify the XML option **ConstantSpecification Package** from path value `/pkg/dt/Ground` to `/pkg/misc/MyGround`, the software opens the following dialog box.



To move AUTOSAR constant specification elements to the new package, click **OK**. To set the new package path without moving the elements, click **Cancel**.

If you programmatically modify a package path, you can use the `MoveElements` property to specify handling of affected elements. The property can be set to `All` (the default), `None`, or `Alert`. If you specify `Alert`, and if packageable elements are affected, the software opens the dialog box with **OK** and **Cancel** buttons.

For example, the following code sets a new constant specification package path without moving existing constant specification elements to the new package.

```
hModel = 'autosar_swc_expfncs';
openExample(hModel);
arProps = autosar.api.getAUTOSARProperties(hModel);
set(arProps, 'XmlOptions', 'ConstantSpecificationPackage', '/pkg/misc/MyGround', ...
    'MoveElements', 'None');
```

Export AUTOSAR Packages

This example shows how to configure and export AUTOSAR packages for an AUTOSAR software component that originated in Simulink.

- 1 Open a model that you configured for the AUTOSAR system target file and that models an AUTOSAR software component. This example uses the example model `autosar_swc_expfncs`.

```
openExample('autosar_swc_expfncs');
```

- 2 Open the AUTOSAR Dictionary and select **XML Options**. Here are initial settings for some of the AUTOSAR package parameters.

View and Edit XML Options	
XML Options Source:	Inlined
Exported XML File Packaging:	Single file
Package Paths	
Datatype Package:	/pkg/dt
Interface Package:	/pkg/if
AUTOSAR Platform Types	
Implementation Platform Types Package:	
User-defined ImplementationDataType References:	BaseTypeReference
Native Declaration:	PlatformTypeName
Platform Type Names:	AUTOSAR4.x
Additional Packages	
ApplicationDataType Package:	
SwBaseType Package:	/pkg/dt/SwBaseTypes
DataTypeMappingSet Package:	
ConstantSpecification Package:	/pkg/dt/Ground

In this example, **Exported XML file packaging** is set to `Single file`, which generates a single, unified ARXML file. If you prefer multiple, modular ARXML files, change the setting to `Modular`.

- 3 Configure packages for one or more AUTOSAR elements that your model exports to ARXML code. For each package, enter a path to define its location in the AR-PACKAGE structure. Click **Apply**.

The example model exports multiple AUTOSAR constant specifications. This example changes the **ConstantSpecification Package** parameter from `/pkg/dt/Ground` to `/pkg/misc/MyGround`.

View and Edit XML Options	
XML Options Source:	Inlined
Exported XML File Packaging:	Single file
Package Paths	
Datatype Package:	/pkg/dt
Interface Package:	/pkg/if
AUTOSAR Platform Types	
Implementation Platform Types Package:	
User-defined ImplementationDataType References:	BaseTypeReference
Native Declaration:	PlatformTypeName
Platform Type Names:	AUTOSAR4.x
Additional Packages	
ApplicationDataType Package:	
SwBaseType Package:	/pkg/dt/SwBaseTypes
DataTypeMappingSet Package:	
ConstantSpecification Package:	/pkg/misc/MyGround

- 4 Generate code for the model.
- 5 Open the generated file `modelName.arxml`. (If you set **Exported XML file packaging** to `Modular`, open the generated file `modelName_datatype.arxml`.)
- 6 Search the XML code for the packages that you configured, for example, using the text AR-PACKAGE or an element name. For the example model, searching `autosar_swc_expcns.arxml` for the text `MyGround` finds the constant specification package and many references to it. Here is a sample code excerpt.

```

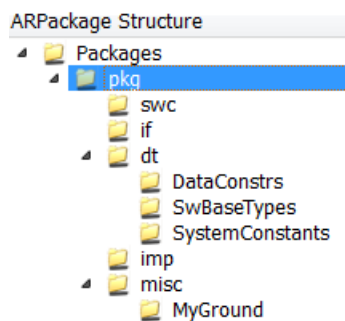
<AR-PACKAGE>
  <SHORT-NAME>MyGround</SHORT-NAME>
  <ELEMENTS>
    <CONSTANT-SPECIFICATION UUID="34e0d5e9-e53a-57d0-0ddf-6294ff4da42e">
      <SHORT-NAME>DefaultInitValue_Double</SHORT-NAME>
      <VALUE-SPEC>
        <NUMERICAL-VALUE-SPECIFICATION>
          <SHORT-LABEL>DefaultInitValue_Double</SHORT-LABEL>
          <VALUE>0</VALUE>
        </NUMERICAL-VALUE-SPECIFICATION>
      </VALUE-SPEC>
    </CONSTANT-SPECIFICATION>
    <CONSTANT-SPECIFICATION UUID="0a7c964c-db7a-529f-b836-1261c80143fd">
      <SHORT-NAME>DefaultInitValu_036fab392deee624</SHORT-NAME>
      <VALUE-SPEC>
        <ARRAY-VALUE-SPECIFICATION>
          <SHORT-LABEL>DefaultInitValu_036fab392deee624</SHORT-LABEL>
          <ELEMENTS>
            <CONSTANT-REFERENCE>
              <SHORT-LABEL>DefaultInitValu_d172891e92bde674</SHORT-LABEL>
              <CONSTANT-REF DEST="CONSTANT-SPECIFICATION"/>/pkg/misc/MyGround/DefaultInitValue_Double</CONSTANT-REF>
            </CONSTANT-REFERENCE>
            <CONSTANT-REFERENCE>
              <SHORT-LABEL>DefaultInitValu_2d898f9704c2f030</SHORT-LABEL>
              <CONSTANT-REF DEST="CONSTANT-SPECIFICATION"/>/pkg/misc/MyGround/DefaultInitValue_Double</CONSTANT-REF>
            </CONSTANT-REFERENCE>
          </ELEMENTS>
        </ARRAY-VALUE-SPECIFICATION>
      </VALUE-SPEC>
    </CONSTANT-SPECIFICATION>
  </ELEMENTS>
</AR-PACKAGE>

```

AR-PACKAGE Location in Exported ARXML Files

Grouping AUTOSAR elements into AUTOSAR packages (AR-PACKAGES) is logically distinct from the ARXML output file packaging that the AUTOSAR configuration parameter **Exported XML file packaging** controls. Whether you set **Exported XML file packaging** to `Single file` or `Modular`, ARXML export preserves the configured AR-PACKAGE structure.

Suppose that you configure the example model `autosar_swc_expfncns` with the following AR-PACKAGE structure. (See the steps in “Export AUTOSAR Packages” on page 4-89). In this configuration, the path of the constant specification package has been changed from the initial model setting, `/pkg/dt/Ground`, to `/pkg/misc/myGround`.



If you export this AR-PACKAGE structure into a single file (**Exported XML file packaging** is set to `Single file`), the exported ARXML code preserves the configured AR-PACKAGE structure.

`autosar_swc_expfncns.arxml`:

```

<AR-PACKAGES>
  <AR-PACKAGE>
    <SHORT-NAME>pkg</SHORT-NAME>
    ...

```

```

        <SHORT-NAME>swc</SHORT-NAME>
        ...
        <SHORT-NAME>if</SHORT-NAME>
        ...
        <SHORT-NAME>dt</SHORT-NAME>
        ...
            <SHORT-NAME>SwBaseTypes</SHORT-NAME>
            ...
        <SHORT-NAME>misc</SHORT-NAME>
        ...
            <SHORT-NAME>MyGround</SHORT-NAME>
            ...
        <SHORT-NAME>imp</SHORT-NAME>
        ...
    </AR-PACKAGE>
</AR-PACKAGES>

```

If you export the same AR-PACKAGE structure into multiple files (**Exported XML file packaging** is set to **Modular**), the exported ARXML code preserves the configured AR-PACKAGE structure, distributed across multiple files.

The exporter maps packageable AUTOSAR elements to ARXML files based on element category, not package path. For example, the exporter maps the data-type-oriented ConstantSpecification package, /pkg/misc/myGround, to the data types ARXML file, autosar_swc_expfncs_datatype.arxml.

autosar_swc_expfncs_component.arxml:

```

<AR-PACKAGES>
  <AR-PACKAGE>
    <SHORT-NAME>pkg</SHORT-NAME>
    ...
      <SHORT-NAME>swc</SHORT-NAME>
      ...
    </AR-PACKAGE>
  </AR-PACKAGES>

```

autosar_swc_expfncs_datatype.arxml:

```

<AR-PACKAGES>
  <AR-PACKAGE>
    <SHORT-NAME>pkg</SHORT-NAME>
    ...
      <SHORT-NAME>dt</SHORT-NAME>
      ...
        <SHORT-NAME>SwBaseTypes</SHORT-NAME>
        ...
      <SHORT-NAME>misc</SHORT-NAME>
      ...
        <SHORT-NAME>MyGround</SHORT-NAME>
        ...
    </AR-PACKAGE>
  </AR-PACKAGES>

```

autosar_swc_expfncs_implementation.arxml:

```

<AR-PACKAGES>
  <AR-PACKAGE>
    <SHORT-NAME>pkg</SHORT-NAME>
    ...
      <SHORT-NAME>imp</SHORT-NAME>
      ...
    </AR-PACKAGE>
  </AR-PACKAGES>

```

autosar_swc_expfncs_interface.arxml:

```

<AR-PACKAGES>
  <AR-PACKAGE>
    <SHORT-NAME>pkg</SHORT-NAME>

```

```
    ...
    <SHORT-NAME>if</SHORT-NAME>
    ...
  </AR-PACKAGE>
</AR-PACKAGES>
```

See Also

Related Examples

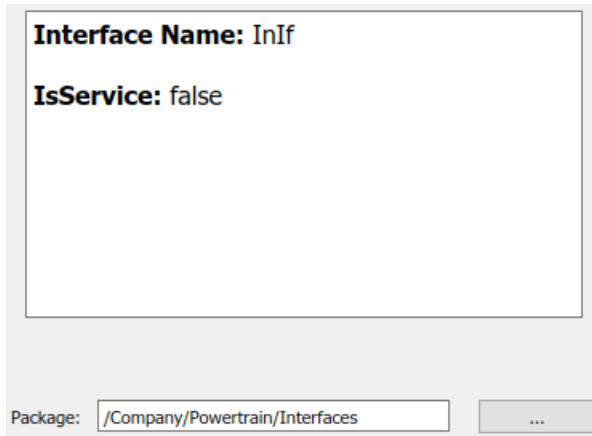
- “Import AUTOSAR XML Descriptions Into Simulink” on page 3-13
- “Import AUTOSAR Adaptive Software Descriptions” on page 6-12
- “Configure AUTOSAR Package for Component, Interface, CompuMethod, or SwAddrMethod” on page 4-94
- “Configure AUTOSAR XML Options” on page 4-43
- “Configure AUTOSAR Adaptive XML Options” on page 6-33
- “Configure AUTOSAR XML Export” on page 4-319
- “Configure AUTOSAR Code Generation” on page 5-7
- “Configure AUTOSAR Adaptive Code Generation” on page 6-73

More About

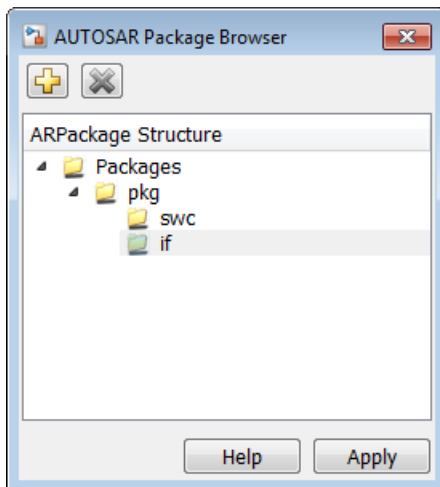
- “AUTOSAR Component Configuration” on page 4-3


Configure AUTOSAR Package for Component, Interface, CompuMethod, or SwAddrMethod

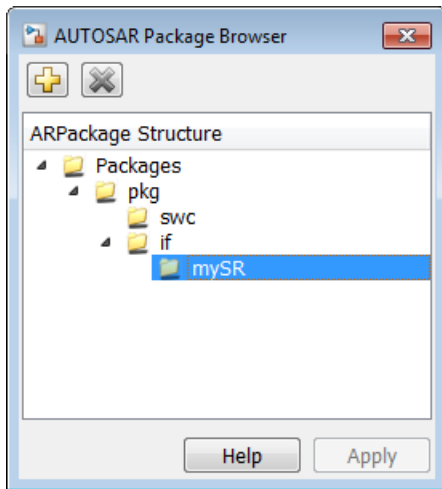
As part of configuring an AUTOSAR component, interface, CompuMethod, or SwAddrMethod, you specify the AUTOSAR package (AR-PACKAGE) to be generated for individual components, interfaces, CompuMethods, or SwAddrMethods in your configuration. For example, here is the AUTOSAR Dictionary view for an individual interface.



You can enter a package path in the **Package** parameter field, or use the AUTOSAR Package Browser to select a package. To open the browser, click the button to the right of the **Package** field. The AUTOSAR Package Browser opens.



In the browser, you can select an existing package, or create and select a new package. To create a new package, select the containing folder for the new package and click the **Add** button . For example, to add a new interface package, select the `if` folder and click the **Add** button. Then select the new subpackage and edit its name.



When you apply your changes in the browser, the interface **Package** parameter value is updated with your selection.

Package:

For more information about AR-PACKAGES, see “Configure AUTOSAR Packages” on page 4-84.

See Also

Related Examples

- “Configure AUTOSAR Elements and Properties” on page 4-8
- “Configure AUTOSAR Adaptive Elements and Properties” on page 6-21
- “Configure and Map AUTOSAR Component Programmatically” on page 4-293

More About

- “AUTOSAR Component Configuration” on page 4-3

Configure AUTOSAR Sender-Receiver Communication

In AUTOSAR port-based sender-receiver (S-R) communication, AUTOSAR software components read and write data to other components or services. To implement S-R communication, AUTOSAR software components define:

- An AUTOSAR sender-receiver interface with data elements.
- AUTOSAR provide and require ports that send and receive data.


In Simulink, you can:

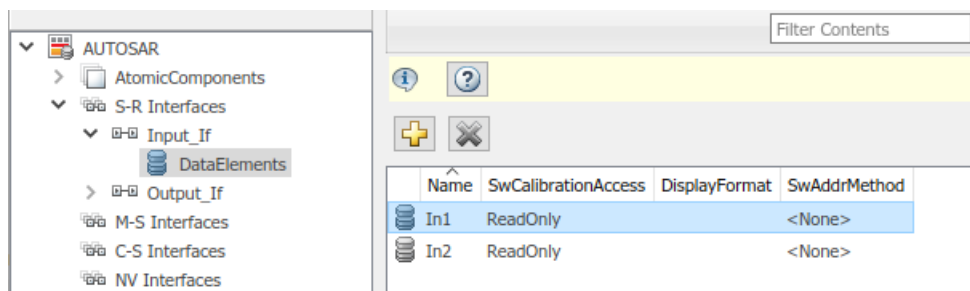
- 1 Create AUTOSAR S-R interfaces and ports by using the AUTOSAR Dictionary.
- 2 Model AUTOSAR provide and require ports by using Simulink root-level outports and inports.
- 3 Map the outports and inports to AUTOSAR provide and require ports by using the Code Mappings editor.

For queued sender-receiver communication, see “Configure AUTOSAR Queued Sender-Receiver Communication” on page 4-112.

Configure AUTOSAR Sender-Receiver Interface

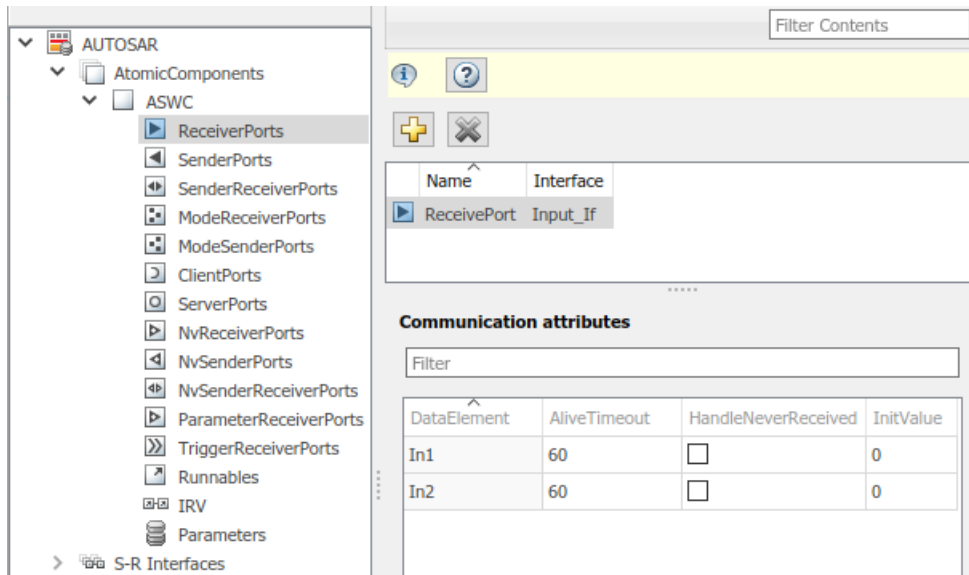
To create an S-R interface and ports in Simulink:

- 1 Open the AUTOSAR Dictionary and select **S-R Interfaces**. Click the **Add** button  to create a new AUTOSAR S-R data interface. Specify its name and the number of associated S-R data elements.
- 2 Select and expand the new S-R interface. Select **DataElements**, and modify the AUTOSAR data element attributes.



- 3 In the AUTOSAR Dictionary, expand the **AtomicComponents** node and select an AUTOSAR component. Expand the component.
- 4 Select and use the **ReceiverPorts**, **SenderPorts**, and **SenderReceiverPorts** views to add AUTOSAR S-R ports that you want to associate with the new S-R interface. For each new S-R port, select the S-R interface you created.

Optionally, examine the communication attributes for each S-R port and modify where required. For more information, see “Configure AUTOSAR Sender-Receiver Port ComSpecs” on page 4-108.




- 5 Open the Code Mappings editor. Select and use the **Inports** and **Outports** tabs to map Simulink inports and outports to AUTOSAR S-R ports. For each inport or outport, select an AUTOSAR port, data element, and data access mode.

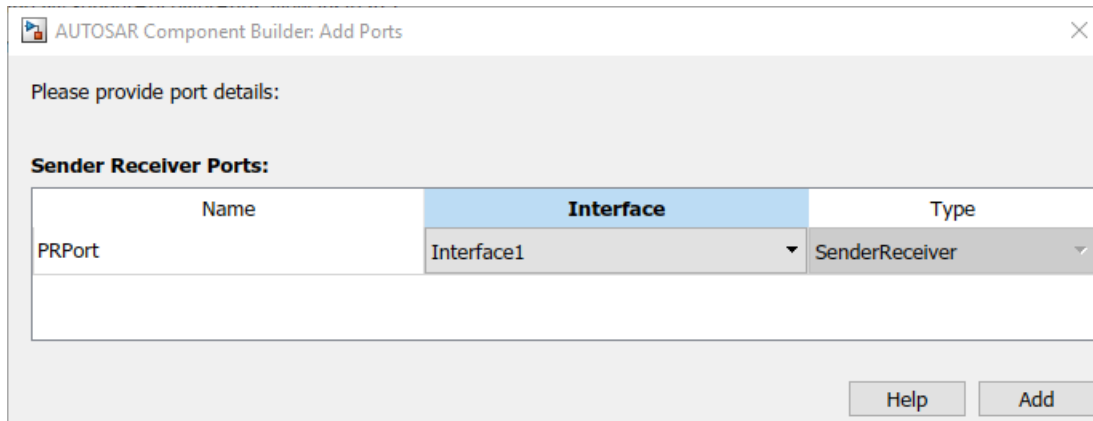
Source	DataAccessMode	Port	Element
In1_1s	ImplicitReceive	ReceivePort	In1
In2_2s	ImplicitReceive	ReceivePort	In2

Configure AUTOSAR Provide-Require Port

AUTOSAR Release 4.1 introduced the AUTOSAR provide-require port (PRPort). Modeling an AUTOSAR PRPort involves using a Simulink inport and outport pair with matching data type, dimension, and signal type. You can associate a PRPort with a sender-receiver (S-R) interface or a nonvolatile (NV) data interface.

To configure an AUTOSAR PRPort for S-R communication in Simulink:

- 1 Open a model that is configured for AUTOSAR, and in which a runnable has an inport and an outport suitable for pairing into an AUTOSAR PRPort. In this example, the RPort_DE1 inport and PPort_DE1 outport both use data type int8, port dimension 1, and signal type real.
- 2 Open the AUTOSAR Dictionary and navigate to the **SenderReceiverPorts** view. (To configure a PRPort for NV communication, use the **NvSenderReceiverPorts** view instead.)
- 3 To add a sender-receiver port, click the **Add** button . In the Add Ports dialog box, specify **Name** as PRPort and select an **Interface** from the list of available S-R interfaces. Click **Add**.




- 4 Open the Code Mappings editor and select the **Inports** tab. To map a Simulink inport to the AUTOSAR sender-receiver port you created, select the inport, set **Port** to the value PRPort, and set **Element** to a data element that the inport and output port will share.

Source	DataAccessMode	Port	Element
RPort_DE1	ImplicitReceive	PRPort	DE1
RPort_DE1_ErrorStatus	ErrorStatus	RPort	DE1
RPort_DE2	ImplicitReceive	RPort	DE2

- 5 Select the **Outports** tab. To map a Simulink output port to the AUTOSAR sender-receiver port you created, select the output port, set **Port** to the value PRPort, and set **Element** to the same data element selected in the previous step.

Source	DataAccessMode	Port	Element
PPort_DE1	ImplicitSend	PRPort	DE1
PPort_DE2	ImplicitSend	PPort	DE2
PPort_DE3	ImplicitSend	PPort	DE3
PPort_DE4	ImplicitSend	PPort	DE4

- 6 Click the **Validate** button  to validate the updated AUTOSAR component configuration. If errors are reported, address them and then retry validation. A common error flagged by validation is mismatched properties between the inport and output port that are mapped to the AUTOSAR PRPort.

Alternatively, you can programmatically add and map a PRPort port using AUTOSAR property and map functions. The following example adds an AUTOSAR PRPort (sender-receiver port) and then maps it to a Simulink inport and output port pair.

```
hModel = 'my_autosar_expfncs';
open_system(hModel)
arProps = autosar.api.getAUTOSARProperties(hModel);
swcPath = find(arProps,[], 'AtomicComponent')

swcPath =
    'ASWC'

add(arProps, 'ASWC', 'SenderReceiverPorts', 'PRPort', 'Interface', 'Interface1')
prportPath = find(arProps,[], 'DataSenderReceiverPort')
```

```

prportPath =
    'ASWC/PRPort'

slMap = autosar.api.getSimulinkMapping(hModel);
mapInport(slMap, 'RPort_DE1', 'PRPort', 'DE1', 'ImplicitReceive')
mapOutport(slMap, 'PPort_DE1', 'PRPort', 'DE1', 'ImplicitSend')
[arPortName, arDataElementName, arDataAccessMode] = getOutport(slMap, 'PPort_DE1')

arPortName =
    PRPort

arDataElementName =
    DE1

arDataAccessMode =
    ImplicitSend

```

Configure AUTOSAR Receiver Port for IsUpdated Service

AUTOSAR defines quality-of-service attributes, such as `ErrorStatus` and `IsUpdated`, for sender-receiver interfaces. The `IsUpdated` attribute allows an AUTOSAR explicit receiver to detect whether a receiver port data element has received data since the last read occurred. When data is idle, the receiver can save computational resources.

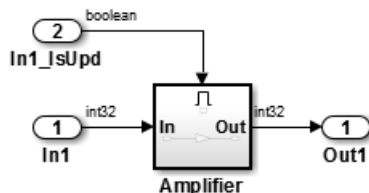
For the sender, the AUTOSAR Runtime Environment (RTE) sets the status of an update flag, indicating whether the data element has been written. The receiver calls the `Rte_IsUpdated_Port_Element` API, which reads the update flag and returns a value indicating whether the data element has been updated since the last read.

In Simulink, you can:

- Import an AUTOSAR receiver port for which `IsUpdated` service is configured.
- Configure an AUTOSAR receiver port for `IsUpdated` service.
- Generate C and ARXML code for an AUTOSAR receiver port for which `IsUpdated` service is configured.

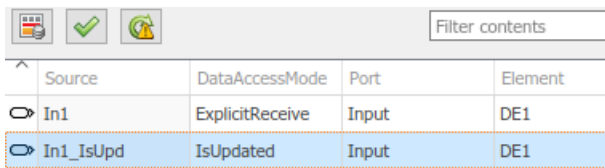
To model `IsUpdated` service in Simulink, you pair an inport that is configured for `ExplicitReceive` data access with a new inport configured for `IsUpdated` data access. To configure an AUTOSAR receiver port for `IsUpdated` service:

- 1 Open a model for which an AUTOSAR sender-receiver interface is configured.
- 2 Identify the inport that corresponds to the AUTOSAR receiver port for which `IsUpdated` service is required. Create a second inport, set its data type to `boolean`, and connect it to the same block. For example:




- 3 Open the Code Mappings editor. Select the **Inports** tab. In the inports view, configure the mapping properties for both inports.
 - a If the data inport is not already configured, set **DataAccessMode** to `ExplicitReceive`. Select **Port** and **Element** values that map the inport to the AUTOSAR receiver port and data element for which `IsUpdated` service is required.

- b** For the quality-of-service inport, set **DataAccessMode** to **IsUpdated**. Select **Port** and **Element** values that exactly match the data inport.



Source	DataAccessMode	Port	Element
In1	ExplicitReceive	Input	DE1
In1_IsUpd	IsUpdated	Input	DE1

- 4** To validate the AUTOSAR component configuration, click the **Validate** button .
- 5** Build the model and inspect the generated code. The generated C code contains an `Rte_IsUpdated` API call.

```
if (Rte_IsUpdated_Input_DE1()) {
    ...
    Rte_Read_Input_DE1(&tmp);
    ...
}
```

The exported ARXML code contains the `ENABLE-UPDATE` setting `true` for the AUTOSAR receiver port.

```
<R-PORT-PROTOTYPE UUID="...">
  <SHORT-NAME>Input</SHORT-NAME>
  <REQUIRED-COM-SPECS>
    <NONQUEUED-RECEIVER-COM-SPEC>
      <DATA-ELEMENT-REF DEST="VARIABLE-DATA-PROTOTYPE">/pkg/if/Input/DE1
      </DATA-ELEMENT-REF>
      ...
      <ENABLE-UPDATE>true</ENABLE-UPDATE>
      ...
    </NONQUEUED-RECEIVER-COM-SPEC>
  </REQUIRED-COM-SPECS>
  ...
</R-PORT-PROTOTYPE>
```

Configure AUTOSAR Sender-Receiver Data Invalidation

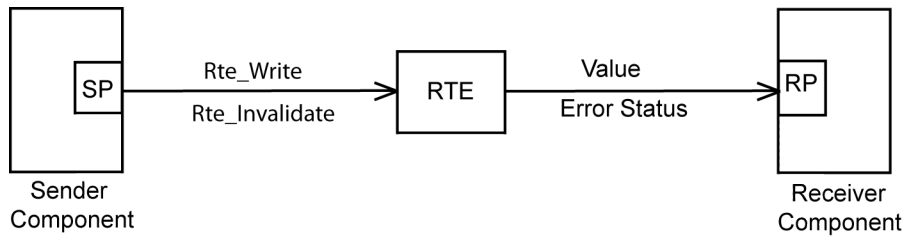
The AUTOSAR standard defines an invalidation mechanism for AUTOSAR data elements used in sender-receiver (S-R) communication. A sender component can notify a downstream receiver component that data in a sender port is invalid. Each S-R data element can have an invalidation policy. In Simulink, you can:

- Import AUTOSAR sender-receiver data elements for which an invalidation policy is configured.
- Use a Signal Invalidation block to model sender-receiver data invalidation for simulation and code generation. Using block parameters, you can specify a signal invalidation policy and an initial value for an S-R data element.
- Generate C code and ARXML descriptions for AUTOSAR sender-receiver data elements for which an invalidation policy is configured.

For each S-R data element, you can set the Signal Invalidation block parameter **Signal invalidation policy** to `Keep`, `Replace`, or `DontInvalidate`. If an input data value is invalid (invalidation control flag is `true`), the resulting action is determined by the value of **Signal invalidation policy**:

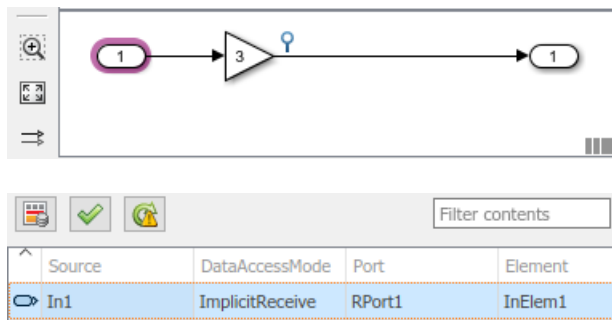
- `Keep` - Replace the input data value with the last valid signal value.
- `Replace` - Replace the input data value with an **Initial value** parameter.

- DontInvalidate - Do not replace the input data value.



To configure an invalidation policy for an AUTOSAR S-R data element in Simulink:

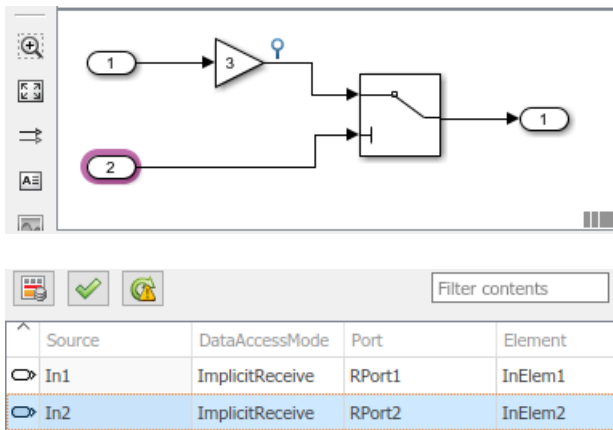
- 1 Open a model for which an AUTOSAR sender-receiver interface is configured. For example, suppose that:
 - A Simulink output named Out is mapped to AUTOSAR sender port PPort and data element OutElem. In the AUTOSAR Dictionary, AUTOSAR sender port PPort selects S-R interface Out, which contains the data element OutElem.
 - A Simulink inport named In1 is mapped to AUTOSAR receiver port RPort1 and data element InElem1. In the AUTOSAR Dictionary, AUTOSAR receiver port RPort1 selects S-R interface In1, which contains the data element InElem1. In the Code Mappings editor, **Imports** tab, here is the mapping for inport In1.



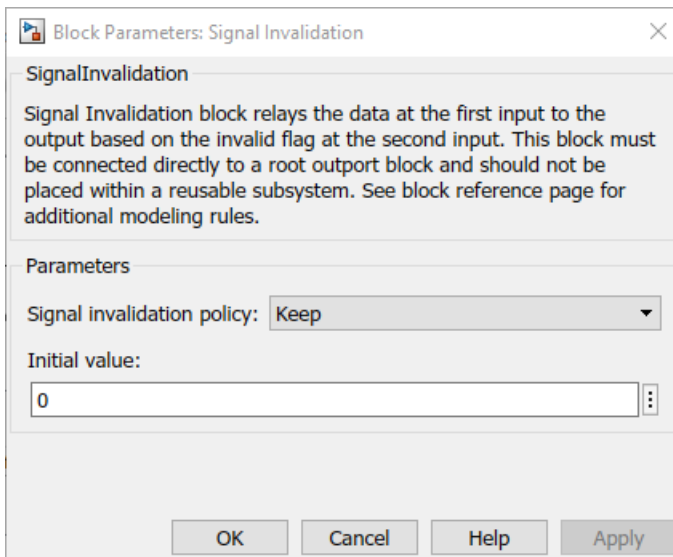
- 2 Add a Signal Invalidation block to the model.
 - a The block must be connected directly to a root output block. Connect the block to root output Out.
 - b Connect the first block input, a data value, to the data path from root inport In1.
 - c For the second block input, an invalidation control flag, add a root inport named In2 to the model. Set its data type to scalar `boolean`. Map the new inport to a second AUTOSAR sender port. If a second AUTOSAR sender port does not exist, use the AUTOSAR Dictionary to create the AUTOSAR port, S-R interface, and data element.

In this example, Simulink inport In2 is mapped to AUTOSAR receiver port RPort2 and data element InElem2. In the AUTOSAR Dictionary, AUTOSAR receiver port RPort2 selects S-R interface In2, which contains the data element InElem2.

Connect the second block input to root inport In2.



- View the Signal Invalidation block parameters dialog box. Examine the **Signal invalidation policy** and **Initial value** attributes. For more information, see the Signal Invalidation block reference page.



- Open the Code Mappings editor and select the **Outports** tab. For the root output Out, verify that the AUTOSAR data access mode is set to **ExplicitSend** or **EndToEndWrite**.

Source	DataAccessMode	Port	Element
Out	ExplicitSend	PPort	OutElem

- To validate the AUTOSAR component configuration, open the Code Mappings editor and click the **Validate** button .
- Build the model and inspect the generated code. When the signal is valid, the generated C code calls `Rte_Write_Port_Element`. When the signal is invalid, the C code calls `Rte_Invalidate_Port_Element`.

```

/* SignalInvalidation: '<Root>/Signal Invalidation' incorporates:
 * Inport: '<Root>/In2'
 */

```



```

if (!Rte_IRead_Runnable_Step_RPort2_InElem2()) {
  /* Outport: '<Root>/Out' */
  (void) Rte_Write_PPort_OutElem(mSignalInvalidation_B.Gain);
} else {
  Rte_Invalidate_PPort_OutElem();
}

```

The exported ARXML code contains the invalidation setting for the data element.

```

<INVALIDATION-POLICY>
  <DATA-ELEMENT-REF DEST="VARIABLE-DATA-PROTOTYPE">/pkg/if/Out/OutElem</DATA-ELEMENT-REF>
  <HANDLE-INVALID>KEEP</HANDLE-INVALID>
</INVALIDATION-POLICY>

```

Configure AUTOSAR S-R Interface Port for End-To-End Protection

AUTOSAR end-to-end (E2E) protection for sender and receiver ports is based on the E2E library. E2E is a C library that you use to transmit data securely between AUTOSAR components. End-to-end protection adds additional information to an outbound data packet. The component receiving the packet can then verify independently that the received data packet matches the sent packet. Potentially, the receiving component can detect errors and take action.

For easier integration of AUTOSAR generated code with AUTOSAR E2E solutions, Embedded Coder supports AUTOSAR E2E protection. In Simulink, you can:

- Import AUTOSAR sender port and receiver ports for which E2E protection is configured.
- Configure an AUTOSAR sender or receiver port for E2E protection.
- Generate C and ARXML code for AUTOSAR sender and receiver ports for which E2E protection is configured.

Simulink supports using either the E2E Transformer method or the E2E Protection Wrapper to implement end-to-end protection in the generated code. You can retrieve which end-to-end protection method is configured by using the function `getDataDefaults`. You set the end-to-end protection method by using the function `setDataDefaults`.

- E2E Transformer:
 - Is invoked by the AUTOSAR runtime environment (RTE). E2E transformer generates `Rte_Write` and `Rte_Read` calls that take an additional transformer error argument, `Rte_TransformerError`, to indicate error status.
 - Is supported when using AUTOSAR schema version 4.2 and later.
- E2E Protection Wrapper:
 - Inserts a wrapper around the `Rte_Write` and `Rte_Read` functions. The body of the E2E protection wrapper that contains the `Rte_Write` and `Rte_Read` calls is implemented external to the generated code.
 - Is the default end-to-end protection method.

Configure E2E protection for individual AUTOSAR sender and receiver ports that use explicit write and read data access modes. When you change the data access mode of an AUTOSAR port from explicit write to end-to-end write, or from explicit read to end-to-end read.

- Simulation behavior is unaffected.
- Code generation is similar to explicit write and read, with these differences:

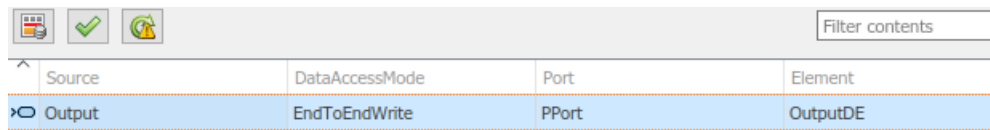
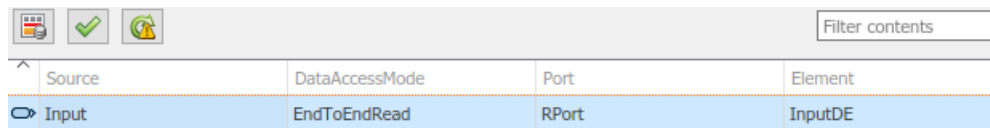
Generated Item	E2E Protection Wrapper	E2E Transformer
Generated Code for Initialization	Calls E2EPW_ReadInit_<Port> or E2EPW_WriteInit_<Port>	None
Generated Code for Function Signature	Uses uint32 E2EPW_Read_<Port>(data*) or (void) E2EPW_Write_<Port>	Uses uint8 Rte_Read_<Port>(data*, Rte_TransformerError*) or (void)Rte_Write_<Port>(data, Rte_TransformerError*)
ARXML Exporter for Receiver and Sender COM-SPECs	Generates property USES-END-TO-END-PROTECTION with value true	Generates property USES-END-TO-END-PROTECTION with value true
ARXML Exporter for Receiver and Sender API Extensions PORT-API-OPTIONS	None	Generates property ERROR-HANDLING with value TRANSFORMER-ERROR-HANDLING


To configure an AUTOSAR sender or receiver port for E2E transformer protection:

- 1 Open a model for which an AUTOSAR sender-receiver interface is configured.
- 2 In the MATLAB Command Window, configure TransformerError as the default E2E protection method.

```
slMap = autosar.api.getSimulinkMapping(modelName);
setDataDefaults(slMap, 'InportsOutports', ...
    'EndToEndProtectionMethod', 'TransformerError');
```

- 3 Open the Configuration Parameters. Verify the AUTOSAR schema version is 4.2 or later.
- 4 Open the Code Mappings editor. Navigate to the Simulink inport or output that models the AUTOSAR receiver or sender port for which you want to configure E2E protection. Select the port.
- 5 Set the AUTOSAR data access mode to EndToEndRead (inport) or EndToEndWrite (output).



- 6 To validate the AUTOSAR component configuration, click the **Validate** button .
- 7 Build the model and inspect the generated code.

The generated C code contains RTE read and write API calls that pass the transformer error argument.

```
void Runnable(void)
{
```

```

Rte_TransformerError transformerError_Input;
float64 tmpRead;
...
/* Inport: '<Root>/Input' */
Rte_Read_RPort_InputDE(&tmpRead, &transformerError_Input);
...
/* Outport: '<Root>/Output'... */
(void) Rte_Write_PPort_OutputDE(data, &transformerError_Input);
...
}

```

The generated header file `Rte_model.h` contains the transformer error declaration.

```

/* Transformer Classes */
typedef enum {
    RTE_TRANSFORMER_UNSPECIFIED = 0x00,
    RTE_TRANSFORMER_SERIALIZER = 0x01,
    RTE_TRANSFORMER_SAFETY = 0x02,
    RTE_TRANSFORMER_SECURITY = 0x03,
    RTE_TRANSFORMER_CUSTOM = 0xff
} Rte_TransformerClass;

typedef uint8 Rte_TransformerErrorCode;

typedef struct {
    Rte_TransformerErrorCode errorCode;
    Rte_TransformerClass transformerClass;
} Rte_TransformerError;

```

The exported ARXML code contains the E2E settings for the AUTOSAR receiver and sender ports.

```

<REQUIRED-COM-SPECS>
  <NONQUEUED-RECEIVER-COM-SPEC>
    ...
    <USES-END-TO-END-PROTECTION>true</USES-END-TO-END-PROTECTION>
    ...
  <NONQUEUED-SENDER-COM-SPEC>
    ...
    <USES-END-TO-END-PROTECTION>true</USES-END-TO-END-PROTECTION>
    ...
</REQUIRED-COM-SPECS>
...
</PORT-API-OPTIONS>
  <PORT-API-OPTION>
    <ERROR-HANDLING>TRANSFORMER-ERROR-HANDLING</ERROR-HANDLING>
    <PORT-REF DEST="R-PORT-PROTOTYPE">/pkg/swc/ASWC/RPort</PORT-REF>
  </PORT-API-OPTION>
  ...
  <PORT-API-OPTION>
    <ERROR-HANDLING>TRANSFORMER-ERROR-HANDLING</ERROR-HANDLING>
    <PORT-REF DEST="P-PORT-PROTOTYPE">/pkg/swc/ASWC/PPort</PORT-REF>
  </PORT-API-OPTION>
  ...
</PORT-API-OPTIONS>

```

To configure an AUTOSAR sender or receiver port for E2E wrapper protection:

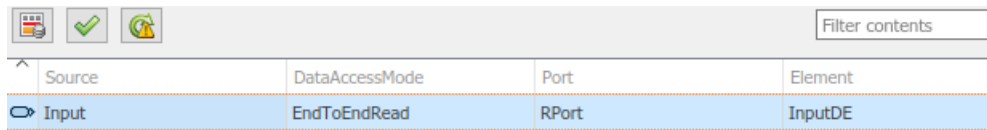
- 1 Open a model for which an AUTOSAR sender-receiver interface is configured.
- 2 In the MATLAB Command Window, configure `ProtectionWrapper` as the default E2E protection method.

```

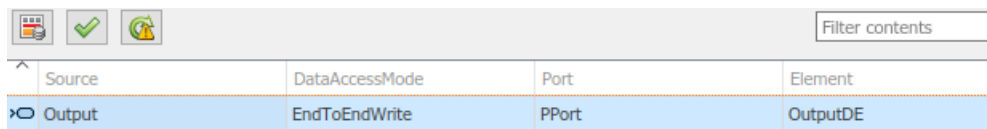
slMap = autosar.api.getSimulinkMapping(modelName);
setDataDefaults(slMap, 'InportsOutports', ...
    'EndToEndProtectionMethod', 'ProtectionWrapper');

```


- 3 Open the Code Mappings editor. Navigate to the Simulink inport or outputport that models the AUTOSAR receiver or sender port for which you want to configure E2E protection. Select the port.
- 4 Set the AUTOSAR data access mode to `EndToEndRead` (inport) or `EndToEndWrite` (outport).



Source	DataAccessMode	Port	Element
Input	EndToEndRead	RPort	InputDE



Source	DataAccessMode	Port	Element
Output	EndToEndWrite	PPort	OutputDE

- 5 To validate the AUTOSAR component configuration, click the **Validate** button .
- 6 Build the model and inspect the generated code. The generated C code contains E2E API calls.

```
void Runnable_Step(void)
{
    ...
    /* Inport: '<Root>/Input' */
    E2EPW_Read_RPort_InputDE(...);
    ...
    /* Outport: '<Root>/Output'... */
    (void) E2EPW_Write_PPport_OutputDE(...);
    ...
}
...
void Runnable_Init(void)
{
    ...
    /* End-to-End (E2E) initialization */
    E2EPW_ReadInit_RPort_InputDE();
    E2EPW_WriteInit_PPport_OutputDE();
    ...
}
```

The exported ARXML code contains the E2E settings for the AUTOSAR receiver and sender ports.

```
<NONQUEUED-RECEIVER-COM-SPEC>
    ...
    <USES-END-TO-END-PROTECTION>true</USES-END-TO-END-PROTECTION>
    ...
<NONQUEUED-SENDER-COM-SPEC>
    ...
    <USES-END-TO-END-PROTECTION>true</USES-END-TO-END-PROTECTION>
    ...
```

Configure AUTOSAR Receiver Port for DataReceiveErrorEvent

In AUTOSAR sender-receiver communication between software components, the Runtime Environment (RTE) raises a `DataReceiveErrorEvent` when the communication layer reports an error in data reception by the receiver component. For example, the event can indicate that the sender component failed to reply within an `AliveTimeout` limit, or that the sender component sent invalid data.

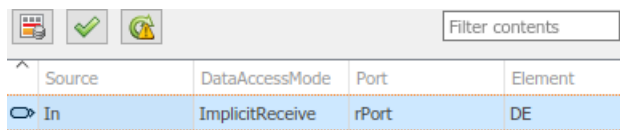
Embedded Coder supports creating `DataReceiveErrorEvents` in AUTOSAR receiver components. In Simulink, you can:

- Import an AUTOSAR DataReceiveErrorEvent definition.
- Define a DataReceiveErrorEvent.
- Generate ARXML code for AUTOSAR receiver ports for which a DataReceiveErrorEvent is configured.

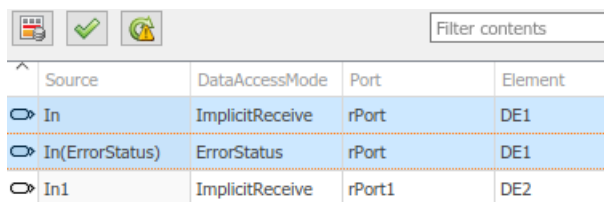
You should configure a DataReceiveErrorEvent for an AUTOSAR receiver port that uses ImplicitReceive, ExplicitReceive, or EndToEndRead data access mode.

To configure an AUTOSAR receiver port for a DataReceiveErrorEvent:


- 1 Open a model for which the receiver side of an AUTOSAR sender-receiver interface is configured.
- 2 Open the Code Mappings editor. Select the **Inports** tab. Select the data inport that is mapped to the AUTOSAR receiver port for which you want to configure a DataReceiveErrorEvent. Set its AUTOSAR data access mode to ImplicitReceive, ExplicitReceive, or EndToEndRead. Here are two examples, without and with a coupled ErrorStatus port.

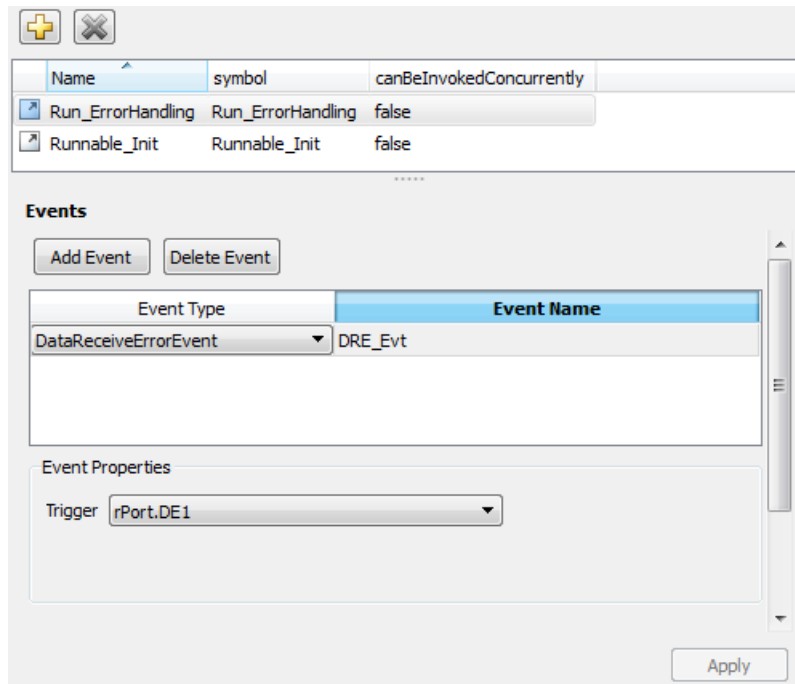


Source	DataAccessMode	Port	Element
In	ImplicitReceive	rPort	DE



Source	DataAccessMode	Port	Element
In	ImplicitReceive	rPort	DE1
In(ErrorStatus)	ErrorStatus	rPort	DE1
In1	ImplicitReceive	rPort1	DE2

- 3 Open the AUTOSAR Dictionary. Expand the **AtomicComponents** node. Expand the receiver component and select **Runnables**.
- 4 In the runnables view, create a runnable to handle DataReceiveErrorEvents.
 - a Click the **Add** button  to add a runnable entry.
 - b Select the new runnable entry to configure its name and other properties.
 - c Go to the **Events** pane, and configure a DataReceiveErrorEvent for the runnable. Click **Add Event**, select type DataReceiveErrorEvent, and enter an event name.
 - d Under **Event Properties**, select the trigger for the event. The selected trigger value indicates the AUTOSAR receiver port and the data element for which the runnable is handling DataReceiveErrorEvents.



Alternatively, you can programmatically create a DataReceiveErrorEvent.

```
arProps = autosar.api.getAUTOSARProperties(mdlname);
add(arProps,ibQName,'Events','DRE_Evt',...
    'Category','DataReceiveErrorEvent','Trigger','rPort.DE1',...
    'StartOnEvent',runnableQName);
```

- 5 Build the model and inspect the generated code. The exported ARXML code defines the error-handling runnable and its triggering event.

```
<EVENTS>
<DATA-RECEIVE-ERROR-EVENT UUID="...">
  <SHORT-NAME>DRE_Evt</SHORT-NAME>
  <START-ON-EVENT-REF DEST="RUNNABLE-ENTITY">
    /Root/mDemoModel_sw/ReceivingASWC/IB/Run_ErrorHandling</START-ON-EVENT-REF>
  <DATA-IREF>
    <CONTEXT-R-PORT-REF DEST="R-PORT-PROTOTYPE">
      /Root/mDemoModel_sw/ReceivingASWC/rPort</CONTEXT-R-PORT-REF>
    <TARGET-DATA-ELEMENT-REF DEST="VARIABLE-DATA-PROTOTYPE">
      /Root/Interfaces/In/DE</TARGET-DATA-ELEMENT-REF>
    </DATA-IREF>
  </DATA-RECEIVE-ERROR-EVENT>
</EVENTS>
...
<RUNNABLES>
...
<RUNNABLE-ENTITY UUID="...">
  <SHORT-NAME>Run_ErrorHandling</SHORT-NAME>
  <MINIMUM-START-INTERVAL>0</MINIMUM-START-INTERVAL>
  <CAN-BE-INVOKED-CONCURRENTLY>false</CAN-BE-INVOKED-CONCURRENTLY>
  ...
  <SYMBOL>Run_ErrorHandling</SYMBOL>
</RUNNABLE-ENTITY>
</RUNNABLES>
```

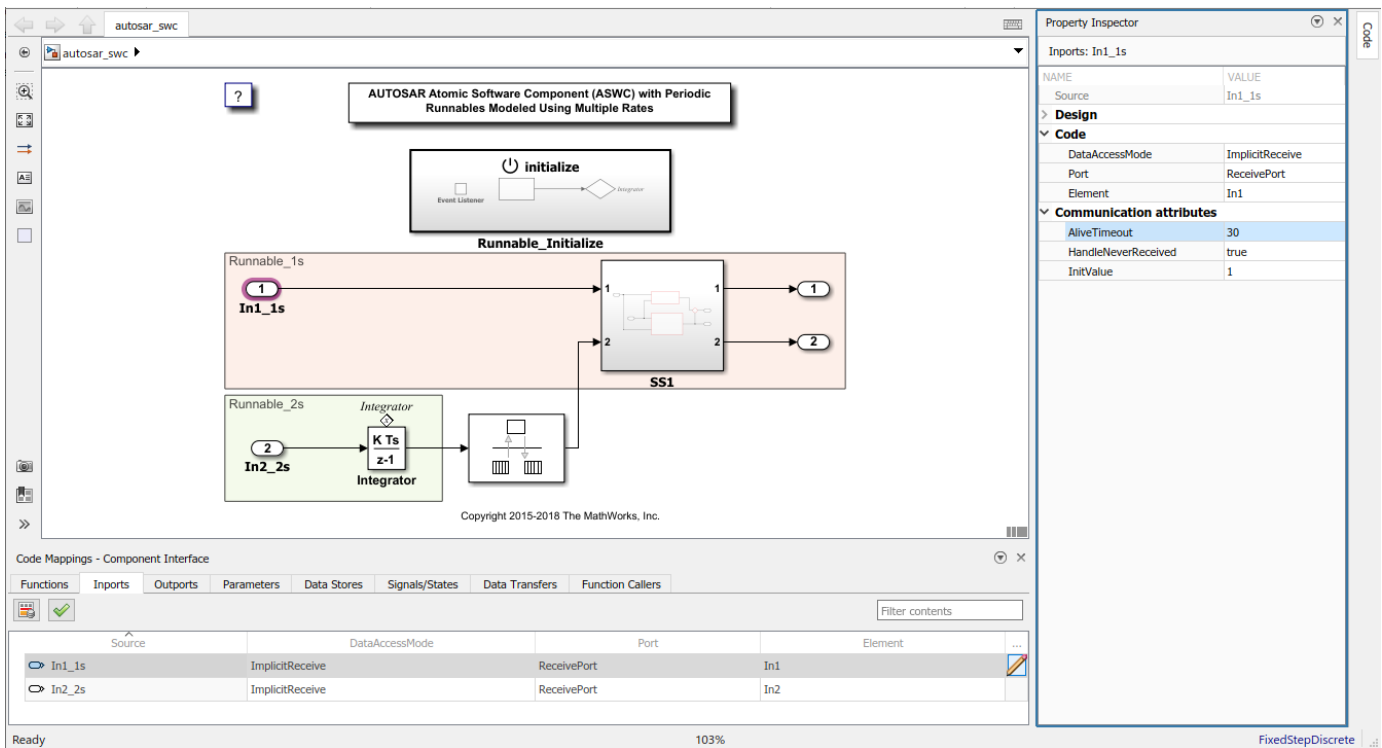
Configure AUTOSAR Sender-Receiver Port ComSpecs

In AUTOSAR software components, a sender or receiver port optionally can specify a communication specification (ComSpec). ComSpecs describe additional communication requirements for port data.

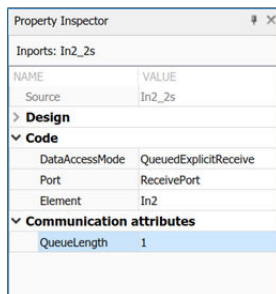
To model AUTOSAR sender and receiver ComSpecs in Simulink, you can:

- Import sender and receiver ComSpecs from ARXML files.
- Create sender and receiver ComSpecs in Simulink.
- For nonqueued sender ports, modify ComSpec attribute `InitValue`.
- For nonqueued receiver ports, modify ComSpec attributes `AliveTimeout`, `HandleNeverReceived`, and `InitValue`.
- For queued receiver ports, modify ComSpec attribute `QueueLength`.
- Export ComSpecs to ARXML files

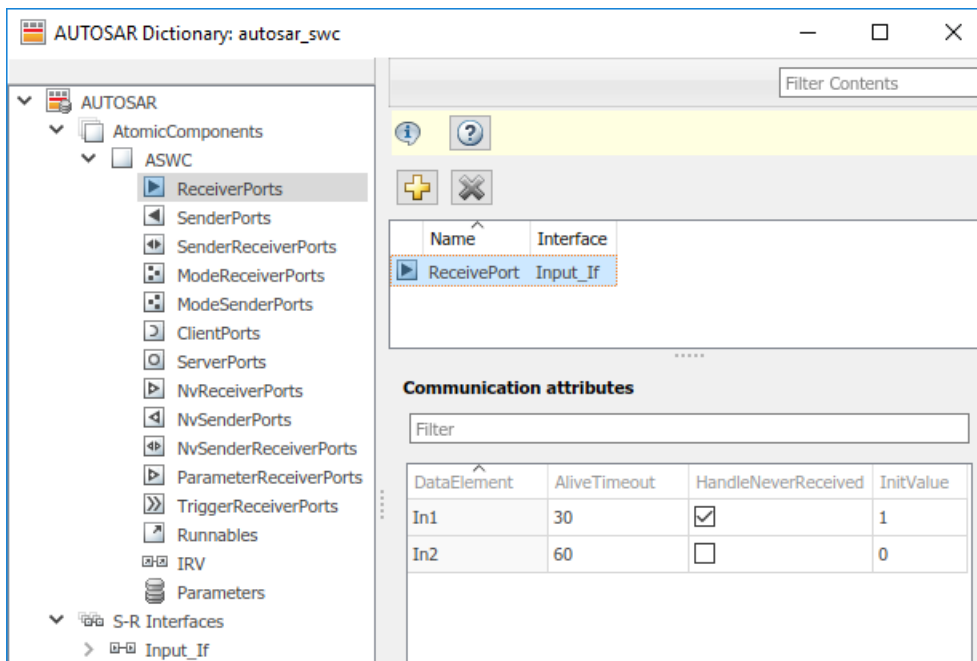
For example, if you create an AUTOSAR receiver port in Simulink, you use the Code Mappings editor to map a Simulink inport to the AUTOSAR receiver port and an S-R data element. You can then select the port and specify its ComSpec attributes.



Here are the properties for a queued receiver port.



If you import or create an AUTOSAR receiver port, you can use the AUTOSAR Dictionary to view and edit the ComSpec attributes of the mapped S-R data elements in the AUTOSAR port.



To programmatically modify ComSpec attributes of an AUTOSAR port, use the AUTOSAR property function set. For example:

```
hModel = 'autosar_swc';
openExample(hModel);
arProps = autosar.api.getAUTOSARProperties(hModel);

% Find ComSpec path
portPath = find(arProps,[],'DataReceiverPort','PathType','FullyQualified');
ifPath = find(arProps,[],'SenderReceiverInterface','Name','Input_If','PathType','FullyQualified');
dataElementPath = find(arProps,ifPath{1},'FlowData','Name','In1','PathType','FullyQualified');
infoPath = find(arProps,portPath{1},'PortInfo',...
    'PathType','FullyQualified','DataElements',dataElementPath{1});
comSpecPath = find(arProps,infoPath{1},'PortComSpec','PathType','FullyQualified');

% Set ComSpec attributes
set(arProps,comSpecPath{1},'AliveTimeout',30,'HandleNeverReceived',true,'InitValue',1);
get(arProps,comSpecPath{1},'AliveTimeout')
get(arProps,comSpecPath{1},'HandleNeverReceived')
get(arProps,comSpecPath{1},'InitValue')
```

To set the QueueLength attribute for a queued receiver port:

```
set(arProps,comSpecPath,'QueueLength',10);
```

When you generate code for an AUTOSAR model that specifies ComSpec attributes, the exported ARXML port descriptions include the ComSpec attribute values.

```
<PORTS>
  <R-PORT-PROTOTYPE UUID="...">
    <SHORT-NAME>ReceivePort</SHORT-NAME>
    <REQUIRED-COM-SPECS>
      <NONQUEUED-RECEIVER-COM-SPEC>
        <DATA-ELEMENT-REF DEST="VARIABLE-DATA-PROTOTYPE">
          /Company/Powertrain/Interfaces/Input_If/In1
        </DATA-ELEMENT-REF>
        ...
        <ALIVE-TIMEOUT>30</ALIVE-TIMEOUT>
        <HANDLE-NEVER-RECEIVED>true</HANDLE-NEVER-RECEIVED>
```



```

...
    <INIT-VALUE>
      <CONSTANT-REFERENCE>
        <SHORT-LABEL>DefaultInitValue_Double_1</SHORT-LABEL>
        <CONSTANT-REF DEST="CONSTANT-SPECIFICATION">
          /Company/Powertrain/Constants/DefaultInitValue_Double_1
        </CONSTANT-REF>
      </CONSTANT-REFERENCE>
    </INIT-VALUE>
  </NONQUEUED-RECEIVER-COM-SPEC>
</REQUIRED-COM-SPECS>
</R-PORT-PROTOTYPE>
</PORTS>
...
<CONSTANT-SPECIFICATION UUID="...">
  <SHORT-NAME>DefaultInitValue_Double_1</SHORT-NAME>
  <VALUE-SPEC>
    <NUMERICAL-VALUE-SPECIFICATION>
      <SHORT-LABEL>DefaultInitValue_Double_1</SHORT-LABEL>
      <VALUE>1</VALUE>
    </NUMERICAL-VALUE-SPECIFICATION>
  </VALUE-SPEC>
</CONSTANT-SPECIFICATION>

```

See Also

Signal Invalidation

Related Examples

- “Model AUTOSAR Communication” on page 2-21
- “Import AUTOSAR XML Descriptions Into Simulink” on page 3-13
- “Configure AUTOSAR Code Generation” on page 5-7

More About

- “AUTOSAR Component Configuration” on page 4-3

Configure AUTOSAR Queued Sender-Receiver Communication

In AUTOSAR queued sender-receiver (S-R) communication, AUTOSAR software components read and write data to other components or services. Data sent by an AUTOSAR sender software component is added to a queue provided by the AUTOSAR Runtime Environment (RTE). Newly received data does not overwrite existing unread data. Later, a receiver software component reads the data from the queue.

To implement queued S-R communication, AUTOSAR software components define:

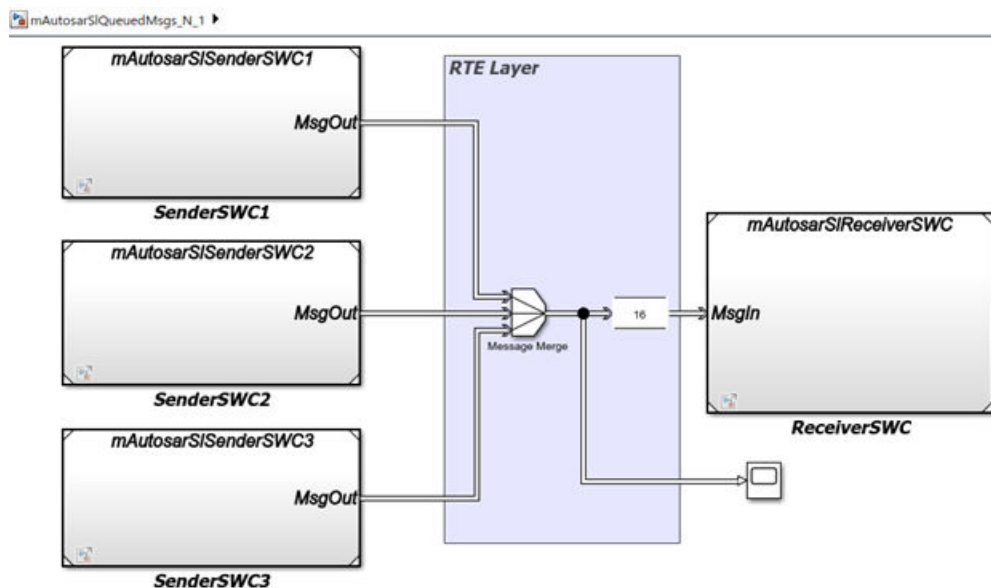
- An AUTOSAR sender-receiver interface with data elements.
- AUTOSAR provide and require ports that send and receive queued data.

In Simulink, you can:

- 1 Create AUTOSAR queued S-R interfaces and ports by using the AUTOSAR Dictionary.
- 2 Model AUTOSAR provide and require ports by using Simulink root-level outports and inports.
- 3 Map the outports and inports to AUTOSAR provide and require ports by using the Code Mappings editor. Set the AUTOSAR data access modes to `QueuedExplicitSend` or `QueuedExplicitReceive`.

To model sending and receiving AUTOSAR data using a queue, use Simulink Send and Receive blocks. If your queued S-R communication implementation involves states or requires decision logic, use Stateflow charts. You can handle errors that occur when the queue is empty or full. You can specify the size of the queue. For more information, see “Simulink Messages Overview”.

You can simulate AUTOSAR queued sender-receiver (S-R) communication between component models, for example, in a composition-level simulation. Data senders and receivers can run at different rates. Multiple data senders can communicate with a single data receiver.



To get started, you can import components with queued S-R interfaces and ports from ARXML files into Simulink, or use Simulink to create the interfaces and ports.

In this section...

“Simulink Workflow for Modeling AUTOSAR Queued Send and Receive” on page 4-113
“Configure AUTOSAR Sender and Receiver Components for Queued Communication” on page 4-114
“Implement AUTOSAR Queued Send and Receive Messaging” on page 4-115
“Configure Simulation of AUTOSAR Queued Sender-Receiver Communication” on page 4-118
“Simulate N-to-1 AUTOSAR Queued Sender-Receiver Communication” on page 4-119
“Simulate Event-Driven AUTOSAR Queued Sender-Receiver Communication” on page 4-122
“Implement AUTOSAR Queued Send and Receive By Using Stateflow Messaging” on page 4-126

Simulink Workflow for Modeling AUTOSAR Queued Send and Receive

This procedure outlines the general workflow for modeling AUTOSAR queued sender and receiver components in Simulink.

- 1** Configure one or more models as AUTOSAR queued sender components, and one model as an AUTOSAR queued receiver component. For each component model, use the AUTOSAR Dictionary and the Code Mappings editor to:
 - a** Create an S-R data interface and its data elements.
 - b** Create a sender or receiver port.
 - c** Map the sender or receiver port to a Simulink outputport for sending or inport for receiving. Set the AUTOSAR data access mode to QueuedExplicitSend or QueuedExplicitReceive.

For example, see “Configure AUTOSAR Sender and Receiver Components for Queued Communication” on page 4-114.

- 2** To implement AUTOSAR queued sender or receiver component behavior, use Simulink Send and Receive blocks. For more information, see “Simulink Messages Overview”.

If your queued S-R communication implementation involves states or requires decision logic, use Stateflow charts.

For more information, see “Implement AUTOSAR Queued Send and Receive Messaging” on page 4-115.

- 3** When you build an AUTOSAR queued sender or receiver component model:
 - Generated C code contains calls to AUTOSAR `Rte_Send_<port>_<DataElement>` or `Rte_Receive_<port>_<DataElement>` APIs.

The generated code handles the status of the message receive calls.

Handling of message send status (such as queue overflow) can only be modeled in Stateflow. The generated code handles message send status only if a queued sender component implements the Stateflow logic.

- Exported ARXML files contain descriptions for queued sender-receiver communication. The generated ComSpec for a queued port includes the port type and the queue length (based on Simulink message property `QueueCapacity`). In the `SwDataDefProps` generated for the queued port data element, `SwImplPolicy` is set to `Queued`.

- 4 To simulate AUTOSAR queued sender-receiver communication in Simulink, create a containing composition, system, or harness model. Include the queued sender and receiver components as referenced models.
- 5 To provide queueing logic between sender and receiver components, you can insert a Simulink Queue block or Stateflow logic. With a Queue block, you can simulate a queue with a specific capacity. If you connect sender and receiver components directly, Simulink inserts a default queue with capacity 1.

For an example of connecting components directly, see the 1-to-1 composition model used in “Configure Simulation of AUTOSAR Queued Sender-Receiver Communication” on page 4-118.


For examples of inserting a Queue block or Stateflow logic between sender and receiver components, see “Simulate N-to-1 AUTOSAR Queued Sender-Receiver Communication” on page 4-119 and “Simulate Event-Driven AUTOSAR Queued Sender-Receiver Communication” on page 4-122.

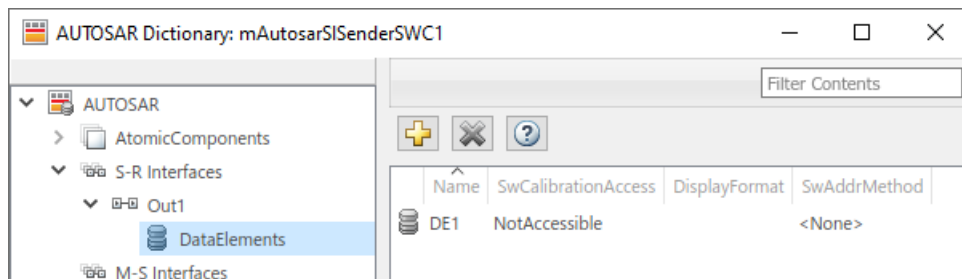
Configure AUTOSAR Sender and Receiver Components for Queued Communication

This example configures AUTOSAR queued sender and receiver components in Simulink. To see these models connected for simulation, see “Configure Simulation of AUTOSAR Queued Sender-Receiver Communication” on page 4-118.

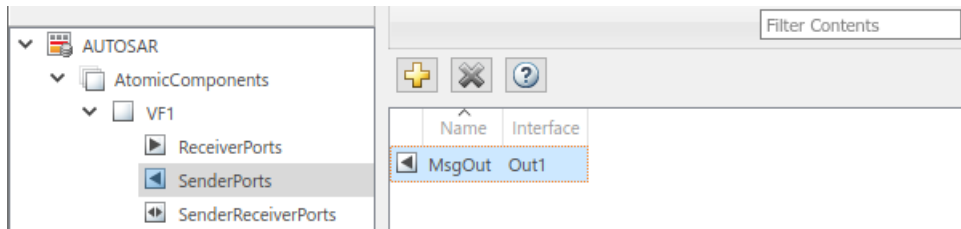
```
openExample('mAutosarSlSenderSWC1')
openExample('mAutosarSlReceiverSWC')
```

Open an AUTOSAR model that you want to configure as a queued sender or receiver component. To create an S-R data interface and a queued sender or receiver port:

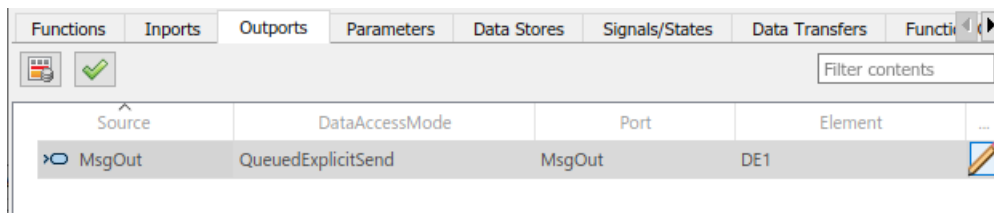
- 1 Open the AUTOSAR Dictionary.
- 2 Select **S-R Interfaces**. To create an S-R data interface, click the **Add** button . Specify its name and the number of associated S-R data elements. This example uses one data element in both the sender and receiver components.
- 3 Select and expand the new S-R interface. Select **DataElements** and modify the data element attributes. This figure shows data element DE1 for the sender component.




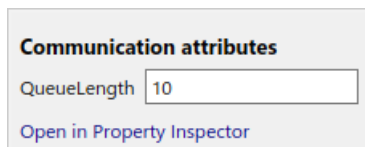
- 4 Expand the **AtomicComponents** node and select an AUTOSAR component. Expand the component.
- 5 Select the **SenderPorts** or **ReceiverPorts** view and use it to add the sender or receiver port you require. For each S-R port, select the S-R interface you created. For the sender component, this figure shows sender port MsgOut, which uses S-R interface Out1.



- 6 Open the Code Mappings editor. Select the **Inports** or **Outports** tab and use it to map a Simulink inport or outport to an AUTOSAR queued S-R port. For each inport or outport, select an AUTOSAR port, data element, and data access mode. Set the AUTOSAR data access mode to `QueuedExplicitSend` or `QueuedExplicitReceive`. In the sender component, this figure shows Simulink outport `MsgOut`, which is mapped to AUTOSAR sender port `MsgOut` and data element `DE1`, with data access mode `QueuedExplicitSend`.



When you map an inport to an AUTOSAR queued receiver port, you can use either the Code Mappings or AUTOSAR Dictionary view of the port to modify its AUTOSAR communication specification (ComSpec) attribute `QueueLength`. Select the inport in Code Mappings and either click the  icon or open the AUTOSAR Dictionary. For more information, see “Configure AUTOSAR Sender-Receiver Port ComSpecs” on page 4-108.



When you build an AUTOSAR queued sender or receiver component model:

- Generated C code contains calls to AUTOSAR `Rte_Send_<port>_<DataElement>` or `Rte_Receive_<port>_<DataElement>` APIs. The generated code handles the status of the message send and receive calls.
- Exported ARXML files contain descriptions for queued sender-receiver communication. The generated ComSpec for a queued port includes the port type and the queue length (based on Simulink message property `QueueCapacity`). In the `SwDataDefProps` generated for the queued port data element, `SwImplPolicy` is set to `Queued`.

To implement the messaging behavior of an AUTOSAR queued sender or receiver component, use Simulink or Stateflow messages. See “Implement AUTOSAR Queued Send and Receive Messaging” on page 4-115 or “Implement AUTOSAR Queued Send and Receive By Using Stateflow Messaging” on page 4-126.

Implement AUTOSAR Queued Send and Receive Messaging

To model AUTOSAR queued sender and receiver component behavior, this example uses:

- Simulink message blocks to implement messaging.
- Stateflow charts to implement decision logic.


This example explains the construction of the example models `mAutosarSlSenderSWC1.slx` and `mAutosarSlReceiverSWC.slx`. These models can be accessed by running the following code at the MATLAB command line.

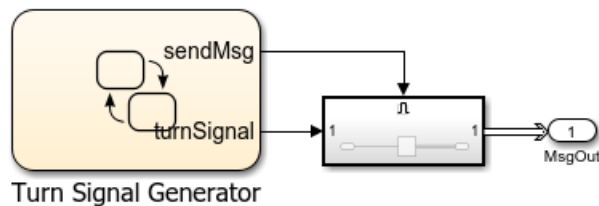
```
openExample('mAutosarSlSenderSWC1')
openExample('mAutosarSlReceiverSWC')
```

Other examples deploy the same sender and receiver models in 1-to-1 and N-to-1 messaging configurations. See “Configure Simulation of AUTOSAR Queued Sender-Receiver Communication” on page 4-118 and “Simulate N-to-1 AUTOSAR Queued Sender-Receiver Communication” on page 4-119.

This figure shows the top level of AUTOSAR queued sender component `mAutosarSlSenderSWC1`. The model contains:

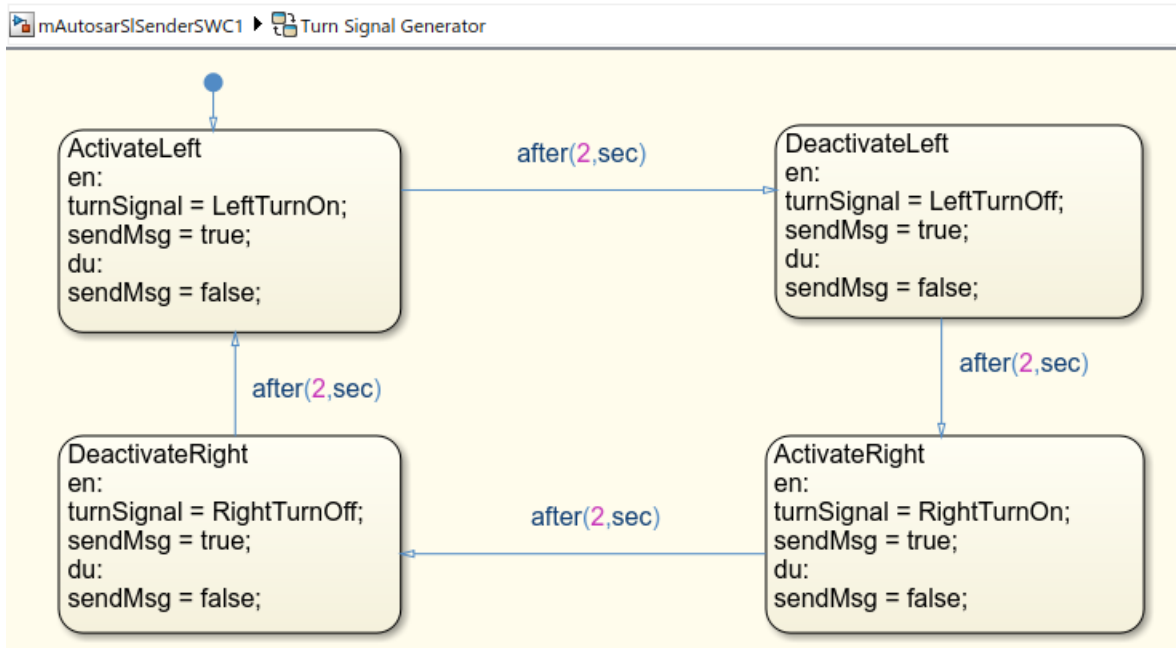
- Stateflow chart `Turn Signal Generator`.
- A Simulink message `Send` block, wrapped in an enabled subsystem.

 `mAutosarSlSenderSWC1` ▶



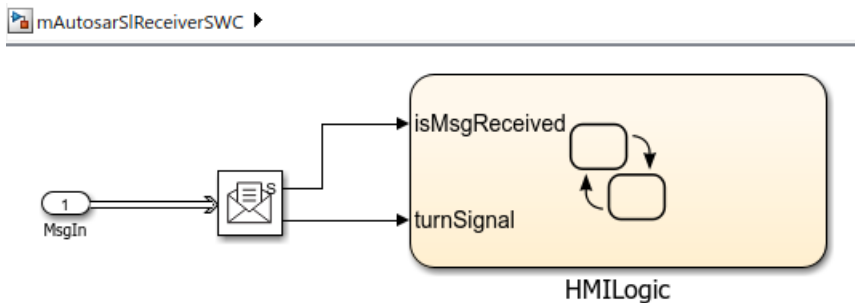
The chart has turn-signal and message-control outputs, which are connected to the enabled subsystem. When the message-control signal becomes a positive value, the subsystem is enabled. Inside the subsystem, the message `Send` block reads the turn-signal data value, and sends a message containing the value to root output `MsgOut`.

This figure shows the logic implemented in the `Turn Signal Generator` chart. The chart has four states - `ActivateLeft`, `DeactivateLeft`, `ActivateRight`, and `DeactivateRight`. Each state contains entry actions that assign a turn-signal data value and set the message-control value to `true`. (See “Communicate with Stateflow Charts by Sending Messages” (Stateflow).) Periodic timing drives the message output.



This figure shows the top level of AUTOSAR queued receiver component mAutosarSlReceiverSWC. The model contains:

- A Simulink message Receive block.
- Stateflow chart HMILogic.



The root inport MsgIn provides a message to the Receive block, which extracts the turn-signal data value from the message. The block then outputs message-received and turn-signal data values to the Stateflow chart.

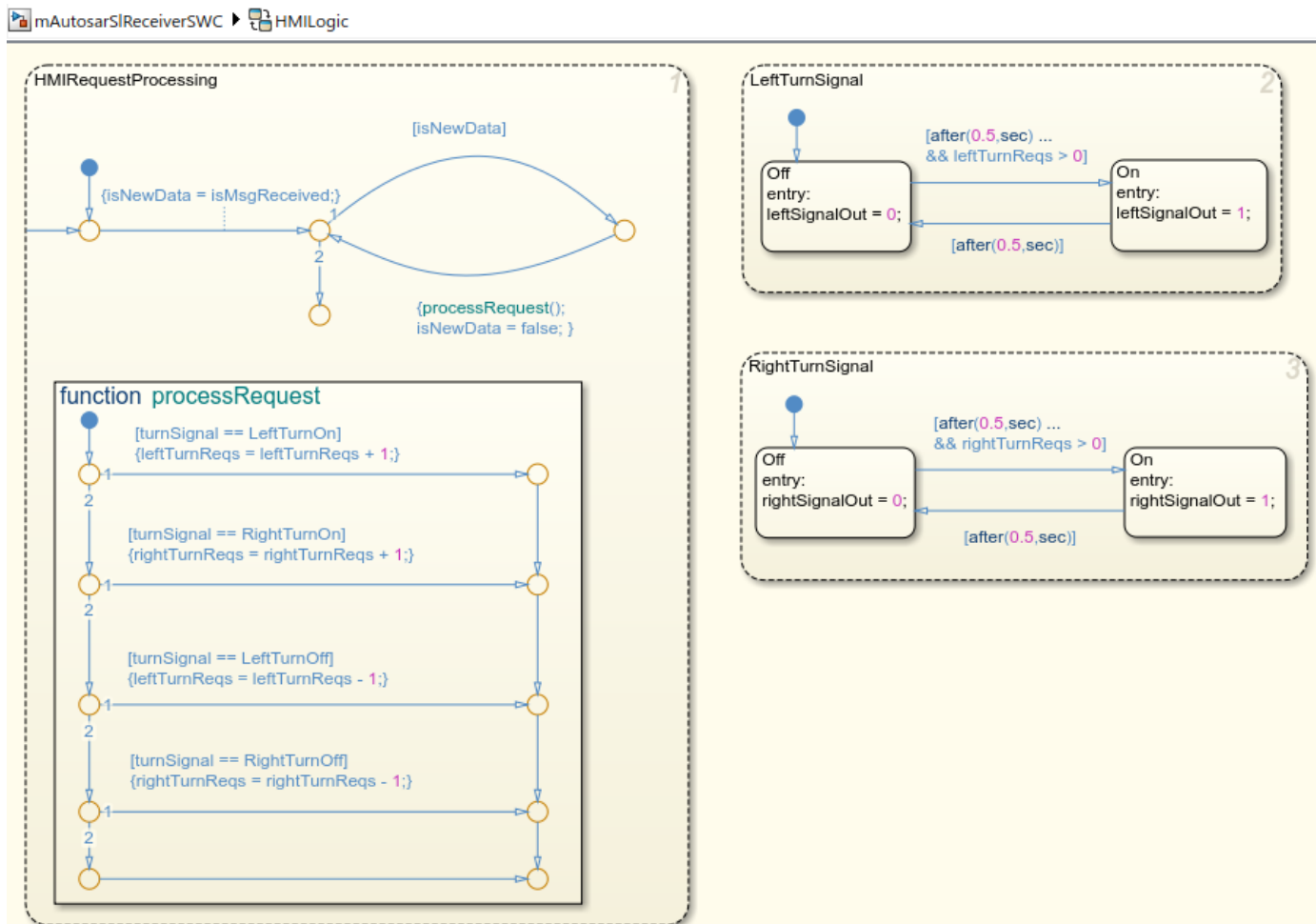
The Receive block parameters are set to their Simulink defaults. For example, **Show receive status** is selected, **Use internal queue** is cleared, and **Value source when queue is empty** is set to Hold last value.

This figure shows the logic implemented in the HMILogic chart. HMILogic contains states HMIRequestProcessing, LeftTurnSignal, and RightTurnSignal.

- HMIRequestProcessing receives the message-received and turn-signal data inputs, sets an isNewData flag, calls a function to process the turn-signal data, and then clears the isNewData flag. The processRequest function tests the received turn-signal data for values potentially set by the message sender -- LeftTurnOn, RightTurnOn, LeftTurnOff, or RightTurnOff. Based

on the value received, the function increments or decrements a request counter variable, either `leftTurnReqs` or `rightTurnReqs`. Periodic timing drives the message input.

- `LeftTurnSignal` and `RightTurnSignal` each contain states `Off` and `On`. They transition from `Off` to `On` based on the value of request counter `leftTurnReqs` or `rightTurnReqs` and a time interval. When the request counter is greater than zero, the charts set a variable, either `leftSignalOut` or `rightSignalOut`, to 1. After a time interval, they transition back to the `Off` state and set `leftSignalOut` or `rightSignalOut` to 0.



For sample implementations of multiple queued sender components in an N-to-1 messaging configuration, see the example models used in “Simulate N-to-1 AUTOSAR Queued Sender-Receiver Communication” on page 4-119.

For sample implementations of event-driven queued messaging, see the example models used in “Simulate Event-Driven AUTOSAR Queued Sender-Receiver Communication” on page 4-122.


Configure Simulation of AUTOSAR Queued Sender-Receiver Communication

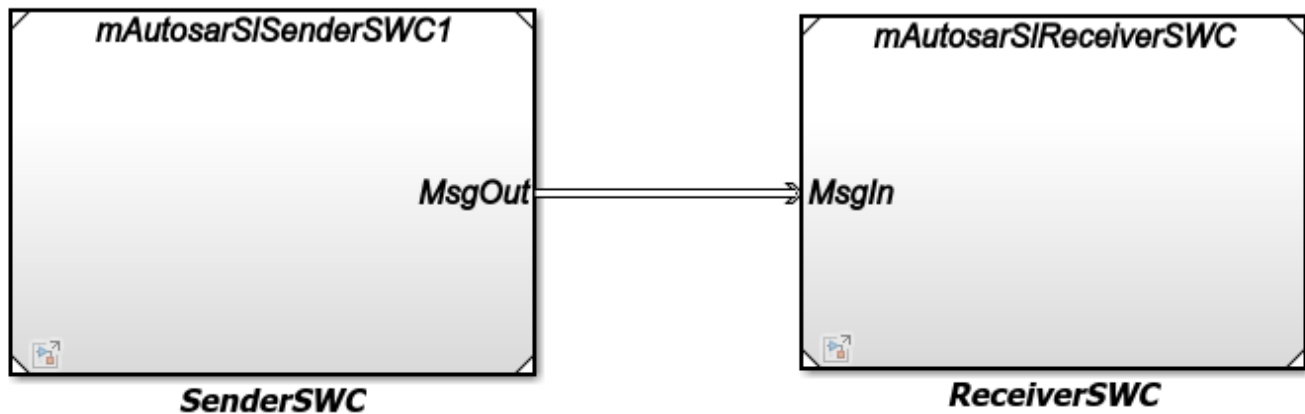
To simulate AUTOSAR queued sender-receiver communication in Simulink®, create a containing composition, system, or harness model. Include the queued sender and receiver components as referenced models.

- If you have one sender component and one receiver component, you potentially can connect the models directly. This example directly connects sender and receiver component models.
- If you are simulating N-to-1 or event-driven messaging, you provide additional logic between sender and receiver component models. For example, see “Simulate N-to-1 AUTOSAR Queued Sender-Receiver Communication” on page 4-119 and “Simulate Event-Driven AUTOSAR Queued Sender-Receiver Communication” on page 4-122.

This example shows a composition-level model that contains queued sender and receiver component models and implements 1-to-1 communication. Periodic timing drives the messaging. The example uses the following three models which can be opened by running the code below:

```
open_system('mAutosarSIQueuedMsgs_1_1.slx') %(top model)
open_system('mAutosarSIReceiverSWC.slx')
open_system('mAutosarSISenderSWC.slx')
```

 mAutosarSIQueuedMsgs_1_1 ▶



Models *mAutosarSISenderSWC1* and *mAutosarSIReceiverSWC* are the same sender and receiver components configured in “Configure AUTOSAR Sender and Receiver Components for Queued Communication” on page 4-114 and implemented in “Implement AUTOSAR Queued Send and Receive Messaging” on page 4-115. Composition-level model *mAutosarSIQueuedMsgs_1_1* includes them as referenced models and connects sender component port *MsgOut* to receiver component port *MsgIn*.

The top model *mAutosarSIQueuedMsgs_1_1* is for simulation only. You can generate AUTOSAR C code and ARXML files for the sender and receiver component models, but not for the containing composition-level model.

Similarly, you can run software-in-the-loop (SIL) simulation for the sender and receiver component models, but not for the composition-level model.

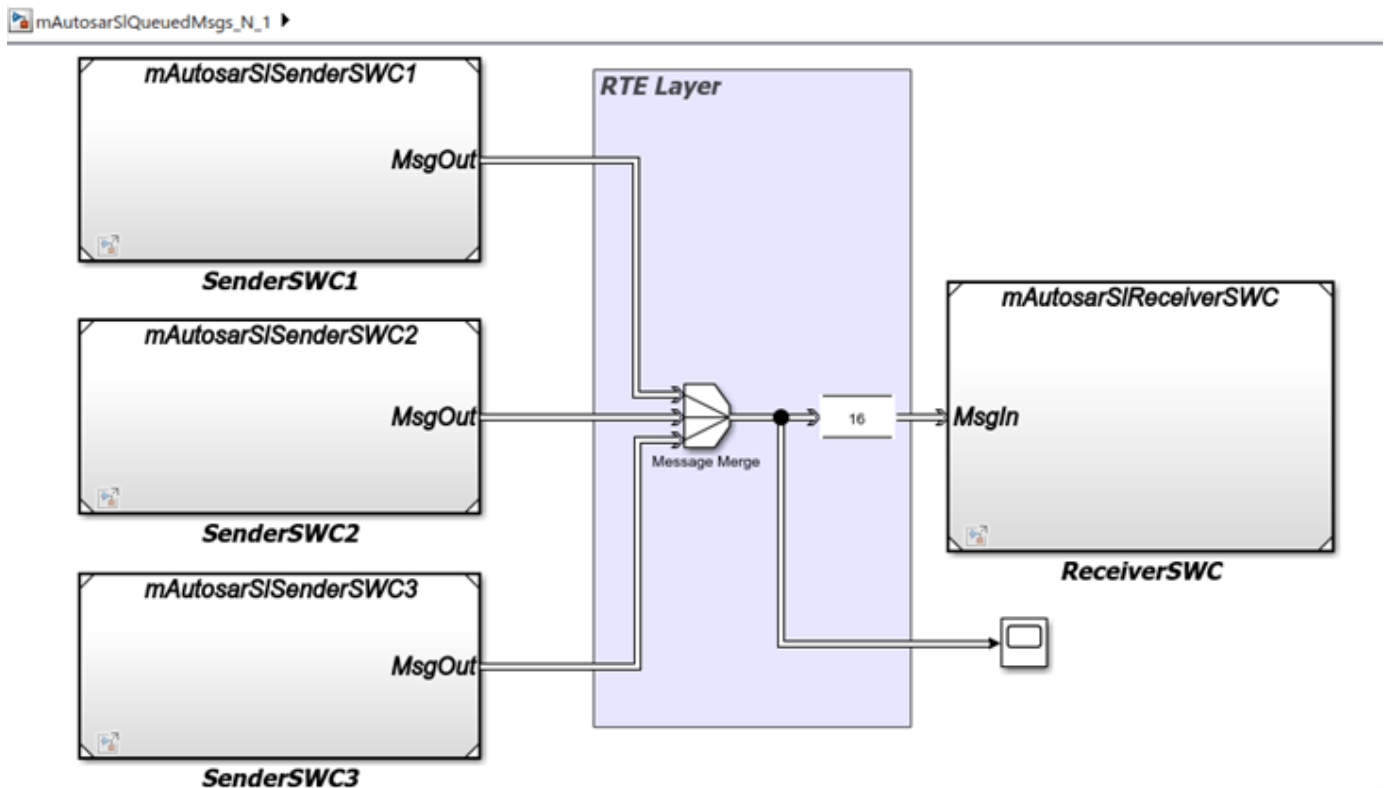
Simulate N-to-1 AUTOSAR Queued Sender-Receiver Communication

This example shows a composition-level model that contains three sender and one receiver component models, and implements N-to-1 communication. Periodic timing drives the messaging. The

example extends the 1-to-1 example by adding two additional sender models and providing flow logic between the senders and receiver.

This example uses four models which can be accessed by running the following lines of code at the MATLAB® command line.

```
open_system('mAutosarSlQueuedMsgs_N_1.slx')
open_system('mAutosarSlSenderSWC1.slx')
open_system('mAutosarSlSenderSWC2.slx')
open_system('mAutosarSlSenderSWC3.slx')
open_system('mAutosarSlReceiverSWC.slx')
```

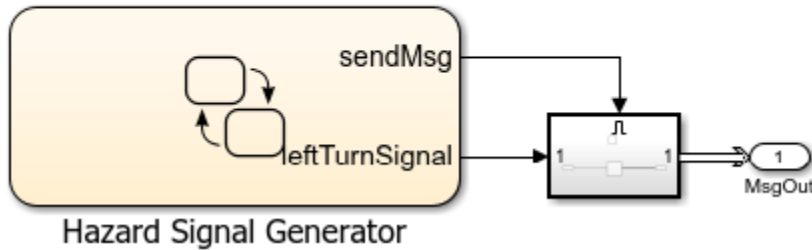


Composition-level model `mAutosarSlQueuedMsgs_N_1` includes three sender components and a receiver component as referenced models. It connects the sender component `MsgOut` ports to intermediate `MsgJoin` processing logic, which in turn connects to a receiver component `MsgIn` port.

Models `mAutosarSlSenderSWC1` and `mAutosarSlReceiverSWC` are the same sender and receiver components configured in “Configure AUTOSAR Sender and Receiver Components for Queued Communication” on page 4-114 and implemented in “Implement AUTOSAR Queued Send and Receive Messaging” on page 4-115. The second and third sender components, `mAutosarSlSenderSWC2` and `mAutosarSlSenderSWC3`, are similar to `mAutosarSlSenderSWC1`, but implement a second type of message input for the receiver to process.

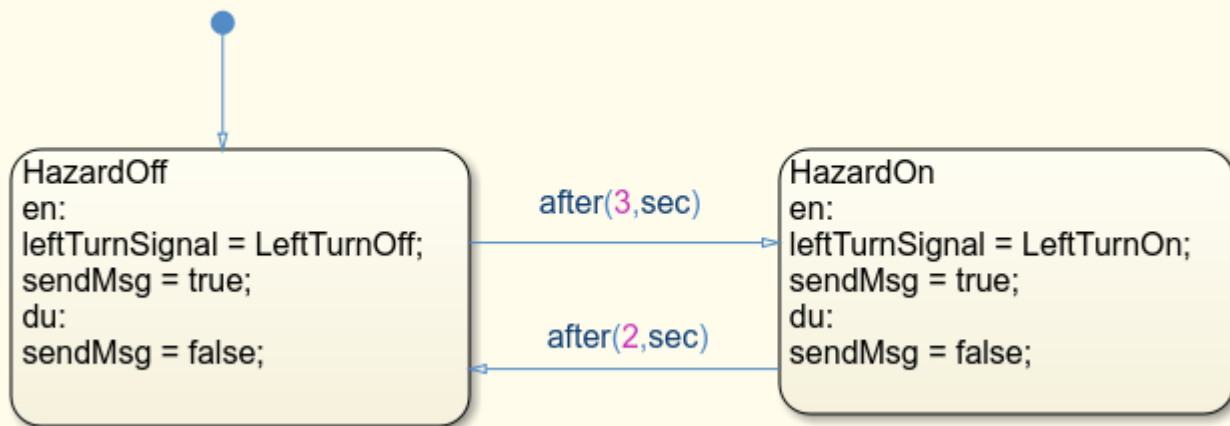
This figure shows the top level of AUTOSAR queued sender component `mAutosarSlSenderSWC2`. It contains Stateflow® chart `Hazard Signal Generator`, which provides logic for left-turn signals. The chart message-line output is connected to Simulink® root outputport `MsgOut`. A corresponding `Hazard Signal Generator` chart to handle right-turn signals appears in sender component `mAutosarSlSenderSWC3`.

mAutosarSIQueuedMsgs_N_1 ▶ SenderSWC2 (mAutosarSISenderSWC2) ▶



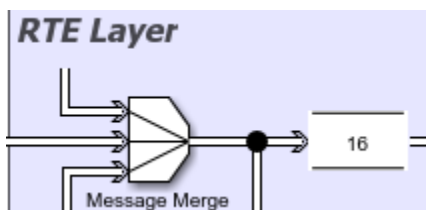
This figure shows the logic implemented in the Hazard Signal Generator chart. The chart has two states - HazardOff and HazardOn. Each state contains entry actions that assign values to message data and send messages. (See “Communicate with Stateflow Charts by Sending Messages” (Stateflow).) Periodic timing drives the message output.

mAutosarSIQueuedMsgs_N_1 ▶ SenderSWC2 (mAutosarSISenderSWC2) ▶ Hazard Signal Generator



Between the sender and receiver components, a Message Merge block and a Queue block provide message merging and queuing.

- The Message Merge block merges 3 message lines and outputs messages to the Queue block.
- The Queue block stores messages from the 3 lines in a queue, based on the order of arrival.
- The queue capacity is set to 16 messages.
- When the queue is full and a message arrives, the block is set to overwrite the oldest message with the incoming message.
- The message sorting policy is set to the policy AUTOSAR supports, first-in first-out (FIFO).



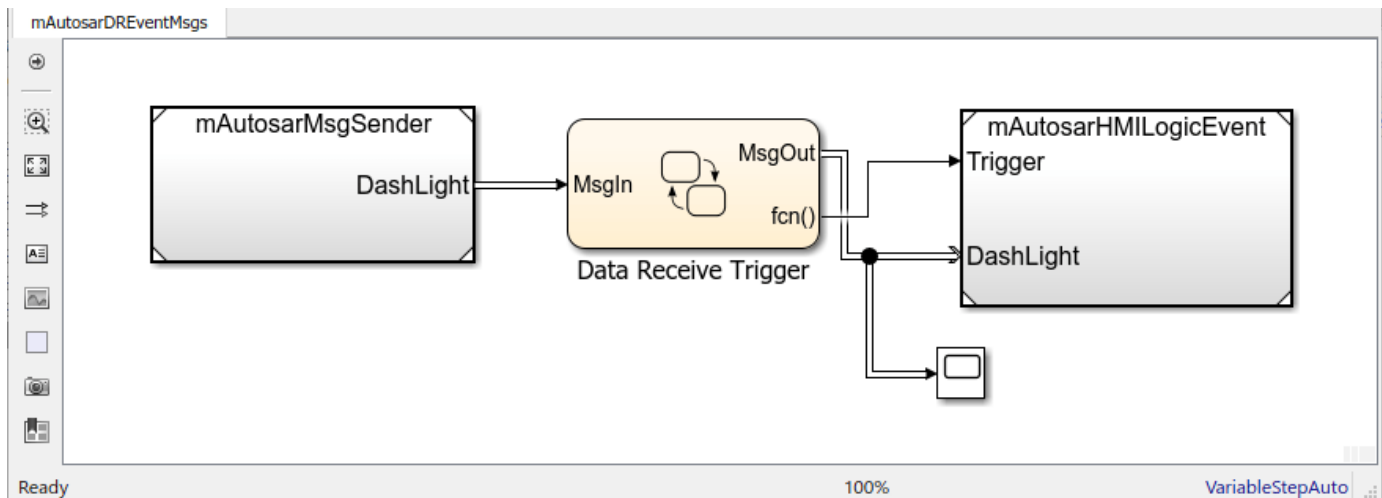
Each element at the head of the queue departs when the downstream ReceiverSWC block is ready to accept it. The top model `mAutosarSlQueuedMsgs_N_1` is for simulation only. You can generate AUTOSAR C code and ARXML files for the referenced sender and receiver component models, but not for the containing composition-level model. Similarly, you can run software-in-the-loop (SIL) simulation for the sender and receiver component models, but not for the composition-level model.

Simulate Event-Driven AUTOSAR Queued Sender-Receiver Communication

This example shows a composition-level model in which a Simulink® function-call input event activates receiver component processing of a queued message. The example is implemented by using Stateflow® messages. For more Stateflow® messaging examples, see “Implement AUTOSAR Queued Send and Receive By Using Stateflow Messaging” on page 4-126.

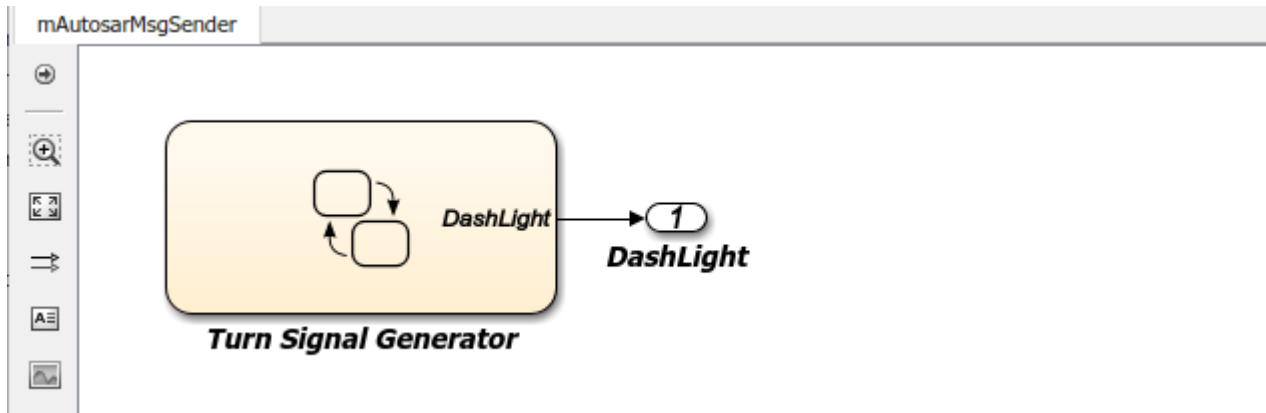
This example using three models which can be accessed by running the following lines of code at the MATLAB® command line:

```
open_system('mAutosarDREventMsgs')
open_system('mAutosarMsgSender')
open_system('mAutosarHMILogicEvent')
```

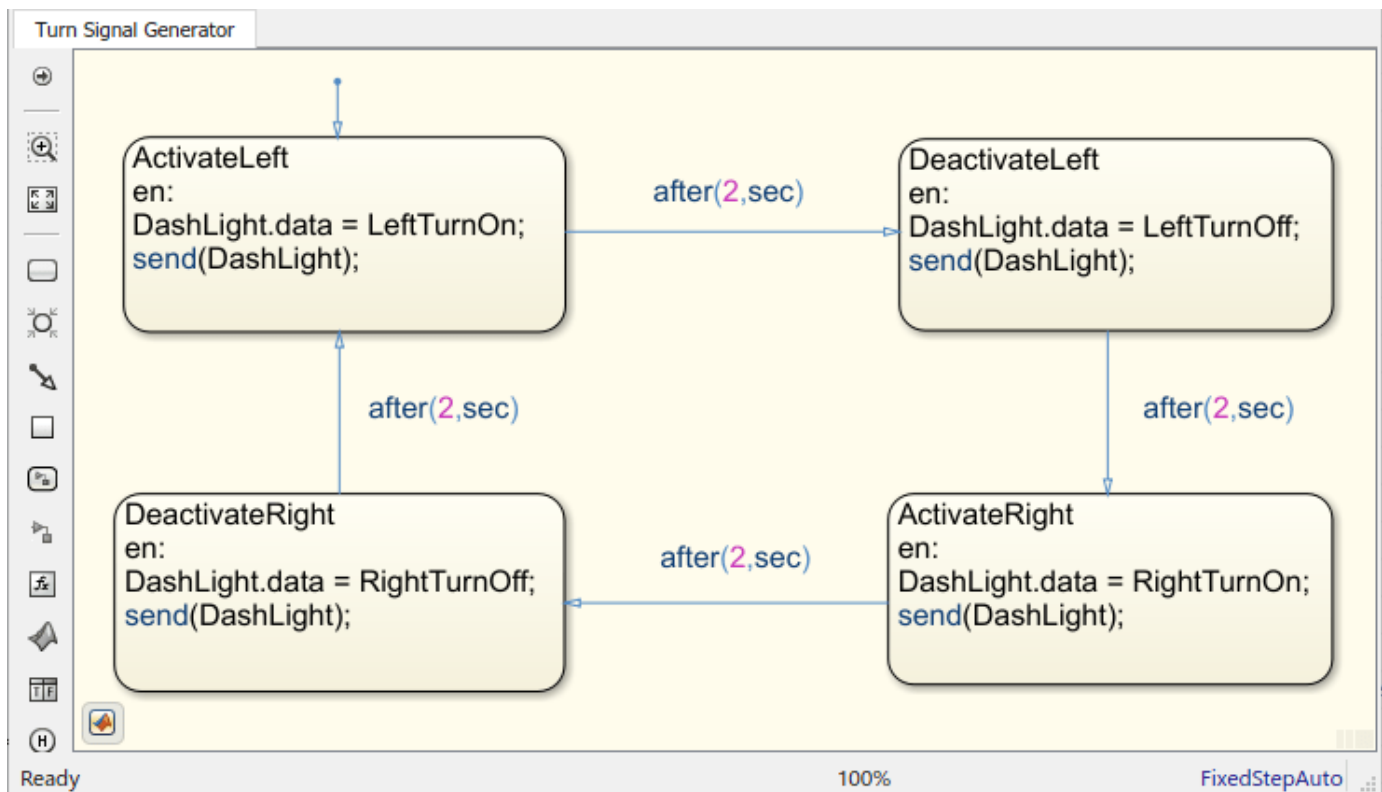


Composition-level model `mAutosarDREventMsgs` includes a sender component and a receiver component as referenced models. It connects the sender message port `DashLight` to intermediate `Data Receive Trigger` logic, which in turn connects to receiver message port `MsgIn` and function trigger port `Trigger`.

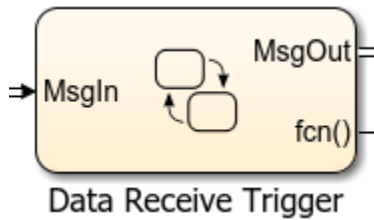
This figure shows the top level of AUTOSAR queued sender component `mAutosarMsgSender`, which contains Stateflow® chart `Turn Signal Generator`. The chart message-line output is connected to Simulink® root outputport `DashLight`. (This sender component is similar to component `mAutosarSenderSWC1` in the Stateflow® 1-to-1 and N-to-1 simulation examples in “Implement AUTOSAR Queued Send and Receive By Using Stateflow Messaging” on page 4-126.)



This figure shows the logic implemented in the Turn Signal Generator chart. The chart has four states - ActivateLeft, DeactivateLeft, ActivateRight, and DeactivateRight. Each state contains entry actions that assign a value to message data and send a message. (See “Communicate with Stateflow Charts by Sending Messages” (Stateflow).) Periodic timing drives the message output.



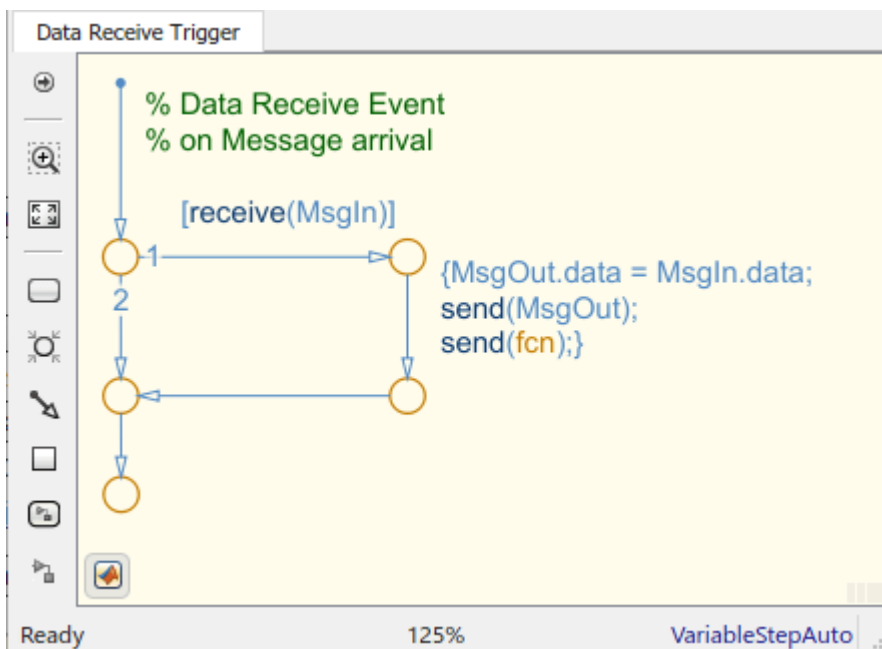
This figure shows the Data Receiver Trigger chart located between the sender and receiver components.



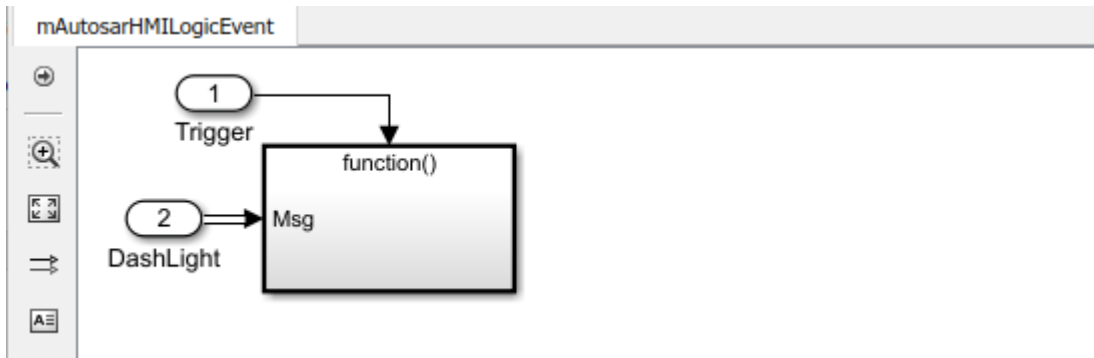
To receive a message, queued receiver logic uses `receive(M)`:

- If a valid message `M` exists, `receive(M)` returns true.
- If a valid message does not exist, the chart removes a message from its associated queue, and `receive(M)` returns true. If `receive(M)` removes a message from the queue, the length of the queue drops by one.
- If message `M` is invalid, and another message could not be removed from the queue, `receive(M)` returns false.

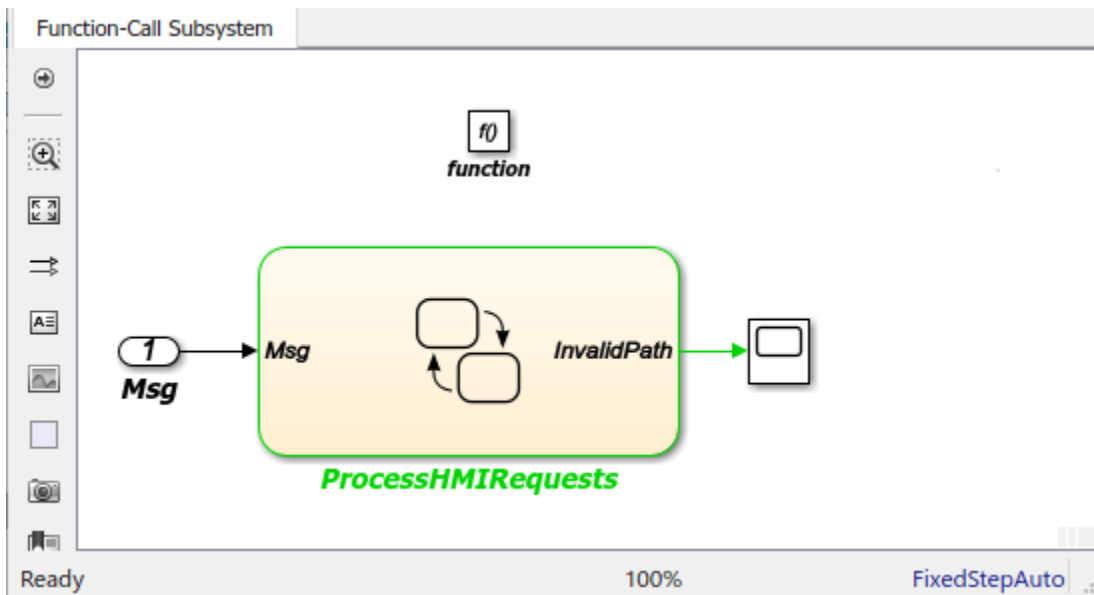
You can place `receive` on a transition (for example, `[receive(M)]`). Or, within a state, use an `if` condition (for example, `if(receive(M))`). For more information, see “Communicate with Stateflow Charts by Sending Messages” (Stateflow).



This figure shows the top level of AUTOSAR queued receiver component `mAutosarHMILogicEvent`, which contains a Simulink® function-call subsystem. The subsystem inports are a function-call trigger and message receiver port `DashLight`, which is configured for AUTOSAR data access mode `QueuedExplicitReceive`.



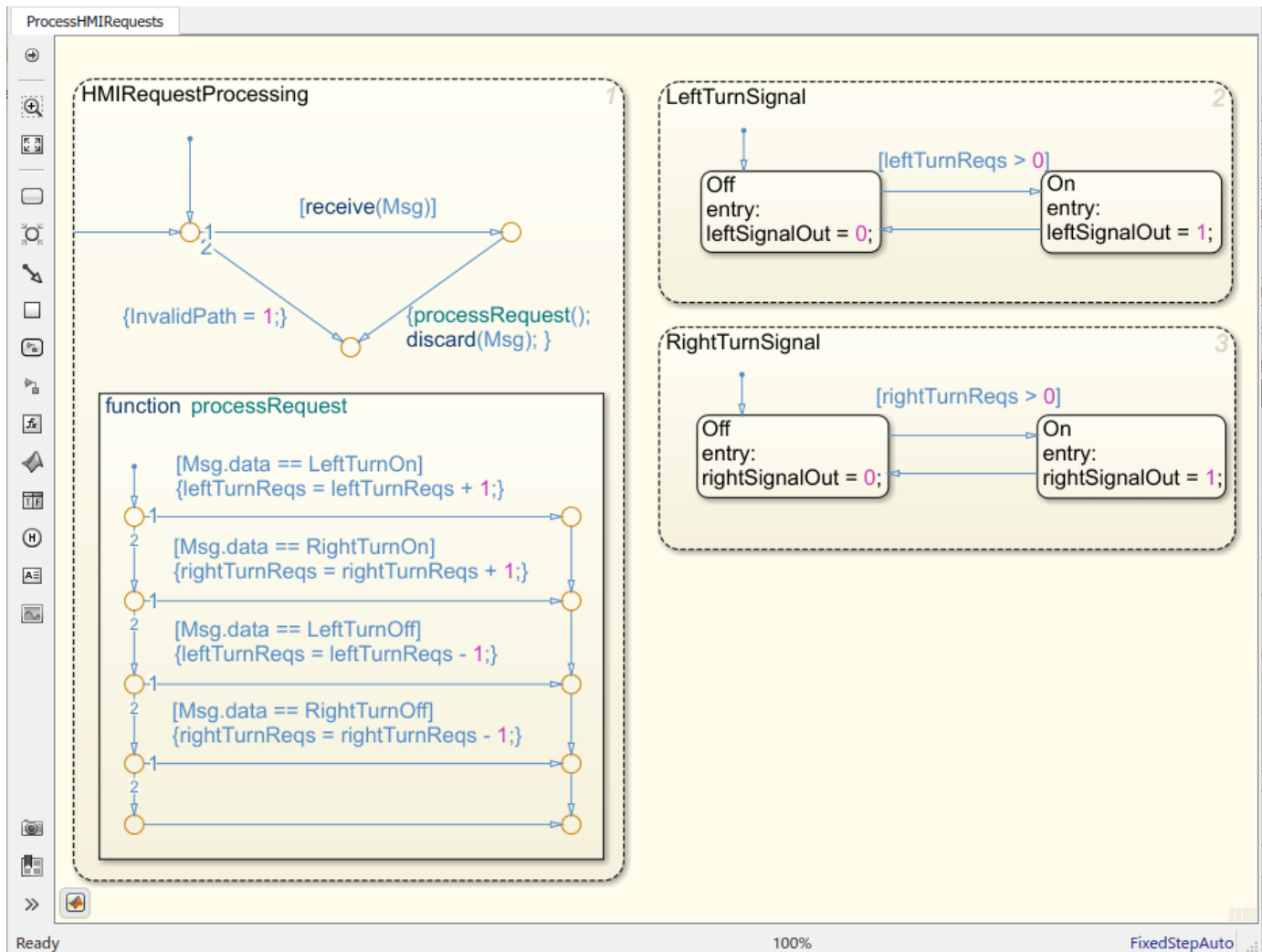
The function-call subsystem contains Stateflow® chart `ProcessHMIRequests` and a Trigger Port block. The chart message-line input is connected to Simulink® root inport `Msg`. A scope is configured to display the value of an `InvalidPath` variable. The Trigger Port block is configured for a function-call trigger and triggered sample time. Function-call input events sent from the Data Receiver Trigger chart in the top model activate the chart.



This figure shows the logic implemented in the `ProcessHMIRequests` chart. `ProcessHMIRequests` contains states `HMIRequestProcessing`, `LeftTurnSignal`, and `RightTurnSignal`. (This receiver chart is similar to chart `HMILogic` in the 1-to-1 and N-to-1 simulation examples.)

- `HMIRequestProcessing` receives a message from the message queue, calls a function to process the message, and then discards the message. The `processRequest` function tests the received message data for values potentially set by the message sender -- `LeftTurnOn`, `RightTurnOn`, `LeftTurnOff`, or `RightTurnOff`. Based on the value received, the function increments or decrements a request counter variable, either `leftTurnReqs` or `rightTurnReqs`. Function-call input events drive the message input. If the chart is incorrectly activated, the `InvalidPath` variable is set to 1.
- `LeftTurnSignal` and `RightTurnSignal` each contain states `Off` and `On`. They transition from `Off` to `On` based on the value of request counter `leftTurnReqs` or `rightTurnReqs`. When the request counter is greater than zero, the charts set a variable, either `leftSignalOut` or

rightSignalOut, to 1. Then they transition back to the Off state and set leftSignalOut or rightSignalOut to 0.



The top model `mAutosarDREventMsgs` is for simulation only. You can generate AUTOSAR C code and ARXML files for the referenced sender and receiver component models, but not for the containing composition-level model. Similarly, you can run software-in-the-loop (SIL) simulation for the sender and receiver component models, but not for the composition-level model.

Implement AUTOSAR Queued Send and Receive By Using Stateflow Messaging

- “Implement AUTOSAR Queued Send and Receive Messaging By Using Stateflow Messages” on page 4-127
- “Configure Simulation of AUTOSAR Queued Sender-Receiver Communication” on page 4-131
- “Simulate N-to-1 AUTOSAR Queued Sender-Receiver Communication” on page 4-132
- “Determine When a Queue Overflows” on page 4-135

Implement AUTOSAR Queued Send and Receive Messaging By Using Stateflow Messages

To implement AUTOSAR queued sender or receiver component behavior, this example uses Stateflow messages. To create a Stateflow chart, follow the general procedure described in “Model Finite State Machines by Using Stateflow Charts” (Stateflow).

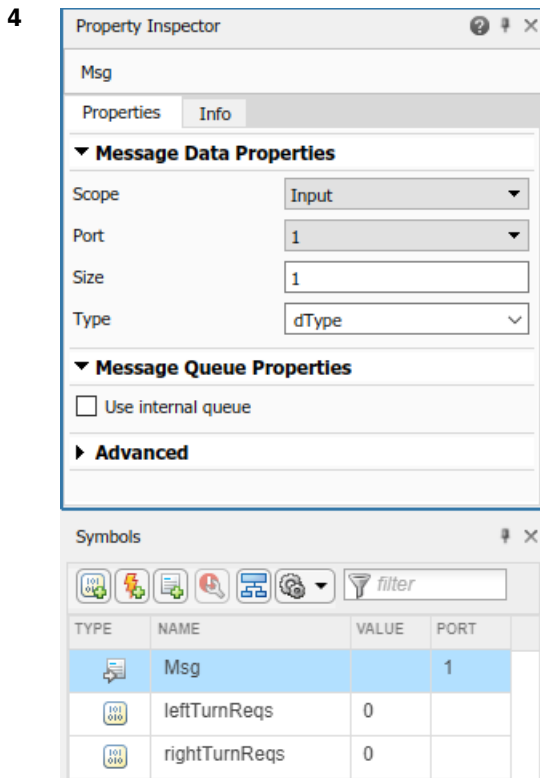
- 1 Add a chart to the AUTOSAR queued sender or receiver component model. Name the chart.
- 2 Open the chart and add message-related states.
- 3 For each state, add entry actions. Supported message keywords include:
 - `send(M)` -- Send message M.
 - `receive(M)` -- Receive message M.
 - `isvalid(M)` -- Check if message M is valid (popped and not discarded).
 - `discard(M)` -- Explicitly discard message M. Messages are implicitly discarded on state exit after a message receive operation completes.
- 4 Add state transition lines and specify transition conditions or events on those lines.
 - Use conditions when you want to transition based on a conditional statement or a change of input value from a Simulink block. For more information, see “Transition Between Operating Modes” (Stateflow).
 - Use events when you want to transition based on a Simulink triggered or function-call input event. For more information, see “Synchronize Model Components by Broadcasting Events” (Stateflow).
- 5 Define data that stores state variables.
- 6 Connect the chart message-line inputs and outputs to Simulink root inports and outports.

For more information, see “Messages” (Stateflow).

In the context of a Stateflow chart, you can modify message properties, such as data type and queue capacity. (For a list of properties, see “Set Properties for a Message” (Stateflow).) You can access message properties in the Property Inspector, a Message properties dialog box, or Model Explorer. To view or modify message properties with the Property Inspector:

- 1 Open a chart that uses messages.
- 2 In the **Modeling** tab, open **Symbols Pane** and **Property Inspector**.
- 3 In the Symbols view, select a message. The Property Inspector displays panes for **Message Data Properties** and **Advanced** properties.

If the chart is in a receiver component, the Property Inspector also displays **Message Queue Properties**. To configure the receiver component to use external AUTOSAR RTE message queues, make sure that property **Use internal queue** is cleared.



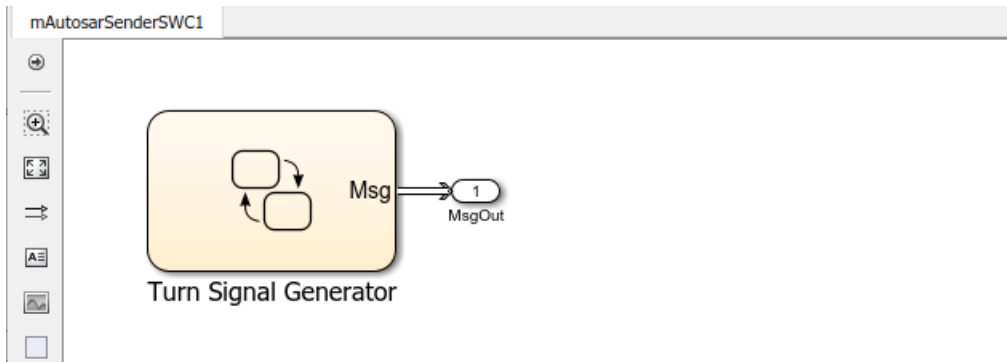
By default, message data type and queue capacity values are inherited from the Stateflow message to which a Simulink root port is attached. Message data can use these Simulink parameter data types: int types, uint types, floating-point types, boolean, Enum, or Bus (struct).

If you use imported bus or enumeration data types in Stateflow charts, typedefs are required for simulation. To generate typedefs automatically, select the Simulink configuration option **Generate typedefs for imported bus and enumeration types**. Otherwise, use Simulink configuration parameter **Simulation Target > Custom Code > Header file** to include header files with the definitions.

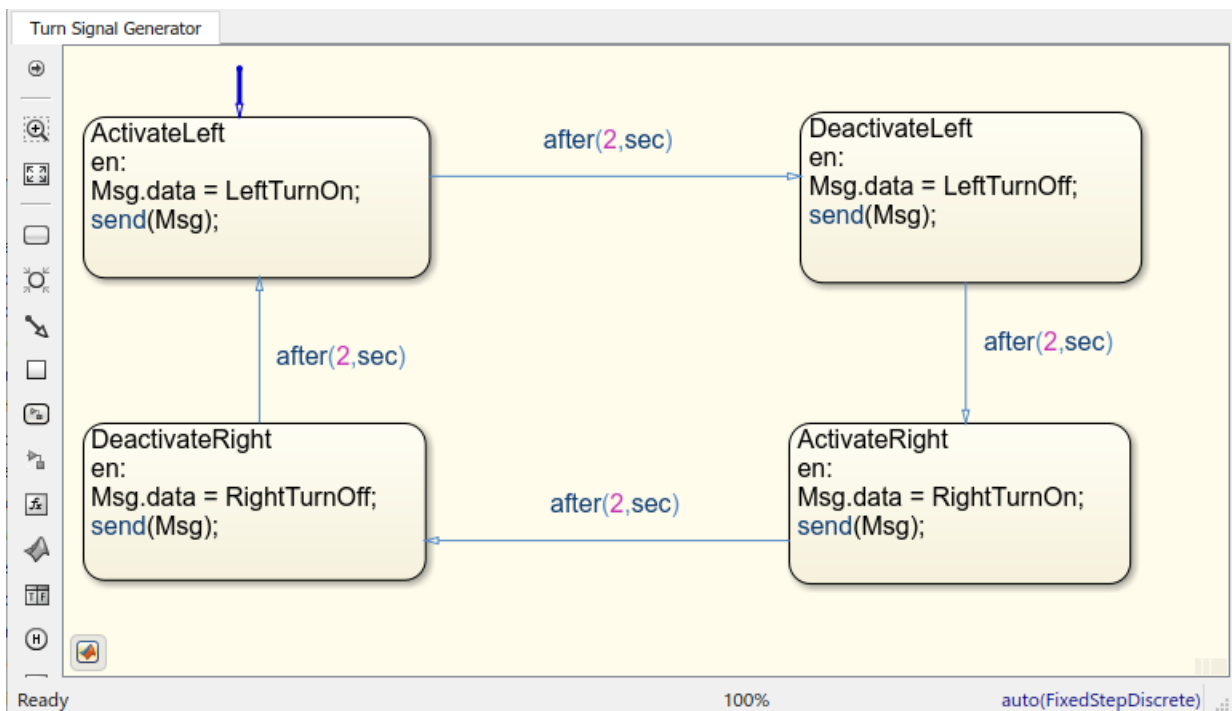
For sample implementations of queued sender and receiver components in a 1-to-1 configuration, see the example component models used in both “Configure AUTOSAR Sender and Receiver Components for Queued Communication” on page 4-114 and “Configure Simulation of AUTOSAR Queued Sender-Receiver Communication” on page 4-118. Models `mAutosarSenderSWC1.slx` and `mAutosarReceiverSWC.slx` can be accessed by running the following lines of code at the MATLAB command line.

```
openExample('mAutosarSenderSWC1.slx')
openExample('mAutosarReceiverSWC.slx')
```

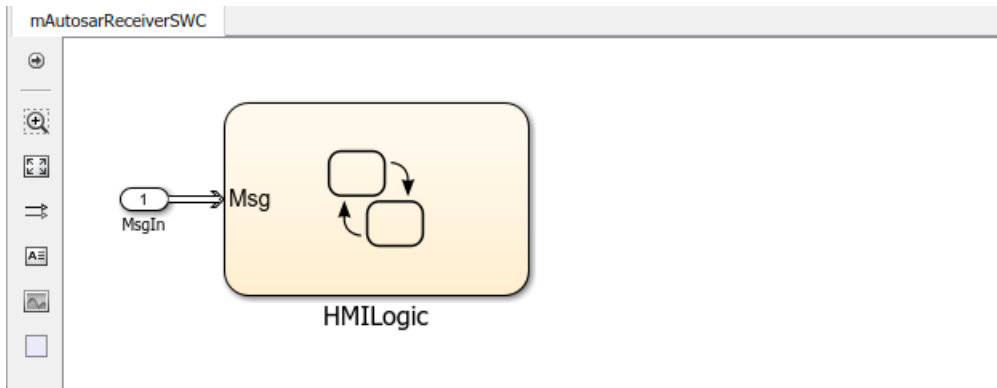
This figure shows the top level of AUTOSAR queued sender component `mAutosarSenderSWC1`, which contains Stateflow chart `Turn Signal Generator`. The chart message-line output is connected to Simulink root outputport `MsgOut`.



This figure shows the logic implemented in the Turn Signal Generator chart. The chart has four states - ActivateLeft, DeactivateLeft, ActivateRight, and DeactivateRight. Each state contains entry actions that assign a value to message data and send a message. (See "Communicate with Stateflow Charts by Sending Messages" (Stateflow).) Periodic timing drives the message output.



This figure shows the top level of AUTOSAR queued receiver component mAutosarReceiverSWC, which contains Stateflow chart HMILogic. The chart message-line input is connected to Simulink root inport MsgIn.



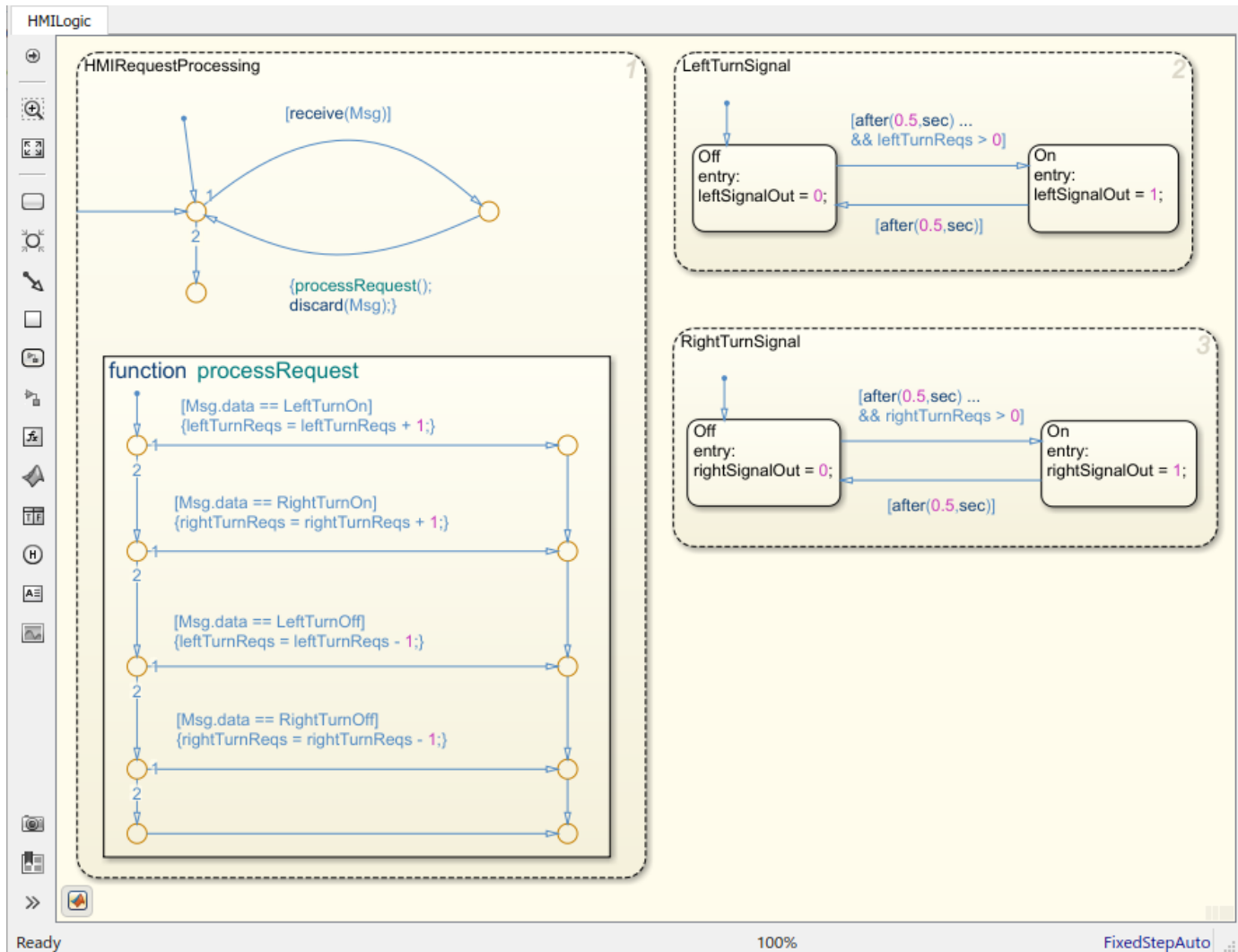
To receive a message, queued receiver logic uses `receive(M)`:

- If a valid message `M` exists, `receive(M)` returns true.
- If a valid message does not exist, the chart removes a message from its associated queue, and `receive(M)` returns true. If `receive(M)` removes a message from the queue, the length of the queue drops by one.
- If message `M` is invalid, and another message could not be removed from the queue, `receive(M)` returns false.

You can place `receive` on a transition (for example, `[receive(M)]`). Or, within a state, use an `if` condition (for example, `if(receive(M))`). For more information, see “Communicate with Stateflow Charts by Sending Messages” (Stateflow).

This figure shows the logic implemented in the `HMILogic` chart. `HMILogic` contains states `HMIRequestProcessing`, `LeftTurnSignal`, and `RightTurnSignal`.

- `HMIRequestProcessing` receives a message from the message queue, calls a function to process the message, and then discards the message. The `processRequest` function tests the received message data for values potentially set by the message sender -- `LeftTurnOn`, `RightTurnOn`, `LeftTurnOff`, or `RightTurnOff`. Based on the value received, the function increments or decrements a request counter variable, either `leftTurnReqs` or `rightTurnReqs`. Periodic timing drives the message input.
- `LeftTurnSignal` and `RightTurnSignal` each contain states `Off` and `On`. They transition from `Off` to `On` based on the value of request counter `leftTurnReqs` or `rightTurnReqs` and a time interval. When the request counter is greater than zero, the charts set a variable, either `leftSignalOut` or `rightSignalOut`, to 1. After a time interval, they transition back to the `Off` state and set `leftSignalOut` or `rightSignalOut` to 0.



For sample implementations of queued sender and receiver components in an N-to-1 configuration, see the example models used in “Simulate N-to-1 AUTOSAR Queued Sender-Receiver Communication” on page 4-119.

For sample implementations of event-driven queued messaging, see the example models used in “Simulate Event-Driven AUTOSAR Queued Sender-Receiver Communication” on page 4-122.

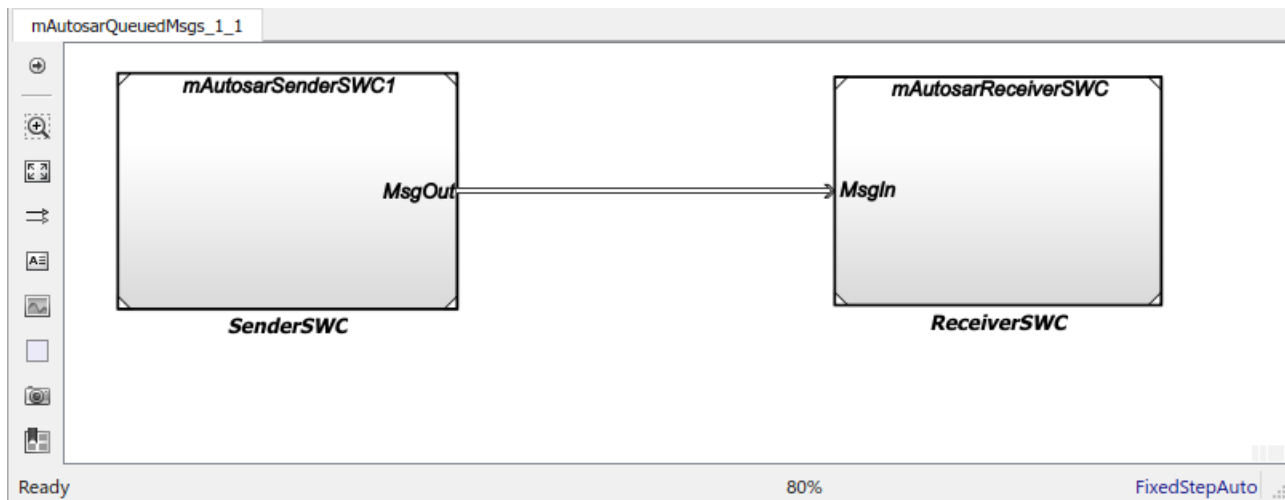
Configure Simulation of AUTOSAR Queued Sender-Receiver Communication

To simulate AUTOSAR queued sender-receiver communication in Simulink, create a containing composition, system, or harness model. Include the queued sender and receiver components as referenced models.

- If you have one sender component and one receiver component, you potentially can connect the models directly. This example directly connects sender and receiver component models.
- If you are simulating N-to-1 or event-driven messaging, you provide additional logic between sender and receiver component models. For example, see “Simulate N-to-1 AUTOSAR Queued Sender-Receiver Communication” on page 4-119 and “Simulate Event-Driven AUTOSAR Queued Sender-Receiver Communication” on page 4-122.

This example shows a composition-level model that contains queued sender and receiver component models and implements 1-to-1 communication. Periodic timing drives the messaging. The example uses three models that can be accessed by running the following lines of code at the MATLAB command line.

```
openExample('mAutosarQueuedMsgs_1_1.slx')
openExample('mAutosarSenderSWC1.slx')
openExample('mAutosarReceiverSWC.slx')
```



Models `mAutosarSenderSWC1` and `mAutosarReceiverSWC` are the same sender and receiver components configured in “Configure AUTOSAR Sender and Receiver Components for Queued Communication” on page 4-114 and implemented in “Implement AUTOSAR Queued Send and Receive Messaging” on page 4-115. Composition-level model `mAutosarQueuedMsgs_1_1` includes them as referenced models and connects sender component port `MsgOut` to receiver component port `MsgIn`.

The top model `mAutosarQueuedMsgs_1_1` is for simulation only. You can generate AUTOSAR C code and ARXML files for the sender and receiver component models, but not for the containing composition-level model.

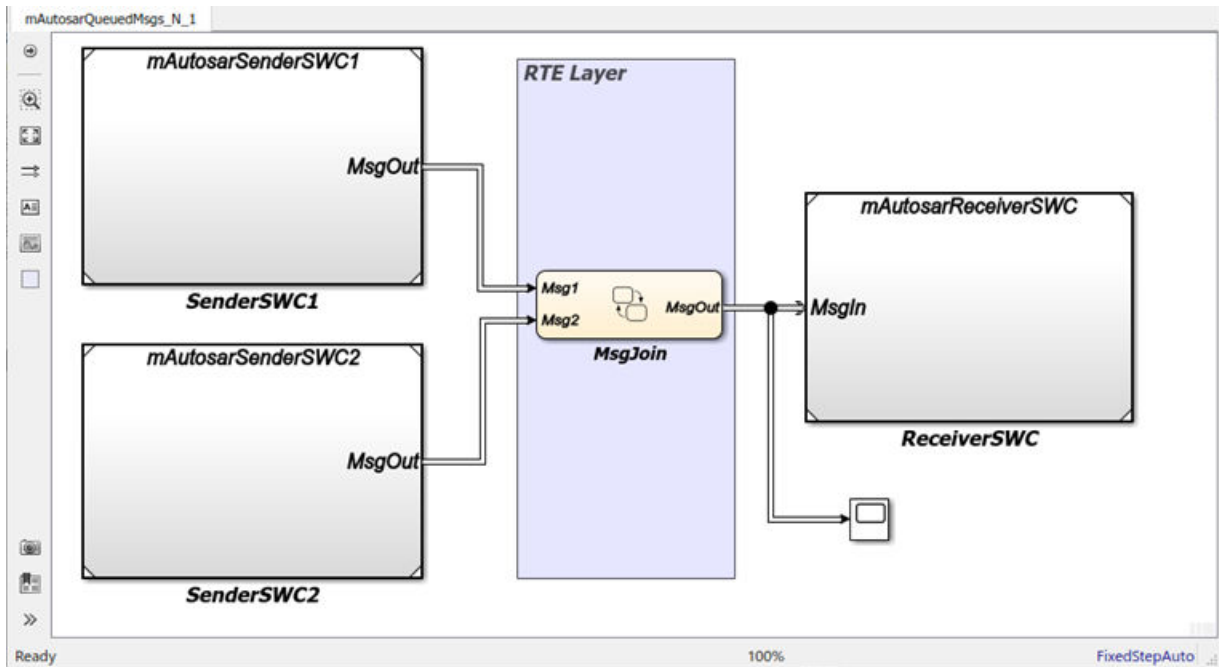
Similarly, you can run software-in-the-loop (SIL) simulation for the sender and receiver component models, but not for the composition-level model.

Simulate N-to-1 AUTOSAR Queued Sender-Receiver Communication

This example shows a composition-level model that contains two sender and one receiver component models, and implements N-to-1 communication. Periodic timing drives the messaging. The example extends the 1-to-1 example by adding a second sender model and providing flow logic between the senders and receiver.

This example uses four models that can be accessed by running the following lines of code at the MATLAB command line.

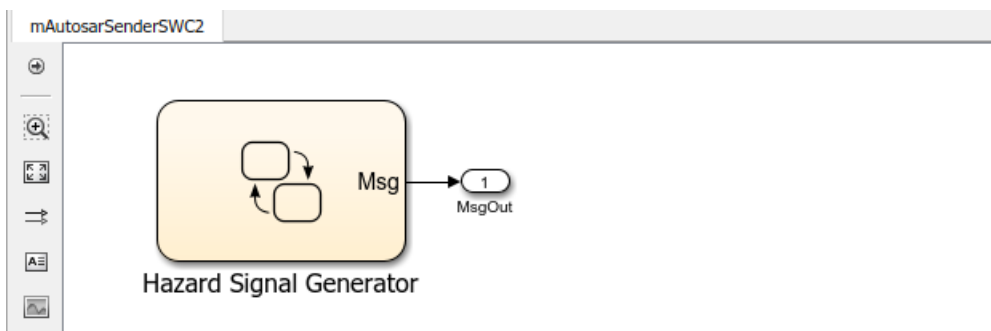
```
openExample('mAutosarQueuedMsgs_N_1.slx') %top model
openExample('mAutosarSenderSWC1.slx')
openExample('mAutosarSenderSWC2.slx')
openExample('mAutosarReceiverSWC.slx')
```



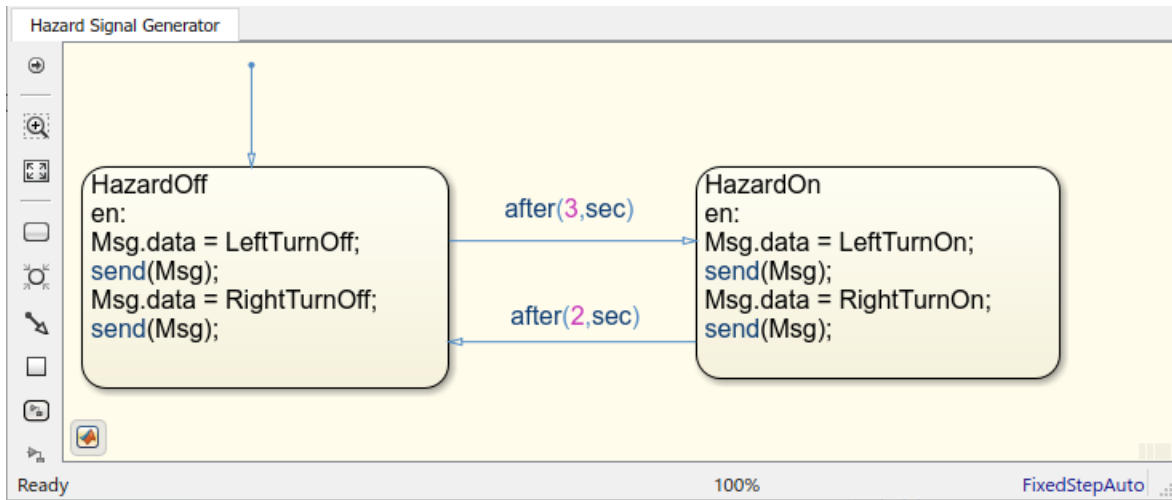
Composition-level model `mAutosarQueuedMsgs_N_1` includes two sender components and a receiver component as referenced models. It connects the sender component `MsgOut` ports to intermediate `MsgJoin` processing logic, which in turn connects to a receiver component `MsgIn` port.

Models `mAutosarSenderSWC1` and `mAutosarReceiverSWC` are the same sender and receiver components configured in “Configure AUTOSAR Sender and Receiver Components for Queued Communication” on page 4-114 and implemented in “Implement AUTOSAR Queued Send and Receive Messaging” on page 4-115. The second sender component, `mAutosarSenderSWC2`, is similar to `mAutosarSenderSWC1`, but implements a second type of message input for the receiver to process.

This figure shows the top level of AUTOSAR queued sender component `mAutosarSenderSWC2`, which contains Stateflow chart `Hazard Signal Generator`. The chart message-line output is connected to Simulink root output `MsgOut`.



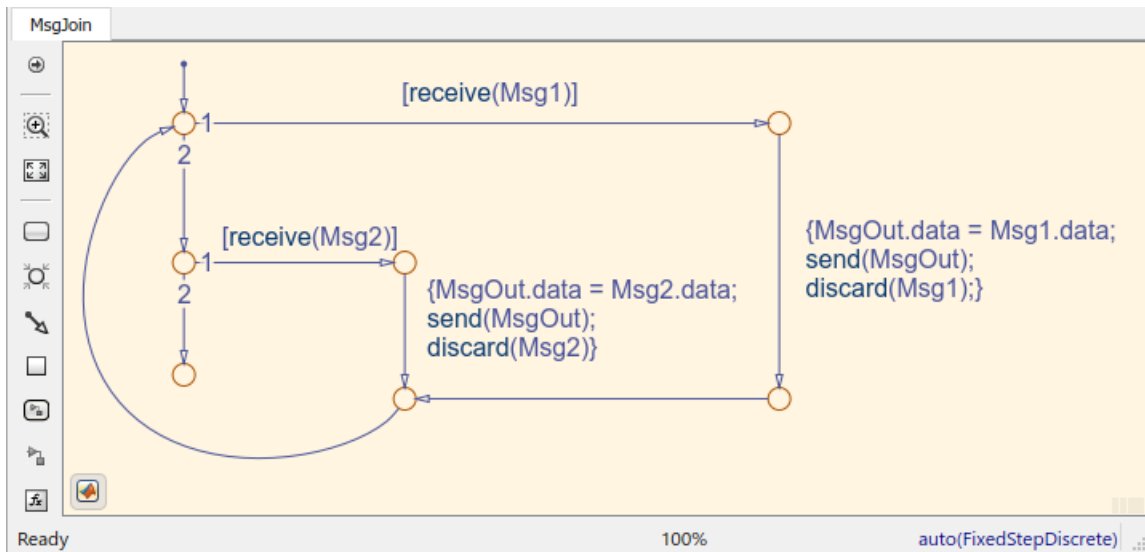
This figure shows the logic implemented in the `Hazard Signal Generator` chart. The chart has two states - `HazardOff` and `HazardOn`. Each state contains entry actions that assign values to message data and send messages. (See “Communicate with Stateflow Charts by Sending Messages” (Stateflow).) Periodic timing drives the message output.



This figure shows the MsgJoin chart located between the sender and receiver components.



This figure shows the logic implemented in the MsgJoin chart. The chart receives queued messages from both sender components and outputs them, one at a time, to the receiver component. Messages from the first sender component, `mAutosarSenderSWC1.slx`, are processed first. For each message received, the chart copies the received message data to the outbound message, sends the data, and discards the received message. (See "Communicate with Stateflow Charts by Sending Messages" (Stateflow).)



The top model `mAutosarQueuedMsgs_N_1` is for simulation only. You can generate AUTOSAR C code and ARXML files for the referenced sender and receiver component models, but not for the containing composition-level model.

Similarly, you can run software-in-the-loop (SIL) simulation for the sender and receiver component models, but not for the composition-level model.

Determine When a Queue Overflows

To check whether a message is lost because it was sent to a queue that was already full, use the Stateflow overflowed operator:

`overflowed(message_name)`

To use the `overflowed` operator, set the model to an `autosar.tlc` target for both simulation and code generation and verify that the inport or outport message connects to an external queue. In each time step, the value of this operator is set when a chart adds a message to, or removes a message from, a queue. It is invalid to use the `overflowed` operator before sending or retrieving a message in the same time step or to check the overflow status of a local message queue.

By default, when a message queue overflows, simulation stops with an error. To prevent a run-time error and allow the `overflowed` operator to dynamically react to dropped messages, set the value of the **Queue Overflow Diagnostic** property to `Warning` or `None`. For more information, see “Queue Overflow Diagnostic” (Stateflow).

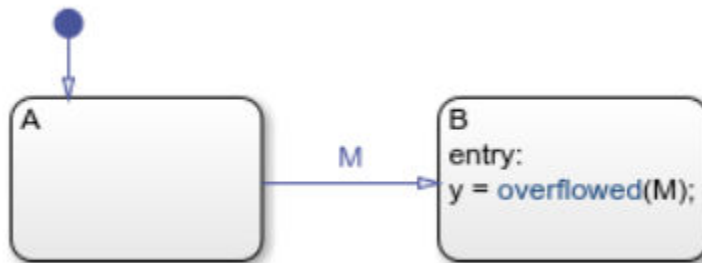
Check for Input Message Overflow

To check the overflow status of an input message queue, first remove a message from the queue. You can:

- Guard a transition with the message and the `overflowed` operator.



- Guard a transition with the message and call the `overflowed` operator in the entry action of the destination state.



- Guard a state on action with the message and call the `overflowed` operator in the action.

```
A
on M:
if overflowed(M) == true
  x = x+1;
end
```

- In a state action, use the receive operator followed by the overflowed operator.

```
A
during:
if receive(M) == true
  if overflowed(M) == true
    y = y+1;
  end
end
```

Calling the overflowed operator before retrieving an input message in the same time step results in a run-time error.

Check for Output Message Overflow

To check the overflow status of an output message queue, first add a message to the queue. You can:

- Use the send operator followed by the overflowed operator.

```
A
entry:
M.data = 3;
send(M);
if overflowed(M) == true
  x = x+1;
end
```

- Use the forward operator followed by the overflowed operator.

```
A
on M1:
forward(M1, M);
if overflowed(M) == true
  x = x+1;
end
```

Calling the overflowed operator before sending or forwarding an output message in the same time step results in a run-time error.

See Also

overflowed

More About

- “Simulink Messages Overview”
- “Messages” (Stateflow)
- “AUTOSAR Communication”
- “AUTOSAR Component Configuration” on page 4-3

Configure AUTOSAR Ports By Using Simulink Bus Ports

In classic and adaptive AUTOSAR software components, you can model AUTOSAR ports by using root-level Simulink bus ports instead of Inport and Outport blocks. Bus port blocks In Bus Element and Out Bus Element can simplify model interfaces. For more information, see “Simplify Subsystem and Model Interfaces with Bus Element Ports”.

Bus port blocks provide a more intuitive way to model AUTOSAR communication ports, interfaces, and groups of data elements. If you model AUTOSAR ports with In Bus Element and Out Bus Element blocks, and type the bus ports by using bus objects, basic properties of AUTOSAR ports, interfaces, and data elements are configured without using the AUTOSAR Dictionary. To manage component interfaces, you configure Simulink bus objects.

You can use root-level bus ports with:

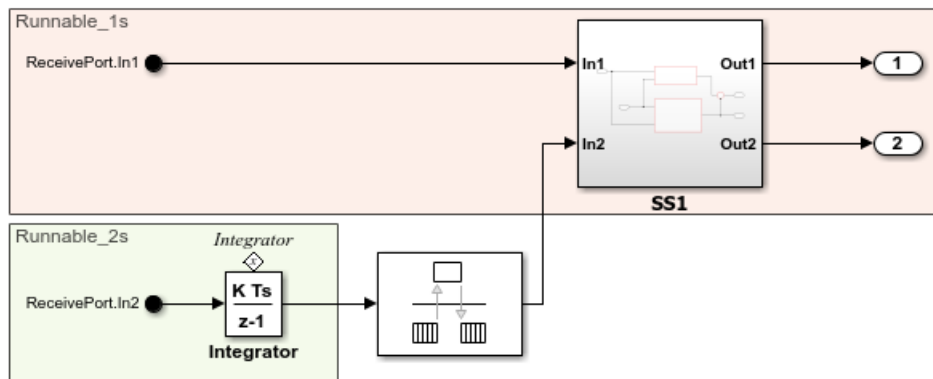
- AUTOSAR software components that use rate-based or export-function modeling styles.
- AUTOSAR signal-based communication.
- AUTOSAR message-based communication, including classic queued sender-receiver (S-R) or adaptive event-based messaging.

In AUTOSAR architecture models, you can link Classic Platform component models that have bus ports and then use the Schedule Editor to schedule the simulation.

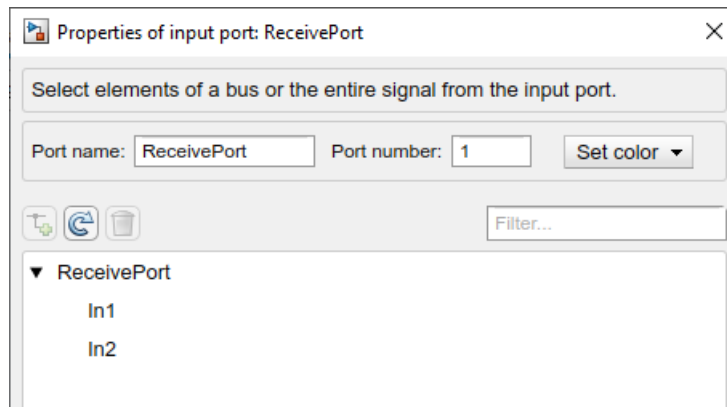
Model AUTOSAR Ports By Configuring Simulink Bus Ports

To configure Simulink bus ports in an AUTOSAR model:

- 1 Create or open an AUTOSAR software component model. The examples in this topic use a writable copy of the example model `autosar_swc`.
- 2 Add two In Bus Element blocks to the model and connect them as root input ports. Configure the bus ports to share the same AUTOSAR port but have different elements. The bus port blocks are automatically mapped to AUTOSAR ports and elements.
 - a Delete the existing Inport blocks in the model.
 - b Create an In Bus Element block. Open the block parameters dialog box. Set **Port name** to `ReceivePort` and the signal name to `In1`.
 - c Connect the block to the first input signal. Make a copy of the block and connect it to the second input signal. In the model canvas (not the block parameters dialog box), click the name of the second block and change `In1` to `In2`.

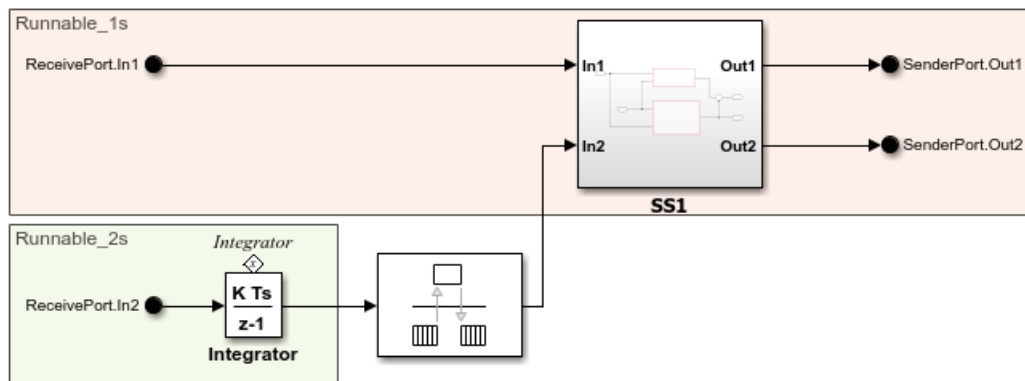


- d The In Bus Element block parameters dialog box now lists both signals.



Edit each signal and set the **Sample time** to 1 and 2, respectively.

- 3 Add two Out Bus Element blocks to the model and connect them as root output ports. Configure the bus ports to share the same AUTOSAR port but have different elements. The bus port blocks are automatically mapped to AUTOSAR ports and elements.
- Delete the existing Outport blocks in the model.
 - Create an Out Bus Element block. Open the block parameters dialog box. Set **Port name** to SenderPort and the signal name to Out1.
 - Connect the block to the first output signal. Make a copy of the block and connect it to the second input signal. In this case, the signal name is automatically set to Out2.



The Out Bus Element block parameters dialog box now lists both signals.

- 4 From the **Apps** tab, open the AUTOSAR Component Designer app.
- Use the Code Mappings editor to verify that the rate-based functions are correctly mapped to AUTOSAR runnables.

Verify that the bus ports are correctly mapped to AUTOSAR ports.

Examine the AUTOSAR data access mode selected for each port. .

- Optionally, open the AUTOSAR Dictionary and view the AUTOSAR component ports, runnables, S-R interfaces, and data elements.

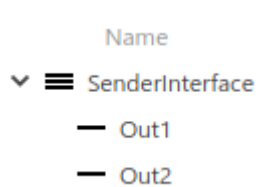
5 If you generate code for the model:

- The generated ARXML file `autosar_swc_component.arxml` describes periodic runnables for each sample rate, named `Runnable_1s` and `Runnable_2s`.
- The generated code file `autosar_swc.c` defines the rate-based functions `Runnable_1s` and `Runnable_2s`.

Model AUTOSAR Interfaces By Typing Bus Ports with Bus Objects

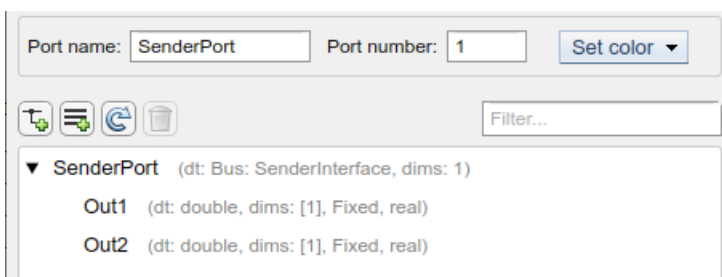
To define an AUTOSAR interface, type a bus port with a bus object. This example uses the same AUTOSAR software component model that was modified in the previous example. The example replaces the sender interface `Output_If` in `autosar_swc` with a new interface named `SenderInterface`.

- 1 With the modified `autosar_swc` open, open the Type Editor. On the **Modeling** tab, in the **Design** gallery, select **Type Editor**.
- 2 In the Type Editor, add a `Simulink.Bus` object and name it `SenderInterface`. Add two `Simulink.BusElement` objects and name them `Out1` and `Out2`.



Optionally, before you exit the dialog, save the `SenderInterface` bus object to a MAT file for later use. The example model `mAutosarSwcBusPorts` does not load a MAT file. Instead, it uses a `PreLoadFcn` model callback to programmatically create the `SenderInterface` bus object.

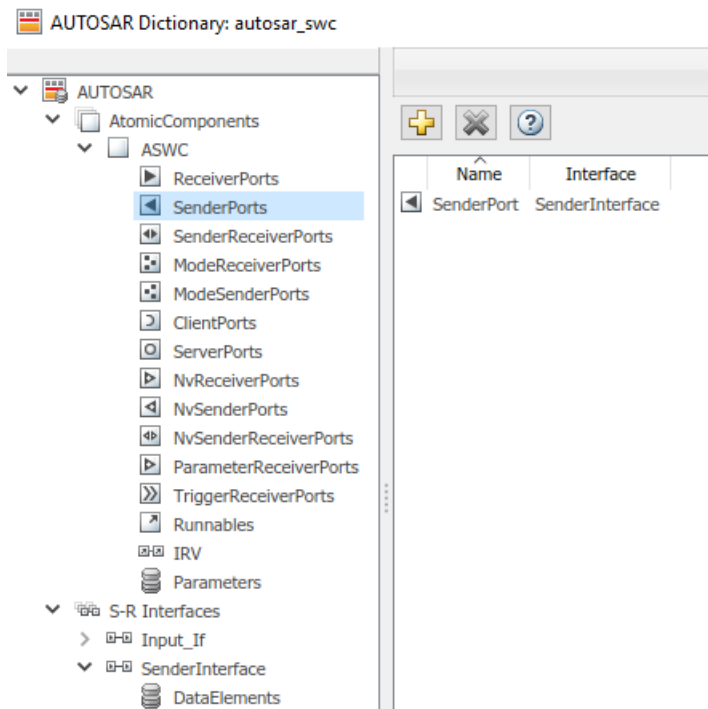
- 3 Open the `SenderPort` block dialog box. Pause on the bus object named `SenderPort` and click the button that appears. Set the **Data type** of the bus object to `Bus: SenderInterface`.



- 4 Because the new interface is replacing an existing mapped interface, you must explicitly delete the existing sender port and sender interface. Open the AUTOSAR Component Designer app and open AUTOSAR Dictionary. Select and delete sender port `SenderPort` and S-R interface `Output_If`.
- 5 To generate and map the new sender interface, either call function `autosar.api.create` to update the model mapping or press **Ctrl+B** to generate model code (requires Embedded Coder). Here is the `autosar.api.create` function call.

```
autosar.api.create('autosar_swc');
```

- 6 Optionally, open the AUTOSAR Dictionary and view the new sender port and S-R interface definitions.



See Also

In Bus Element | Out Bus Element | `autosar.api.create`

Related Examples

- “Model AUTOSAR Communication” on page 2-21
- “Define AUTOSAR Component Behavior by Creating or Linking Models” on page 8-27

More About

- “Simplify Subsystem and Model Interfaces with Bus Element Ports”

Configure AUTOSAR Client-Server Communication

In Simulink, you can model AUTOSAR client-server communication for simulation and code generation. For information about the Simulink blocks you use and the high-level workflow, see “Client-Server Interface” on page 2-24.

To model AUTOSAR servers and clients, you can do either or both of the following:

- Import AUTOSAR servers and clients from ARXML code into a model.
- Configure AUTOSAR servers and clients from Simulink blocks.

This topic provides examples of AUTOSAR server and client configuration that start from Simulink blocks.

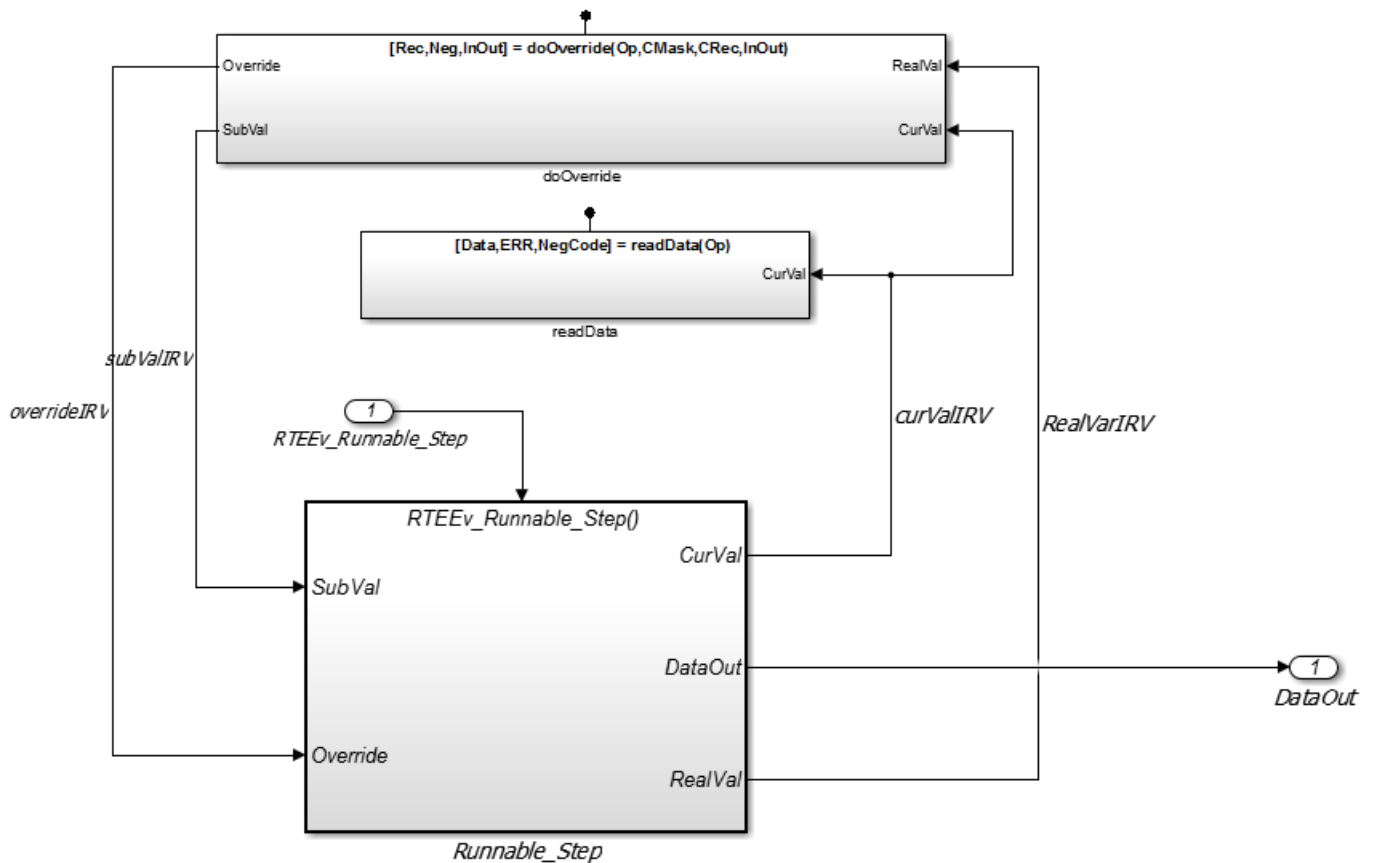
Configure AUTOSAR Server

This example shows how to configure a Simulink Function block as an AUTOSAR server.

- `mControllerWithInterface_server.slx`
- `ExampleApplicationErrType.m`

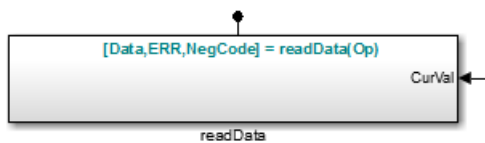
If you copy the files to a working folder, collocate the MATLAB file with the model file.

- 1** Open a model in which you want to create and configure an AUTOSAR server, or open the example model `mControllerWithInterface_server.slx`.
- 2** Add a Simulink Function block to the model. The example model provides two Simulink Function blocks, `doOverride` and `readData`.



- Configure the Simulink Function block to implement a server function. Configure a function prototype and implement the server function algorithm.

In the example model, the contents of the Simulink Function block named readData implement a server function named readData.

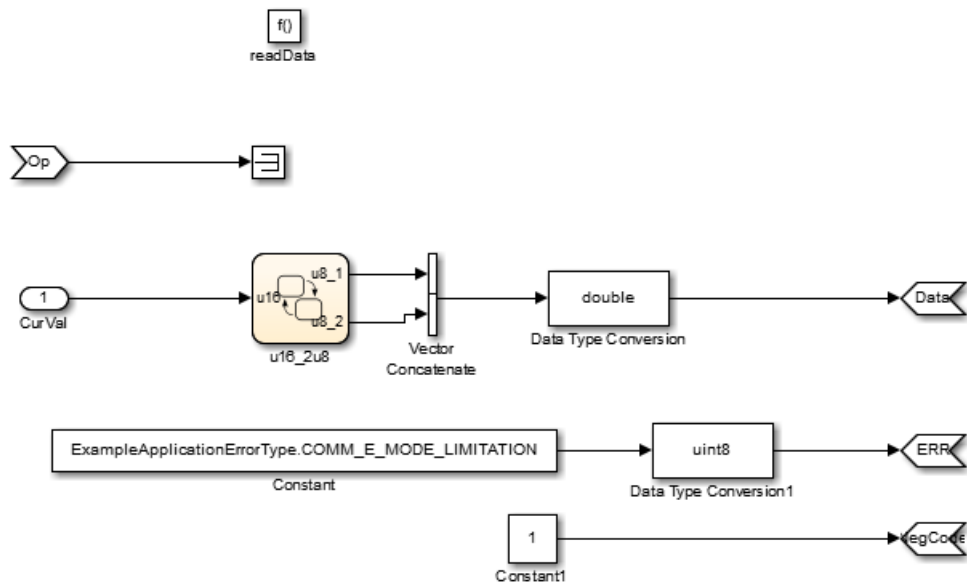


The contents include:

- Trigger block readData, representing a trigger port for the server function. In the Trigger block properties, **Trigger type** is set to function-call. Also, the option **Treat as Simulink function** is selected.
- Argument Inport block Op and Argument Outport blocks Data, ERR, and NegCode, corresponding to the function prototype [Data, ERR, NegCode]=readData(Op).

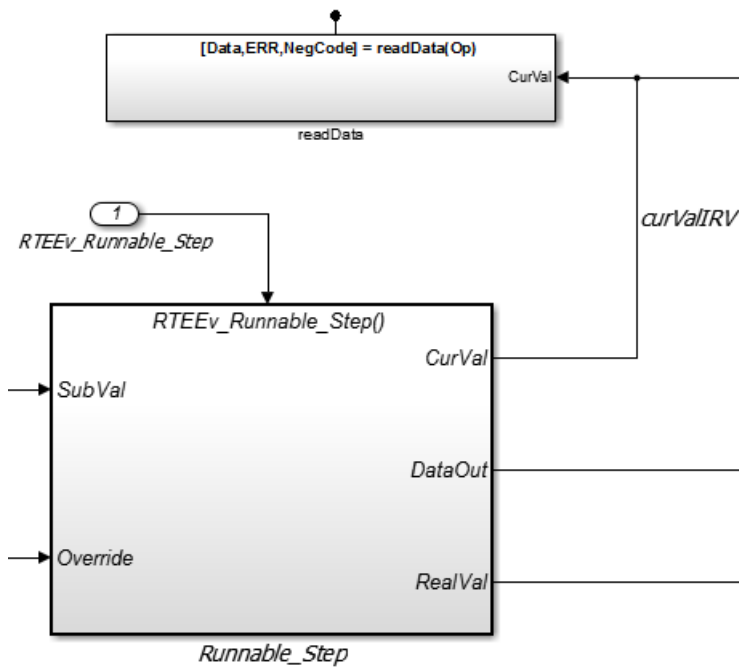
Note When configuring server function arguments, you must specify signal data type, port dimensions, and signal type on the **Signal Attributes** tab of the inport and outport blocks. The AUTOSAR configuration fails validation if signal attributes are absent for server function arguments.

- Blocks implementing the `readData` function algorithm. In this example, a few simple blocks provide `Data`, `ERR`, and `NegCode` output values with minimal manipulation. A `Constant` block represents the value of an application error defined for the server function. The value of `Op` passed by the caller is ignored. In a real-world application, the algorithm could perform a more complex manipulation, for example, selecting an execution path based on the passed value of `Op`, producing output data required by the application, and checking for error conditions.



- 4 When the server function is working in Simulink, set up the Simulink Function block in a model configured for AUTOSAR. For example, configure the current model for AUTOSAR or copy the block into an AUTOSAR model.

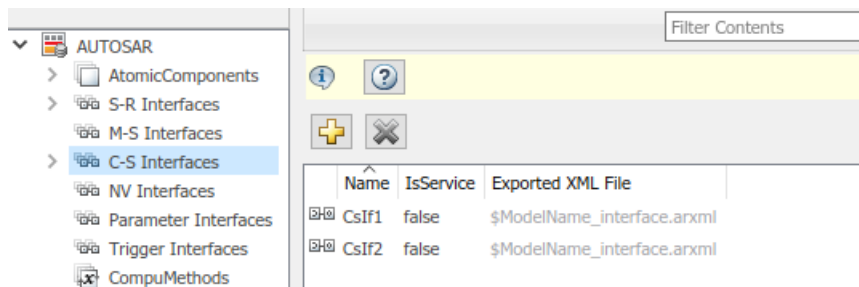
The example model is an AUTOSAR model, into which the Simulink Function block `readData` has been copied. In place of a meaningful `Op` input value for the `readData` function, Simulink data transfer line `CurVal` provides an input value that is used in the function algorithm.




5 The required elements to configure an AUTOSAR server, in the general order they are created, are:

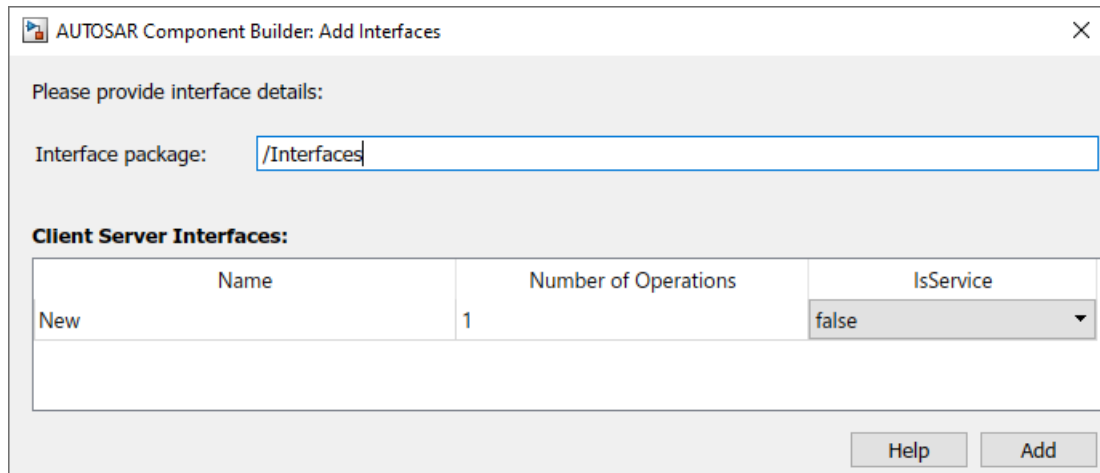
- AUTOSAR client-server (C-S) interface
- One or more AUTOSAR operations for which the C-S interface handles client requests
- AUTOSAR server port to receive client requests for a server operation
- For each server operation, an AUTOSAR server runnable to execute client requests

Open the AUTOSAR Dictionary. To view AUTOSAR C-S interfaces in the model, go to the **C-S Interfaces** view. The example model already contains client-server interfaces.



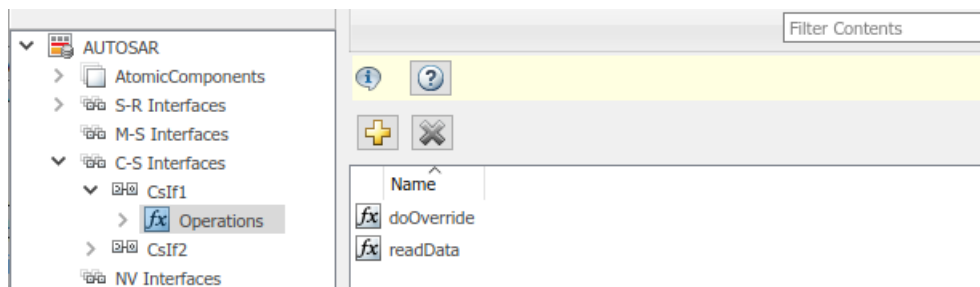
If a C-S interface does not yet exist in your model, create one.

- In the C-S interfaces view, click the **Add** button . This action opens the Add Interfaces dialog box.
- In the dialog box, name the new C-S Interface, and specify the number of operations you intend to associate with the interface. Leave other parameters at their defaults. Click **Add**. The new interface appears in the C-S interfaces view.




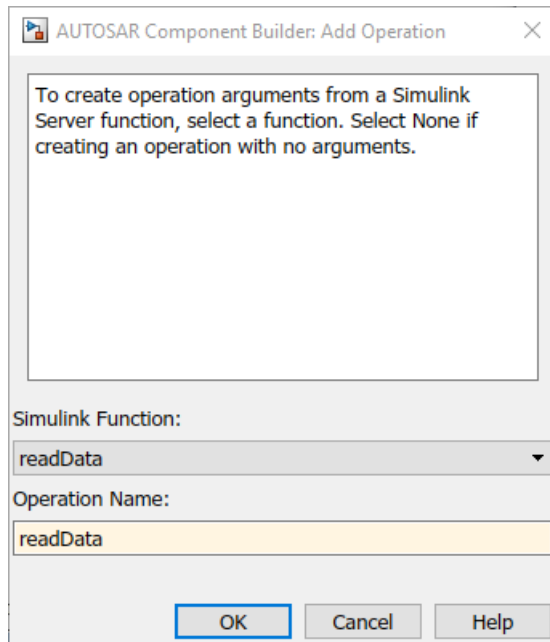
- 6 Under **C-S Interfaces**, create one or more AUTOSAR server operations for which the C-S interface handles client requests. Each operation corresponds to a Simulink server function in the model.

Expand **C-S Interfaces** and expand the individual C-S interface to which you want to add a server operation. (In the example model, expand CsIf1.) To view operations for the interface, select **Operations**. The example model already contains AUTOSAR server operations named doOverride and readData.

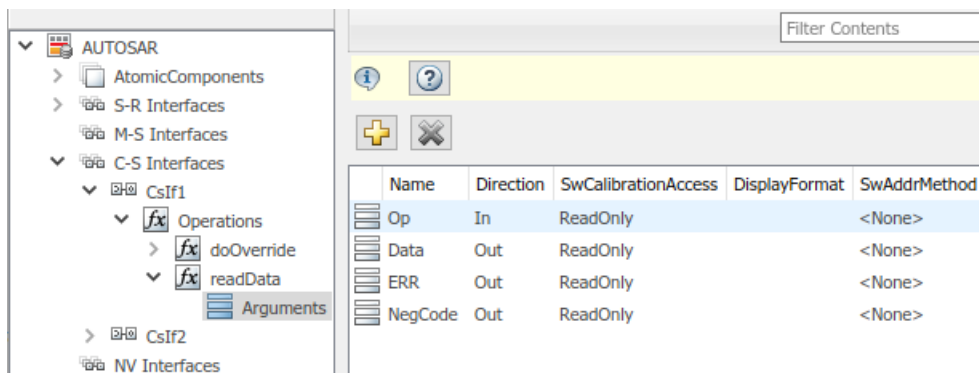


If a server operation does not yet exist in your model, create one. (If your C-S interface contains a placeholder operation named `Operation1`, you can safely delete it.)

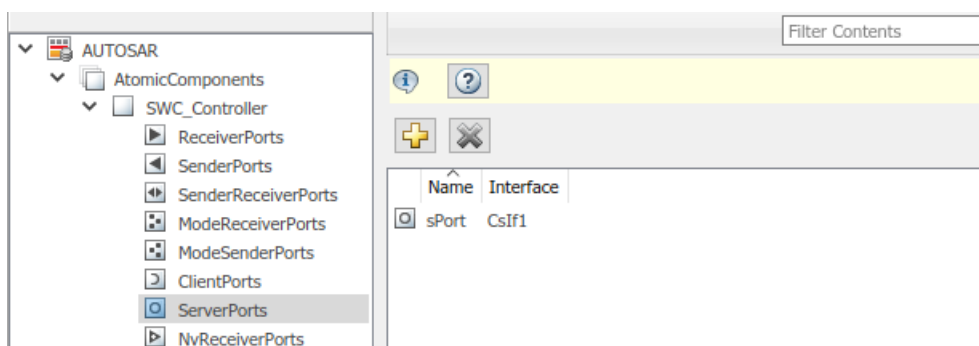
- a In the operations view, click the **Add** button . This action opens the Add Operation dialog box.
- b In the dialog box, enter the **Operation Name**. Specify the name of the corresponding Simulink server function.
- c If the corresponding Simulink server function has arguments, select the function in the **Simulink Function** list. This action causes AUTOSAR operation arguments to be automatically created based on the Simulink server function arguments. Click **OK**. The operation and its arguments appear in the operations view.




- 7 Examine the arguments listed for the AUTOSAR server operation. Expand **Operations**, expand the individual operation (for example, `readData`), and select **Arguments**. The listed arguments correspond to the Simulink server function prototype.

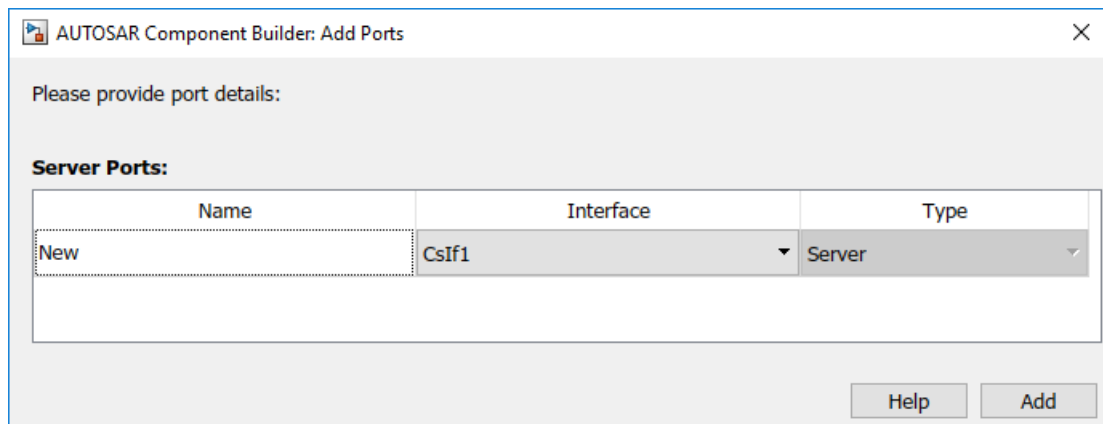


- 8 To view AUTOSAR server ports in the model, go to the server ports view. Expand **AtomicComponents**, expand the individual component that you are configuring, and select **ServerPorts**. The example model already contains an AUTOSAR server port named `sPort`.




If a server port does not yet exist in your model, create one.

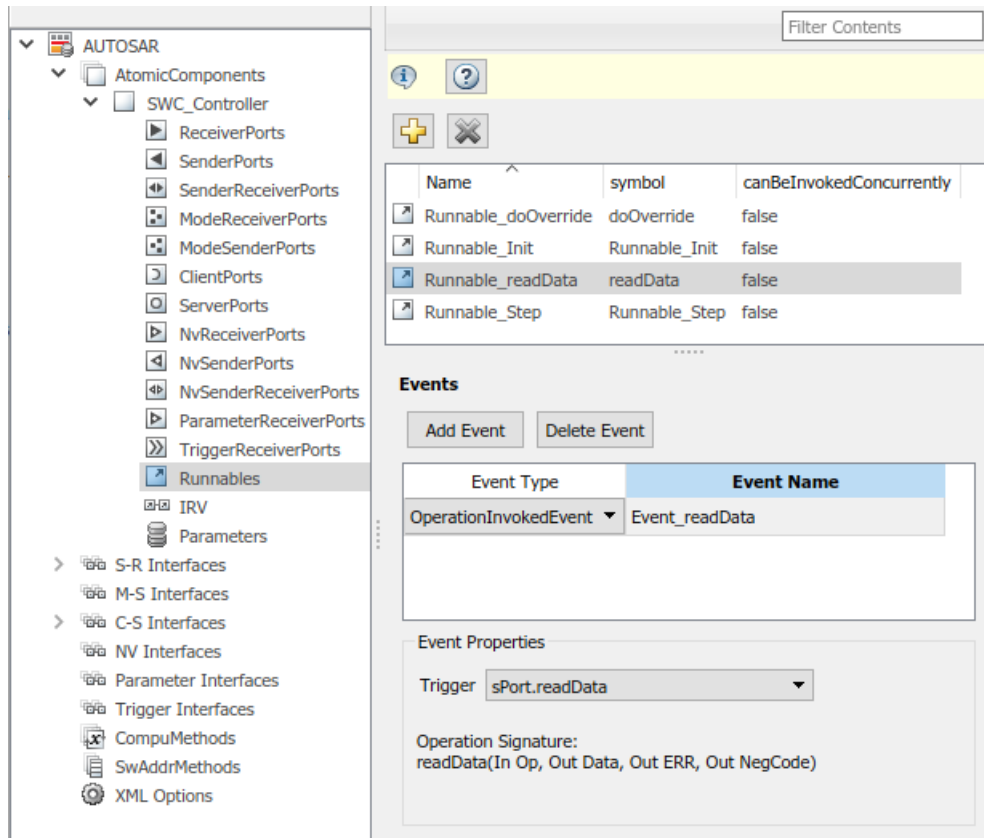
- a In the server ports view, click the **Add** button . This action opens the Add Ports dialog box.
- b In the dialog box, name the new server port, and select the C-S interface for which you configured a server operation. Click **Add**. The new port appears in the server ports view.



- 9 For each AUTOSAR server operation, configure an AUTOSAR server runnable to execute client requests. To view AUTOSAR runnables in the model, select **Runnables**. The example model already contains a server runnable for readData, named Runnable_readData.

If a suitable server runnable does not yet exist in your model, create one.

- a In the runnables view, click the **Add** button . This action adds a table entry for a new runnable.
- b Select the new runnable and configure its name and symbol. (In the example model, the **symbol** name for Runnable_readData is the function name readData.)
- c Create an operation-invoked event to trigger the server runnable. (The example model defines event event_readData for server runnable Runnable_readData.)
 - i Under **Events**, click **Add Event**. Select the new event.
 - ii For **Event Type**, select OperationInvokedEvent.
 - iii Enter the **Event Name**.
 - iv Under **Event Properties**, select a **Trigger** value that corresponds to the server port and C-S operation previously created for the server function. (In the example model, the **Trigger** value selected for Runnable_readData is sPort.readData, combining server port sPort with operation readData.) Click **Apply**.



This step completes the configuration of an AUTOSAR server in the AUTOSAR Dictionary view of the configuration.

- 10 Switch to the Code Mappings editor view of the configuration and map the Simulink server function to the AUTOSAR server runnable.
 - a Open the Code Mappings editor. Select the **Functions** tab.
 - b Select the Simulink server function. To map the function to an AUTOSAR runnable, click on the **Runnable** field and select the corresponding runnable from the list of available server runnables. In the example model, the Simulink function `readData` is mapped to AUTOSAR runnable `Runnable_readData`.

Source	Runnable
fx Exported Function:RTEEv_Runnable_Step	Runnable_Step
fx Initialize	Runnable_Init
fx Simulink Function:doOverride	Runnable_doOverride
fx Simulink Function:readData	Runnable_readData

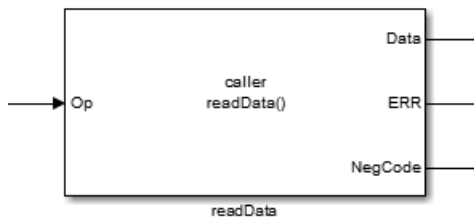
- 11 To validate the AUTOSAR component configuration, click the **Validate** button . If errors are reported, fix the errors, and retry validation. Repeat until validation succeeds.
- 12 Generate C and ARXML code for the model.

After you configure an AUTOSAR server, configure a corresponding AUTOSAR client invocation, as described in “Configure AUTOSAR Client” on page 4-150.

Configure AUTOSAR Client

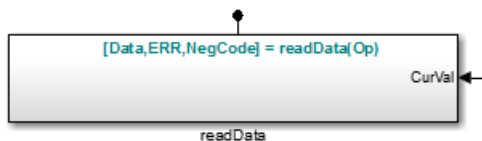
After you configure an AUTOSAR server, as described in “Configure AUTOSAR Server” on page 4-142, configure a corresponding AUTOSAR client invocation. This example shows how to configure a Function Caller block as an AUTOSAR client invocation. The example uses the example model `mControllerWithInterface_client.slx`.

- 1 Open a model in which you want to create and configure an AUTOSAR client, or open the example model `mControllerWithInterface_client.slx`.
- 2 Add a Function Caller block to the model. The example model provides a Simulink Function block named `readData`, which is located inside `Runnable3_Subsystem`.

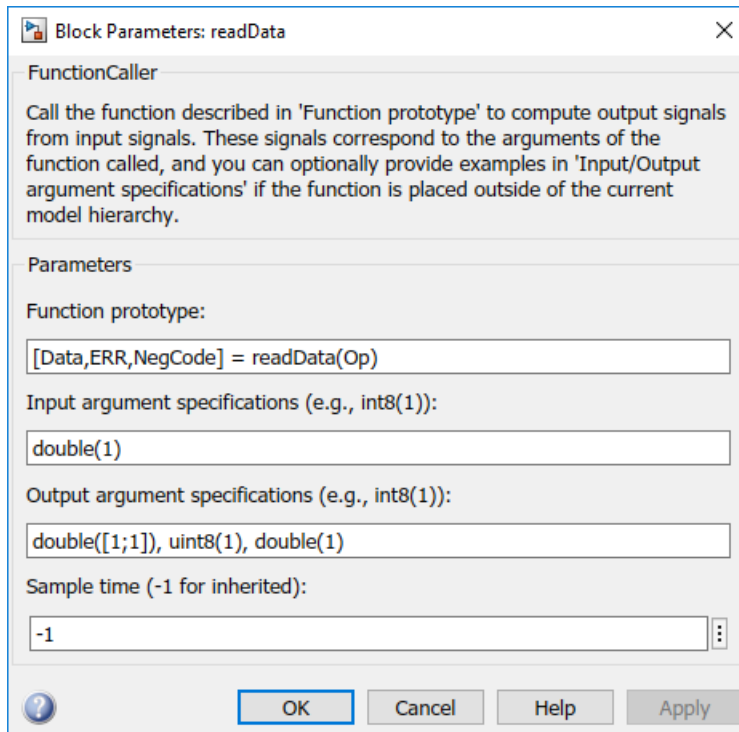


- 3 Configure the Function Caller block to call a corresponding Simulink Function block. Double-click the block to open it, and edit the block parameters to specify the server function prototype.

In the example model, the `readData` Function Caller parameters specify a function prototype for the `readData` server function used in the AUTOSAR server example, “Configure AUTOSAR Server” on page 4-142. Here is the `readData` function from the server example.



The Function Caller parameters include function prototype and argument specification fields. The function name in the prototype must match the **Operation Name** specified for the corresponding server operation. See the operation creation step in “Configure AUTOSAR Server” on page 4-142. The argument types and dimensions also must match the server function arguments.

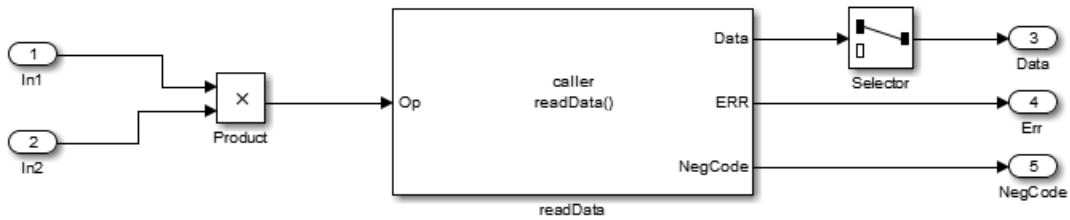



Note If you want to simulate the function invocation at this point, you must place the Function Caller block in a common model or test harness with the corresponding Simulink Function block. Simulation is not required for this example.

- 4 When the function invocation is completely formed in Simulink, set up the Function Caller block in a model configured for AUTOSAR. For example, configure the current model for AUTOSAR or copy the block into an AUTOSAR model.

Tip If you create (or copy) a Function Caller block in a model before you map and configure the AUTOSAR component, you have the option of having the software populate the AUTOSAR operation arguments for you, rather than creating the arguments manually. To have the arguments created for you, along with a fully-configured AUTOSAR client port and a fully mapped Simulink function caller, use the AUTOSAR Component Quick Start to create a default component. For more information, see “Create AUTOSAR Software Component in Simulink” on page 3-2.

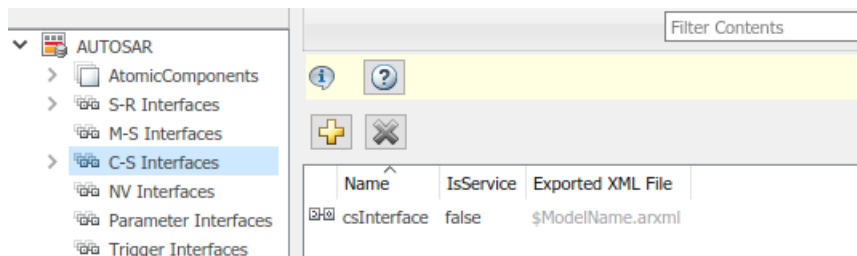
The example model is an AUTOSAR model, into which the Function Caller block `readData` has been copied. The block is connected to inports, outports, and signal lines matching the function argument data types and dimensions.




Note Whenever you add or change a Function Caller block in an AUTOSAR model, update function callers in the AUTOSAR configuration. Open the Code Mappings editor. In the dialog box, click the **Update** button . This action loads or updates Simulink data transfers, function callers, and numeric types in your model. After updating, the function caller you added appears in the **Function Callers** tab of the Code Mappings editor.

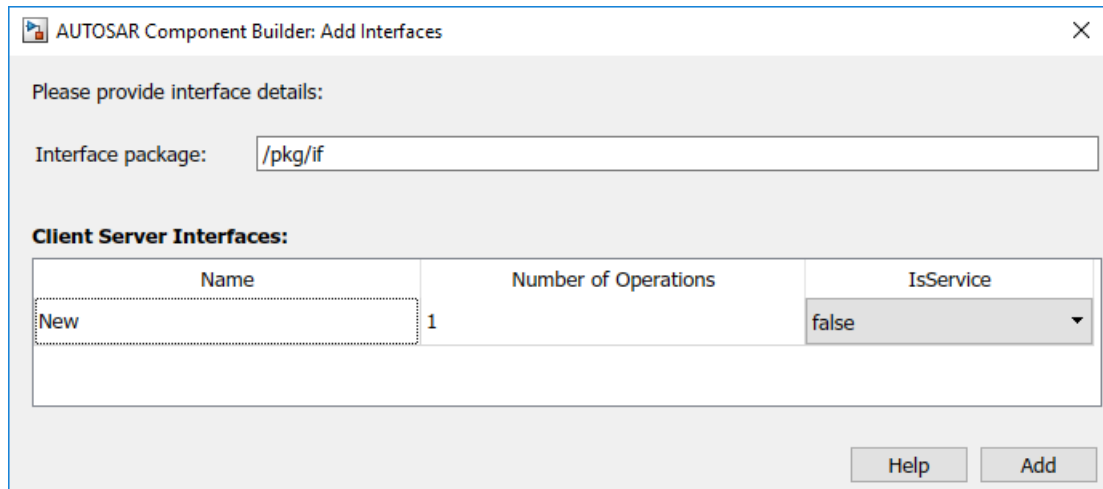
- 5 The required elements to configure an AUTOSAR client, in the general order they should be created, are:
- AUTOSAR client-server (C-S) interface
 - One or more AUTOSAR operations matching the Simulink server functions that you defined in the AUTOSAR server model
 - AUTOSAR client port to receive client requests for a server operation offered by the C-S interface

Open the AUTOSAR Dictionary. To view AUTOSAR C-S interfaces in the model, go to the **C-S Interfaces** view. The example model already contains a client-server interface named **csInterface**.



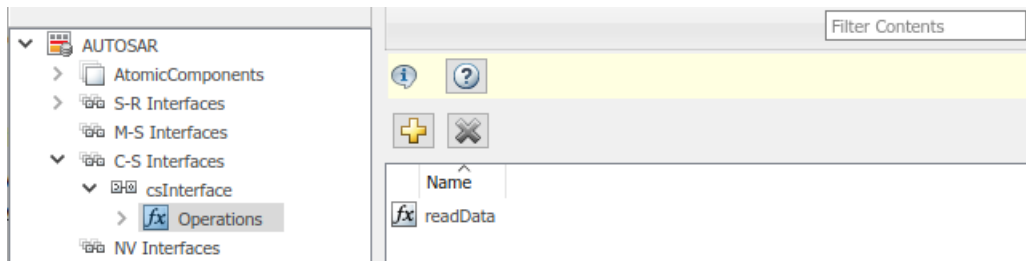
If a C-S interface does not yet exist in the AUTOSAR configuration, create one.

- In the C-S interfaces view, click the **Add** button . This action opens the Add Interfaces dialog box.
- In the dialog box, name the new C-S Interface, and specify the number of operations you intend to associate with the interface. Leave other parameters at their defaults. Click **Add**. The new interface appears in the C-S interfaces view.




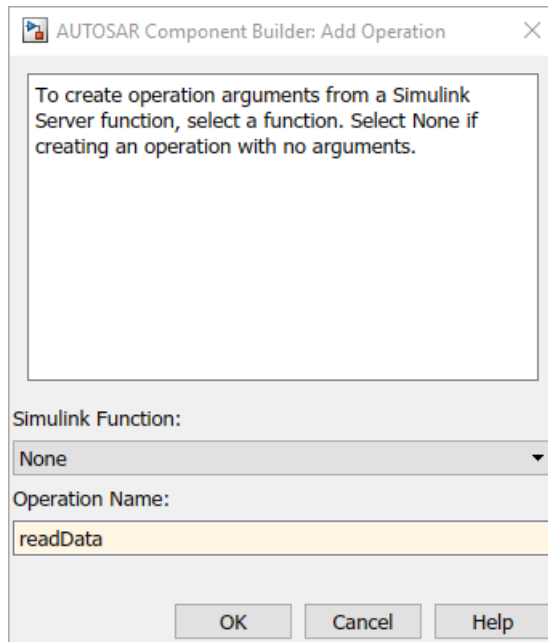
- 6 Under **C-S Interfaces**, create one or more AUTOSAR operations matching the Simulink server functions that you defined in the AUTOSAR server model.


Expand **C-S Interfaces** and expand the individual C-S interface to which you want to add an AUTOSAR operation. (In the example model, expand CsInterface.) To view operations for the interface, select **Operations**. The example model already contains an AUTOSAR operation named readData.

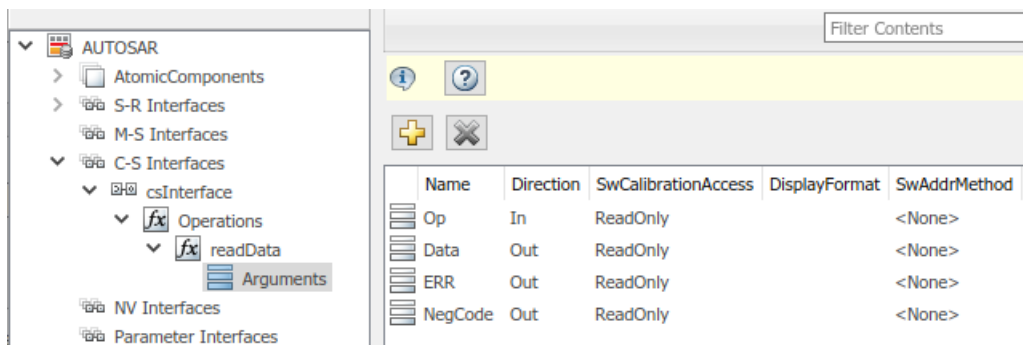


If an AUTOSAR operation does not yet exist in your model, create one. (If your C-S interface contains a placeholder operation named `Operation1`, you can safely delete it.)

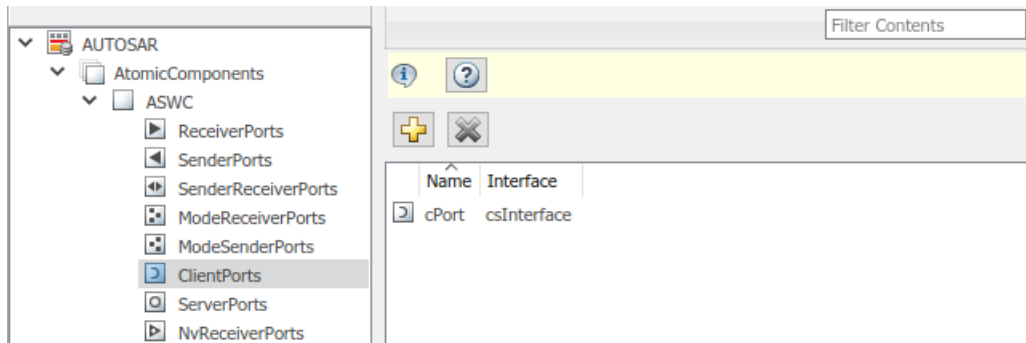
- a In the operations view, click the **Add** button . This action opens the Add Operation dialog box.
- b In the dialog box, enter the **Operation Name**. Specify the name of the corresponding Simulink server function. Leave **Simulink Function** set to **None**, because the client model does not contain the Simulink server function block. Click **OK**. The new operation appears in the operations view.




- 7 Add the AUTOSAR operation arguments.
 - a Expand **Operations**, expand the individual operation (for example, `readData`), and select **Arguments**.
 - b In the arguments view, click the **Add** button  one time for each function argument. For example, for `readData`, click the **Add** button four times, for arguments `Op`, `Data`, `ERR`, and `NegCode`. Each click creates one new argument entry.
 - c Select each argument entry and set the argument **Name** and **Direction** to match the function prototype.

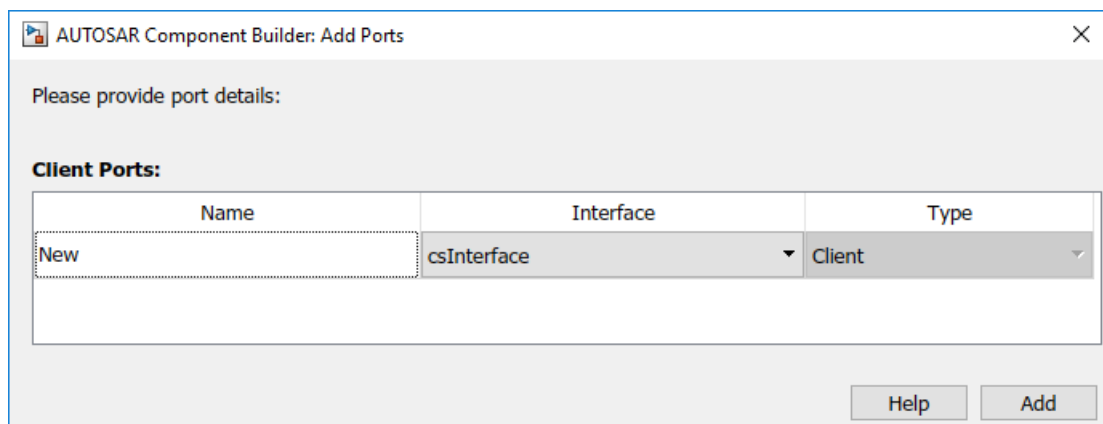


- 8 To view AUTOSAR client ports in the model, go to the client ports view. Expand **AtomicComponents**, expand the individual component that you are configuring, and select **ClientPorts**. The example model already contains an AUTOSAR client port named `cPort`.



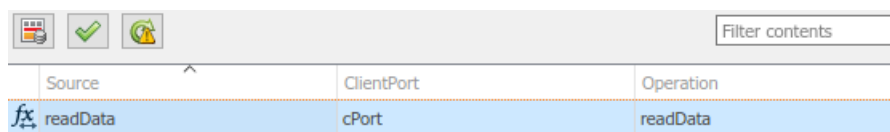
If a client port does not yet exist in your model, create one.


- a In the client ports view, click the **Add** button . This action opens the Add Ports dialog box.
- b In the dialog box, name the new client port, and select a C-S interface. Click **Add**. The new port appears in the client ports view.



This step completes the configuration of an AUTOSAR client in the AUTOSAR Dictionary view of the configuration.

- 9 Switch to the Code Mappings editor view of the configuration and map the Simulink function caller to an AUTOSAR client port and C-S operation.
 - a Open the Code Mappings editor. Select the **Function Callers** tab.
 - b Select the Simulink function caller. Click on the **ClientPort** field and select a port from the list of available AUTOSAR client ports. Click on the **Operation** field and select an operation from the list of available AUTOSAR C-S operations. In the example model, the Simulink function caller `readData` is mapped to AUTOSAR client port `cPort` and C-S operation `readData`.



- 10 To validate the AUTOSAR component configuration, click the **Validate** button . If errors are reported, fix the errors, and retry validation. Repeat until validation succeeds.

- 11 Generate C and ARXML code for the model.

Configure AUTOSAR Client-Server Error Handling

AUTOSAR defines an application error status mechanism for client-server error handling. An AUTOSAR server returns error status, with a value matching a predefined possible error. An AUTOSAR client receives and responds to the error status. An AUTOSAR software component that follows client-server error handling guidelines potentially provides error status to AUTOSAR Basic Software, such as a Diagnostic Event Manager (DEM).

In Simulink, you can:

- Import ARXML code that implements client-server error handling.
- Configure error handling for a client-server interface.
- Generate C and ARXML code for client-server error handling.

If you import ARXML code that implements client-server error handling, the importer creates error status ports at the corresponding server call-point (Function Caller block) locations.

To implement AUTOSAR client-server error handling in Simulink:

- 1 Define the possible error status values that the AUTOSAR server returns in a Simulink data type. Define one or more error codes in the range 0-63, inclusive. The underlying storage of the data type must be an unsigned 8-bit integer. The data scope must be `Exported`. For example, define an enumeration type `appErrType`:

```
classdef(Enumeration) appErrType < uint8

    enumeration
        SUCCESS(0)
        ERROR(1)
        COMM_MODE_LIMITATION(2)
        OVERFLOW(3)
        UNDERFLOW(4)
        VALUE_MOD3(5)
    end

    methods (Static = true)
        function descr = getDescription()
            descr = 'Definition of application error type.';
        end

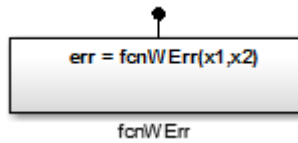
        function hdrFile = getHeaderFile()
            hdrFile = '';
        end

        function retVal = addClassNameToEnumNames()
            retVal = false;
        end

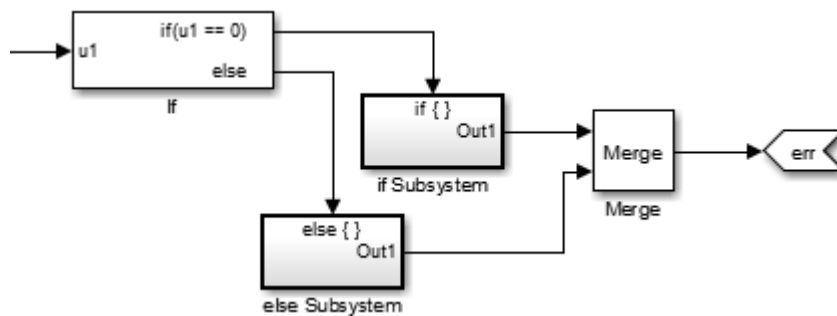
        function dataScope = getDataScope()
            dataScope = 'Exported';
        end
    end
end
```

Note The Simulink data type that you define to represent possible errors in the model does not directly impact the AUTOSAR possible errors that are imported and exported in ARXML code. To modify the exported possible errors for a C-S interface or C-S operation, use AUTOSAR properties functions. This topic provides examples.

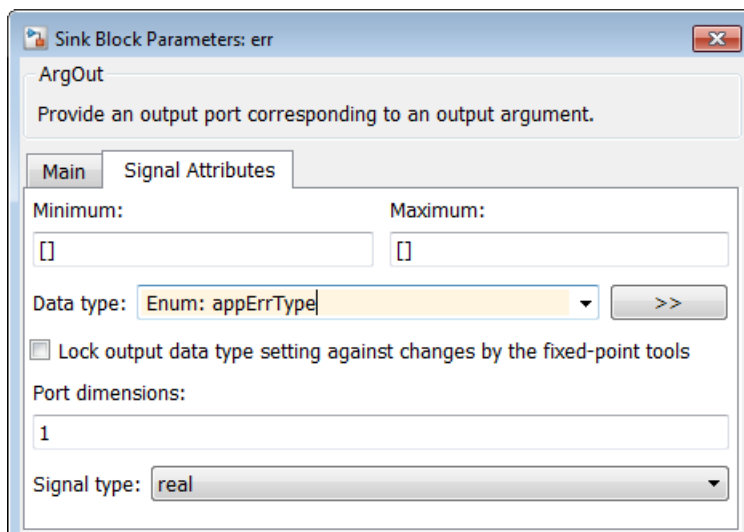
- Define an error status output argument for the Simulink Function block that models the AUTOSAR server. Configure the error status argument as the only function output or add it to other outputs. For example, here is a Simulink Function block that returns an error status value in output `err`.




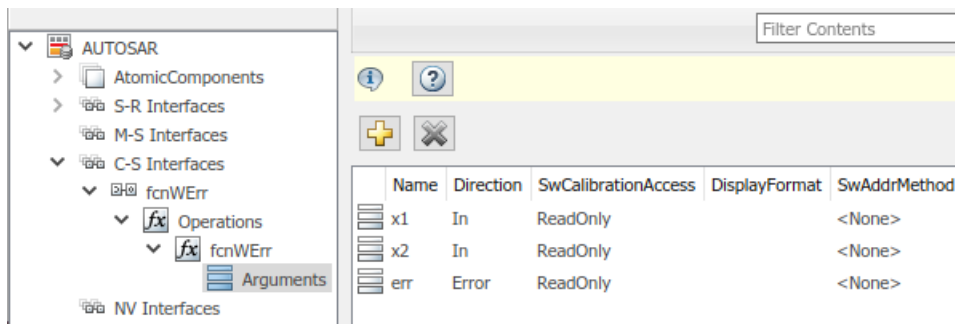
The Simulink Function block implements an algorithm to return error status.



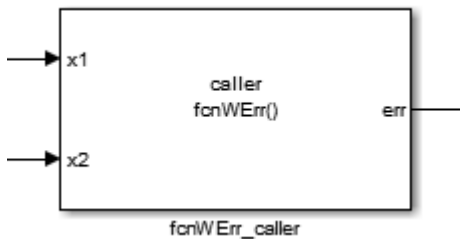
- Reference the possible error values type in the model. In the Argument Outport block parameters for the error outport, specify the error status data type, in this case, `appErrType`. Set **Port dimensions** to 1 and **Signal type** to `real`.



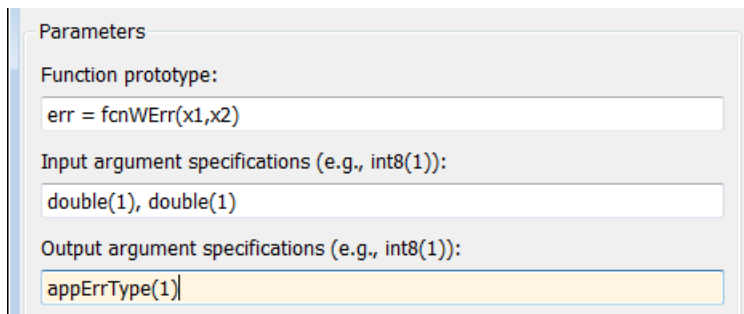
- Configure the AUTOSAR properties of the error argument in the client-server interface. Open the AUTOSAR Dictionary, expand **C-S Interfaces**, and navigate to the **Arguments** view of the AUTOSAR operation. To add an argument, click the **Add** button . Configure the argument name and set **Direction** to Error.



- 5 Create an error port in each Function Caller block that models an AUTOSAR client invocation. For example, here is a Function Caller block that models an invocation of `fcnWErr`.



In the Function Caller block parameters, specify the same error status data type.



Configure the AUTOSAR properties of the error argument to match the information in the AUTOSAR Dictionary, **Arguments** view, shown in Step 4.

The generated C code for the function reflects the configured function signature and the logic defined in the model for handling the possible errors.

```
appErrType fcnWErr(real_T x1, real_T x2)
{
    appErrType rty_err_0;
    if (...) == 0.0) {
        rty_err_0 = ...;
    } else {
        rty_err_0 = ...;
    }

    return rty_err_0;
}
```

Additionally, for the enumeration type class definition used in this example, the build generates header file `appErrType.h`, containing the possible error type definitions.

The exported ARXML code contains the possible error definitions, and references to them.

```
<POSSIBLE-ERRORS>
  <APPLICATION-ERROR ...>
    <SHORT-NAME>SUCCESS</SHORT-NAME>
    <ERROR-CODE>0</ERROR-CODE>
  </APPLICATION-ERROR>
  <APPLICATION-ERROR ...>
    <SHORT-NAME>ERROR</SHORT-NAME>
    <ERROR-CODE>1</ERROR-CODE>
  </APPLICATION-ERROR>
  ...
  <APPLICATION-ERROR ...>
    <SHORT-NAME>UNDERFLOW</SHORT-NAME>
    <ERROR-CODE>4</ERROR-CODE>
  </APPLICATION-ERROR>
  <APPLICATION-ERROR ...>
    <SHORT-NAME>VALUE_MOD3</SHORT-NAME>
    <ERROR-CODE>5</ERROR-CODE>
  </APPLICATION-ERROR>
</POSSIBLE-ERRORS>
```

You can use AUTOSAR property functions to programmatically modify the possible errors that are exported in ARXML code, and to set the **Direction** property of a C-S operation argument to `Error`.

The following example adds UNDERFLOW and VALUE_MOD3 to the possible errors for a C-S interface named `fcnWErr`.

```
>> arProps = autosar.api.getAUTOSARProperties(bdroot)
>> get(arProps, 'fcnWErr', 'PossibleError')
ans =
    'fcnWErr/SUCCESS'    'fcnWErr/ERROR'    'fcnWErr/COMM_MODE...'
    'fcnWErr/OVERFLOW'
>> get(arProps, 'fcnWErr/OVERFLOW', 'errorCode')
ans =
    3
>> add(arProps, 'fcnWErr', 'PossibleError', 'UNDERFLOW')
>> set(arProps, 'fcnWErr/UNDERFLOW', 'errorCode', 4)
>> add(arProps, 'fcnWErr', 'PossibleError', 'VALUE_MOD3')
>> set(arProps, 'fcnWErr/VALUE_MOD3', 'errorCode', 5)
>> get(arProps, 'fcnWErr', 'PossibleError')
ans =
    'fcnWErr/SUCCESS'    'fcnWErr/ERROR'    'fcnWErr/COMM_MODE...'
    'fcnWErr/OVERFLOW'    'fcnWErr/UNDERFLOW'    'fcnWErr/VALUE_MOD3'
```

You can also access possible errors on a C-S operation. The following example lists possible errors for operation `fcnWErr` on C-S interface `fcnWErr`.

```
>> arProps = autosar.api.getAUTOSARProperties(bdroot)
>> get(arProps, 'fcnWErr/fcnWErr', 'PossibleError')
ans =
    'fcnWErr/SUCCESS'    'fcnWErr/ERROR'    'fcnWErr/COMM_MODE...'
    'fcnWErr/OVERFLOW'
```

The following example sets the direction of C-S operation argument `err` to `Error`.

```
>> arProps = autosar.api.getAUTOSARProperties(bdroot)
>> set(arProps, 'fcnWErr/fcnWErr/err', 'Direction', 'Error')
>> get(arProps, 'fcnWErr/fcnWErr/err', 'Direction')
ans =
    Error
```

Concurrency Constraints for AUTOSAR Server Runnables

The following blocks and modeling patterns are incompatible with concurrent execution of an AUTOSAR server runnable.

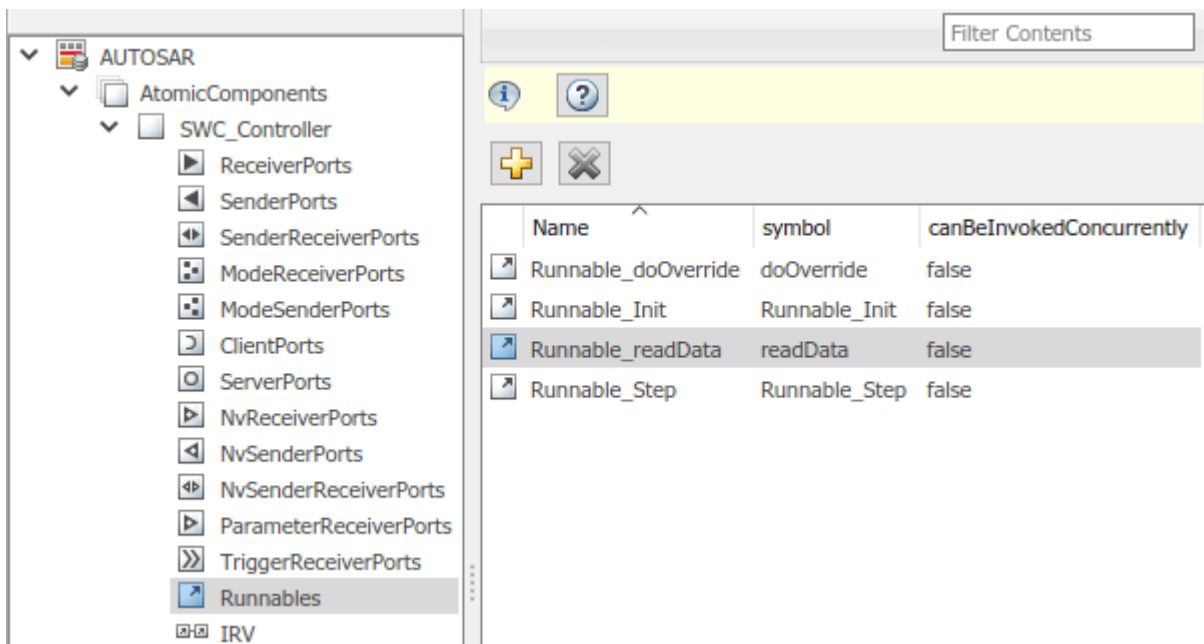
Blocks that are inside a Simulink® function:

- Blocks with state, such as Unit Delay.
- Blocks with zero-crossing logic, such as Triggered Subsystem and Enabled Subsystem.
- Stateflow® charts.
- Other Simulink Function blocks.
- Noninlined subsystems.
- Legacy C function calls with side effects.

Modeling patterns inside a Simulink® function:

- Writing to a data store memory (for example, per-instance-memory).
- Writing to a global block signal (for example, static memory).

To enforce concurrency constraints for AUTOSAR server runnables, use the runnable property `canBeInvokedConcurrently`. The property is located in the **Runnables** view in the AUTOSAR Dictionary.



When `canBeInvokedConcurrently` is set to true for a server runnable, AUTOSAR validation checks for blocks and modeling patterns that are incompatible with concurrent execution of a server runnable. If a Simulink® function contains an incompatible block or modeling pattern, validation reports errors. If `canBeInvokedConcurrently` is set to false, validation does not check for blocks and modeling patterns that are incompatible with concurrent execution of a server runnable.

You can set `canBeInvokedConcurrently` to true only for an AUTOSAR server runnable — that is, a runnable with an `OperationInvokedEvent`. The property `canBeInvokedConcurrently` is not supported for runnables with other event triggers, such as timing events. If `canBeInvokedConcurrently` is set to true for a nonserver runnable, AUTOSAR validation fails.

To programmatically set the runnable property `canBeInvokedConcurrently`, use the AUTOSAR property function set. The following example sets the runnable property

`canBeInvokedConcurrently` to `true` for an AUTOSAR server runnable named `Runnable_readData`.

```
open_system('mControllerWithInterface_server')
arProps = autosar.api.getAUTOSARProperties('mControllerWithInterface_server');
SRPath = find(arProps,[],'Runnable','Name','Runnable_readData')

SRPath = 1x1 cell array
    {'SWC_Controller/ControllerWithInterface_ar/Runnable_readData'}

invConc = get(arProps,'SWC_Controller/ControllerWithInterface_ar/Runnable_readData',...
    'canBeInvokedConcurrently')

invConc = logical
    0

set(arProps,'SWC_Controller/ControllerWithInterface_ar/Runnable_readData',...
    'canBeInvokedConcurrently',true)
invConc = get(arProps,'SWC_Controller/ControllerWithInterface_ar/Runnable_readData',...
    'canBeInvokedConcurrently')

invConc = logical
    1
```

Configure and Map AUTOSAR Server and Client Programmatically

To programmatically configure AUTOSAR properties of AUTOSAR client-server interfaces, use AUTOSAR property functions such as `set` and `get`.

To programmatically configure Simulink to AUTOSAR mapping information for AUTOSAR clients and servers, use these functions:

- `getFunction`
- `getFunctionCaller`
- `mapFunction`
- `mapFunctionCaller`

For example scripts that use AUTOSAR property and map functions, see “Configure AUTOSAR Client-Server Interfaces” on page 4-314.

See Also

[Simulink Function](#) | [Function Caller](#) | [Trigger](#) | [Argument Inport](#) | [Argument Outport](#)

Related Examples

- “Client-Server Interface” on page 2-24
- “Configure AUTOSAR Client-Server Interfaces” on page 4-314
- “Import AUTOSAR XML Descriptions Into Simulink” on page 3-13
- “Configure AUTOSAR Code Generation” on page 5-7

More About

- “Model AUTOSAR Communication” on page 2-21
- “AUTOSAR Component Configuration” on page 4-3

Configure AUTOSAR Mode-Switch Communication

AUTOSAR mode-switch (M-S) communication relies on a mode manager and connected mode users. The mode manager is an authoritative source for software components to query the current mode and to receive notification when the mode changes (switches). A mode manager can be provided by AUTOSAR Basic Software (BSW) or implemented as an AUTOSAR software component. A mode manager implemented as a software component is called an application mode manager. A software component that queries the mode manager and receives notifications of mode switches is a mode user.

Configure Mode Receiver Port and Mode-Switch Event for Mode User

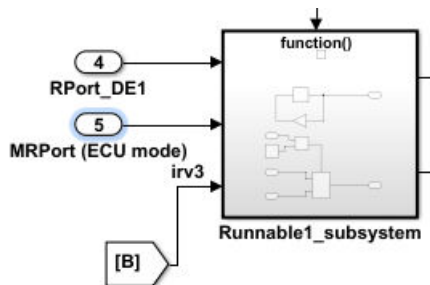
To model a mode user software component, use an AUTOSAR mode receiver port and a mode-switch event. The mode receiver port uses a mode-switch (M-S) interface to connect and communicate with a mode manager, which provides notifications of mode changes. You configure a mode-switch event to respond to a specified mode change by activating an associated runnable. This example shows how to configure an AUTOSAR mode-receiver port, mode-switch event, and related elements for a mode user.

Note This example does not implement a meaningful algorithm for controlling component execution based on the current ECU mode.

- 1 Open a writable copy of the example model `autosar_swc_expfcns`.
- 2 Declare a mode declaration group — a group of mode values — using Simulink enumeration. Specify the storage type as an unsigned integer. Enter the following command in the MATLAB Command Window:

```
Simulink.defineIntEnumType('mdgEcuModes', ...
    {'Run', 'Sleep'}, [0;1], ...
    'Description', 'Mode declaration group for ECU modes', ...
    'DefaultValue', 'Run', ...
    'HeaderFile', 'Rte_Type.h', ...
    'AddClassNameToEnumNames', false, ...
    'StorageType', 'uint16');
```

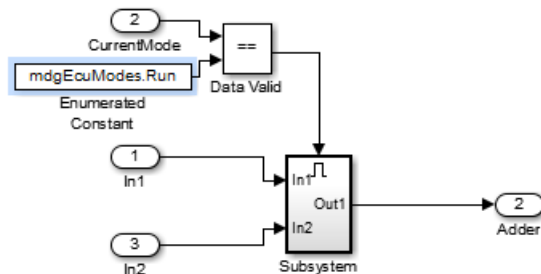
- 3 Rename the Simulink inport `RPort_DE1` (ErrorStatus) to `MRPort` (ECU mode). For example, open the Model Data Editor (on the **Modeling** tab, click **Model Data Editor**). Use the **Source** column to rename the inport. In a later step, you will map this inport to an AUTOSAR mode-receiver port.




- 4 Next, apply the mode declaration group `mdgEcuModes` to the inport. In the Model Data Editor, for the inport, set **Data Type** to Enum: `mdgEcuModes`. Additionally, set **Complexity** to auto.

Model Data									
Inports/Outports		Signals	Data Stores	States	Parameters				
Source	#	Signal Name	Data Type	Min	Max	Dimensions	Complexity	Sample Time	Unit
Runnable2	2		Inherit: auto			1	real	1	inherit
Runnable3	3		Inherit: auto			1	real	10	inherit
RPort_DE1	4		double			1	real	-1	inherit
MRPort (ECU m...	5		Enum: mdgEcu...			1	auto	-1	inherit

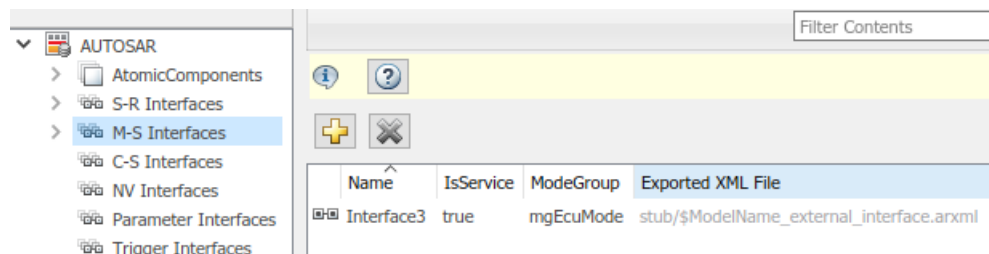
- 5 In the model window, open the function-call subsystem named `Runnable1_subsystem` and make the following changes:
 - a Rename inport `ErrorStatus` to `CurrentMode`.
 - b Replace Constant block `RTE_E_OK` with an Enumerated Constant block. (The Enumerated Constant block can be found in the Sources block group.) Double-click the block to open its block parameters dialog box. Set **Output data type** to `Enum: mdgEcuModes` and set **Value** to `mdgEcuModes.Run`. Click **OK**.




- 6 Add an AUTOSAR mode-switch interface to the model. Open the AUTOSAR Dictionary. Select **M-S Interfaces**. Click the **Add** button . In the Add Interfaces dialog box, specify **Name** as `Interface3` and specify **ModeGroup** as `mgEcuMode`.

The **IsService** property of an M-S interface defaults to `true`. For the purposes of this example, you can leave **IsService** at its default setting, unless you have a reason to change it.

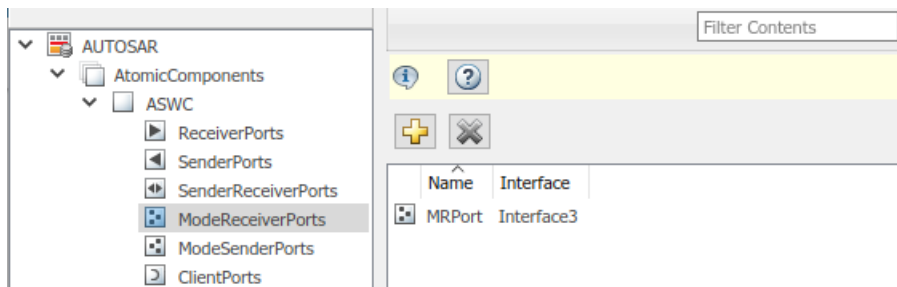
Click **Add**.



The value you specify for the AUTOSAR mode group is used in a later step, when you map a Simulink inport to an AUTOSAR mode-receiver port and element.


- 7 Add an AUTOSAR mode-receiver port to the model. Expand **AtomicComponents**, expand component **ASWC**, and select **ModeReceiverPorts**. To open the Add Ports dialog box, click the **Add** button . In the Add Ports dialog box, specify **Name** as `MRPort`. **Interface** is already set

to **Interface3** (the only available value in this configuration), and **Type** is already set to **ModeReceiver**. Click **Add**.



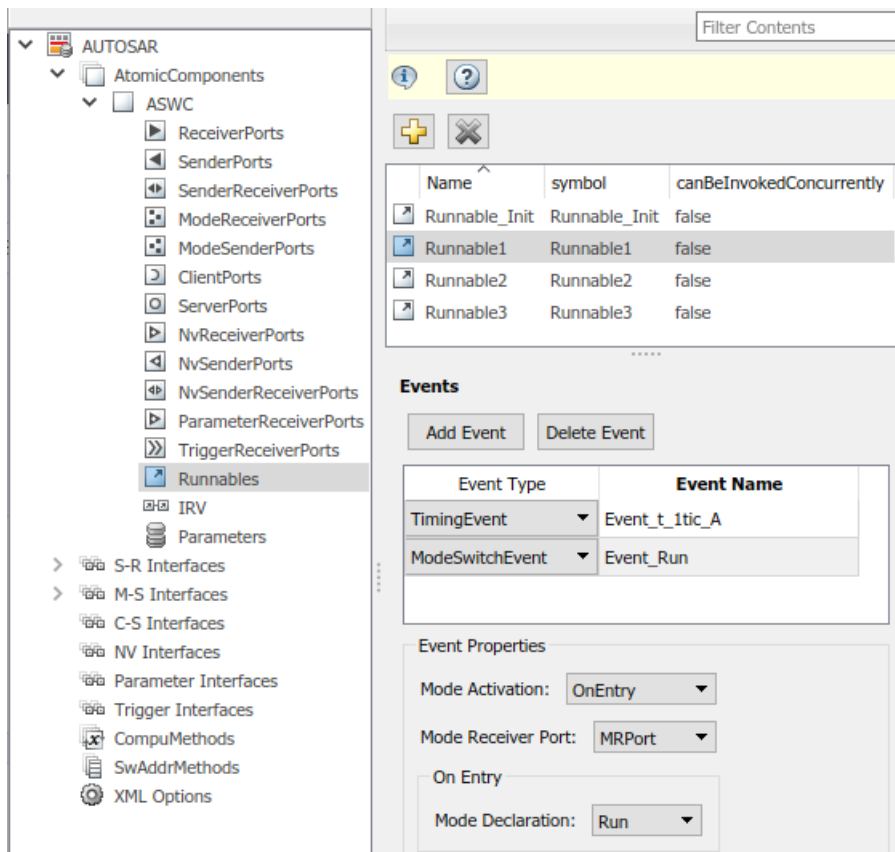
- 8 In the Code Mappings editor, map the Simulink inport **MRPort (ECU mode)** to the AUTOSAR mode-receiver port and element. Open the Code Mappings editor and select the **Inports** tab. In the row for inport **MRPort (ECU mode)**, set **DataAccessMode** to **ModeReceive**, set **Port** to **MRPort**, and set **Element** to **mgEcuMode**. (The AUTOSAR element value matches the **ModeGroup** value you specified when you added AUTOSAR mode-switch interface **Interface3**.)

Source	DataAccessMode	Port	Element
RPort_DE1	ImplicitReceive	RPort	DE1
MRPort (ECU mode)	ModeReceive	MRPort	mgEcuMode
RPort_DE2	ImplicitReceive	RPort	DE2

This step completes the AUTOSAR mode-receiver port configuration. Click the **Validate** button  to validate the AUTOSAR component configuration. If errors are reported, address them and then retry validation. When the model passes validation, save the model.

Note The remaining steps create an AUTOSAR mode-switch event and set it up to trigger activation of an AUTOSAR runnable. If you intend to use ECU modes to control program execution, without using an event to activate a runnable, you can skip the remaining steps and implement the required flow-control logic in your design.


- 9 To add an AUTOSAR mode-switch event for a runnable:
- a Open the AUTOSAR Dictionary. Expand **AtomicComponents**, expand the **ASWC** component, and select **Runnables**. In the list of runnables, select **Runnable1**. This selection activates an **Events** configuration pane for the runnable.
 - b To add an event to the list of events for **Runnable1**, click **Add Event**. For the new event, set **Event Type** to **ModeSwitchEvent**. (This activates an **Event Properties** subpane.) Specify **Event Name** as **Event_Run**.
 - c In the **Event Properties** subpane, set **Mode Activation** to **OnEntry**, set **Mode Receiver Port** to **MRPort**, and set **Mode Declaration** to **Run**. Click **Apply**.



- 10 Open the Code Mappings editor and select the **Functions** tab. In this example model, Simulink entry-point functions have already been mapped to AUTOSAR runnables, including the runnable `Runnable1`, to which you just added a mode-switch event.

The screenshot shows the Code Mappings editor with the 'Functions' tab selected. The table below shows the mapping between Simulink source functions and AUTOSAR runnables:

Source	Runnable
<code>fx</code> Exported Function:Runnable1	Runnable1
<code>fx</code> Exported Function:Runnable2	Runnable2
<code>fx</code> Exported Function:Runnable3	Runnable3
<code>fx</code> Initialize	Runnable_Init

- 11 This completes the AUTOSAR mode-switch event configuration. Click the **Validate** button  to validate the AUTOSAR component configuration. If errors are reported, address them and then retry validation. When the model passes validation, save the model. Optionally, you can generate XML and C code from the model and inspect the results.

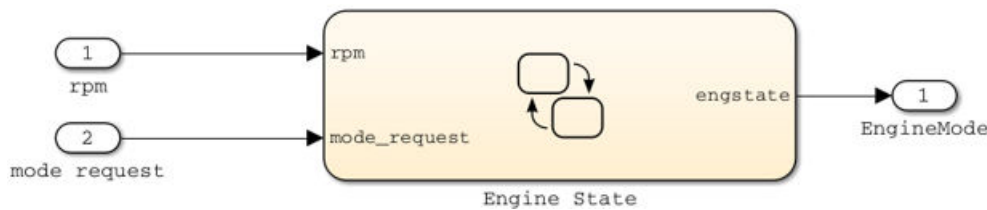
Configure Mode Sender Port and Mode Switch Point for Application Mode Manager

To model an application mode manager software component, use an AUTOSAR mode sender port. Mode sender ports use a mode-switch (M-S) interface to output a mode switch to connected mode user components.

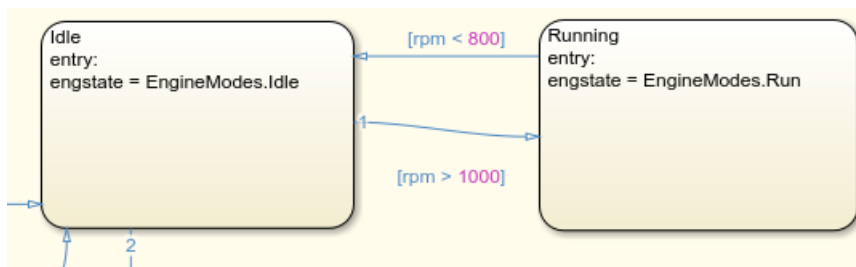
You model the mode sender port as a model root output port, which is mapped to an AUTOSAR mode sender port and a mode-switch (M-S) interface. The output data type is an enumeration class with an unsigned integer storage type, representing an AUTOSAR mode declaration group.


This example shows how to configure a mode sender port and related elements for an application mode manager.

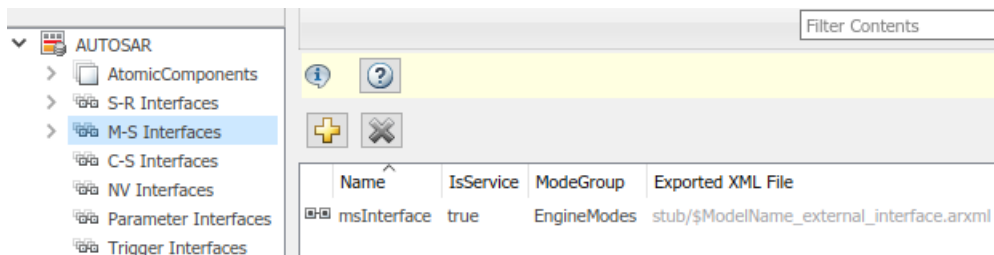
- 1 Open a model configured for AUTOSAR code generation. This example uses a model that contains Stateflow logic for maintaining engine state. The model outputs the current engine mode value.




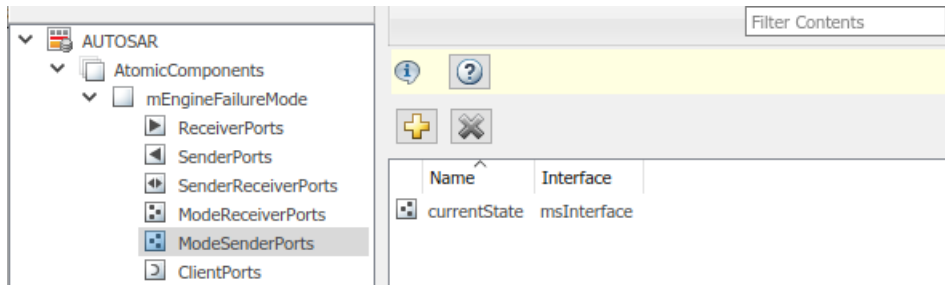
- 2 Declare a mode declaration group — a group of mode values. You can declare mode values with Simulink enumeration. In this example, the Stateflow logic defines EngineModes values Off, Crank, Stall, Idle, and Run. For example:



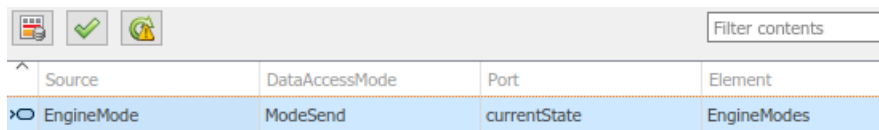
- 3 Add an AUTOSAR M-S interface to the model. Open the AUTOSAR Dictionary and select **M-S Interfaces**. Click the **Add** button . In the Add Interfaces dialog box, set **isService** to **true** and enter a **ModeGroup** name. In this example, the mode declaration group is EngineModes.



- 4 Add an AUTOSAR mode sender port to the model. Expand **AtomicComponents**, expand the component, and select **ModeSenderPorts**. Click the **Add** button . In the Add Ports dialog box, set **Interface** to the name of the M-S interface you created.



- Map the Simulink output that outputs the mode value to the AUTOSAR mode sender port you created. Open the Code Mappings editor and select the **Outports** tab. To map the output to the AUTOSAR mode sender port, set **DataAccessMode** to **ModeSend**, select the **Port** name, and for **Element**, select the mode declaration group name that you specified for the M-S interface.



- Generate code for the model.

The ARXML code includes referenced ModeSwitchPoints, ModeSwitchInterfaces, and ModeDeclarationGroups. For example, the following ARXML code describes the ModeSwitchPoint for the AUTOSAR mode sender port.

```
<RUNNABLE-ENTITY>
...
<MODE-SWITCH-POINTS>
  <MODE-SWITCH-POINT UUID="...">
    <SHORT-NAME>OUT_currentState_EngineModes</SHORT-NAME>
    <MODE-GROUP-IREF>
      <CONTEXT-P-PORT-REF DEST="P-PORT-PROTOTYPE">/pkg/swc/mEngineFailureMode/currentState
    </CONTEXT-P-PORT-REF>
    <TARGET-MODE-GROUP-REF DEST="MODE-DECLARATION-GROUP-PROTOTYPE">
      /pkg/if/msInterface/EngineModes</TARGET-MODE-GROUP-REF>
    </MODE-GROUP-IREF>
  </MODE-SWITCH-POINT>
</MODE-SWITCH-POINTS>
...
</RUNNABLE-ENTITY>
```

The C code includes Rte_Switch API calls to communicate mode switches to other software components. For example, the following code communicates an EngineModes mode switch.

```
/* Outport: '<Root>/EngineMode' */
Rte_Switch_currentState_EngineModes(mEngineFailureMode_B.engstate);
```

See Also

Related Examples

- “Configure AUTOSAR Mode-Switch Interfaces” on page 4-317
- “Configure Disabled Mode for AUTOSAR Runnable Event” on page 4-198

More About


- “Mode-Switch Interface” on page 2-25
- “AUTOSAR Component Configuration” on page 4-3

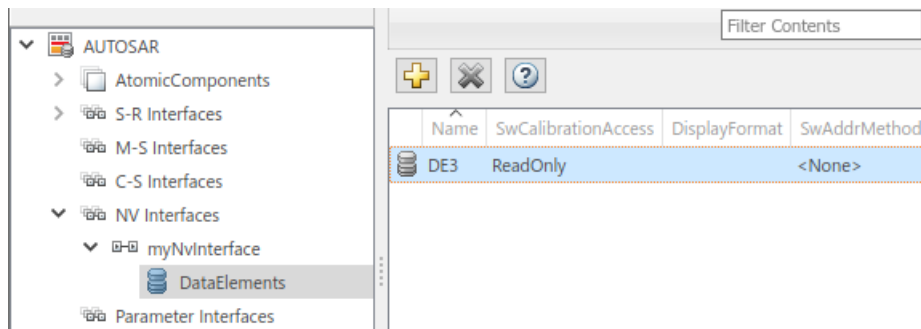
Configure AUTOSAR Nonvolatile Data Communication

The AUTOSAR standard defines port-based nonvolatile (NV) data communication, in which an AUTOSAR software component reads and writes data to AUTOSAR nonvolatile components. To implement NV data communication, AUTOSAR software components define provide and require ports that send and receive NV data. For more information about modeling software component access to AUTOSAR nonvolatile memory, see “Model AUTOSAR Nonvolatile Memory” on page 2-40.

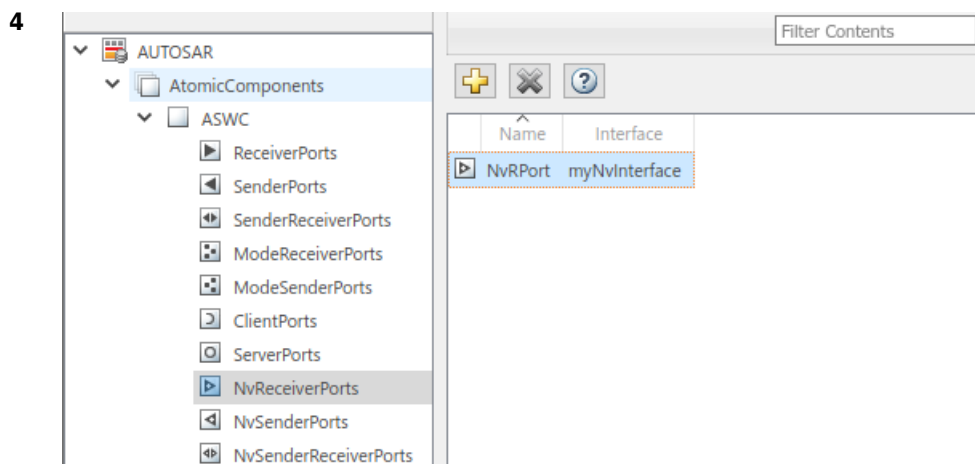
In Simulink, you can create AUTOSAR NV interfaces and ports, and map Simulink inports and outports to AUTOSAR NV ports. You model AUTOSAR NV ports with Simulink inports and outports, in the same manner described in “Sender-Receiver Interface” on page 2-21.

To create an NV data interface and ports in Simulink:

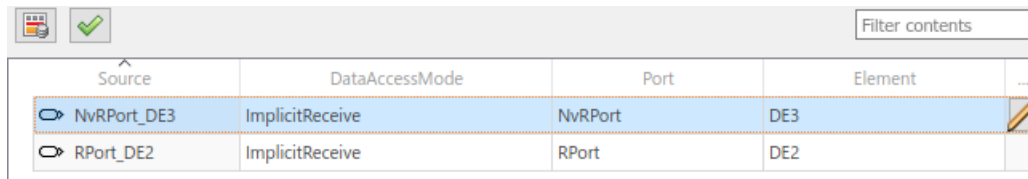
- 1 Add an AUTOSAR NV interface to the model. Open the AUTOSAR Dictionary and select **NV Interfaces**. Click the **Add** button . In the Add Interfaces dialog box, specify the interface name and the number of associated NV data elements.
- 2 Select and expand the new NV interface. Select **DataElements** and modify the data element attributes.



- 3 Add AUTOSAR NV ports to the model. Expand **AtomicComponents** and expand the component. Select and use the **NvReceiverPorts**, **NvSenderPorts**, and **NvSenderReceiverPorts** views to add the NV ports you require. For each NV port, select the NV interface you created.



- 5 Map Simulink inports and outports to the AUTOSAR NV ports you created. Open the Code Mappings editor. Select and use the **Inports** and **Outports** tabs to map the ports. For each import or output, select an AUTOSAR port, data element, and data access mode.



Source	DataAccessMode	Port	Element	...
NvRPort_DE3	ImplicitReceive	NvRPort	DE3	
RPort_DE2	ImplicitReceive	RPort	DE2	

To programmatically configure AUTOSAR NV data communication elements, use the AUTOSAR property and mapping functions. For example, the following MATLAB code adds an AUTOSAR NV data interface and an NV receiver port to an open model. It then maps a Simulink inport to the AUTOSAR NV receiver port.

```
% Add AUTOSAR NV data interface myNvInterface with NV data element DE3
arProps = autosar.api.getAUTOSARProperties(hModel);
addPackageableElement(arProps, 'NvDataInterface', '/pkg/if', 'myNvInterface');
add(arProps, 'myNvInterface', 'DataElements', 'DE3');

% Add AUTOSAR NV receiver port NvRPort, associated with myNvInterface
add(arProps, 'ASWC', 'NvReceiverPorts', 'NvRPort', 'Interface', 'myNvInterface');

% Map Simulink inport NvRPort_DE3 to AUTOSAR port/element pair NvRPort and DE3
slMap = autosar.api.getSimulinkMapping(hModel);
mapInport(slMap, 'NvRPort_DE3', 'NvRPort', 'DE3', 'ImplicitReceive');
```

See Also

Related Examples

- “Nonvolatile Data Interface” on page 2-28
- “Configure AUTOSAR Sender-Receiver Interfaces” on page 4-312
- “Import AUTOSAR XML Descriptions Into Simulink” on page 3-13
- “Configure AUTOSAR Code Generation” on page 5-7

More About

- “Model AUTOSAR Nonvolatile Memory” on page 2-40
- “Model AUTOSAR Communication” on page 2-21
- “AUTOSAR Component Configuration” on page 4-3


Configure AUTOSAR Port Parameters for Communication with Parameter Component

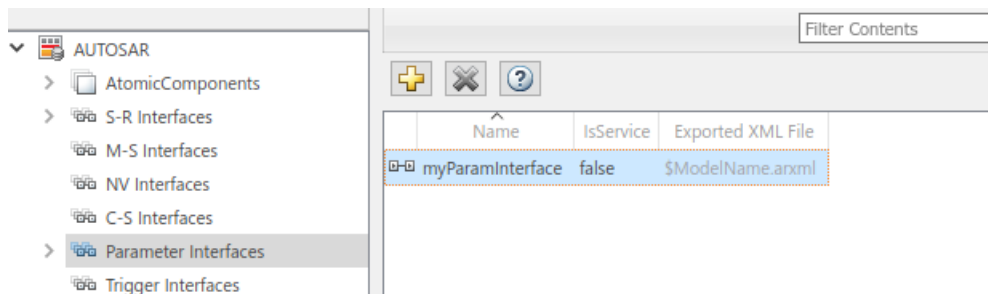
The AUTOSAR standard defines port-based parameters for parameter communication. AUTOSAR parameter communication relies on a parameter software component (ParameterSwComponent) and one or more atomic software components that require port-based access to parameter data. The ParameterSwComponent represents memory containing AUTOSAR parameters and provides parameter data to connected atomic software components. For information about port-based parameter workflows, see “Port Parameters” on page 2-35.

In Simulink, you can model the receiver side of AUTOSAR port-based parameter communication. To configure an AUTOSAR atomic software component as a parameter receiver:

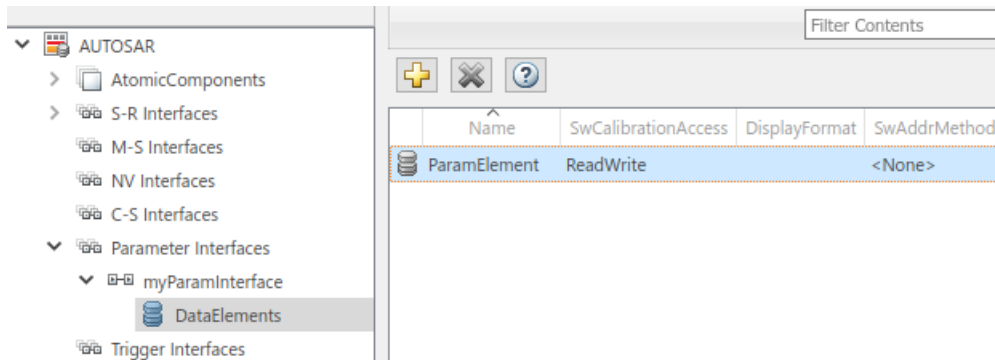
- 1 In an AUTOSAR component model, in the AUTOSAR Dictionary, create an AUTOSAR parameter interface, parameter data elements, and a parameter receiver port.
- 2 In the Simulink model workspace, create a parameter, mark it as an argument, and set an initial value. You can use Simulink parameter, lookup table, and breakpoint objects.
- 3 Map the Simulink model workspace parameter or lookup table to an AUTOSAR parameter receiver port and parameter interface data element. Use the **Parameters** tab of the Code Mappings editor or the `mapParameter` function.


This example shows how to configure an AUTOSAR software component as a receiver for parameter communication.

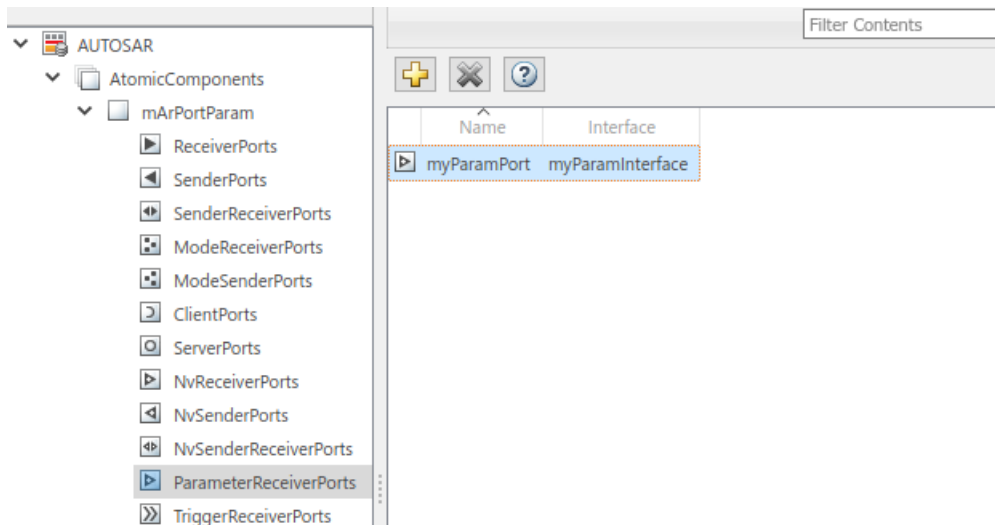
- 1 Open a model configured for AUTOSAR code generation in which the software component requires port-based access to parameter data.
- 2 Open the AUTOSAR Dictionary. To add a parameter interface to the model, select the **Parameter Interfaces** view and click the **Add** button . In the Add Interfaces dialog box, specify the name of the new interface and set **Number of Data Elements** to 1. Click **Add**.



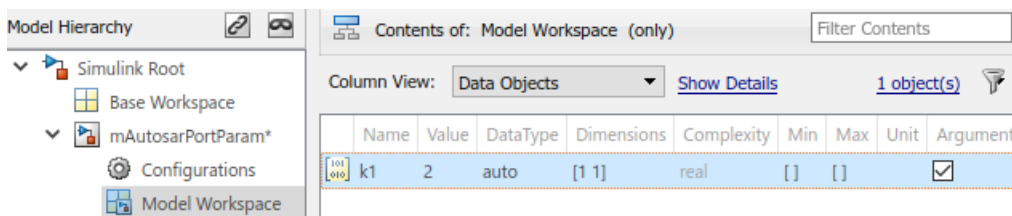
- 3 Expand **Parameter Interfaces** and select the **DataElements** view. Examine and modify the properties of the associated data element that you created, including its name.



- Expand **AtomicComponents** and expand the component. To add a parameter receiver port to the model, go to the **ParameterReceiverPorts** view and click the **Add** button . In the Add Ports dialog box, specify the name of the new port and set **Interface** to the name of the parameter interface that you created. Click **Add**.

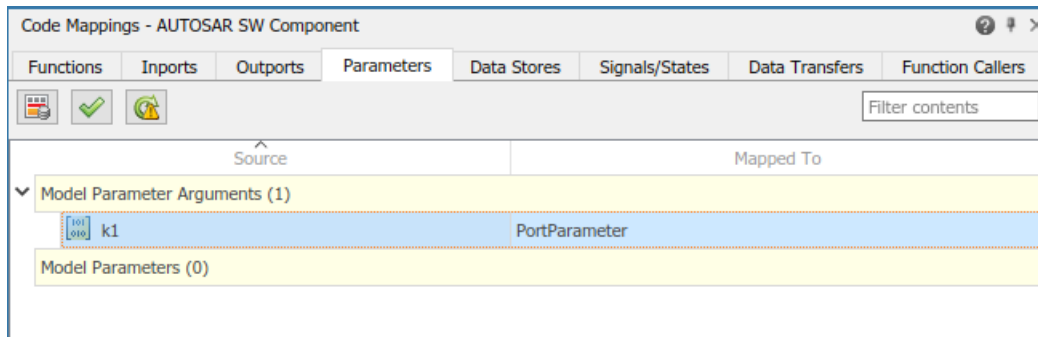


- In the Simulink model workspace, create a data object for the parameter. For example, use Model Explorer. With the data object selected, set the **Name** and **Value** fields. To configure the parameter as a model argument (that is, unique to each instance of a multi-instance model), select the **Argument** check box.




Reference the data object name in the model. For example, enter k1 in the **Gain** parameter field of a Gain block.

- Open the Code Mappings editor and select the **Parameters** tab. In the **Model Parameter Arguments** group, select the parameter data object that you created. In the **Mapped To** menu, select AUTOSAR parameter type PortParameter.



7

To view and modify other code and calibration attributes for the parameter, click the  icon.

- a Set **Port** to the name of the parameter receiver port that you configured in the AUTOSAR Dictionary.
- b Set **DataElement** to the name of the parameter interface data element that you configured in the AUTOSAR Dictionary.

For more information, see “Map Model Workspace Parameters to AUTOSAR Component Parameters” on page 4-54.

8

When you generate code for the AUTOSAR component model:

- The exported ARXML files contain descriptions of the parameter receiver component, parameter interface, parameter data element, and parameter receiver port.

```
<PARAMETER-INTERFACE UUID="...">
  <SHORT-NAME>myParamInterface</SHORT-NAME>
  <IS-SERVICE>false</IS-SERVICE>
  <PARAMETERS>
    <PARAMETER-DATA-PROTOTYPE UUID="...">
      <SHORT-NAME>ParamElement</SHORT-NAME>
      ...
    </PARAMETER-DATA-PROTOTYPE>
  </PARAMETERS>
</PARAMETER-INTERFACE>
```

- The generated C code contains AUTOSAR port parameter Rte function calls.

```
/* Model step function */
void mArPortParam_Step(void)
{
  ...
  Rte_IWrite_mArPortParam_Step_Out2_Out2(Rte_Prm_myParamPort_ParamElement() *
    Rte_IRead_mArPortParam_Step_In2_In2());
}
```

At run time, the software can access the parameter data element as a port-based parameter.

See Also

getParameter | mapParameter

Related Examples

- “Map Model Workspace Parameters to AUTOSAR Component Parameters” on page 4-54
- “Port Parameters” on page 2-35

More About

- “Model AUTOSAR Communication” on page 2-21
- “Model AUTOSAR Component Behavior” on page 2-31

Configure Receiver for AUTOSAR External Trigger Event Communication

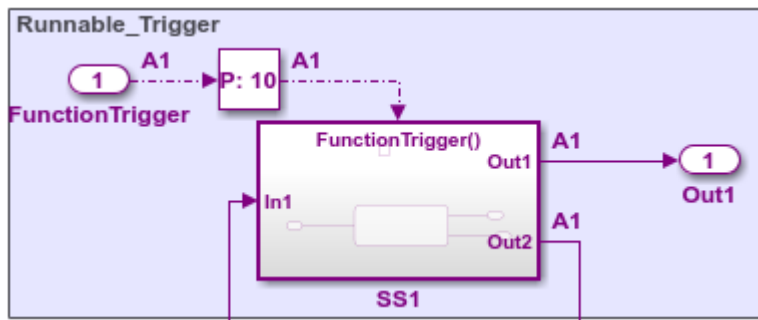
The AUTOSAR standard defines external trigger event communication, in which an AUTOSAR software component or service signals an external trigger occurred event (ExternalTriggerOccurredEvent) to another component. The receiving component activates a runnable in response to the event.


In Simulink, you can model the receiver portion of AUTOSAR external trigger event communication. Select a component that you want to react to an external trigger. In the component, you create a trigger interface, a trigger receiver port to receive an ExternalTriggerOccurredEvent, and a runnable that the event activates.

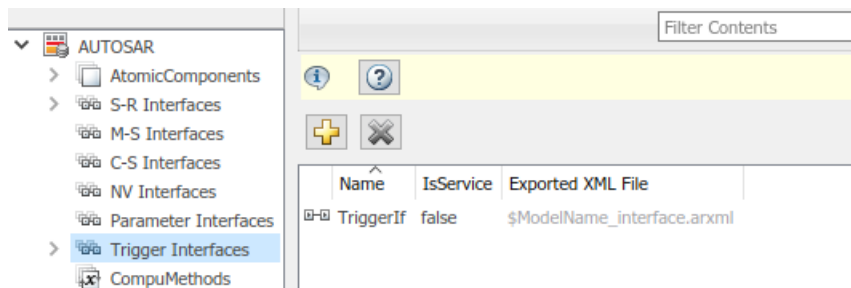
This example shows how to configure an AUTOSAR software component as a receiver for external trigger event communication.

- 1 Open a model configured for AUTOSAR code generation, in which you want to activate a runnable based on receiving an AUTOSAR ExternalTriggerOccurredEvent.

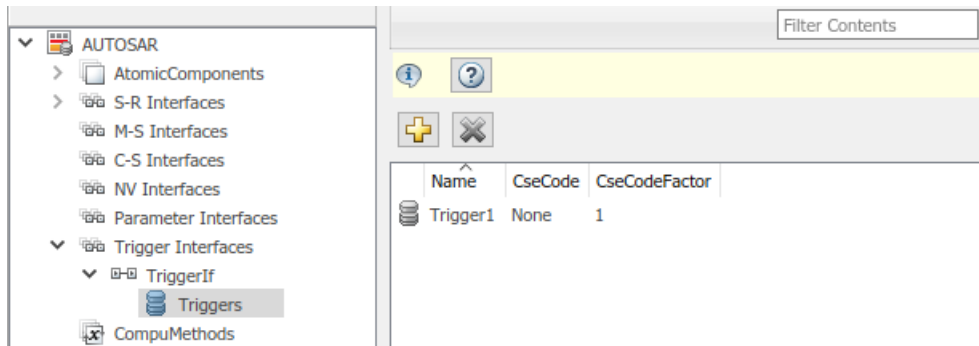
For a sample model that uses external trigger event communication, see `autosar_swc_fcncalls`. In `autosar_swc_fcncalls`, asynchronous function-call subsystem SS1 models an AUTOSAR runnable. An ExternalTriggerOccurredEvent activates the runnable.




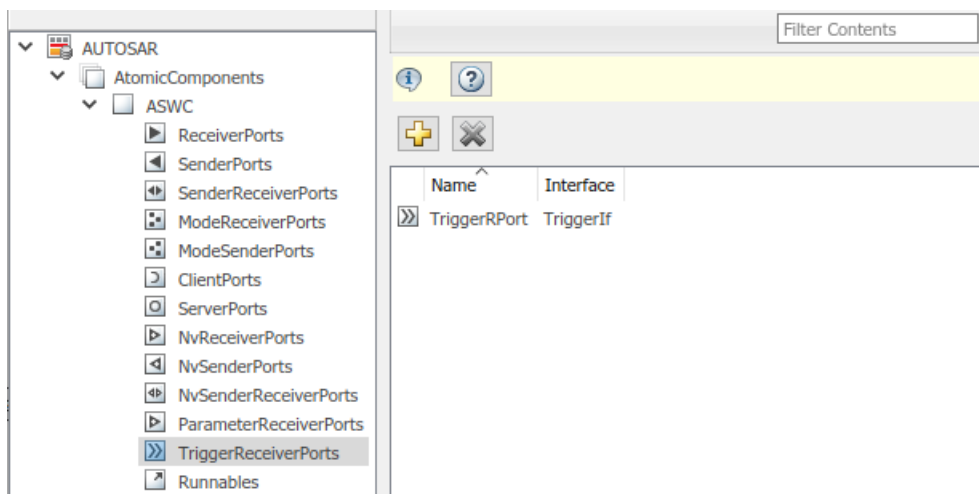
- 2 Open the AUTOSAR Dictionary. Select the **Trigger Interfaces** view and use the **Add** button  to add a trigger interface to the model. In the Add Interfaces dialog box, specify the name of the new interface and set **Number of Triggers** to 1.



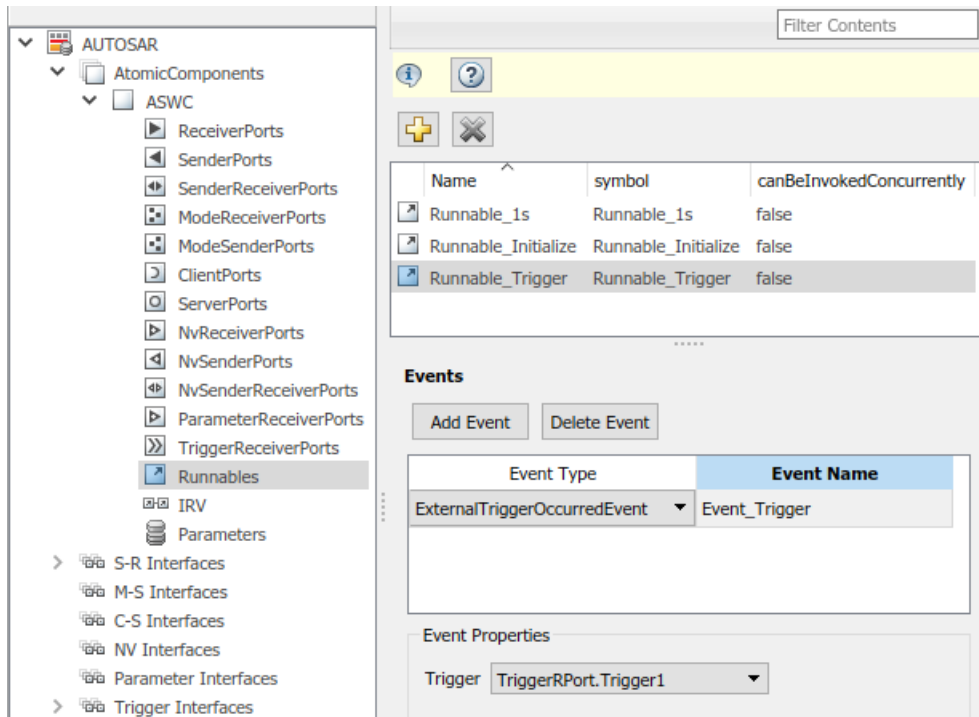
- 3 Expand **Trigger Interfaces** and select the **Triggers** view. Examine the properties of the associated trigger. For an asynchronous (nonperiodic) trigger, set **CseCode** to **None**, indicating an unspecified trigger period. For more information about specifying trigger periods, click the help button in the triggers view.



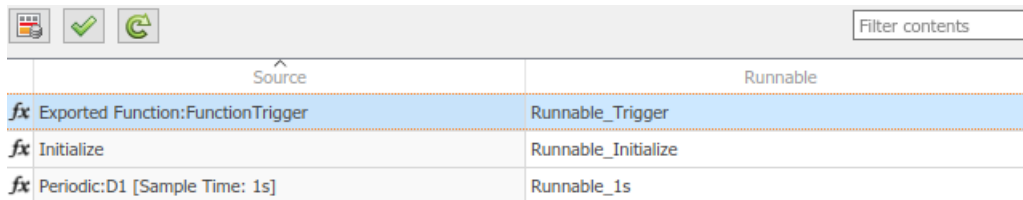
- Expand **AtomicComponents** and expand the component. Select the **TriggerReceiverPorts** view and use the **Add** button  to add a trigger receiver port to the model. In the Add Ports dialog box, specify the name of the new port and set **Interface** to the name of the trigger interface you created.



- Select the **Runnables** view and select the runnable that you want to activate based on receiving an AUTOSAR ExternalTriggerOccurredEvent. In the **Events** subpane, set **Event Type** to ExternalTriggerOccurredEvent. To display event properties, select the event name. For **Trigger**, select the value corresponding to the trigger receiver port and trigger you created.



- To complete the trigger receiver configuration, open the Code Mappings editor and select the **Functions** tab. Select the Simulink entry-point function for the subsystem that models the AUTOSAR ExternalTriggerOccurredEvent runnable. In the **Runnable** field, select the runnable name.



See Also

Related Examples

- “Add Top-Level Asynchronous Trigger to Periodic Rate-Based System” on page 4-193
- “Import AUTOSAR XML Descriptions Into Simulink” on page 3-13
- “Configure AUTOSAR Code Generation” on page 5-7

More About

- “Model AUTOSAR Communication” on page 2-21
- “AUTOSAR Component Configuration” on page 4-3

Configure AUTOSAR Runnables and Events

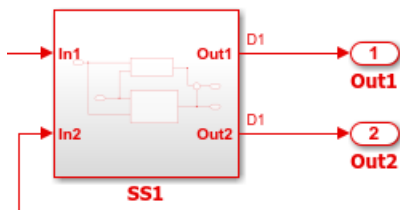
The internal behavior of an AUTOSAR software component is implemented by a set of runnable entities (runnables). A runnable is a sequence of operations provided by the component that can be started by the run-time environment (RTE). The component configures an event to activate each runnable - for example, a timing event, data received, a client request, a mode change, component startup or shutdown, or a trigger.

In Simulink, you can configure these types of AUTOSAR events.

Event Type	Workflow	Example
DataReceivedEvent	Sender-receiver (S-R) communication	“Configure Events for Runnable Activation” on page 4-307
DataReceiveErrorEvent	Sender-receiver (S-R) communication	“Configure AUTOSAR Receiver Port for DataReceiveErrorEvent” on page 4-106
ExternalTrigger-OccurredEvent	External trigger event communication	“Configure Receiver for AUTOSAR External Trigger Event Communication” on page 4-175
InitEvent	Activation of initialization runnable	“Configure AUTOSAR Initialization Runnable (R4.1)” on page 4-196
ModeSwitchEvent	Mode-switch (M-S) communication	“Configure AUTOSAR Mode-Switch Communication” on page 4-163
OperationInvokedEvent	Client-server (C-S) communication	“Configure AUTOSAR Client-Server Communication” on page 4-142
TimingEvent	Periodic activation of runnable	“Configure AUTOSAR TimingEvent for Periodic Runnable” on page 4-304

To configure an AUTOSAR runnable in Simulink:

- 1 Open a model that is configured for AUTOSAR code generation. This example uses a writable copy of the example model `autosar_sw_c`.
- 2 In the model, create or identify a root-level Simulink subsystem or function that implements a sequence of operations. The subsystem or function must generate an entry-point function in C code. In `autosar_sw_c`, the subsystem SS1 generates rate-based model step function `Runnable_1s`.



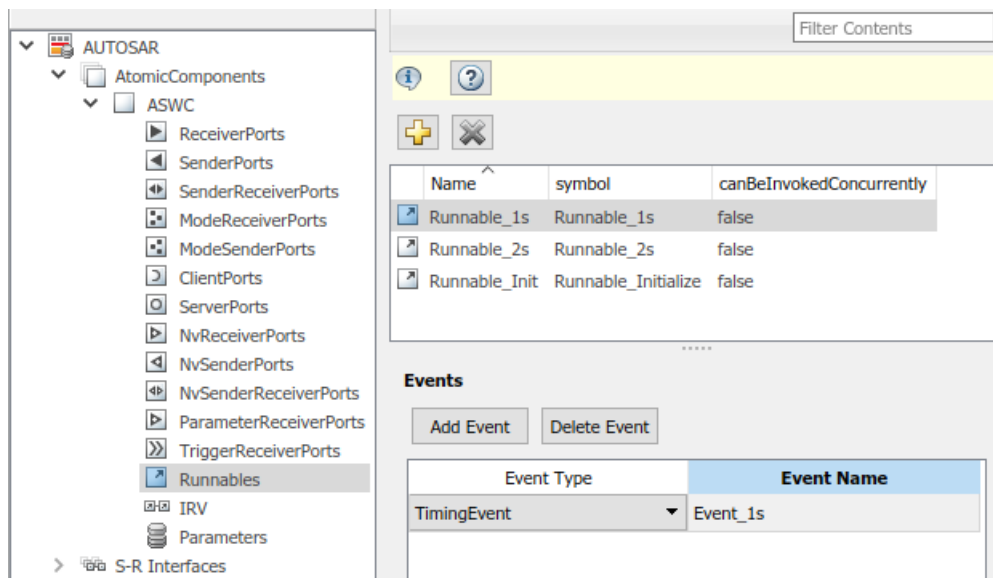
- 3 Create or identify an AUTOSAR runnable to which to map the Simulink entry point function. Open the AUTOSAR Dictionary. Expand **AtomicComponents**, expand the component, and select the **Runnables** view. If you need to create a new AUTOSAR runnable, click the plus sign. The model `autosar_sw_c` contains the periodic runnable `Runnable_1s`.
- 4 Select the row containing the runnable and configure its properties, including name and symbol. The AUTOSAR runnable symbol-name that you specify is exported in ARXML descriptions and C

code. For an AUTOSAR server runnable, set the runnable property `canBeInvokedConcurrently` to designate whether to enforce concurrency constraints. For nonservice runnables, leave `canBeInvokedConcurrently` set to `false`. For more information, see “Concurrency Constraints for AUTOSAR Server Runnables” on page 4-159.

- 5 Configure an event to activate the runnable. Go to the **Events** pane for the selected runnable. If you need to create an event, click **Add Event**. Enter an event name and set the event type.

The steps to configure an event depend on the type of event. If the event relies on a communication interface, such as data received (sender-receiver) or client request (client-server), you must first configure the communication interface before configuring the event.

In the model `autosar_sw`, the periodic runnable `Runnable_1s` is activated by a `TimingEvent` named `Event_1s`.



- 6 Map the Simulink entry-point function to the AUTOSAR runnable. Open the Code Mappings editor and select the **Functions** tab. For model `autosar_sw`, select the periodic function with a 1s sample time and map it to AUTOSAR runnable `Runnable_1s`.



To see the results of AUTOSAR runnable and event configuration in ARXML descriptions and C code, build the model.

If an AUTOSAR software component model contains multiple runnables, you can configure the order in which runnables execute. For more information, see “Configure AUTOSAR Runnable Execution Order” on page 4-181.

See Also

Related Examples

- “Configure AUTOSAR Runnable Execution Order” on page 4-181
- “Import AUTOSAR XML Descriptions Into Simulink” on page 3-13
- “Modeling Patterns for AUTOSAR Runnables” on page 2-10
- “Model AUTOSAR Software Components” on page 2-3
- “Configure AUTOSAR Code Generation” on page 5-7

More About

- “AUTOSAR Component Configuration” on page 4-3

Configure AUTOSAR Runnable Execution Order

For AUTOSAR Classic Platform software components that contain multiple runnables, the AUTOSAR Timing Extensions specification defines execution order constraints. These constraints specify the execution order of runnable entities within a component. You can view and manipulate the constraints at the component level or, in AUTOSAR architecture models, at the Virtual Function Bus (VFB) level.

In Simulink, you can:

- Import component- and VFB-level execution order constraints from ARXML files.
- Open an AUTOSAR component or architecture model and use the Schedule Editor to modify the execution order of runnables.
- Export component- and VFB-level execution order constraints to ARXML files.
- In a component model, update execution order constraints by importing ARXML changes.

In an AUTOSAR software component model, use the Schedule Editor to schedule and specify the execution order of the runnables belonging to that component. The Schedule Editor displays partitions in a model, the data connections between them, and the order of those partitions. In AUTOSAR component models, partitions correspond to runnable entities that execute independently. In the editor, you can:

- View a graphical representation of runnables as partitions in an AUTOSAR component.
- Create partitions and map them to AUTOSAR runnables.
- Directly specify the execution order of runnables.

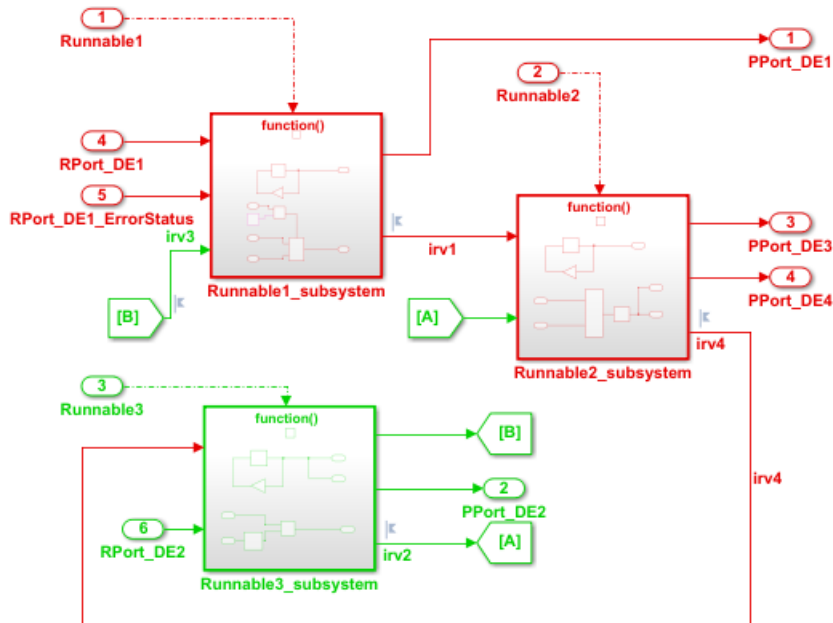
The Schedule Editor supports multiple modeling styles, including rate-based and export-function modeling. For more information, see “Using the Schedule Editor” and “Create Partitions”. You can also use the Schedule Editor in AUTOSAR architecture modeling. See “Configure AUTOSAR Scheduling and Simulation” on page 8-38.

In a standalone AUTOSAR component model, to open the Schedule Editor, open the **Modeling** tab and select **Schedule Editor**. For the runnables in an AUTOSAR component model, the Schedule Editor initially displays implicit partitions, created based on the component modeling style. You can view and configure the implicit partitions or create explicit partitions and map them to new or existing AUTOSAR runnables.

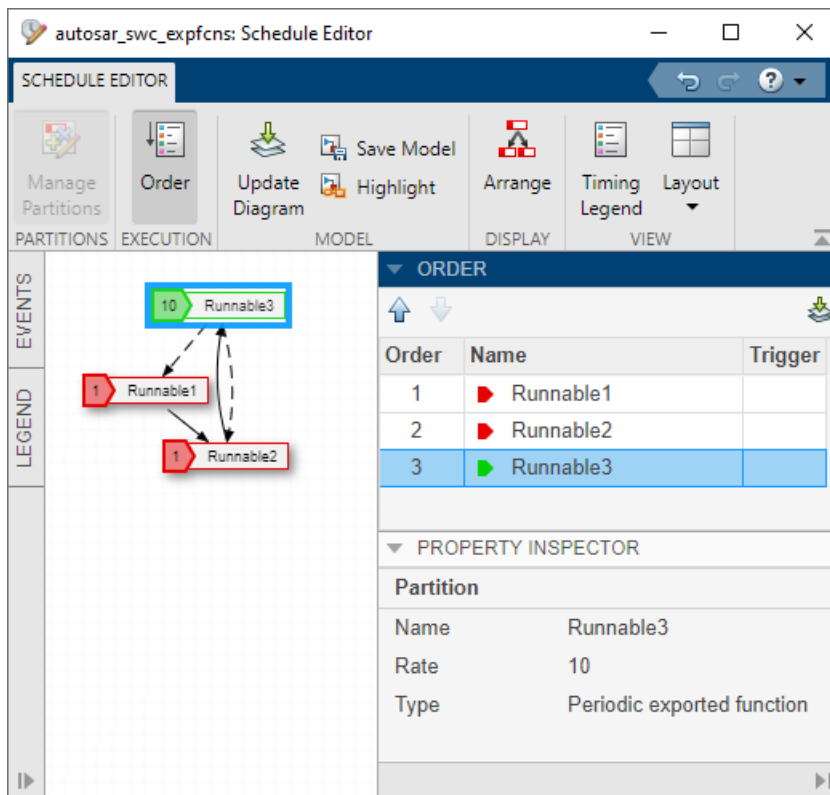
To view and configure implicit partitions:

- 1 Open AUTOSAR example model `autosar_swc_expfcns`, which uses Simulink exported functions to model AUTOSAR runnables.

```
openExample('autosar_swc_expfcns');
```



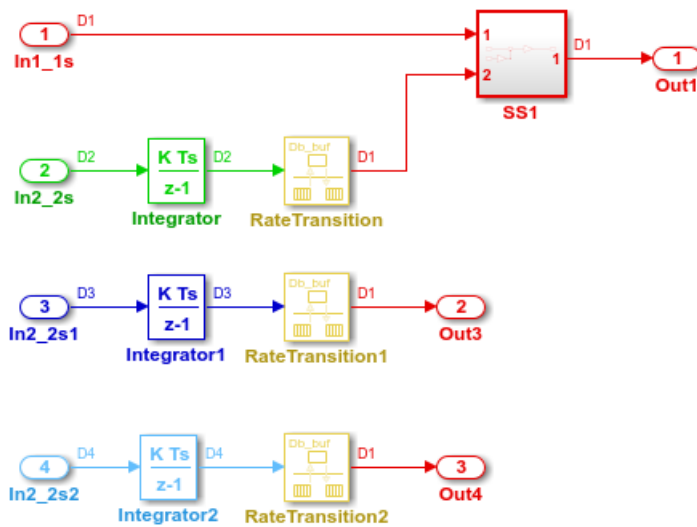
- Open the **Modeling** tab and select **Schedule Editor**. The Schedule Editor displays the periodic exported functions, which map to AUTOSAR runnables, as implicit partitions.



Use the editor controls to reorder the partitions. For example, in the Order section, click directional arrows or drag table entries.

To create an explicit partition in an AUTOSAR software component:

- 1 In AUTOSAR example model `mAutosarMultitasking.slx`, periodic runnables that have multiple sample rates are modeled.

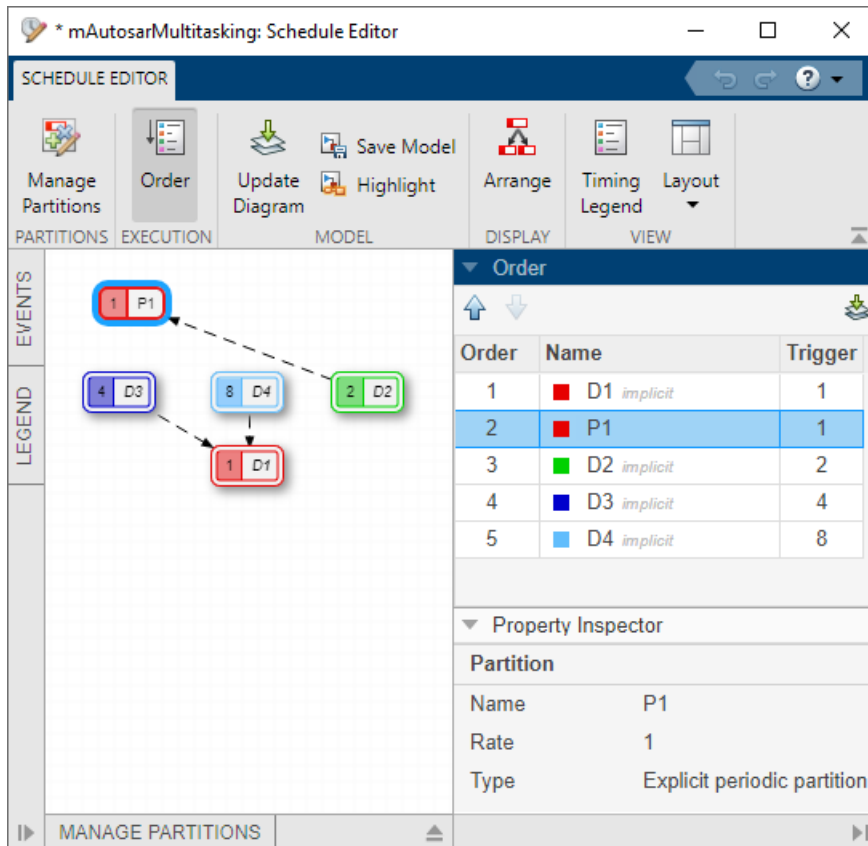



Initially, from a Schedule Editor perspective, the model contains implicit partitions D1, D2, D3, and D4.

Order	Name	Trigger
1	D1 <i>implicit</i>	1
2	D2 <i>implicit</i>	2
3	D3 <i>implicit</i>	4
4	D4 <i>implicit</i>	8

Property Inspector	
Partition	
Name	D1
Rate	1
Type	Implicit periodic partition

- 2 To create a partition, open the block parameters dialog box for the SS1 subsystem. With **Treat as atomic unit** selected, set parameter **Schedule as** to **Periodic partition**. Specify a partition name, such as P1, and sample time 1. Click **Apply**. Update the model diagram.
- 3 Open the **Modeling** tab and select **Schedule Editor**. The Schedule Editor displays the explicit periodic partition in the model.



- 4 In the model window, open the Code Mappings editor and select the **Functions** tab. Map the P1 partition function to an AUTOSAR runnable.
 - a If the configuration does not contain an AUTOSAR runnable to map, add a runnable. Open the AUTOSAR Dictionary, Runnables view, and click the **Add** button . For this example, create runnable `Runnable_P1`. Then select the runnable and create a timing event.
 - b In the **Functions** tab, map P1 to `Runnable_P1`.

Code Mappings - AUTOSAR SW Component							
Functions	Inports	Outputs	Parameters	Data Stores	Signals/States	Data Transfers	Function Callers
^		Runnable					
fx Initialize							Runnable_Init
fx Partition:P1							Runnable_P1
fx Periodic:D1 [Sample Time: 1s]							Runnable_Step
fx Periodic:D2 [Sample Time: 2s]							Runnable_Step1
fx Periodic:D3 [Sample Time: 4s]							Runnable_Step2
fx Periodic:D4 [Sample Time: 8s]							Runnable_Step3

Building an AUTOSAR model that contains execution order constraints exports component timing information. If you set the AUTOSAR Dictionary XML option **Exported XML File Packaging** to **Modular**, the timing information is exported into the file `modelname_timing.arxml`. This ARXML code shows the execution order constraint exported for the runnables in `mAutosarMultitasking`, based on the Schedule Editor configuration.

```
<SWC-TIMING UUID="...">
  <SHORT-NAME>mAutosarMultitasking</SHORT-NAME>
  <TIMING-REQUIREMENTS>
    <EXECUTION-ORDER-CONSTRAINT UUID="...">
      <SHORT-NAME>EOC</SHORT-NAME>
      <ORDERED-ELEMENTS>
        <EOC-EXECUTABLE-ENTITY-REF UUID="...">
          <SHORT-NAME>Runnable_Step</SHORT-NAME>
          <EXECUTABLE-REF DEST="RUNNABLE-ENTITY">
            /pkg/swc/mAutosarMultitasking/IB/Runnable_Step
          </EXECUTABLE-REF>
          <SUCCESSOR-REFS>
            <SUCCESSOR-REF DEST="EOC-EXECUTABLE-ENTITY-REF">
              /Timing/mAutosarMultitasking/EOC/Runnable_P1
            </SUCCESSOR-REF>
          </SUCCESSOR-REFS>
        </EOC-EXECUTABLE-ENTITY-REF>
        <EOC-EXECUTABLE-ENTITY-REF UUID="...">
          <SHORT-NAME>Runnable_P1</SHORT-NAME>
          <EXECUTABLE-REF DEST="RUNNABLE-ENTITY">
            /pkg/swc/mAutosarMultitasking/IB/Runnable_P1
          </EXECUTABLE-REF>
          <SUCCESSOR-REFS>
            <SUCCESSOR-REF DEST="EOC-EXECUTABLE-ENTITY-REF">
              /Timing/mAutosarMultitasking/EOC/Runnable_Step1
            </SUCCESSOR-REF>
          </SUCCESSOR-REFS>
        </EOC-EXECUTABLE-ENTITY-REF>
        <EOC-EXECUTABLE-ENTITY-REF UUID="...">
          <SHORT-NAME>Runnable_Step1</SHORT-NAME>
          <EXECUTABLE-REF DEST="RUNNABLE-ENTITY">
            /pkg/swc/mAutosarMultitasking/IB/Runnable_Step1
          </EXECUTABLE-REF>
          <SUCCESSOR-REFS>
            <SUCCESSOR-REF DEST="EOC-EXECUTABLE-ENTITY-REF">
              /Timing/mAutosarMultitasking/EOC/Runnable_Step2
            </SUCCESSOR-REF>
          </SUCCESSOR-REFS>
        </EOC-EXECUTABLE-ENTITY-REF>
        <EOC-EXECUTABLE-ENTITY-REF UUID="...">
          <SHORT-NAME>Runnable_Step2</SHORT-NAME>
          <EXECUTABLE-REF DEST="RUNNABLE-ENTITY">
            /pkg/swc/mAutosarMultitasking/IB/Runnable_Step2
          </EXECUTABLE-REF>
          <SUCCESSOR-REFS>
```

```
        <SUCCESSOR-REF DEST="EOC-EXECUTABLE-ENTITY-REF">
          /Timing/mAutosarMultitasking/EOC/Runnable_Step3
        </SUCCESSOR-REF>
      </SUCCESSOR-REFS>
    </EOC-EXECUTABLE-ENTITY-REF>
  <EOC-EXECUTABLE-ENTITY-REF UUID="...">
    <SHORT-NAME>Runnable_Step3</SHORT-NAME>
    <EXECUTABLE-REF DEST="RUNNABLE-ENTITY">
      /pkg/swc/mAutosarMultitasking/IB/Runnable_Step3
    </EXECUTABLE-REF>
  </EOC-EXECUTABLE-ENTITY-REF>
</ORDERED-ELEMENTS>
</EXECUTION-ORDER-CONSTRAINT>
</TIMING-REQUIREMENTS>
<BEHAVIOR-REF DEST="SWC-INTERNAL-BEHAVIOR">
  /pkg/swc/mAutosarMultitasking/IB
</BEHAVIOR-REF>
</SWC-TIMING>
```

See Also

Schedule Editor

Related Examples

- “Configure AUTOSAR Runnables and Events” on page 4-178
- “Using the Schedule Editor”
- “Create Partitions”
- “Configure AUTOSAR Scheduling and Simulation” on page 8-38

More About

- “Modeling Patterns for AUTOSAR Runnables” on page 2-10
- “Model AUTOSAR Runnables Using Exported Functions” on page 2-18

Configure AUTOSAR Initialize, Reset, or Terminate Runnables

AUTOSAR applications sometimes require complex logic to execute during system initialization, reset, and termination sequences. To model startup, reset, and shutdown processing in an AUTOSAR software component, use the Simulink blocks Initialize Function and Terminate Function.

The Initialize Function and Terminate Function blocks can control execution of a component in response to initialize, reset, or terminate events. For more information, see “Using Initialize, Reinitialize, Reset, and Terminate Functions”, and “Startup, Reset, and Shutdown” on page 2-8.

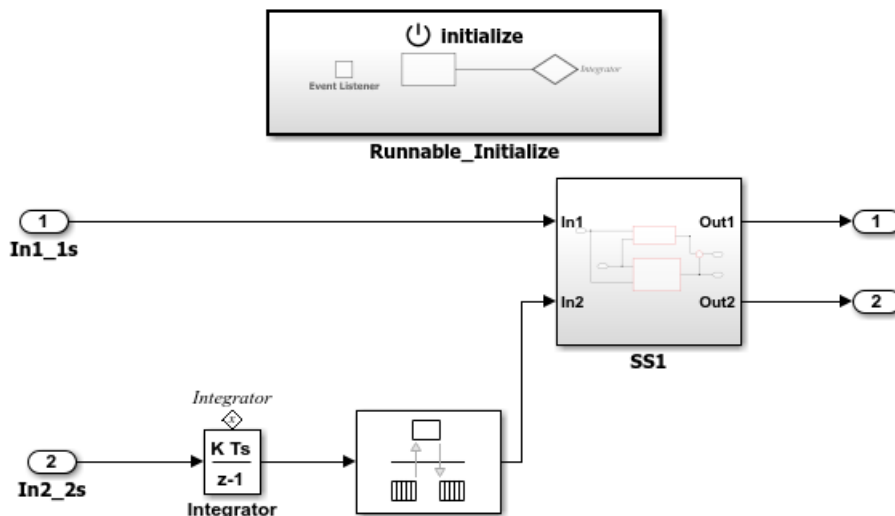
In an AUTOSAR model, you map each Simulink initialize, reset, or terminate entry-point function to an AUTOSAR runnable. For each runnable, configure the AUTOSAR event that activates the runnable. In general, you can select any AUTOSAR event type except `TimingEvent`. The runnables work with any AUTOSAR component modeling style. (However, software-in-the-loop simulation of AUTOSAR initialize, reset, or terminate runnables works only with exported function modeling.)

This example shows how to configure an AUTOSAR software component for simple startup and termination processing, using the Initialize Function and Terminate Function blocks.

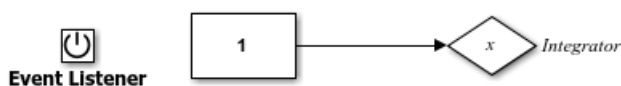
- 1 Open a model that is configured for AUTOSAR code generation. This example uses a writable copy of the example model `autosar_sw_c`.

```
openExample("autosar_sw_c");
```

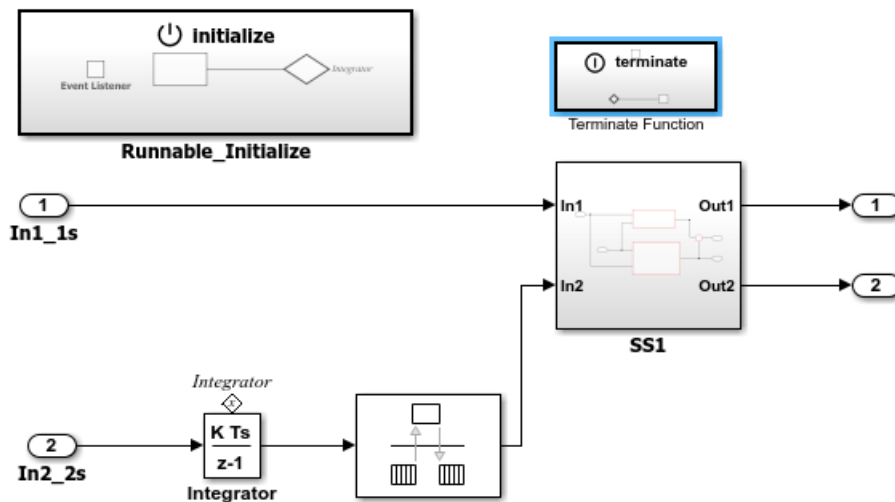
Add an Initialize Function block to the model.



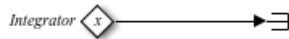
- 2 In the Initialize Function block, develop the logic that is required to execute during component initialization, using the techniques described in “Using Initialize, Reinitialize, Reset, and Terminate Functions”.



- 3 Add a Terminate Function block to the model.



- 4 In the Terminate Function block, develop the logic that is required to execute during component termination, using the techniques described in “Using Initialize, Reinitialize, Reset, and Terminate Functions”.

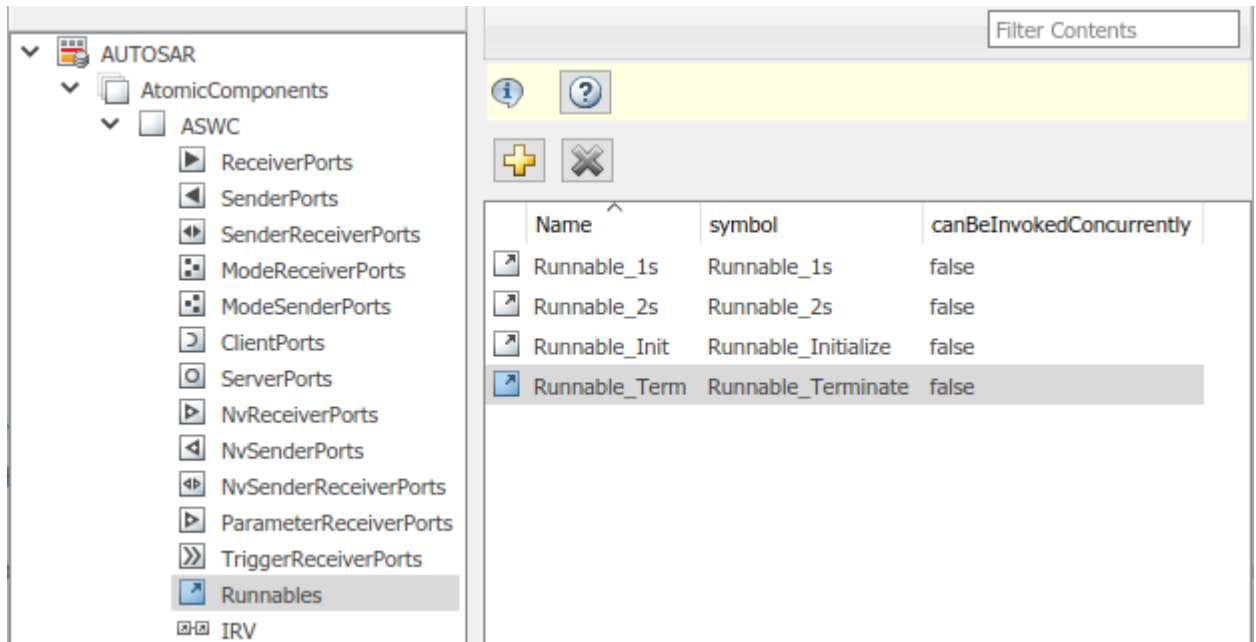


In this example, the Terminator block is a placeholder for saving the state value.

- 5 Add a terminate entry-point function to the model. In the Configuration Parameters dialog box, in the **Code Generation > Interface** pane, under **Advanced parameters**, select the option **Terminate function required**. Click **Apply**.
- 6 Open the Code Mappings editor. To update the Simulink to AUTOSAR mapping of the model, click the **Update** button . The mapping now reflects the addition of the Initialize Function and Terminate Function blocks and enabling of a terminate entry-point function.
- 7 Open the AUTOSAR Dictionary. Expand **AtomicComponents**, expand the component, and select the **Runnables** view.

The runnables list already contains an initialization runnable, created as part of the initial Simulink representation of the AUTOSAR software component. Use the **Add** button to add a terminate runnable to the component. Select each runnable and configure its name and properties.

The runnable **symbol** value shown in the runnables view becomes the runnable function name. The runnable **Name** value is used in the names of RTE access methods generated for the runnable.



- 8 For both the initialize and terminate runnables, configure an AUTOSAR event that activates the runnable.

This example defines a `ModeSwitchEvent` for each runnable. Using a `ModeSwitchEvent` requires creating a model declaration group, a mode-switch (M-S) interface, and a mode receiver port for the model. For more information, see “Configure AUTOSAR Mode-Switch Communication” on page 4-163.

In the runnables view, click the initialize runnable name to display and modify its associated event properties. Add and configure an event.

The screenshot displays two parts of the software interface. The top part is a table with three columns: 'Name', 'symbol', and 'canBeInvokedConcurrently'. It lists four runnables: Runnable_1s, Runnable_2s, Runnable_Init, and Runnable_Term. The 'Runnable_Init' row is selected. The bottom part is the 'Events' configuration panel, which includes 'Add Event' and 'Delete Event' buttons, a table for event configuration, and 'Event Properties' for the selected event.

Name	symbol	canBeInvokedConcurrently
Runnable_1s	Runnable_1s	false
Runnable_2s	Runnable_2s	false
Runnable_Init	Runnable_Initialize	false
Runnable_Term	Runnable_Terminate	false

Events

Add Event Delete Event

Event Type	Event Name
ModeSwitchEvent	EventInit

Event Properties

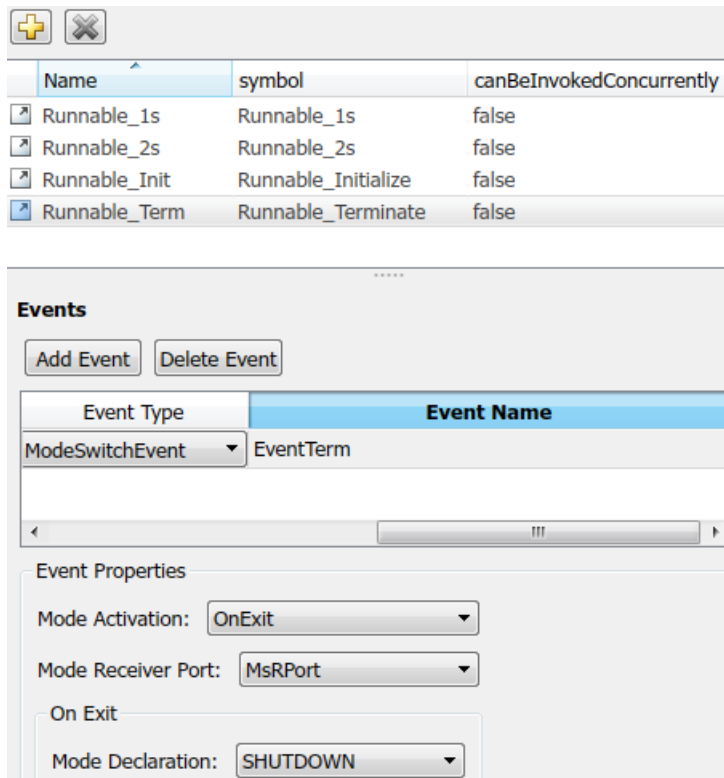
Mode Activation: OnEntry

Mode Receiver Port: MsRPort

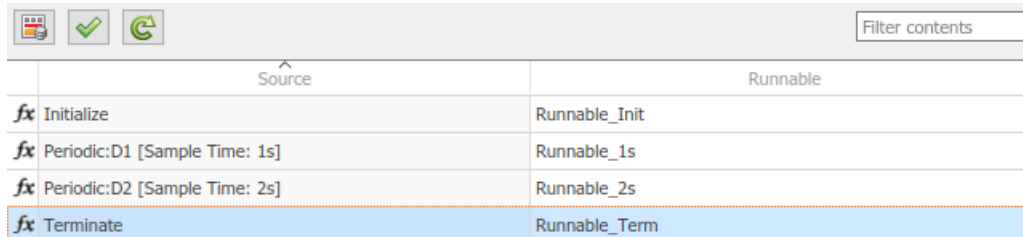
On Entry

Mode Declaration: STARTUP

In the runnables view, click the terminate runnable name to display and modify its associated event properties. Add and configure an event.



- 9 Open the Code Mappings editor and select the **Functions** tab. Select the Simulink initialize and terminate functions and map them to the AUTOSAR initialize and terminate runnables that you configured.



- 10 Build the model and examine the generated code.
- The exported ARXML code contains an AUTOSAR runnable for each initialize, reset, or terminate subsystem in the model, with the specified AUTOSAR runnable name and symbol. The runnable description includes each AUTOSAR data access point and server call point associated with the runnable.
 - The generated C code contains RTE access methods for parameters, states, function callers, and external I/O associated with the runnable.

See Also

Initialize Function | Terminate Function | Event Listener | State Writer | State Reader

Related Examples

- “Using Initialize, Reinitialize, Reset, and Terminate Functions”
- “Create Test Harness to Generate Function Calls”
- “Configure AUTOSAR Mode-Switch Communication” on page 4-163

More About

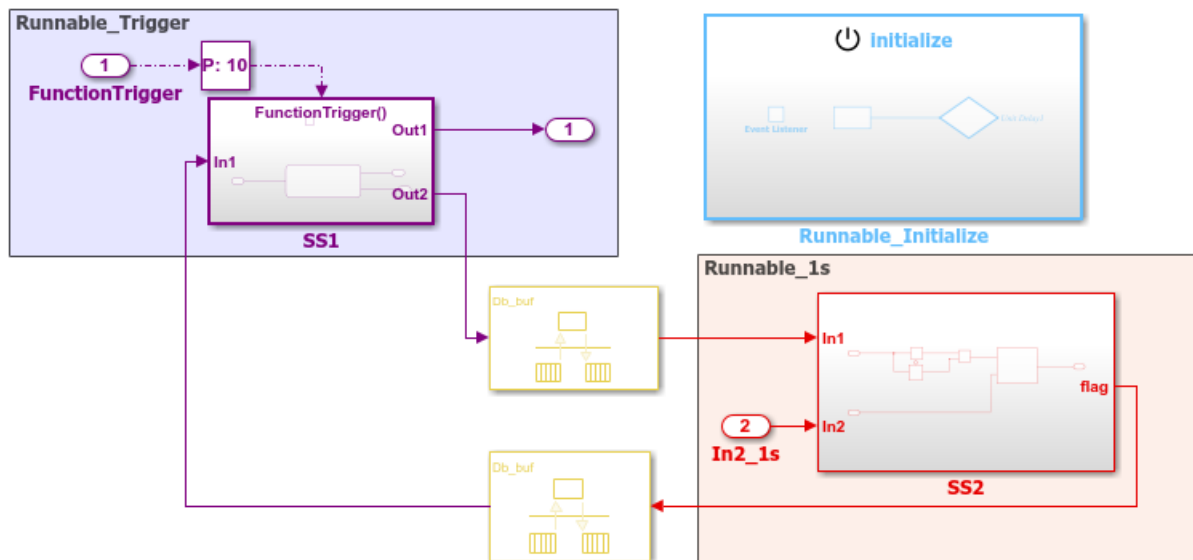
- “Startup, Reset, and Shutdown” on page 2-8
- “Configure Generated C Function Interface for Model Entry-Point Functions” (Embedded Coder)
- “AUTOSAR Component Configuration” on page 4-3

Add Top-Level Asynchronous Trigger to Periodic Rate-Based System

In Simulink, you can model an AUTOSAR software component in which an asynchronous function-call runnable interacts with periodic rate-based runnables. This type of component uses both periodic and asynchronous rates (sample times).

The approach can be used to model the JMAAB complex control model type beta (β) architecture. This architecture is described in the document *Control Algorithm Modeling Guidelines Using MATLAB, Simulink, and Stateflow*, which is available from the MathWorks® website at <https://www.mathworks.com/solutions/mab-guidelines.html>.

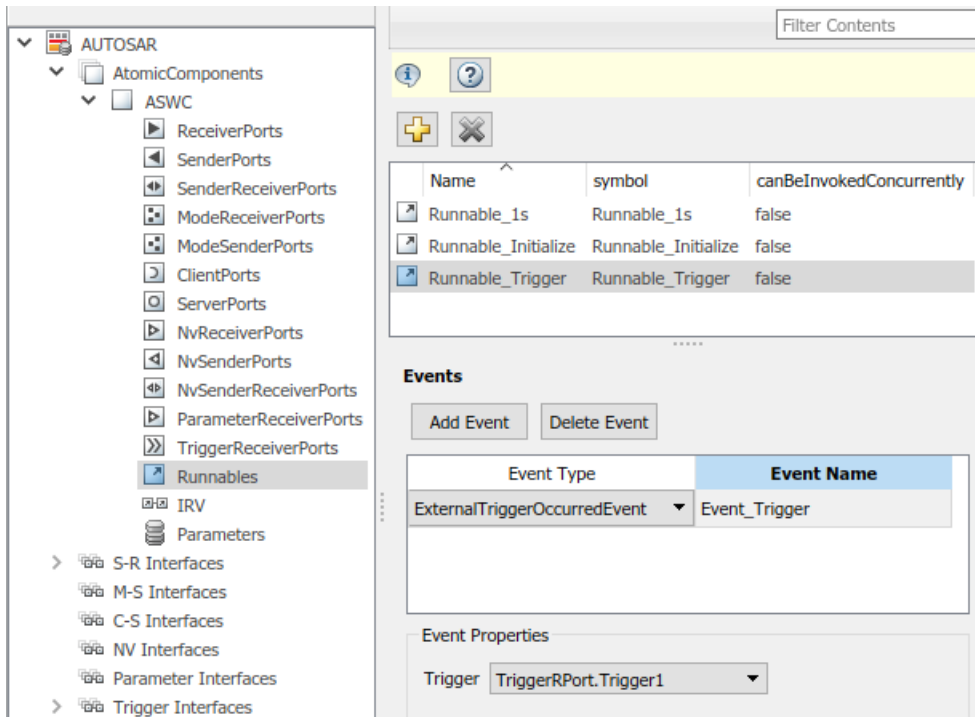
In JMAAB type beta modeling, at the top level of a control model, you place function layers above scheduling layers. For example, here is an AUTOSAR example model, `autosar_sw_cncalls`. In this model, an asynchronous function-call runnable at the top level of the model interacts with a periodic rate-based runnable.



Some guidelines apply to AUTOSAR modeling of the JMAAB type beta controller layout:

- IRVs must be modeled with Rate Transition blocks.
- Function-call subsystems must have asynchronous rates. (In the function-call subsystem Trigger block, **Sample time type** must be triggered, not periodic.)
- For each asynchronous function-call subsystem, you must insert an Asynchronous Task Specification task block between the function-call root inport and the subsystem.

Here is the AUTOSAR Dictionary view of the runnables. An event triggers the asynchronous function-call runnable. The event must be of type `DataReceivedEvent`, `DataReceiveErrorEvent`, `ModeSwitchEvent`, `InitEvent`, or `ExternalTriggerOccurredEvent`.



In this example, an `ExternalTriggerOccurredEvent` activates the AUTOSAR runnable. A trigger interface delivers the event to a trigger receiver port. For more information about `ExternalTriggerOccurredEvents`, see “Configure Receiver for AUTOSAR External Trigger Event Communication” on page 4-175.

Here is the Code Mappings editor view of the Simulink entry-point functions. The functions are mapped to AUTOSAR function-trigger, initialization, and periodic runnables, respectively.

Source	Runnable
fx Exported Function:FunctionTrigger	Runnable_Trigger
fx Initialize	Runnable_Initialize
fx Periodic:D1 [Sample Time: 1s]	Runnable_1s

See Also

Rate Transition | Asynchronous Task Specification

Related Examples

- “Configure Receiver for AUTOSAR External Trigger Event Communication” on page 4-175
- “Modeling Patterns for AUTOSAR Runnables” on page 2-10
- “Configure AUTOSAR Code Generation” on page 5-7

More About

- “Model AUTOSAR Software Components” on page 2-3

- “AUTOSAR Component Configuration” on page 4-3

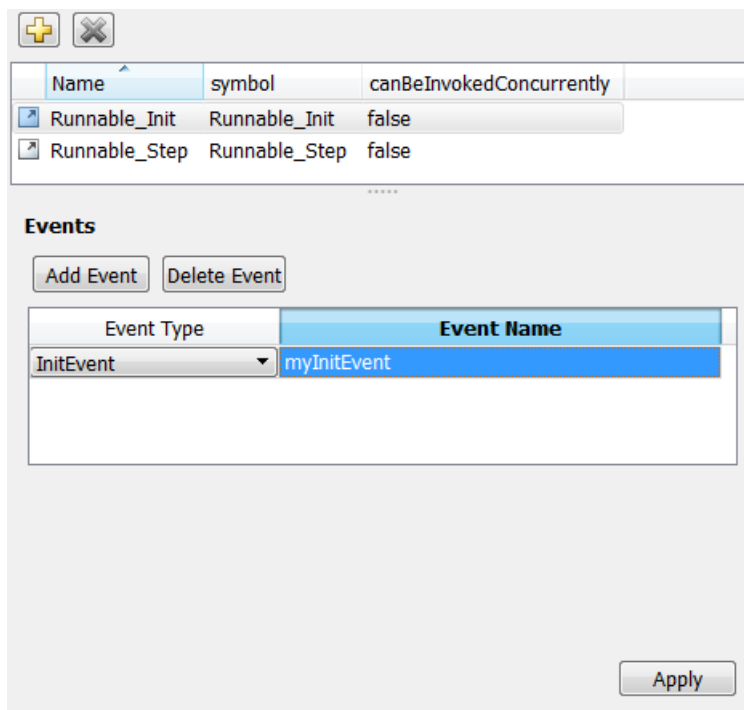
Configure AUTOSAR Initialization Runnable (R4.1)

AUTOSAR Release 4.1 introduced the AUTOSAR initialization event (`InitEvent`). You can use an `InitEvent` to designate an AUTOSAR runnable as an initialization runnable, and then map an initialization function to the runnable. Using an `InitEvent` to initialize a software component is a potentially simpler and more efficient than using AUTOSAR mode management, in which you define a `ModeDeclarationGroup` with a mode for setting up and initializing a software component. (For more information on the mode management approach, see “Configure AUTOSAR Mode-Switch Communication” on page 4-163.)

If you import ARXML code that describes a runnable with an `InitEvent`, the ARXML importer configures the runnable in Simulink as an initialization runnable.

Alternatively, you can configure a runnable to be the initialization runnable in Simulink. For example,

- 1 Open a model configured for AUTOSAR.
- 2 Open the Configuration Parameters dialog box, go to **Code Generation > AUTOSAR Code Generation Options**, and verify that the selected AUTOSAR schema version is 4.1 or higher.
- 3 Open the AUTOSAR Dictionary. Navigate to a software component, and select the **Runnables** view.
- 4 Select a runnable to configure as an initialization runnable, and click **Add Event**. From the **Event Type** drop-down list, select `InitEvent`, and specify the **Event Name**. In this example, initialization event `myInitEvent` is configured for runnable `Runnable_Init`.



You can configure at most one `InitEvent` for a runnable.

- 5 Open the Code Mappings editor and select the **Functions** tab.
- 6 To map an initialization function to the initialization runnable, select the function. From the **Runnable** drop-down list, select the runnable for which you configured an `InitEvent`. In this

example, Simulink entry-point function `Initialize` is mapped to AUTOSAR runnable `Runnable_Init`.

	Source	Runnable
fx	Initialize	Runnable_Init
fx	Periodic:D1 [Sample Time: 0.1s]	Runnable_Step

When you export ARXML code from a model containing an initialization runnable, the ARXML exporter generates an `InitEvent` that is mapped to the initialization runnable and function. For example:

```
<EVENTS>
  <INIT-EVENT UUID="...">
    <SHORT-NAME>myInitEvent</SHORT-NAME>
    <START-ON-EVENT-REF DEST="RUNNABLE-ENTITY"/.../Runnable_Init</START-ON-EVENT-REF>
  </INIT-EVENT>
</EVENTS>
```

See Also

Related Examples

- “Configure AUTOSAR Runnables and Events” on page 4-178
- “Configure AUTOSAR Runnables” on page 4-21

Configure Disabled Mode for AUTOSAR Runnable Event

In an AUTOSAR software component model, you can set the `DisabledMode` property of a runnable event to potentially prevent a runnable from running in certain modes.

Given a model containing a mode receiver port and defined mode values, you can programmatically get and set the `DisabledMode` property of a `TimingEvent`, `DataReceivedEvent`, `ModeSwitchEvent`, `OperationInvokedEvent`, `DataReceiveErrorEvent`, or `ExternalTriggerOccurredEvent`. The property is not supported for an `InitEvent`.

The value of the `DisabledMode` property is either `''` (no disabled modes) or one or more mode values of the form `'mrPortName.modeName'`. To set the `DisabledMode` property of a runnable event in your model, use the AUTOSAR property function `set`.

The following example sets the `DisabledMode` property for a timing event named `Event_t_1tic_B`. The `set` function call potentially disables the event for modes `STARTUP` and `SHUTDOWN`, which are defined on mode-receiver port `myMRPort`.

```
hModel = 'mAutosarMsConfigAfter';
open_system(hModel)
arProps = autosar.api.getAUTOSARProperties(hModel);
eventPaths = find(arProps,[], 'TimingEvent')

eventPaths = 1x2 cell
    {'ASWC/Behavior/Event_t_1tic_B'}    {'ASWC/Behavior/Event_t_10tic'}

dsblModes = get(arProps,eventPaths{1}, 'DisabledMode')

dsblModes =

    1x0 empty cell array

set(arProps,eventPaths{1}, 'DisabledMode', {'myMRPort.STARTUP', 'myMRPort.SHUTDOWN'});
dsblModes = get(arProps,eventPaths{1}, 'DisabledMode')

dsblModes = 1x2 cell
    {'myMRPort.STARTUP'}    {'myMRPort.SHUTDOWN'}
```

When you export ARXML files for the model, the timing event description for `Event_t_1tic_B` includes a `DISABLED-MODE-IREFS` section that references the mode-receiver port, the mode declaration group, and each disabled mode.

The software preserves the `DisabledMode` property of a runnable event across round trips between an AUTOSAR authoring tool (AAT) and Simulink®.

See Also

Related Examples

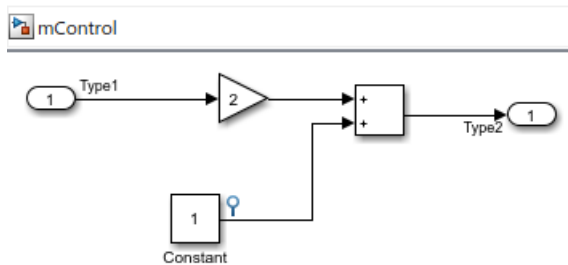
- “Configure AUTOSAR Runnables and Events” on page 4-178
- “Configure AUTOSAR Mode-Switch Communication” on page 4-163

Configure Internal Data Types for AUTOSAR IncludedDataTypeSets

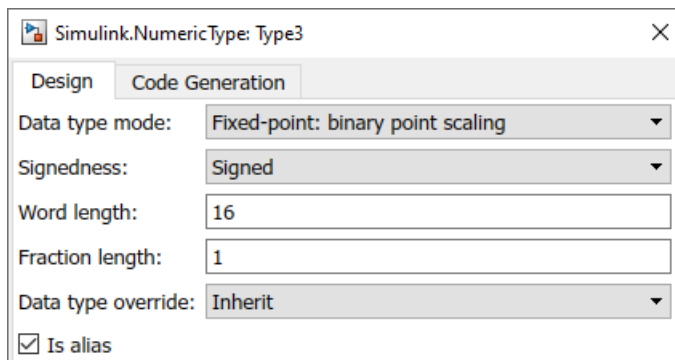
In an AUTOSAR software component model, you can import and export an ARXML description of an AUTOSAR included data type set (IncludedDataTypeSet). An IncludedDataTypeSet is defined as part of the internal behavior of a software component. It contains references to AUTOSAR data type definitions that are internal to a component and not present in the component interface descriptions. The referenced internal data type definitions can be shared among multiple software components, as described in “Import and Reference Shared AUTOSAR Element Definitions” on page 3-29. For information about IncludedDataTypeSet workflows, see “Included Data Type Sets” on page 2-32.

In Simulink, you can configure an AUTOSAR internal data type to be exported in an ARXML IncludedDataTypeSet and generated in a C header file. In your AUTOSAR component model, create a data type object, use it to describe the internal data type, and map the data type to header file Rte_Type.h. For example:

- 1 Create or open an AUTOSAR software component model in which blocks are used internally and are not part of the component model interface. For example, here is a model in which a Constant block is not connected to a model import or output.



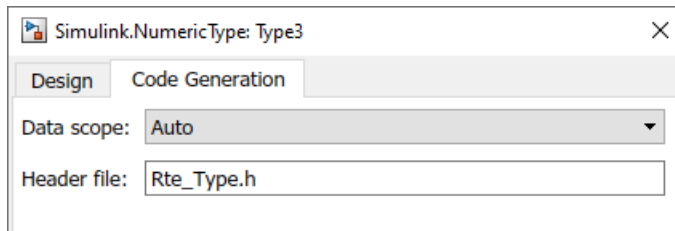
- 2 Create a data type object, Type3, in the base workspace or in a data dictionary. Select the **Is alias** check box.



For AUTOSAR IncludedDataTypeSet export, Simulink supports these data types:

- Numeric
- Alias
- Bus

- Fixed-point
 - Enumerated
- 3 Map the Type3 data type to header file Rte_Type.h. In the data type object dialog box, **Code Generation** tab, set **Header file** to Rte_Type.h. Click **Apply**.



- 4 To reference Type3 in the model, enter Type3 in the Constant block parameter field **Output data type**.

Type3 is internal to the component and is not exported in component interface descriptions. Because you mapped it to header file Rte_Type.h, it is exported in an IncludedDataSet description.

- 5 Build the model. In the exported ARXML, inside the description of the component internal behavior, an IncludedDataSet description references the internal data type.

```
<INCLUDED-DATA-TYPE-SETS>
  <INCLUDED-DATA-TYPE-SET>
    <DATA-TYPE-REFS>
      <DATA-TYPE-REF DEST="APPLICATION-PRIMITIVE-DATA-TYPE">
        /Control_pkg/Control_dt/ImplDataTypes/Type3
      </DATA-TYPE-REF>
    </DATA-TYPE-REFS>
  </INCLUDED-DATA-TYPE-SET>
</INCLUDED-DATA-TYPE-SETS>
```

The generated Rte_Type.h header file contains an entry for the internal data type.

```
/* AUTOSAR Implementation data types, specific to software component */
typedef sint16 Type1;
typedef sint16 Type2;
typedef sint16 Type3;
typedef void* Rte_Instance;
```

See Also

Related Examples

- “Model AUTOSAR Component Behavior” on page 2-31
- “Import and Reference Shared AUTOSAR Element Definitions” on page 3-29

More About

- “Included Data Type Sets” on page 2-32

Configure AUTOSAR Per-Instance Memory


To model AUTOSAR per-instance memory (PIM) for AUTOSAR applications, you import per-instance memory definitions from ARXML files or create per-instance memory content in Simulink. For information about the high-level PIM workflow, see “Per-Instance Memory” on page 2-34.

AUTOSAR typed per-instance memory (`ArTypedPerInstanceMemory`) defines an AUTOSAR typed memory block that is available for each instance of an AUTOSAR software component. In the AUTOSAR run-time environment, calibration tools can access `arTypedPerInstanceMemory` blocks for calibration and measurement.

To model AUTOSAR PIM, you can use Simulink block signals, discrete states, or data stores in your model.

Configure Block Signals and States as AUTOSAR Typed Per-Instance Memory

To generate `arTypedPerInstanceMemory` blocks for Simulink block signal and discrete state data in your AUTOSAR model, open the Code Mappings editor and select the **Signals/States** tab. Select signals and states and map them to `arTypedPerInstanceMemory`. For example:

- 1 Open an AUTOSAR model that contains signals or states for which you want to generate `arTypedPerInstanceMemory` blocks. This example uses model `autosar_swc_counter`.
- 2 In the AUTOSAR Code perspective, open the Code Mappings editor and select the **Signals/States** tab. In the list of available signals, select `sum_out`. Selecting a signal highlights the signal in the model diagram. In the **Mapped To** drop-down list, select `ArTypedPerInstanceMemory`. To view and modify AUTOSAR attributes for the per-instance memory, click the  icon. For more information about signal code and calibration attributes, see “Map Block Signals and States to AUTOSAR Variables” on page 4-58. If you are mapping signals and states in a submodel referenced by a component, see “Map Submodel Signals and States to AUTOSAR Variables” on page 4-70.

This model maps Simulink model workspace parameters and signals to AUTOSAR parameters and variables for AUTOSAR run-time calibration and measurement.

Copyright 1994-2018 The MathWorks, Inc.

Source	Mapped To	Path
Signals (3)		
equal_to_count	StaticMemory	autosar_sw_counter
sum_out	ArTypedPerInstanceMemory	autosar_sw_counter
switch_out	Auto	autosar_sw_counter
States (1)		
X	StaticMemory	autosar_sw_counter

- In the **Signals/States** tab, from the list of available states, select state X. In the **Mapped To** drop-down list, select **ArTypedPerInstanceMemory**. To view and modify AUTOSAR attributes for the per-instance memory, click the icon.


When you generate code:

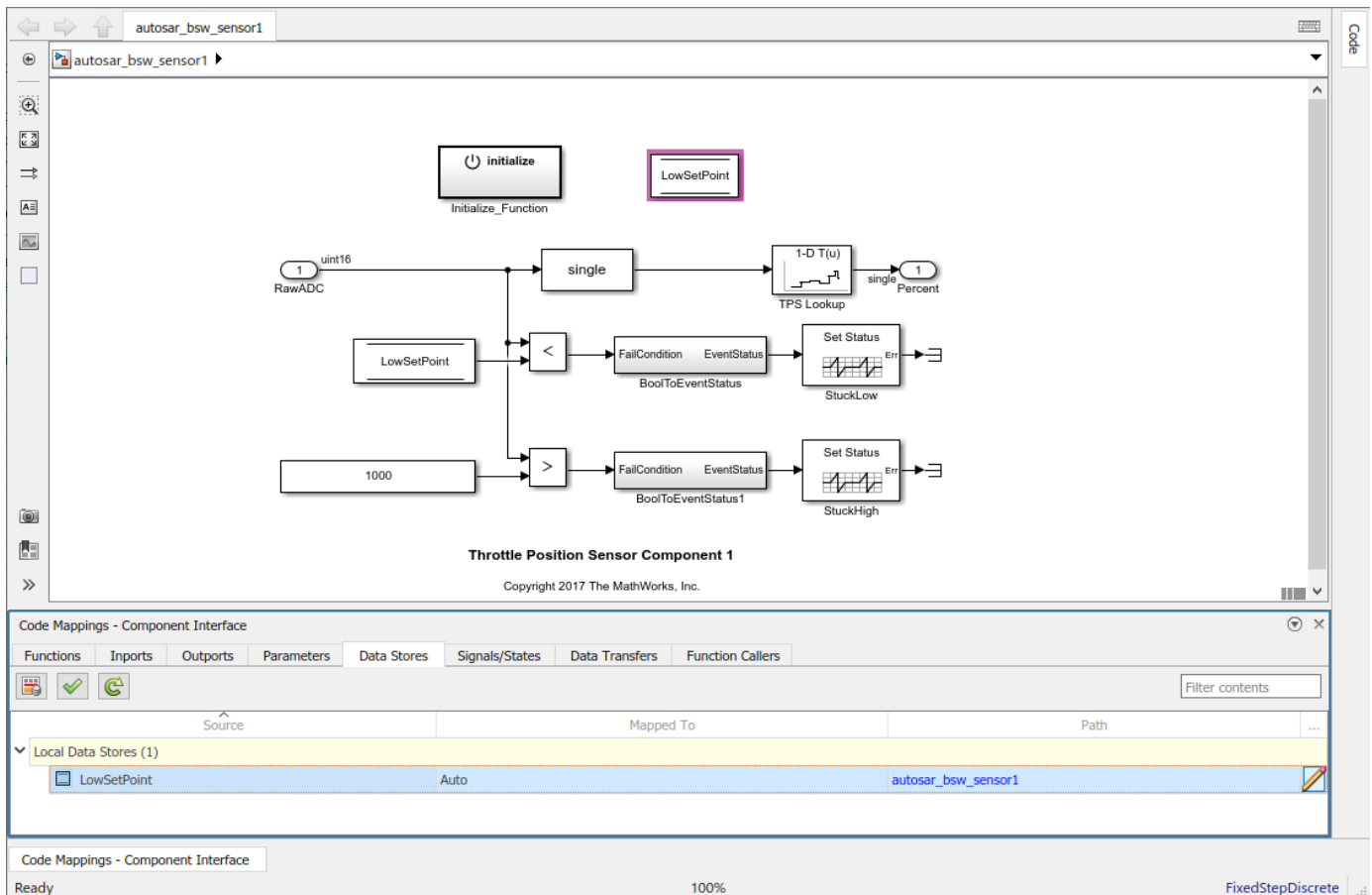
- Exported ARXML files contain AR-TYPED-PER-INSTANCE-MEMORYS descriptions for signals and states that you configured as **ArTypedPerInstanceMemory**.
- Generated C code contains `Rte_Pim_*` API calls for signal and state variables.

For referenced models within an AUTOSAR component model, Embedded Coder maps internal signals and states for model reference code generation. Internal signals and states map to AUTOSAR **ArTypedPerInstanceMemory** for multi-instance model reference or to AUTOSAR **StaticMemory** for single-instance model reference.

Configure Data Stores as AUTOSAR Typed Per-Instance Memory

To generate **ArTypedPerInstanceMemory** blocks for Simulink data store memory blocks in your AUTOSAR model, open the Code Mappings editor and select the **Data Stores** tab. Select data stores and map them to **ArTypedPerInstanceMemory**. For example:

- 1 Open an AUTOSAR model that contains data stores that you want to generate arTypedPerInstanceMemory blocks for. This example uses model `autosar_bsw_sensor1`.
- 2 In the AUTOSAR Code perspective, open the Code Mappings editor and select the **Data Stores** tab. In the list of available data stores, select data store `LowSetPoint`. Selecting a data store highlights the data store memory block in the model diagram. In the **Mapped To** drop-down list, select `ArTypedPerInstanceMemory`. To view and modify AUTOSAR attributes for the per-instance memory, click the  icon. For more information about data store code and calibration attributes, see “Map Data Stores to AUTOSAR Variables” on page 4-56. If you are mapping data stores in a submodel referenced by a component, see “Map Submodel Data Stores to AUTOSAR Variables” on page 4-69.



The screenshot displays the AUTOSAR Code perspective for the model `autosar_bsw_sensor1`. The main window shows a block diagram of the **Throttle Position Sensor Component 1**. Key elements include an `initialize` block, a `LowSetPoint` data store (highlighted with a purple border), a `single` block, a `1-D T(u)` block, and two `FailCondition` blocks. The `LowSetPoint` data store is connected to a `<` comparison block, which is connected to a `FailCondition` block. The `FailCondition` block is connected to a `Set Status` block. The `Set Status` block is connected to a `StuckLow` block. The `1000` block is connected to a `>` comparison block, which is connected to a `FailCondition` block. The `FailCondition` block is connected to a `Set Status` block. The `Set Status` block is connected to a `StuckHigh` block.

The **Code Mappings - Component Interface** window is open, showing the **Data Stores** tab. The `Local Data Stores (1)` list contains the `LowSetPoint` data store. The `Mapped To` column shows `Auto` and the `Path` column shows `autosar_bsw_sensor1`. A pencil icon is visible in the `Path` column, indicating that the mapping can be edited.

When you generate code:

- Exported ARXML files contain AR-TYPED-PER-INSTANCE-MEMORYS descriptions for data stores that you configured as `ArTypedPerInstanceMemory`.
- Generated C code contains `Rte_Pim_*` API calls for data store variables.


When you build your model, the XML files that are generated define an exclusive area for each Data Store Memory block that references per-instance memory. Every runnable that accesses per-instance memory runs inside the corresponding exclusive area. If multiple AUTOSAR runnables have access to the same Data Store Memory block, the exported AUTOSAR specification enforces data consistency

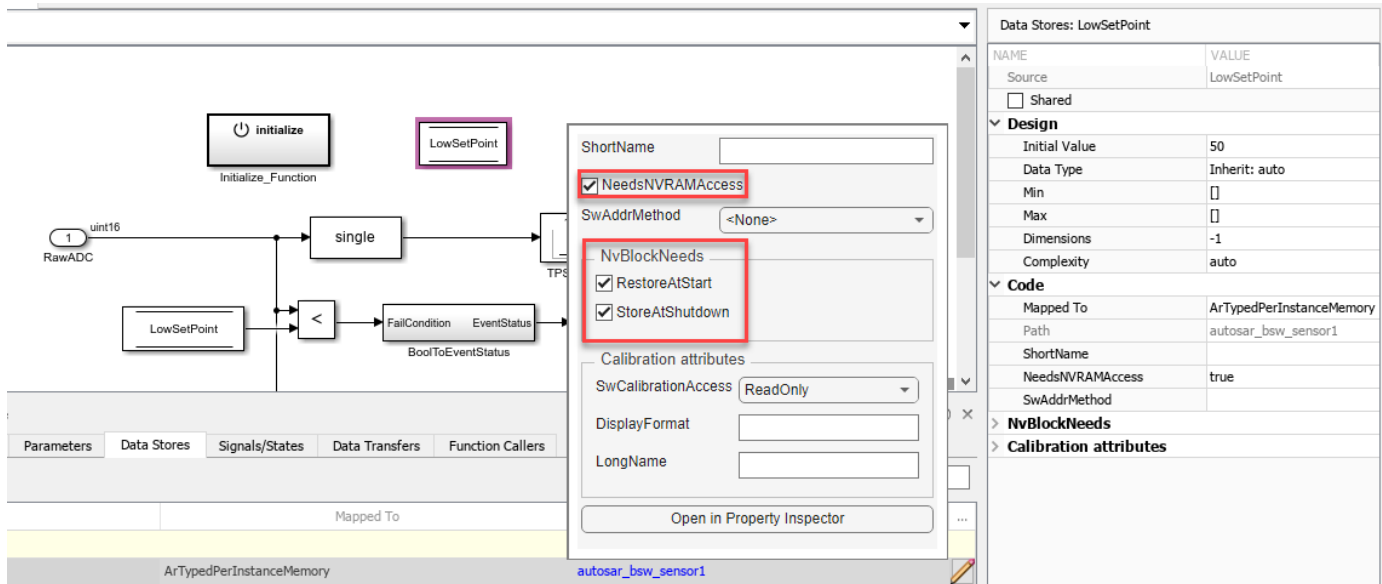
by using an AUTOSAR exclusive area. With this specification, the runnables have mutually exclusive access to the per-instance memory global data, which prevents data corruption.

In the AUTOSAR attributes of the per-instance memory, if you select **needsNVRAMAccess**, a **SERVICE-NEEDS** entry is declared in XML files. The entry indicates that the per-instance memory is a RAM mirror block and requires service from the NvM manager module. For more information about modeling software component access to AUTOSAR nonvolatile memory, see “Model AUTOSAR Nonvolatile Memory” on page 2-40.

Configure Data Stores to Preserve State Information at Startup and Shutdown

To facilitate bottom-up and round-trip workflows, you can configure NVRAM block state data to be read out at startup and written away at shutdown by configuring the NvBlockNeeds properties **RestoreAtStart** and **StoreAtShutdown**. To set these properties, you must configure your model data stores as **ArTypedPerInstanceMemory** and set the property **needsNVRAMAccess** as **true**.

To set these parameters interactively, you can use the Code Mappings editor by selecting the pencil icon , or by using the **NvBlockNeeds** section of the Property Inspector:



NAME	VALUE
Source	LowSetPoint
<input type="checkbox"/> Shared	
Design	
Initial Value	50
Data Type	Inherit: auto
Min	[]
Max	[]
Dimensions	-1
Complexity	auto
Code	
Mapped To	ArTypedPerInstanceMemory
Path	autosar_bsw_sensor1
ShortName	
NeedsNVRAMAccess	true
SwAddrMethod	
NvBlockNeeds	
Calibration attributes	

To configure these parameters programmatically, you can configure the mapping object by using the `getDataStore` function:

```
mappingObj = autosar.api.getSimulinkMapping(modelName);
mappingObj.mapDataStore(dsmBlockPath, 'ArTypedPerInstanceMemory', ...
'NeedsNVRAMAccess', 'true', ...
'RestoreAtStart', 'true', ...
'StoreAtShutdown', 'true');
```

See Also

`getDataStore` | `getSignal` | `getState` | `mapDataStore` | `mapSignal` | `mapState` | Data Store Memory

Related Examples

- “Map Block Signals and States to AUTOSAR Variables” on page 4-58
- “Map Submodel Signals and States to AUTOSAR Variables” on page 4-70
- “Map Data Stores to AUTOSAR Variables” on page 4-56
- “Map Submodel Data Stores to AUTOSAR Variables” on page 4-69
- “Model AUTOSAR Component Behavior” on page 2-31

More About

- “Per-Instance Memory” on page 2-34
- “Model AUTOSAR Nonvolatile Memory” on page 2-40

Configure AUTOSAR Static Memory

To model AUTOSAR static memory for AUTOSAR applications, you import static memory definitions from ARXML files or create static memory content in Simulink. For information about the high-level static memory workflow, see “Static and Constant Memory” on page 2-34.


AUTOSAR static memory (`StaticMemory`) corresponds to Simulink internal global signals. In the AUTOSAR run-time environment, calibration tools can access `StaticMemory` blocks for calibration and measurement.

To model AUTOSAR static memory, you can use Simulink block signals, discrete states, or data stores in your model.

Configure Block Signals and States as AUTOSAR Static Memory

To generate `StaticMemory` blocks for Simulink block signal and discrete state data in your AUTOSAR model, open the Code Mappings editor and select the **Signals/States** tab. Select signals and states and map them to `StaticMemory`. For example:

- 1 Open an AUTOSAR model that contains signals or states that you want to generate `StaticMemory` blocks for. This example uses model `autosar_sw_c_counter`.
- 2 In the AUTOSAR Code perspective, open the Code Mappings editor and select the **Signals/States** tab. In the list of available signals, select `equal_to_count`. Selecting a signal highlights the signal in the model diagram. In the **Mapped To** drop-down list, select `StaticMemory`. To

view and modify AUTOSAR attributes for the static memory, click the  icon. For more information about signal code and calibration attributes, see “Map Block Signals and States to AUTOSAR Variables” on page 4-58.


This model maps Simulink model workspace parameters and variables for AUTOSAR

Copyright 1994-2018 T

Code Mappings - Component Interface

Source	Mapped To
equal_to_count	StaticMemory

ShortName: SM_equal_to_count
 Volatile
 AdditionalNativeTypeQualifier: my_qualifier
 SwAddrMethod: VAR
 Calibration attributes
 SwCalibrationAccess: ReadOnly
 DisplayFormat:
 LongName:
 Open in Property Inspector

- 3 Select the **Signals/States** tab, and then select state X. From the **Mapped To** drop-down list, select **StaticMemory**. To view and modify AUTOSAR attributes for the static memory, click the  icon.


When you generate code:

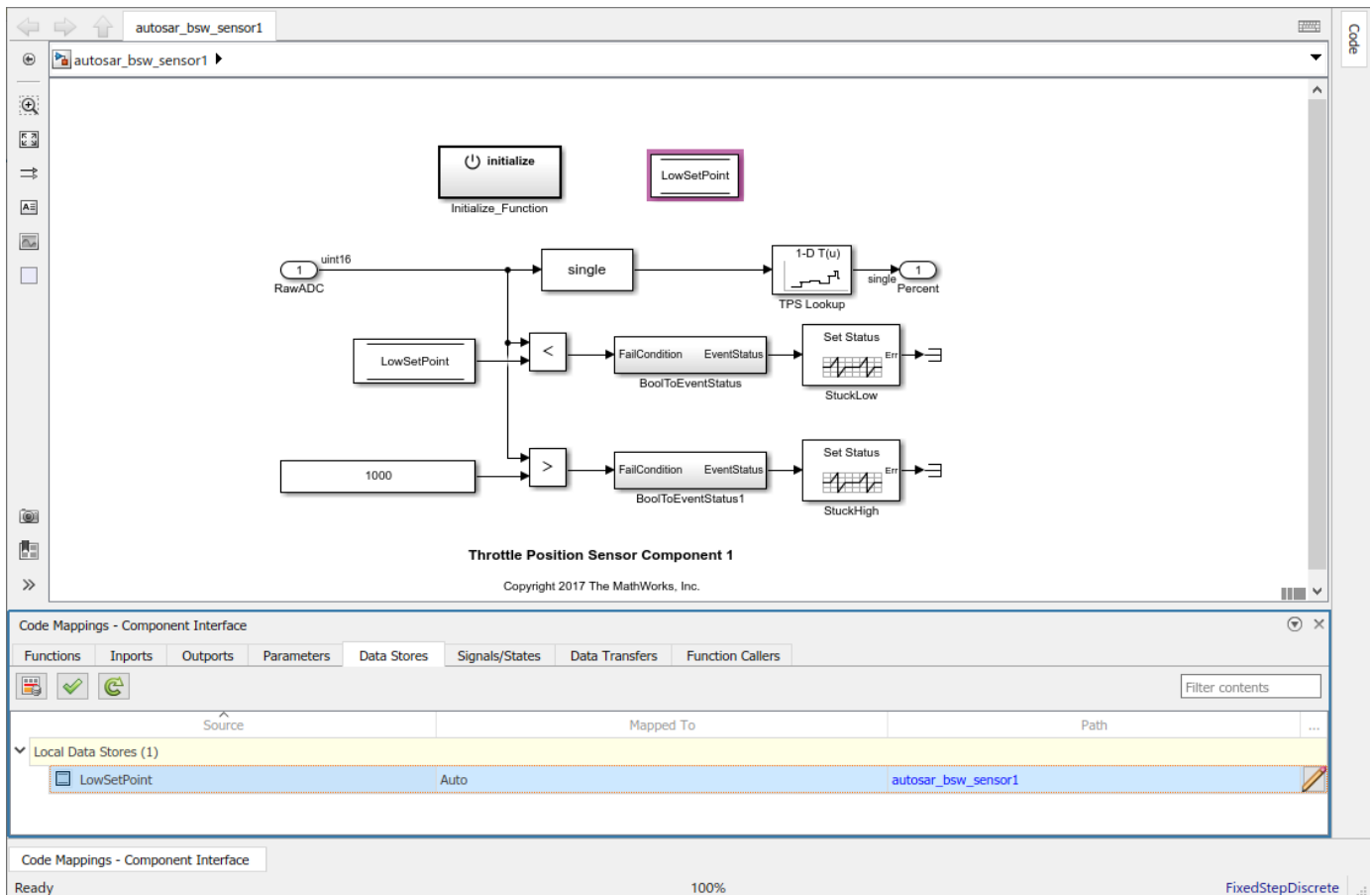
- Exported ARXML files contain **STATIC-MEMORYS** descriptions for signals and states that you configured as **StaticMemory**.
- Generated C code declares and references the static memory variables.

For referenced models within an AUTOSAR component model, Embedded Coder maps internal signals and states for model reference code generation. Internal signals and states map to AUTOSAR **ArTypedPerInstanceMemory** for multi-instance model reference or to AUTOSAR **StaticMemory** for single-instance model reference.

Configure Data Stores as AUTOSAR Static Memory

To generate **StaticMemory** blocks for Simulink data store memory blocks in your AUTOSAR model, open the Code Mappings editor and select the **Data Stores** tab. Select data stores and map them to **StaticMemory**. For example:

- 1 Open an AUTOSAR model that contains data stores that you want to generate `StaticMemory` blocks for. This example uses model `autosar_bsw_sensor1`.
- 2 In the AUTOSAR Code perspective, open the Code Mappings editor and select the **Data Stores** tab. From the list of available data stores, select data store `LowSetPoint`. Selecting a data store highlights the data store memory block in the model diagram. From the **Mapped To** drop-down list, select `StaticMemory`. To view and modify AUTOSAR attributes for the static memory, click the  icon. For more information about data store code and calibration attributes, see “Map Data Stores to AUTOSAR Variables” on page 4-56.



When you generate code:

- Exported ARXML files contain `STATIC-MEMORYS` descriptions for data stores that you configured as `StaticMemory`.
- Generated C code declares and references the static memory variables.

Note AUTOSAR Blockset does not support static memory code generation for data stores in referenced models.

See Also

[getDataStore](#) | [getSignal](#) | [getState](#) | [mapDataStore](#) | [mapSignal](#) | [mapState](#) | [Data Store Memory](#)

Related Examples

- “Map Block Signals and States to AUTOSAR Variables” on page 4-58
- “Map Data Stores to AUTOSAR Variables” on page 4-56
- “Model AUTOSAR Component Behavior” on page 2-31

More About


- “Static and Constant Memory” on page 2-34

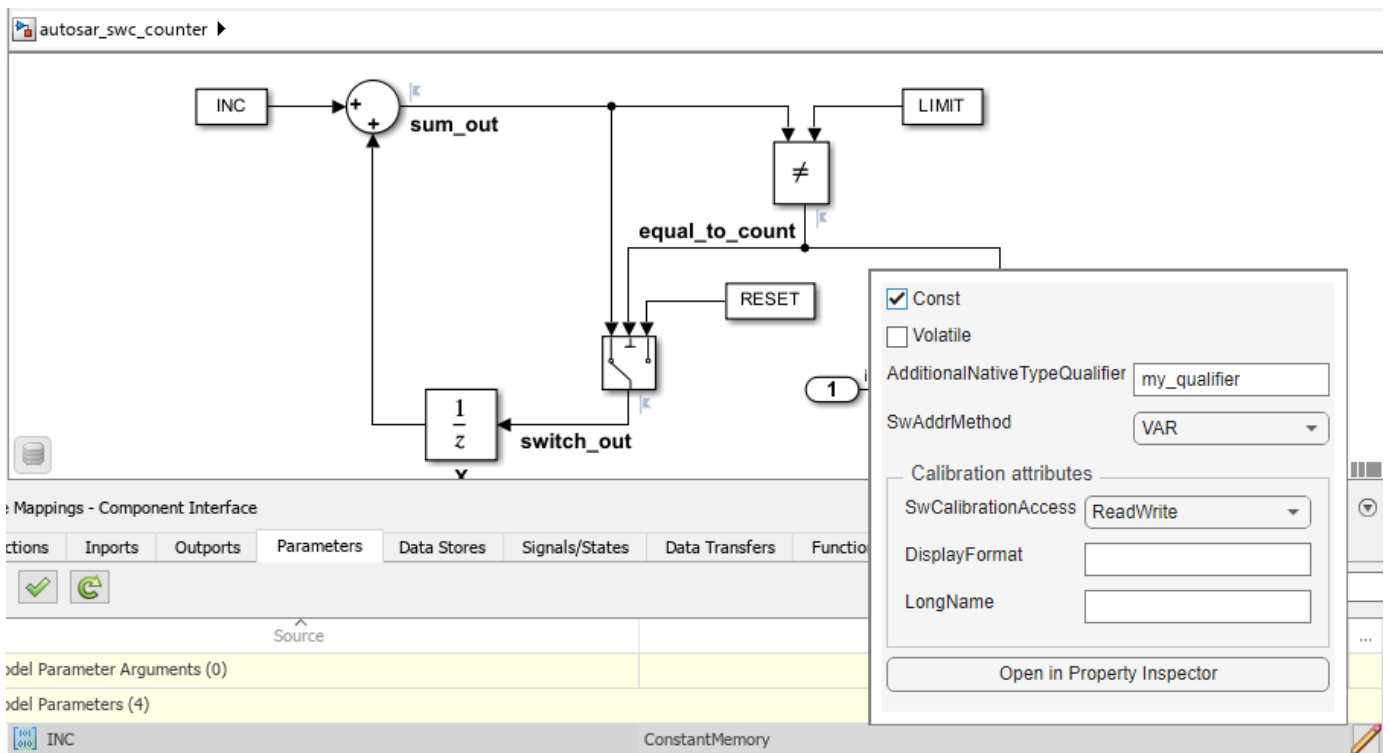
Configure AUTOSAR Constant Memory

You can model AUTOSAR constant memory for AUTOSAR applications. To model AUTOSAR constant memory, import constant memory definitions from ARXML files or create constant memory content in Simulink. For information about the high-level constant memory workflow, see “Static and Constant Memory” on page 2-34.

AUTOSAR constant memory (`ConstantMemory`) corresponds to Simulink internal global parameters. In the AUTOSAR run-time environment, calibration tools can access `ConstantMemory` blocks for calibration and measurement.

To model AUTOSAR constant memory, you can use Simulink model workspace parameters in your model. To generate `ConstantMemory` blocks for model workspace parameter data in your AUTOSAR model, open the Code Mappings editor. Use the **Parameters** tab to map parameters to `ConstantMemory`. For example:

- 1 Open an AUTOSAR model that contains model workspace parameters for which you want to generate `ConstantMemory` blocks. This example uses model `autosar_swc_counter`.
- 2 In the AUTOSAR Code perspective, open the Code Mappings editor and select the **Parameters** tab. In the list of available parameters, select `INC`. In the **Mapped To** drop-down list, select `ConstantMemory`. To view and modify AUTOSAR attributes for the constant memory, click the  icon. For more information about parameter code and calibration attributes, see “Map Model Workspace Parameters to AUTOSAR Component Parameters” on page 4-54.



The screenshot shows the Simulink Code Mappings editor for the model `autosar_swc_counter`. The main workspace displays a Simulink block diagram with blocks for `INC`, `sum_out`, `LIMIT`, `equal_to_count`, `RESET`, `switch_out`, and a `1/z` block. The `Parameters` tab is selected, showing a list of model parameters. The parameter `INC` is selected, and its `Mapped To` property is set to `ConstantMemory`. A pencil icon is clicked to open the `ConstantMemory` configuration dialog. The dialog shows the following settings:

- Const**
- Volatile**
- AdditionalNativeTypeQualifier: `my_qualifier`
- SwAddrMethod: `VAR`
- Calibration attributes:
 - SwCalibrationAccess: `ReadWrite`
 - DisplayFormat: (empty)
 - LongName: (empty)
- Open in Property Inspector button

The bottom of the editor shows the `Mappings - Component Interface` table with the following data:

Model Parameter Arguments (0)	Model Parameters (4)
	<code>INC</code> ConstantMemory

When you generate code:

- Exported ARXML files contain `CONSTANT-MEMORYS` descriptions for parameters that you configured as `ConstantMemory`.

- Generated C code declares and references the constant memory parameters.

See Also

[getParameter](#) | [mapParameter](#)

Related Examples

- “Map Model Workspace Parameters to AUTOSAR Component Parameters” on page 4-54
- “Model AUTOSAR Component Behavior” on page 2-31

More About

- “Static and Constant Memory” on page 2-34

Configure AUTOSAR Shared or Per-Instance Parameters

In this section...

“Configure Model Workspace Parameters as AUTOSAR Shared Parameters” on page 4-212

“Configure Model Workspace Parameters as AUTOSAR Per-Instance Parameters” on page 4-213

You can model AUTOSAR shared parameters (`SharedParameters`) and per-instance parameters (`PerInstanceParameters`) for use in AUTOSAR software components that potentially are instantiated multiple times. Shared parameter values are shared among all instances of a component. Per-instance parameter values are unique and private to each component instance. In the AUTOSAR run-time environment, calibration tools can access `SharedParameters` and `PerInstanceParameters` for calibration and measurement.

To model AUTOSAR shared or per-instance parameters, import parameter definitions from ARXML files or create parameter content in Simulink. For information about the high-level shared and per-instance parameters workflow, see “Shared and Per-Instance Parameters” on page 2-35 .

To model AUTOSAR parameters in Simulink, you use model workspace parameters.


Configure Model Workspace Parameters as AUTOSAR Shared Parameters

To model AUTOSAR shared parameters in Simulink:

- 1 Open an AUTOSAR model that contains a model workspace parameter for which you want to generate an AUTOSAR `SharedParameter`. This example uses model `autosar_sw_counter`.
- 2 To model an AUTOSAR shared parameter in Simulink, configure a model workspace parameter that is not a model argument (that is, not unique to each instance of a multi-instance model). For example, in the Model Explorer view of the parameter, clear the **Argument** property. In example model `autosar_sw_counter`, clear the **Argument** property for parameter K. Leave the parameter **StorageClass** set to `Auto`.

Name	Value	DataType	Dimensions	Complexity	Min	Max	Unit	Argument	StorageClass
INC	1	uint8	[1 1]	real	[]	[]		<input type="checkbox"/>	Auto
K	2	uint8	[1 1]	real	[]	[]		<input type="checkbox"/>	Auto
LIMIT	16	uint8	[1 1]	real	[]	[]		<input type="checkbox"/>	Auto
RESET	0	uint8	[1 1]	real	[]	[]		<input type="checkbox"/>	Auto

- 3 In the AUTOSAR Code perspective, open the Code Mappings editor and select the **Parameters** tab. In the list of available parameters, select K. In the **Mapped To** drop-down list, select parameter type `SharedParameter`. To view and modify AUTOSAR attributes for the shared

parameter, click the  icon. For more information about parameter code and calibration attributes, see “Map Model Workspace Parameters to AUTOSAR Component Parameters” on page 4-54.

This model maps Simulink model workspace parameters and signals to AUTOSAR parameters and variables for AUTOSAR run-time calibration and measurement.

Copyright 1994-2018 The MathWorks, Inc.

Source	Mapped To
Model Parameter Arguments (0)	
Model Parameters (4)	
INC	ConstantMemory
K	SharedParameter
LIMIT	Auto
RESET	Auto

When you generate code:

- Exported ARXML files contain SHARED-PARAMETERS descriptions for parameters that you configured as SharedParameter.
- Generated C code contains Rte_CData calls where shared parameters are used.

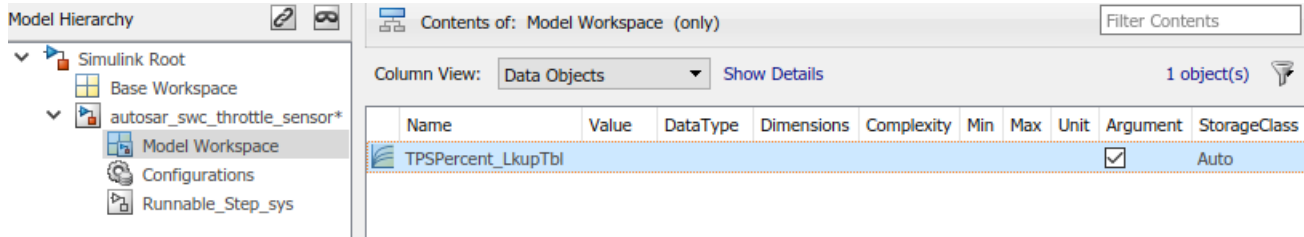
```
autosar_swc_counter_B.Gain = Rte_CData_K() *
Rte_IRead_Runnable_Step_RPort_InData();
```


Configure Model Workspace Parameters as AUTOSAR Per-Instance Parameters

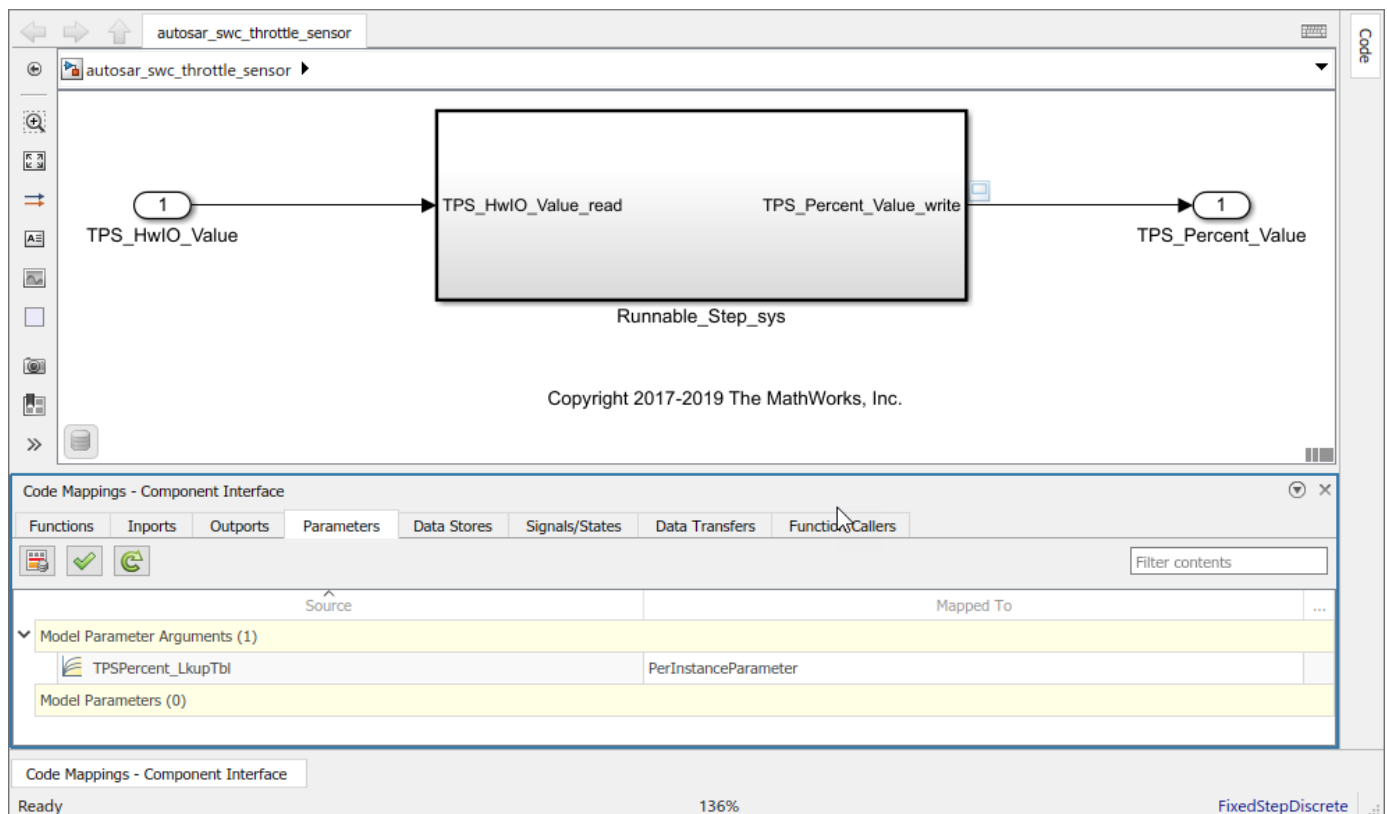
To model AUTOSAR per-instance parameters in Simulink:

- 1 Open an AUTOSAR model that contains a model workspace parameter for which you want to generate an AUTOSAR PerInstanceParameter. This example uses model `autosar_swc_throttle_sensor`. This model is part of AUTOSAR composition model `autosar_composition`, which contains two instances of `autosar_swc_throttle_sensor`.
- 2 To model an AUTOSAR per-instance parameter in Simulink, configure a model workspace parameter that is a model argument (that is, unique to each instance of a multi-instance model).

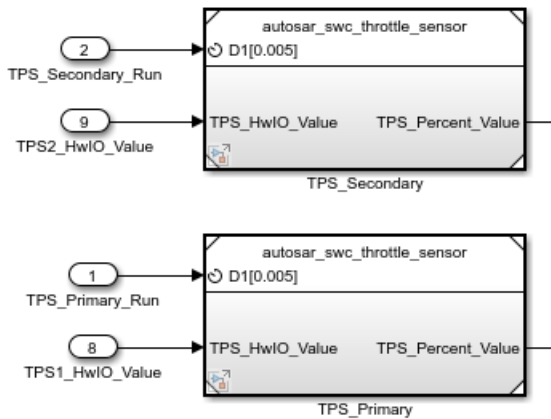
For example, in the Model Explorer view of the parameter, select the **Argument** property. In example model `autosar_swc_throttle_sensor`, select the **Argument** property for parameter `TPSPercent_LkupTbl`. Leave the parameter **StorageClass** set to Auto.



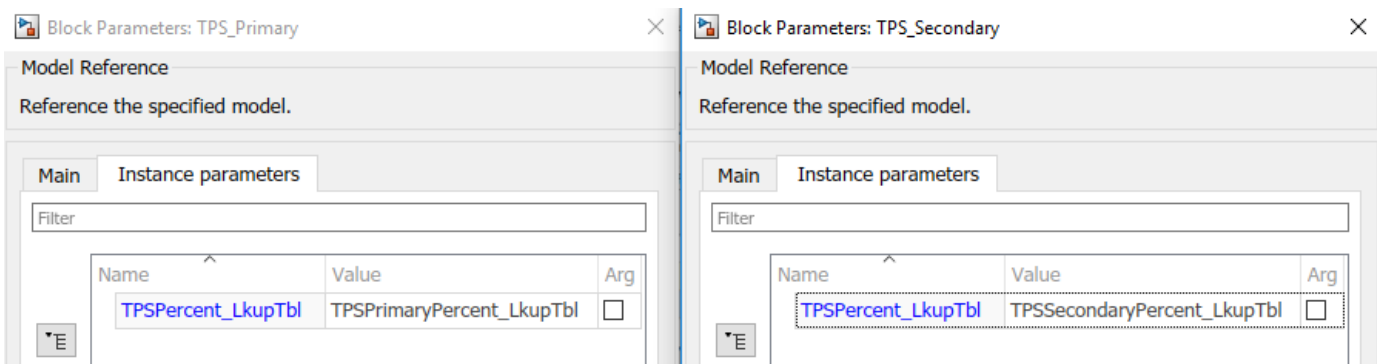
- In the AUTOSAR Code perspective, open the Code Mappings editor and select the **Parameters** tab. Select parameter `TPSPercent_LkupTbl`. In the **Mapped To** drop-down list, select parameter type `PerInstanceParameter`. To view and modify AUTOSAR attributes for the per-instance parameter, click the  icon. For more information about parameter code and calibration attributes, see “Map Model Workspace Parameters to AUTOSAR Component Parameters” on page 4-54. If you are mapping parameters in a submodel referenced by a component, see “Map Submodel Parameters to AUTOSAR Component Parameters” on page 4-68.



AUTOSAR example model `autosar_composition` is a composition model that contains several components, including two instances of component model `autosar_swc_throttle_sensor`.



If you open `autosar_composition`, you can right-click the Model blocks that represent instances of `autosar_swc_throttle_sensor`. If you open up each Model block dialog box, **Instance Parameters** tab, and view them together, notice that each Model block uses a different value for the per-instance parameter.



When you generate code:

- Exported ARXML files contain PER-INSTANCE-PARAMETERS descriptions for parameters that you configured as `PerInstanceParameter`.
- Generated C code contains `Rte_CData` calls where per-instance parameters are used.

```
Rte_IWriteRunnableStepTPSPercentValue(self, look1_iflf_linlcpw((float32)
rtb_DataTypeConversion, (Rte_CData_TPSPercent_LkupTbl(self))->BP1,
(Rte_CData_TPSPercent_LkupTbl(self))->Table, I0U));
```

See Also

[getParameter](#) | [mapParameter](#)

Related Examples

- “Map Model Workspace Parameters to AUTOSAR Component Parameters” on page 4-54
- “Map Submodel Parameters to AUTOSAR Component Parameters” on page 4-68
- “Model AUTOSAR Component Behavior” on page 2-31

More About

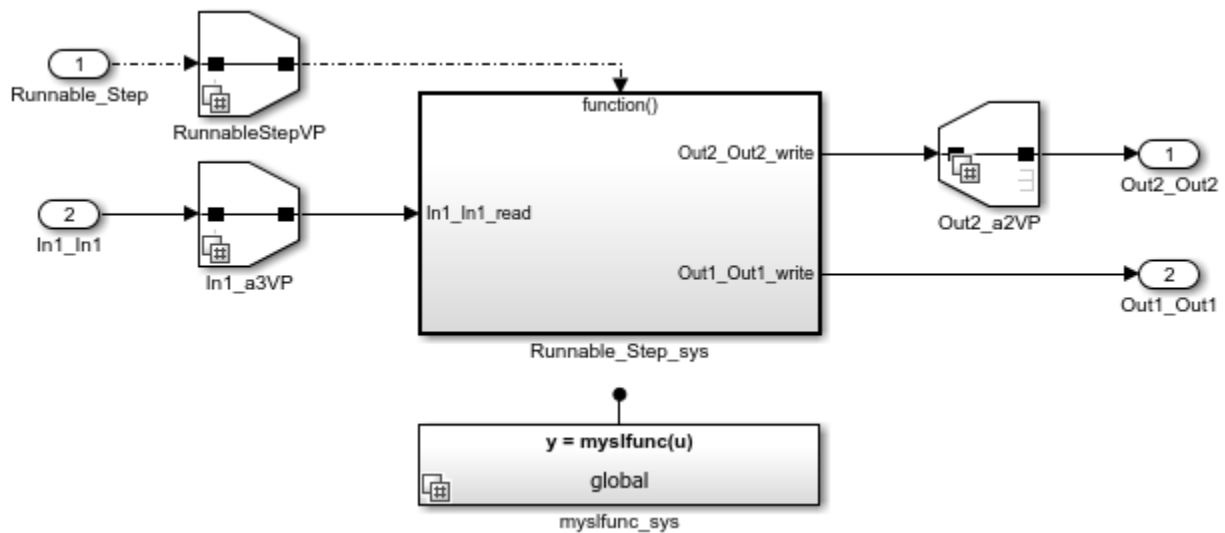
- “Shared and Per-Instance Parameters” on page 2-35

Configure Variants for AUTOSAR Elements

AUTOSAR software components can use `VariationPoint` elements to enable or disable AUTOSAR elements, such as ports and runnables, based on defined conditions. In Simulink®, to configure variants that enable or disable AUTOSAR ports and runnables:

- Use Variant Sink and Variant Source blocks to define variant condition logic and propagate variant conditions.
- Use `AUTOSAR.Parameter` data objects with storage class `SystemConstant` to model AUTOSAR system constants. The system constants represent the condition values that enable or disable ports and runnables.

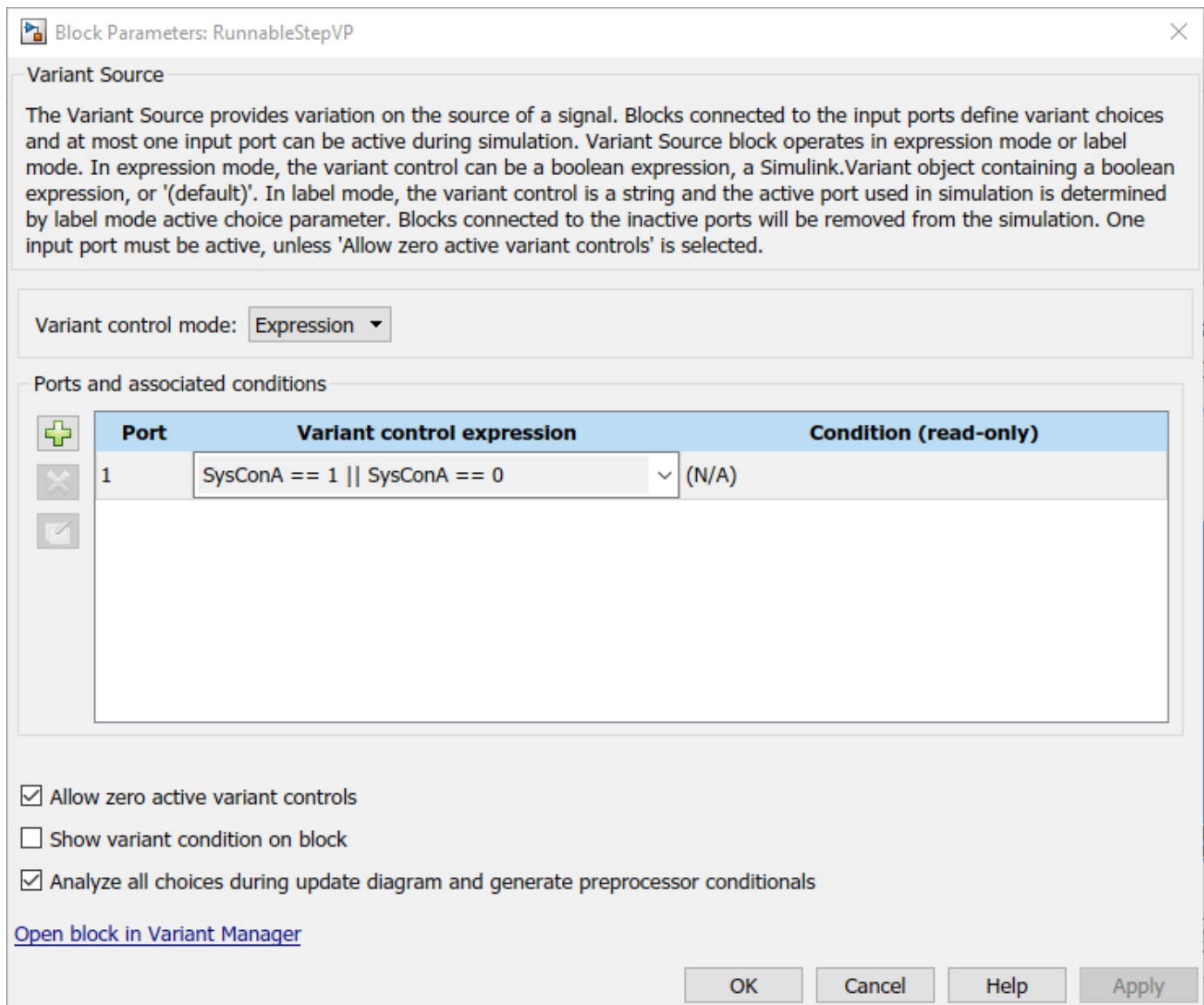
```
open_system('mAutosarInlineVariant.slx');
```



To model an AUTOSAR system constant, the model defines `AUTOSAR.Parameter` data object `SysConA`:

```
SysConA = AUTOSAR.Parameter;
SysConA.CoderInfo.StorageClass = 'Custom';
SysConA.CoderInfo.CustomStorageClass = 'SystemConstant';
SysConA.DataType = 'int32';
SysConA.Value = 1;
```

Each Variant Source or Variant Sink block defines variant condition logic, which is based on the system constant value. You can specify an expression or a `Simulink.Variant` object containing an expression. Here is the variant condition logic for Variant Source block `RunnableStepVP`.



When you generate code for the model:

- The exported ARXML code contains definitions for variation point proxies and variation points. In this example, the VARIATION-POINT-PROXY entry has a short-name c0, which is referenced in the generated C code. SysConA appears as a system constant representing the associated condition value.

```
<VARIATION-POINT-PROXYS>
  <VARIATION-POINT-PROXY UUID="...">
    <SHORT-NAME>c0</SHORT-NAME>
    <CATEGORY>CONDITION</CATEGORY>
    <CONDITION-ACCESS BINDING-TIME="PRE-COMPILE-TIME">
      <SYSC-REF DEST="SW-SYSTEMCONST">/mInlineVariant_pkg/mInlineVariant_dt/SystemConstants/
      == 0 ||
      <SYSC-REF DEST="SW-SYSTEMCONST">/mInlineVariant_pkg/mInlineVariant_dt/SystemConstants/
      == 1</CONDITION-ACCESS>
```

```

</VARIATION-POINT-PROXY>
</VARIATION-POINT-PROXYS>

```

VARIATION-POINT entries appear for AUTOSAR ports, runnables, and runnable accesses to external data.

```

<R-PORT-PROTOTYPE UUID="...">
  <SHORT-NAME>In1</SHORT-NAME>
  <VARIATION-POINT>
    <SHORT-LABEL>In1_a3VP</SHORT-LABEL>
    <SW-SYSCOND BINDING-TIME="PRE-COMPILE-TIME">
      <SYSC-REF DEST="SW-SYSTEMCONST">/mInlineVariant_pkg/mInlineVariant_dt/SystemConstants/
        == 0 ||
      <SYSC-REF DEST="SW-SYSTEMCONST">/mInlineVariant_pkg/mInlineVariant_dt/SystemConstants/
        == 1</SW-SYSCOND>
    </VARIATION-POINT>
    ...
  </R-PORT-PROTOTYPE>

```

- In the RTE compatible C code, short-name c0 is encoded in the names of preprocessor symbols used in the variant condition logic. For example:

```

#if Rte_SysCon_c0
...
#endif

```

For more information, see “Variant Systems” (Embedded Coder) (Embedded Coder) and “Variant Systems”.

See Also

AUTOSAR.Parameter | Variant Sink | Variant Source

Related Examples

- “Model AUTOSAR Variants” on page 2-37

More About

- “Variant Systems” (Embedded Coder)
- “System Constants” on page 2-33

Configure Variants for AUTOSAR Runnable Implementations

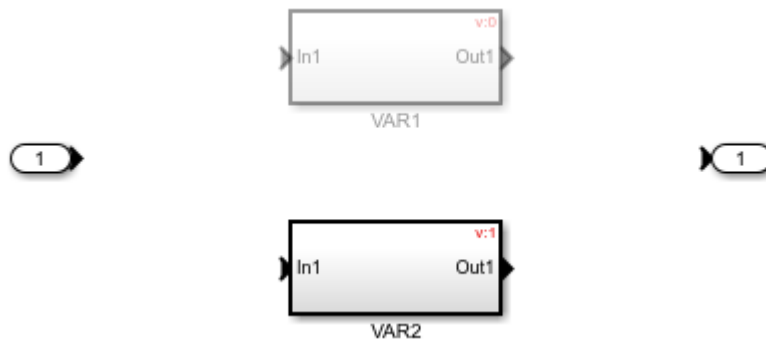
To vary the implementation of an AUTOSAR runnable, AUTOSAR software components can specify variant condition logic inside a runnable. In Simulink®, to model variant condition logic inside a runnable:

- To represent variant implementations of a subsystem or model and define variant condition logic, use a Variant Subsystem, Variant Model, Variant Assembly Subsystem block.
- To model AUTOSAR system constants, use `AUTOSAR.Parameter` data objects with storage class `SystemConstant`. The system constants represent the condition values that determine the active subsystem or model implementation.

For example, here is an AUTOSAR component model that contains a Variant Subsystem block, which models two variant implementations of a subsystem. You can open the model by running the following code at the MATLAB® command line.

```
open_system("mAutosarVariantSubsystem")
```

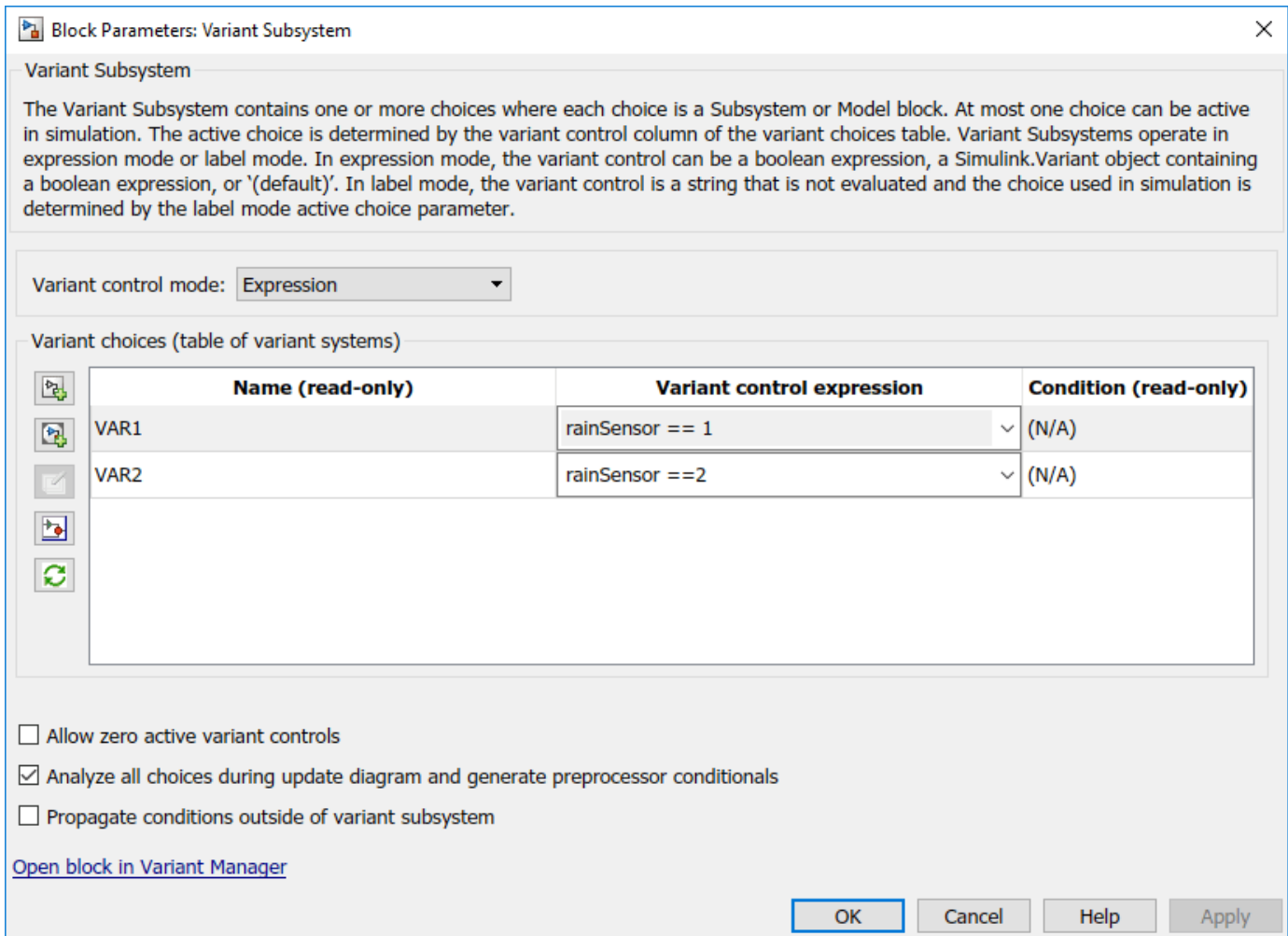
The variant choices are subsystems VAR1 and VAR2. The blocks are not connected because connectivity is determined during simulation, based on the active variant.



To model an AUTOSAR system constant, the model defines `AUTOSAR.Parameter` data object `rainSensor`:

```
rainSensor = AUTOSAR.Parameter;
rainSensor.CoderInfo.StorageClass = 'Custom';
rainSensor.CoderInfo.CustomStorageClass = 'SystemConstant';
rainSensor.DataType = 'uint8';
rainSensor.Value = 2;
```

The Variant Subsystem block dialog box defines the variant condition logic, which is based on the system constant value. You can specify an expression or a `Simulink.Variant` object containing an expression.



When you generate code for the model:

- In the ARXML code, the variant choices appear as VARIATION-POINT-PROXY entries with short-names c0 and c1. rainSensor appears as a system constant representing the associated condition value. For example:

```
<VARIATION-POINT-PROXYS>
  <VARIATION-POINT-PROXY UUID="...">
    <SHORT-NAME>c0</SHORT-NAME>
    <CATEGORY>CONDITION</CATEGORY>
    <CONDITION-ACCESS BINDING-TIME="PRE-COMPILE-TIME">
      <SYSC-REF DEST="SW-SYSTEMCONST">/vss_pkg/vss_dt/SystemConstants/rainSensor</SYSC-REF>
      == 1</CONDITION-ACCESS>
    </VARIATION-POINT-PROXY>
  <VARIATION-POINT-PROXY UUID="...">
    <SHORT-NAME>c1</SHORT-NAME>
    <CATEGORY>CONDITION</CATEGORY>
    <CONDITION-ACCESS BINDING-TIME="PRE-COMPILE-TIME">
      <SYSC-REF DEST="SW-SYSTEMCONST">/vss_pkg/vss_dt/SystemConstants/rainSensor</SYSC-REF>
      == 2</CONDITION-ACCESS>
    </VARIATION-POINT-PROXY>
</VARIATION-POINT-PROXYS>
```

- In the RTE compatible C code, short-names `c0` and `c1` are encoded in the names of preprocessor symbols used in the variant condition logic. For example:

```
#if Rte_SysCon_c0
...
#elif Rte_SysCon_c1
...
#endif
```

See Also

Variant Subsystem | AUTOSAR.Parameter

Related Examples

- “Model AUTOSAR Variants” on page 2-37

More About

- “System Constants” on page 2-33

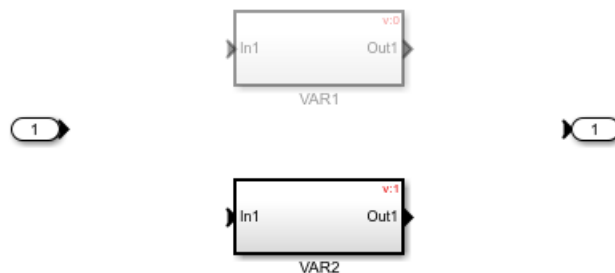
Export Variation Points for AUTOSAR Calibration Data

You can export variation points for AUTOSAR calibration data, including:

- Parameters — Calibration, shared internal, instance-specific, or constant memory
- Per-instance memory — C- typed or AR-typed
- Inter-runnable variables (IRVs) — Implicit or explicit

You can model calibration data in combination with different types of variant conditions. Model the variant conditions by using Variant Source and Variant Sink blocks, Variant Subsystem blocks, or model reference variants. When you build your model, the exported AUTOSAR XML (ARXML) files contain the conditionally used data elements and their variation points.

For example, suppose that you open an AUTOSAR component model containing a Variant Subsystem block, which models two variant implementations of a subsystem, VAR1 and VAR2.



Block Parameters: Variant Subsystem ✕

Variant Subsystem

The Variant Subsystem contains one or more choices where each choice is a Subsystem or Model block. At most one choice can be active in simulation. The active choice is determined by the variant control column of the variant choices table. Variant Subsystems operate in expression mode or label mode. In expression mode, the variant control can be a boolean expression, a Simulink.Variant object containing a boolean expression, or '(default)'. In label mode, the variant control is a string that is not evaluated and the choice used in simulation is determined by the label mode active choice parameter.

Variant control mode: Expression

Variant choices (table of variant systems)

	Name (read-only)	Variant control expression	Condition (read-only)
+	VAR1	rainSensor == 1	(N/A)
+	VAR2	rainSensor ==2	(N/A)

In VAR1, which is enabled when `rainSensor == 1`, you define a parameter named `scale` and reference it in a block. When you build the model, the exported ARXML contains a variation point description for the parameter. In the variation point `SHORT-LABEL`, the parameter name is prefixed with `vp`. In this example, the description indicates that the `scale` parameter is used when variant condition `rainSensor == 1` is true.

```
<VARIATION-POINT>
  <SHORT-LABEL>vpscale</SHORT-LABEL>
  <SW-SYSCOND BINDING-TIME="PRE-COMPILE-TIME"><SYSC-REF DEST="SW-SYSTEMCONST">
```

```
/vss_pkg/vss_dt/SystemConstants/rainSensor</SYSC-REF> == 1</SW-SYSCOND>  
</VARIATION-POINT>
```

See Also

Variant Sink | Variant Source | Variant Subsystem

Related Examples

- “Configure Variants for AUTOSAR Elements” on page 4-217
- “Configure Variants for AUTOSAR Runnable Implementations” on page 4-220

More About

- “Model AUTOSAR Variants” on page 2-37

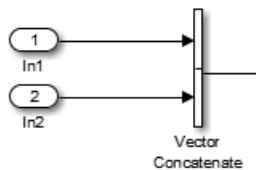
Configure Dimension Variants for AUTOSAR Array Sizes

AUTOSAR software components can flexibly specify the dimensions of an AUTOSAR element, such as a port, by using a symbolic reference to a system constant. The system constant defines the array size of the port data type. To model AUTOSAR elements with variant array sizes in Simulink:

- Create blocks that represent AUTOSAR elements.
- To represent array size values, add `AUTOSAR.Parameter` data objects with storage class `SystemConstant`.
- To specify an array size for an AUTOSAR element, reference an `AUTOSAR.Parameter` data object.

With variant array sizes, you can modify array size values in system constants between model simulations, without regenerating code for simulation. When you build the model, the generated C and ARXML code contains symbols corresponding to variant array sizes.

Suppose that you create a Simulink inport `In1` to represent an AUTOSAR receiver port with a variant array size.



To model the AUTOSAR system constant that specifies the dimensions of `In1`, create an `AUTOSAR.Parameter` data object, `SymDimA`, with storage class `SystemConstant`. The data type must be a signed 32-bit integer type.

```
SymDimA = AUTOSAR.Parameter;
SymDimA.CoderInfo.StorageClass = 'custom';
SymDimA.CoderInfo.CustomStorageClass = 'SystemConstant';
SymDimA.DataType = 'int32';
SymDimA.Min = 1;
SymDimA.Max = 100;
SymDimA.Value = 5;
```

In the dialog box for inport block `In1`, **Signal Attributes** tab, **Port dimensions** field, enter the parameter name, `SymDimA`.

Port dimensions (-1 for inherited):

To allow symbolic dimensions to propagate throughout the model, you must select the model configuration option **Allow symbolic dimension specification**.

When you generate code for the model, the name of the system constant, `SymDimA`, appears in C and ARXML code to represent the variant array size. Here is a sample of the generated C code:

```
/* SignalConversion generated from: '<Root>/Vector Concatenate' */
for (i = 0; i < Rte_SysCon_SymDimA; i++) {
    rtb_VectorConcatenate[i] = tmpIRRead[i];
}
```

Here is a sample of the exported ARXML descriptions:

```
<MAX-NUMBER-OF-ELEMENTS BINDING-TIME="PRE-COMPILE-TIME">
  <SYSC-REF DEST="SW-SYSTEMCONST">/varDim_pkg/dt/SystemConstants/SymDimA</SYSC-REF>
</MAX-NUMBER-OF-ELEMENTS>
```

See Also

AUTOSAR.Parameter | **Allow symbolic dimension specification**

Related Examples

- “Implement Symbolic Dimensions for Array Sizes in Generated Code” (Embedded Coder)
- “Model AUTOSAR Variants” on page 2-37

More About

- “System Constants” on page 2-33

Control AUTOSAR Variants with Predefined Value Combinations

To define the values that control variation points in an AUTOSAR software component, components use the following AUTOSAR elements:

- `SwSystemconst` — Defines a system constant that serves as an input to control a variation point.
- `SwSystemconstantValueSet` — Specifies a set of system constant values to apply to an AUTOSAR software component.
- `PredefinedVariant` — Describes a combination of system constant values, among potentially multiple valid combinations, to apply to an AUTOSAR software component.

For example, in ARXML code, you can define `SwSystemconst`s for automobile features, such as Transmission, Headlight, Sunroof, and Turbocharge. Then a `PredefinedVariant` can map feature combinations to automobile model variants, such as Basic, Economy, Senior, Sportive, and Junior.

Suppose that you have an ARXML specification of an AUTOSAR software component. If the ARXML files also define a `PredefinedVariant` or `SwSystemconstantValueSet`s for controlling variation points in the component, you can resolve the variation points at model creation time. Specify a `PredefinedVariant` or `SwSystemconstantValueSet`s with which the importer can initialize `SwSystemconst` data.

Typical steps include:

- 1 Get a list of the `PredefinedVariants` or `SwSystemconstantValueSet`s defined in the ARXML file.

```
>> obj = arxml.importer('mySWC.arxml');
>> find(obj, '/', 'PredefinedVariant', 'PathType', 'FullyQualified');
ans =
    '/pkg/body/Variants/Basic'
    '/pkg/body/Variants/Economy'
    '/pkg/body/Variants/Senior'
    '/pkg/body/Variants/Sportive'
    '/pkg/body/Variants/Junior'

>> obj = arxml.importer('mySWC.arxml');
>> find(obj, '/', 'SystemConstValueSet', 'PathType', 'FullyQualified')
ans =
    '/pkg/body/SystemConstantValues/A'
    '/pkg/body/SystemConstantValues/B'
    '/pkg/body/SystemConstantValues/C'
    '/pkg/body/SystemConstantValues/D'
```

- 2 Create a model from the ARXML file, and specify a `PredefinedVariant` or one or more `SwSystemconstantValueSet`s.

This example specifies `PredefinedVariant Senior`, which describes a combination of values for Transmission, Headlight, Sunroof, and Turbocharge.

```
>> createComponentAsModel(obj, compNames{1}, 'ModelPeriodicRunnablesAs', 'AtomicSubsystem', ...
    'PredefinedVariant', '/pkg/body/Variants/Senior');
```

This example specifies `SwSystemconstantValueSet`s A and B, which together provide values for `SwSystemconst`s in the AUTOSAR software component.

```
>> createComponentAsModel(obj, compNames{1}, 'ModelPeriodicRunnablesAs', 'AtomicSubsystem', ...
    'SystemConstValueSet', {'/pkg/body/SystemConstantValues/A', '/pkg/body/SystemConstantValues/B'});
```

- 3 During model creation, the ARXML importer creates AUTOSAR.Parameter data objects, with **Storage class** set to `SystemConstant`. The importer initializes the system constant data with values based on the specified `PredefinedVariant` or `SwSystemconstantValueSet`s.

After model creation, you can run simulations and generate code based on the combination of variation point input values that you specified.

In Simulink, you can redefine the `SwSystemconst` data that controls variation points without recreating the model. Call the AUTOSAR property function `createSystemConstants`, and specify a different imported `PredefinedVariant` or a different cell array of `SwSystemconstantValueSets`. The function creates a set of system constant data objects with the same names as the original objects. You can run simulations and generate code based on the revised combination of variation point input values.

This example creates a set of system constant data objects with names and values based on imported `PredefinedVariant` `'/pkg/body/Variants/Economy'`.

```
arProps = autosar.api.getAUTOSARProperties(hModel);  
createSystemConstants(arProps, '/pkg/body/Variants/Economy');
```

Building the model exports previously imported `PredefinedVariants` and `SwSystemconstantValueSets` to ARXML code.

See Also

Related Examples

- “Model AUTOSAR Variants” on page 2-37
- “Configure Variants for AUTOSAR Elements” on page 4-217

More About

- “System Constants” on page 2-33

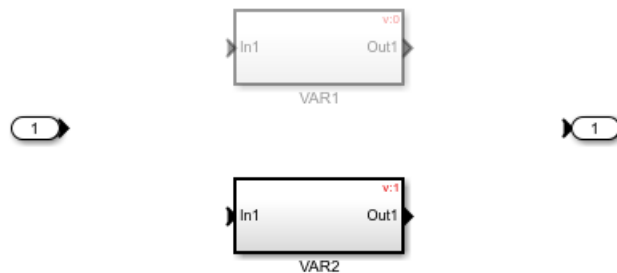
Configure Postbuild Variant Conditions for AUTOSAR Software Components

AUTOSAR software components use variants to enable or disable AUTOSAR interfaces or implementations in the execution path based on defined conditions. Variation points in a component present a choice between two or more variants. Postbuild variant binding enables you to configure AUTOSAR variants modeled in Simulink to activate on or after the AUTOSAR software component startup by using an AUTOSAR run-time environment (RTE) function call. You can now:

- Import AUTOSAR software components that contain postbuild variation points from ARXML files.
- Import shared `PostBuildVariantCriterion` and `PostBuildVariantCondition` definitions from ARXML files.
- Model AUTOSAR postbuild variation points in component models.
- Export ARXML variant descriptions that define `PostBuildVariantCriteria`s and `PostBuildVariantConditions`.
- Generate C code with AUTOSAR `Rte_PbCon` function calls.

You can create postbuild variants at startup for AUTOSAR component models as long as the components contain Simulink variant blocks that model AUTOSAR variants. Alternatively, you can import ARXML files that contain `PostBuild` conditions and have AUTOSAR blockset create the parameter objects for each defined `PostBuildVariantCriterion` and the relevant `Variant Source` and `Variant Sink` blocks with startup variant activation times.

For an example of how to create and configure postbuild conditions, open the model from `matlabroot/help/toolbox/autosar/examples/mAutosarVariantSubsystem.slx`.

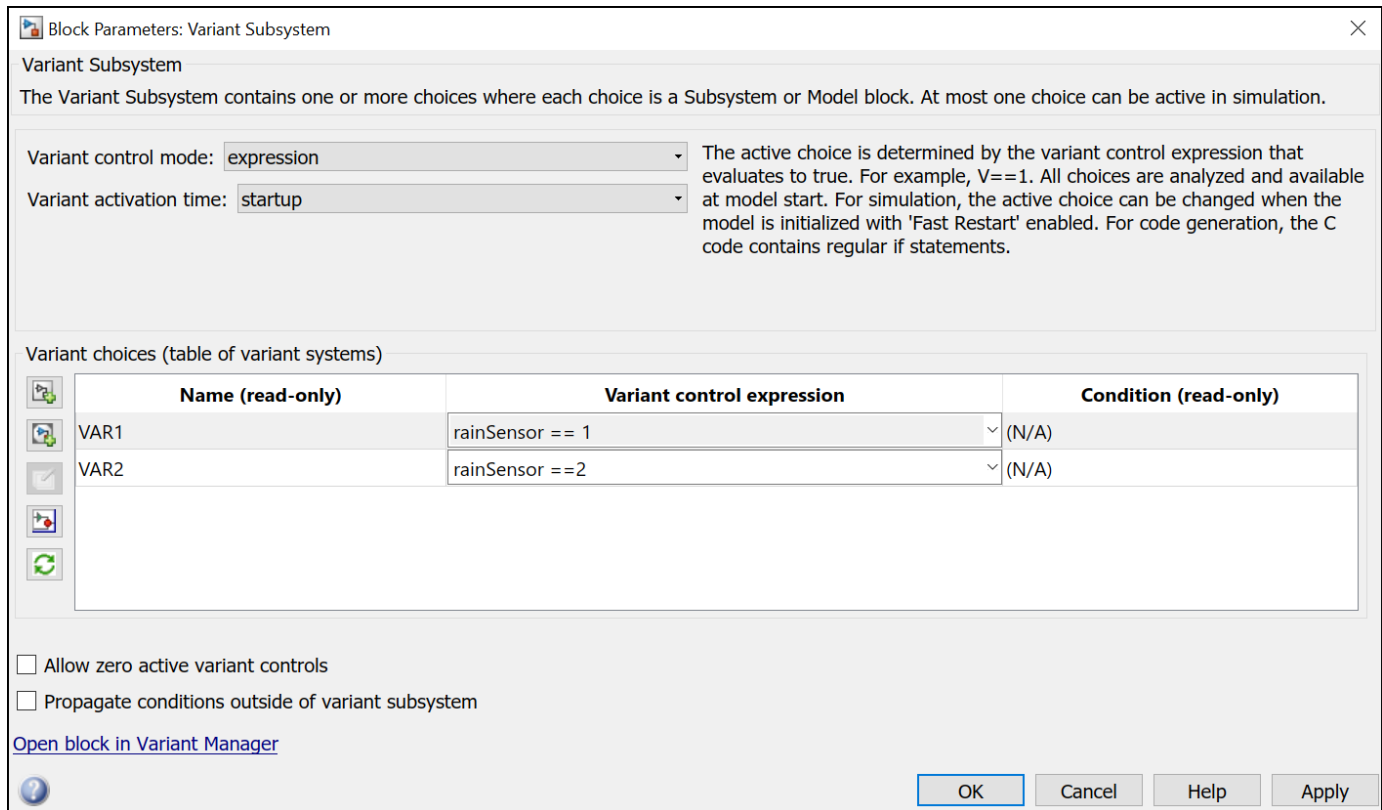


To specify the startup variant activation time, open the Block Parameters of the variant blocks in the component model and configure the **Variant activation time** to `Startup`. For this model, configure the `Variant Subsystem` block.

To model a postbuild condition, create a MATLAB variable. This model already defines a variant condition as a `AUTOSAR.Parameter` data object, `rainSensor`. Configure this object as a MATLAB variable postbuild condition:

```
rainSensor = 2;
```

The `Variant Subsystem` block dialog box defines the variant condition logic, which is based on the postbuild constant value. You can specify an expression or a `Simulink.Variant` object containing an expression.



When you generate code and export ARXML for the model:

- The exported ARXML includes `PostBuildVariantCriterion` and `PostBuildVariantCondition` descriptions for the postbuild variant criteria and conditions that you defined.

```
<POST-BUILD-VARIANT-CONDITIONS>
  <POST-BUILD-VARIANT-CONDITION>
    <MATCHING-CRITERION-REF DEST="POST-BUILD-VARIANT-CRITERION">
      /vss_pkg/vss_dt/PostBuildCriteria/rainSensor
    </MATCHING-CRITERION-REF>
    <VALUE>1</VALUE>
  </POST-BUILD-VARIANT-CONDITION>
</POST-BUILD-VARIANT-CONDITIONS>
```

You can use the AUTOSAR Dictionary XML Options to generate the `PostBuildVariantCriteria`s and associated `ValueSets` as a package.

- The generated AUTOSAR C code includes `Rte_PbCon` function calls to resolve postbuild conditions for variant binding.

```
void Runnable_Step(void)
{
  ...
  /* Outputs for Atomic SubSystem: '<Root>/Variant Subsystem' */
  if (Rte_PbCon_mAutosarVariantSubsystem_c0()) {
  ...
  } else if (Rte_PbCon_mAutosarVariantSubsystem_c1()) {
  ...
  ...
}
```



```
    }  
    /* End of Outputs for SubSystem: '<Root>/Variant Subsystem' */  
    ...  
}
```

For software-in-the-loop (SIL) simulation, in the stub folder, the model build generates stub implementations of the Rte_PbCon functions used to resolve the post-build conditions.

See Also

[AUTOSAR.Parameter](#) | [Variant Sink](#) | [Variant Source](#) | [Variant Subsystem](#)

Related Examples

- “Configure Variants for AUTOSAR Elements” on page 4-217
- “Configure Variants for AUTOSAR Runnable Implementations” on page 4-220

More About

- “Model AUTOSAR Variants” on page 2-37
- “Activate Variant During Different Stages of Simulation and Code Generation Workflow”

Configure Variant Parameter Values for AUTOSAR Elements

AUTOSAR software components can flexibly specify the parameter value of an AUTOSAR element by using variant parameters. To model AUTOSAR elements with variable parameter values in Simulink:

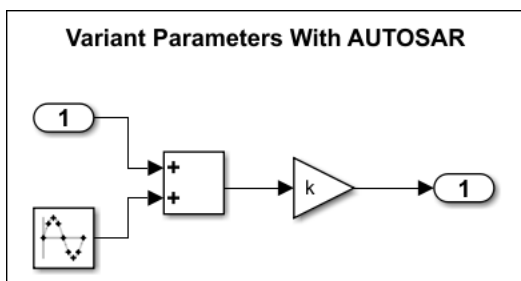
- Create blocks that represent AUTOSAR elements.
- Represent varying parameter values by adding `Simulink.VariantVariable` data objects.
- Model AUTOSAR system constants by using `AUTOSAR.Parameter` data objects. The AUTOSAR data objects represent the condition values that determine the active value of variant parameters.
- Associate an activation time with the AUTOSAR system constants by using `Simulink.VariantControl` data objects. The activation time determines at which stage of code generation you can modify the variant parameter values.

With variant parameters, you can modify parameter values prior to code compile or at model startup. When you build the model, the generated C code contains values and conditions corresponding to variant parameters. The exported ARXML code contains the variant choices as `VARIATION-POINT-PROXY` entries and the variant control variable as a system constant representing the associated condition value.

Specify Variant Parameters at Precompile Time

This example shows how to generate a code for an AUTOSAR element that contains variant parameters without needing to regenerate the code for each set of values. In the generated code, the variant parameter values are enclosed in preprocessor conditionals `#if` and `#elif` that enable you to switch between the values prior to code compile.

For example, here is an AUTOSAR component model that contains a variant parameter, `k`, which models multiple values for the Gain block. You can open the model from `matlabroot/help/toolbox/autosar/examples/mAutosarVariantParameter.slx`.



This parameter defines multiple values for the **Gain** parameter and associates each value with variant condition logic. You can specify the variant condition logic as an expression or a `Simulink.Variant` object containing an expression.

```
ADAPTIVE = Simulink.Variant('MySys == 10');
LINEAR = Simulink.Variant('MySys == 1');
NONLINEAR = Simulink.Variant('MySys == 2');
k = Simulink.VariantVariable('Choices',{ 'ADAPTIVE', 10 , 'LINEAR', 1, 'NONLINEAR', 3});
```

To model an AUTOSAR system constant, the model defines the `AUTOSAR.Parameter` data object `tmpSysCon`.

```
tmpSysCon = AUTOSAR.Parameter(int32(1));
tmpSysCon.CoderInfo.StorageClass = 'Custom';
tmpSysCon.CoderInfo.CustomStorageClass = 'SystemConstant';
```

The value of tmpSysCon determines the active value of k.

```
MySys = Simulink.VariantControl('Value', tmpSysCon, 'ActivationTime', 'code compile');
```

When you generate code for the model:

- In the ARXML code, the variant choices appear as VARIATION-POINT-PROXY entries with short-names ADAPTIVE, LINEAR, and NONLINEAR. MySys appears as a system constant representing the associated condition value.

```
<VARIATION-POINT-PROXYS>
  <VARIATION-POINT-PROXY UUID="744b1a40-2029-54ae-fba9-79a6ca104b8c">
    <SHORT-NAME>ADAPTIVE</SHORT-NAME>
    <CATEGORY>CONDITION</CATEGORY>
    <CONDITION-ACCESS BINDING-TIME="PRE-COMPILE-TIME"><SYSC-REF DEST="SW-SYSTEMCONST"/>/DataTypes/SystemConstants/My
  </VARIATION-POINT-PROXY>
  <VARIATION-POINT-PROXY UUID="af1f057b-45e6-58f7-7e12-b66857813de6">
    <SHORT-NAME>LINEAR</SHORT-NAME>
    <CATEGORY>CONDITION</CATEGORY>
    <CONDITION-ACCESS BINDING-TIME="PRE-COMPILE-TIME"><SYSC-REF DEST="SW-SYSTEMCONST"/>/DataTypes/SystemConstants/My
  </VARIATION-POINT-PROXY>
  <VARIATION-POINT-PROXY UUID="6ba924d2-49e1-5948-cbd1-c0990240bb21">
    <SHORT-NAME>NONLINEAR</SHORT-NAME>
    <CATEGORY>CONDITION</CATEGORY>
    <CONDITION-ACCESS BINDING-TIME="PRE-COMPILE-TIME"><SYSC-REF DEST="SW-SYSTEMCONST"/>/DataTypes/SystemConstants/My
  </VARIATION-POINT-PROXY>
</VARIATION-POINT-PROXYS>
```

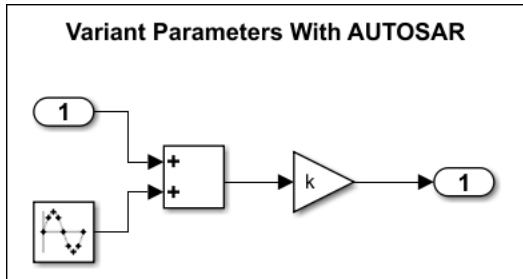
- In the RTE compatible C code, the values of k are enclosed in preprocessor conditionals #if and #elif. When you compile this code, Simulink evaluates the condition expressions. Based on the condition expression that evaluates to true, the gain value associated with that condition logic becomes active and compiles the code only for that gain value. You can then change the value of the variant control variable MySys to compile the code for a different gain parameter value. You are not required to regenerate the code for a different value of gain.

```
Parameters rtP = {
#if Rte_SysCon_ADAPTIVE || Rte_SysCon_LINEAR || Rte_SysCon_NONLINEAR
  /* Variable: k
   * Referenced by: '<Root>/Gain'
   */
  #if Rte_SysCon_ADAPTIVE
    10.0
  #elif Rte_SysCon_LINEAR
    1.0
  #elif Rte_SysCon_NONLINEAR
    3.0
  #endif
  #define PARAMETERS_VARIANT_EXISTS
  #endif
  #ifndef PARAMETERS_VARIANT_EXISTS
    0
  #endif /* PARAMETERS_VARIANT_EXISTS undefined */
};
```

Specify Variant Parameters at Postbuild Time

This example shows how to generate a runnable for an AUTOSAR element that runs for different sets of variant parameter values without needing to recompile the code for each set of values. In the generated code, the variant parameter values are enclosed in regular if conditions that enable you to switch between the values at model startup.

For example, here is an AUTOSAR component model that contains a variant parameter, *k*, which models multiple values for the Gain block. You can open the model from `matlabroot/help/toolbox/autosar/examples/mAutosarVariantParameter.slx`.



This parameter defines multiple values for the **Gain** parameter and associates each value with a variant condition logic. You can specify the variant condition logic as an expression or a `Simulink.Variant` object containing an expression.

```
ADAPTIVE = Simulink.Variant('MyPBCrit == 10');
LINEAR = Simulink.Variant('MyPBCrit == 1');
NONLINEAR = Simulink.Variant('MyPBCrit == 2');
k = Simulink.VariantVariable('Choices', {'ADAPTIVE', 10, 'LINEAR', 1, 'NONLINEAR', 3});
```

The value of `MyPBCrit` determines the active value of *k*.

```
MyPBCrit = Simulink.VariantControl('Value', 1, 'ActivationTime', 'startup');
```

When you generate code for the model:

- In the ARXML code, the variant choices appear as `VARIATION-POINT-PROXY` entries with short-names `ADAPTIVE`, `LINEAR`, and `NONLINEAR`. `MyPBCrit` appears as a system constant representing the associated condition value.

```
<VARIATION-POINT-PROXYS>
  <VARIATION-POINT-PROXY UUID="e773053e-d2a7-568c-768b-fee924d1fad6">
    <SHORT-NAME>ADAPTIVE</SHORT-NAME>
    <CATEGORY>CONDITION</CATEGORY>
    <POST-BUILD-VARIANT-CONDITIONS>
      <POST-BUILD-VARIANT-CONDITION>
        <MATCHING-CRITERION-REF DEST="POST-BUILD-VARIANT-CRITERION">/DataTypes/PostBuildCriteriaions/MyPBCrit</MATCHING-CRITERION-REF>
        <VALUE>10</VALUE>
      </POST-BUILD-VARIANT-CONDITION>
    </POST-BUILD-VARIANT-CONDITIONS>
  </VARIATION-POINT-PROXY>
  <VARIATION-POINT-PROXY UUID="3ce69abb-b974-591d-c7a8-180c64bedfb5">
    <SHORT-NAME>LINEAR</SHORT-NAME>
    <CATEGORY>CONDITION</CATEGORY>
    <POST-BUILD-VARIANT-CONDITIONS>
      <POST-BUILD-VARIANT-CONDITION>
        <MATCHING-CRITERION-REF DEST="POST-BUILD-VARIANT-CRITERION">/DataTypes/PostBuildCriteriaions/MyPBCrit</MATCHING-CRITERION-REF>
        <VALUE>1</VALUE>
      </POST-BUILD-VARIANT-CONDITION>
    </POST-BUILD-VARIANT-CONDITIONS>
  </VARIATION-POINT-PROXY>
  <VARIATION-POINT-PROXY UUID="b4b96126-4744-5093-360b-3965883aeeda">
    <SHORT-NAME>NONLINEAR</SHORT-NAME>
    <CATEGORY>CONDITION</CATEGORY>
    <POST-BUILD-VARIANT-CONDITIONS>
      <POST-BUILD-VARIANT-CONDITION>
        <MATCHING-CRITERION-REF DEST="POST-BUILD-VARIANT-CRITERION">/DataTypes/PostBuildCriteriaions/MyPBCrit</MATCHING-CRITERION-REF>
        <VALUE>2</VALUE>
      </POST-BUILD-VARIANT-CONDITION>
    </POST-BUILD-VARIANT-CONDITIONS>
  </VARIATION-POINT-PROXY>
</VARIATION-POINT-PROXYS>
```

- In the RTE compatible C code, the values of `k` are enclosed in regular `if` conditions. When you execute the runnable built from this code, Simulink evaluates the condition expressions. Based on the condition expression that evaluates to `true`, the gain value associated with that condition logic becomes active and the runnable executes only for that gain value. You can then change the value of the variant control variable `MyPBCrit` to execute the runnable for a different gain parameter value. You are not required to recompile the code to build the runnable for a different gain parameter value.

```
void mBasic_Init(void)
{
    /* Variant Parameters startup activation time */
    if (Rte_PbCon_ADAPTIVE()) {
        rtP.k = 10.0;
    } else if (Rte_PbCon_LINEAR()) {
        rtP.k = 1.0;
    } else if (Rte_PbCon_NONLINEAR()) {
        rtP.k = 3.0;
    }
}
```

See Also

“Use Variant Parameters to Reuse Block Parameters with Different Values”

Related Examples

- “Create a Simple Variant Parameter Model”
- “Model AUTOSAR Variants” on page 2-37

Configure AUTOSAR CompuMethods

AUTOSAR software components use computation methods (CompuMethods) to convert between the internal values and physical representation of AUTOSAR data. Common uses for CompuMethods are linear data scaling and calibration and measurement.

Embedded Coder imports AUTOSAR CompuMethods described in ARXML code and preserves them across round-trips between an AUTOSAR authoring tool (AAT) and Simulink. In Simulink, you can modify imported CompuMethods or create and configure new CompuMethods.

This topic provides examples of configuring AUTOSAR CompuMethods in Simulink.

In this section...

“Configure AUTOSAR CompuMethod Properties” on page 4-236

“Create AUTOSAR CompuMethods” on page 4-237

“Configure CompuMethod Direction for Linear Functions” on page 4-238

“Export CompuMethod Unit References” on page 4-239

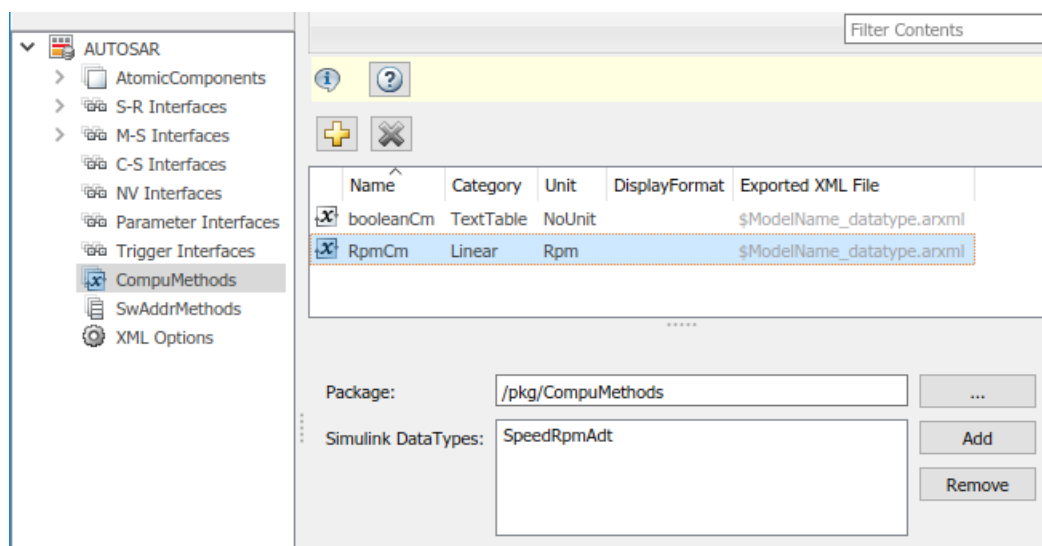
“Modify Linear Scaling for SCALE_LINEAR_AND_TEXTTABLE CompuMethod” on page 4-240

“Configure Rational Function CompuMethod for Dual-Scaled Parameter” on page 4-241

Configure AUTOSAR CompuMethod Properties

You can configure AUTOSAR CompuMethod properties in your model, either graphically or programmatically. The CompuMethod properties you can modify include name, category, unit, display format, AUTOSAR package, and Simulink data types.

To configure a CompuMethod using the graphical interface, open the AUTOSAR Dictionary and select the **CompuMethods** view. This view displays the modifiable CompuMethods in the model, whether imported from ARXML code or created in Simulink.



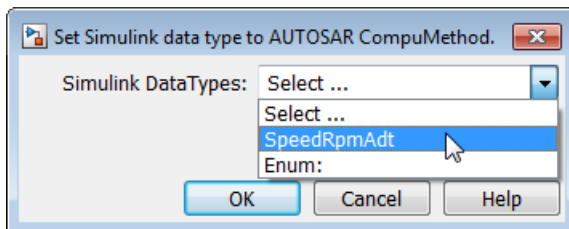
Select a CompuMethod and edit the available fields.

- **Name** — Specify name text
- **Category** — Select *Identical*, *Linear*, *RatFunc*, *TextTable*, or *LinearAndTextTable* (see “CompuMethod Categories for Data Types” on page 2-48)
- **Unit** — Select from units available in the model
- **DisplayFormat** — Optionally specify format to be used by calibration and measurement tools to display the data. Use an ANSI C `printf` format specifier string. For example, `%2.1d` specifies a signed decimal number, with a minimum width of two characters and maximum precision of one digit. The string produces a displayed value such as 12.2. For more information about constructing a format specifier string, see “Configure DisplayFormat” on page 4-264.
- **Package** — Specify path of AUTOSAR package to be generated for CompuMethods
- **Simulink DataTypes** — Specify list of Simulink data types that reference the CompuMethod

To modify the AUTOSAR package for a CompuMethod, you can do either of the following:

- Enter a package path in the **Package** parameter field.
- To open the AUTOSAR Package Browser, click the button to the right of the **Package** field. Use the browser to navigate to an existing package or create and select a package. When you select a package in the browser and click **Apply**, the CompuMethod **Package** parameter value is updated with your selection. For more information about the AUTOSAR Package Browser, see “Configure AUTOSAR Package for Component, Interface, CompuMethod, or SwAddrMethod” on page 4-94.

To associate a CompuMethod with a Simulink data type used in the model, select a CompuMethod and click the **Add** button to the right of **Simulink DataTypes**. This action opens a dialog box with a list of available data types. In the list of values, select a `Simulink.NumericType` or `Simulink.AliasType`, or enter the name of a Simulink enumerated type. To add the type to the **Simulink DataTypes** list, click **OK**.



To set the **Simulink DataTypes** property programmatically, open the model and use an AUTOSAR property `set` function call similar to the following:

```
arProps=autosar.api.getAUTOSARProperties('cmSpeed');
set(arProps, '/pkg/CompuMethods/RpmCm', 'SLDataTypes', {'SpeedRpmAdt'})
sltypes=get(arProps, '/pkg/CompuMethods/RpmCm', 'SLDataTypes')

sltypes =
    'SpeedRpmAdt'
```

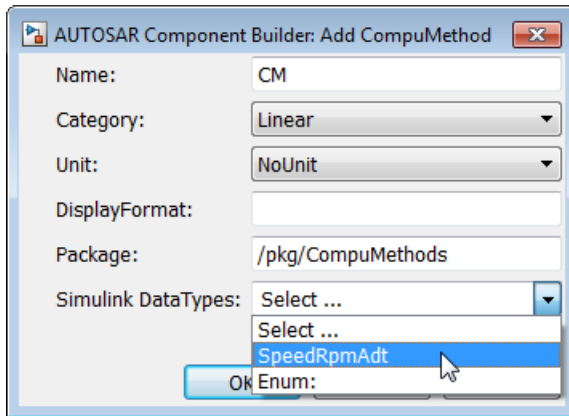
Create AUTOSAR CompuMethods

You can create AUTOSAR CompuMethods in your model, either graphically or programmatically. To create an AUTOSAR CompuMethod using the graphical interface, open the AUTOSAR Dictionary and select the **CompuMethods** view. To open the Add CompuMethod dialog box, click the **Add** button



. Configure the initial properties for the CompuMethod, such as name, category, unit, display format for calibration, AUTOSAR package to generate, and associated Simulink data type. When you

click **OK**, the CompuMethods view in the AUTOSAR Dictionary is updated with the new CompuMethod.



When you generate code, the exported ARXML code contains the CompuMethod definition and references to it.

Configure CompuMethod Direction for Linear Functions

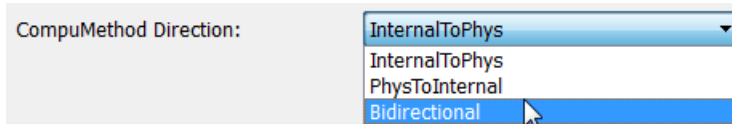
For designs originated in Simulink, you can control properties for an exported CompuMethod, including the direction of CompuMethod conversion between internal and physical representations of a value. Using either the AUTOSAR Dictionary or the AUTOSAR property set function, you can specify one of the following CompuMethod direction values:

- `InternalToPhys` (default) — Generate CompuMethod sections for conversion of internal values into their physical representations.
- `PhysToInternal` — Generate CompuMethod sections for conversion of physical values into their internal representations.
- `Bidirectional` — Generate CompuMethod sections for both internal-to-physical and physical-to-internal conversion directions.

To specify CompuMethod direction in the MATLAB Command Window, use an AUTOSAR property set function call similar to the following:

```
hModel = 'autosar_sw_c_expfcns';
openExample(hModel);
arProps=autosar.api.getAUTOSARProperties(hModel);
set(arProps,'XmlOptions','CompuMethodDirection','Bidirectional');
get(arProps,'XmlOptions','CompuMethodDirection')
```

To specify CompuMethod direction in the AUTOSAR Dictionary, select **XML Options**. Select a value for parameter **CompuMethod Direction**. Click **Apply**.



When you generate code for your model, the CompuMethods in the exported ARXML code contain the requested directional sections. For example, here is a CompuMethod generated with the CompuMethod direction set to `Bidirectional`.


```

<COMPU-METHOD UUID="...">
  <SHORT-NAME>COMPU_EngSpdValue</SHORT-NAME>
  <CATEGORY>LINEAR</CATEGORY>
  <COMPU-INTERNAL-TO-PHYS>
  <COMPU-SCALES>
    <COMPU-SCALE>
      <SHORT-LABEL>intToPhys</SHORT-LABEL>
      <LOWER-LIMIT INTERVAL-TYPE="CLOSED">0</LOWER-LIMIT>
      <UPPER-LIMIT INTERVAL-TYPE="CLOSED">24000</UPPER-LIMIT>
      <COMPU-RATIONAL-COEFFS>
        <COMPU-NUMERATOR>
          <V>0</V>
          <V>1</V>
        </COMPU-NUMERATOR>
        <COMPU-DENOMINATOR>
          <V>8</V>
        </COMPU-DENOMINATOR>
      </COMPU-RATIONAL-COEFFS>
    </COMPU-SCALE>
  </COMPU-SCALES>
</COMPU-INTERNAL-TO-PHYS>
<COMPU-PHYS-TO-INTERNAL>
<COMPU-SCALES>
  <COMPU-SCALE>
    <SHORT-LABEL>physToInt</SHORT-LABEL>
    <LOWER-LIMIT INTERVAL-TYPE="CLOSED">0</LOWER-LIMIT>
    <UPPER-LIMIT INTERVAL-TYPE="CLOSED">3000</UPPER-LIMIT>
    <COMPU-RATIONAL-COEFFS>
      <COMPU-NUMERATOR>
        <V>0</V>
        <V>8</V>
      </COMPU-NUMERATOR>
      <COMPU-DENOMINATOR>
        <V>1</V>
      </COMPU-DENOMINATOR>
    </COMPU-RATIONAL-COEFFS>
  </COMPU-SCALE>
</COMPU-SCALES>
</COMPU-PHYS-TO-INTERNAL>
</COMPU-METHOD>

```

Note CompuMethods of category TEXTTABLE, which are generated for Boolean or enumerated data types, use only InternalToPhys, regardless of the direction parameter setting.

Export CompuMethod Unit References

The ARXML importer preserves unit and physical dimension information found in imported CompuMethods. The software preserves CompuMethod unit and physical dimension information across round-trips between an AUTOSAR authoring tool (AAT) and Simulink.

For designs originated in Simulink, the exporter generates a unit reference for each CompuMethod. By convention, each CompuMethod references a unit named NoUnit. For example, here is a Boolean data type CompuMethod and the unit it references.

```

<COMPU-METHOD UUID="...">
  <SHORT-NAME>COMPU_Boolean</SHORT-NAME>
  <CATEGORY>TEXTTABLE</CATEGORY>
  <UNIT-REF DEST="UNIT">/mymodel_pkg/mymodel_dt/NoUnit</UNIT-REF>
  ...
</COMPU-METHOD>
<UNIT UUID="...">
  <SHORT-NAME>NoUnit</SHORT-NAME>
  <FACTOR-SI-TO-UNIT>1</FACTOR-SI-TO-UNIT>
  <OFFSET-SI-TO-UNIT>0</OFFSET-SI-TO-UNIT>
</UNIT>

```

Providing a unit for each exported CompuMethod helps support calibration and measurement tool use of exported AUTOSAR data.

Modify Linear Scaling for SCALE_LINEAR_AND_TEXTTABLE CompuMethod

You can import and export an AUTOSAR CompuMethod that uses LINEAR and TEXTTABLE scaling. Importing application data types that reference CompuMethods of category SCALE_LINEAR_AND_TEXTTABLE creates Simulink.NumericType or Simulink.AliasType data objects in the Simulink workspace. In Simulink, you can modify the LINEAR scaling for the CompuMethods. The TEXTTABLE scaling is read-only.

For example, here is a CompuMethod with one LINEAR scale and two TEXTTABLE scales.

```
<COMPU-METHOD>
  <SHORT-NAME>COMPU_myType</SHORT-NAME>
  <CATEGORY>SCALE_LINEAR_AND_TEXTTABLE</CATEGORY>
  <COMPU-INTERNAL-TO-PHYS>
    <COMPU-SCALES>
      <<COMPU-SCALE>
        <COMPU-RATIONAL-COEFFS>
          <COMPU-NUMERATOR>
            <V>30</V>
            <V>2</V>
          </COMPU-NUMERATOR>
          <COMPU-DENOMINATOR>
            <V>1</V>
          </COMPU-DENOMINATOR>
        </COMPU-RATIONAL-COEFFS>
      </COMPU-SCALE>
      <COMPU-SCALE>
        <LOWER-LIMIT INTERVAL-TYPE="CLOSED">350</LOWER-LIMIT>
        <UPPER-LIMIT INTERVAL-TYPE="CLOSED">350</UPPER-LIMIT>
        <COMPU-CONST>
          <VT>SensorError</VT>
        </COMPU-CONST>
      </COMPU-SCALE>
      <COMPU-SCALE>
        <LOWER-LIMIT INTERVAL-TYPE="CLOSED">351</LOWER-LIMIT>
        <UPPER-LIMIT INTERVAL-TYPE="CLOSED">351</UPPER-LIMIT>
        <COMPU-CONST>
          <VT>SignalNotAvailable</VT>
        </COMPU-CONST>
      </COMPU-SCALE>
    </COMPU-SCALES>
  </COMPU-INTERNAL-TO-PHYS>
</COMPU-METHOD>
```

When you import the CompuMethod into a model, the importer creates a Simulink.NumericType with linear scaling. To modify the linear scaling, open the Simulink.NumericType data object and modify its fields.

Simulink.NumericType: myType	
Data type mode:	Fixed-point: slope and bias scaling
Signedness:	Signed
Word length:	16
Slope:	2
Bias:	30

For read-only access to the TEXTTABLE scaling information, use AUTOSAR property get function calls similar to the following:

```

>> arProps=autosar.api.getAUTOSARProperties('mySWC');
>> % Get literals for COMPU_myType TEXTTABLE scales
>> get(arProps, '/simple_ar_package/simple_ar_dt/COMPU_myType', 'CellOfEnums')
ans =
    'SensorError'    'SignalNotAvailable'
>> % Get internal values for COMPU_myType TEXTTABLE scales
>> get(arProps, '/simple_ar_package/simple_ar_dt/COMPU_myType', 'IntValues')
ans =
    350    351

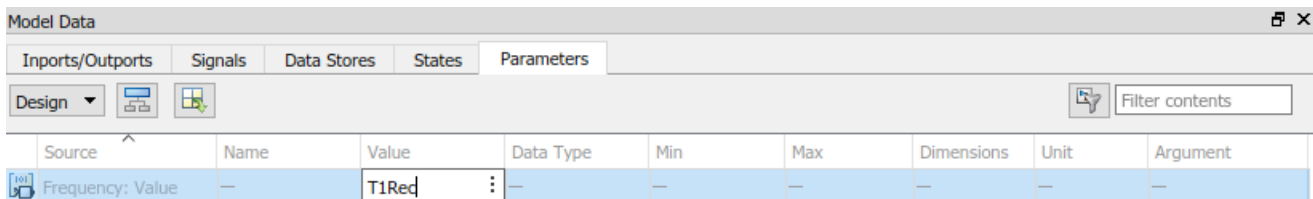
```


Configure Rational Function CompuMethod for Dual-Scaled Parameter

For an AUTOSAR dual-scaled parameter, which stores two scaled values of the same physical value, the software generates the CompuMethod category RAT_FUNC. The computation method can be a first-order rational function.

To configure and generate a dual-scaled parameter:

- 1 Open an AUTOSAR model. For the purposes of this example, create a Constant block from which to reference an AUTOSAR dual-scaled parameter. In the model, connect the Constant block to a Simulink output.
- 2 Open the Model Data Editor (on the **Modeling** tab, click **Model Data Editor**) and select the **Parameters** tab. Find the parameter entry for the Constant block. Use the **Value** column to reference the name of a dual-scaled parameter. This example uses the parameter name T1Rec.



- 3 Create the T1Rec data object. In the Model Data Editor, to the right of the value T1Rec, click the action button  and select **Create**.

In the Create New Data dialog box, set **Value** to `AUTOSAR.DualScaledParameter` and click **Create**. An `AUTOSAR.DualScaledParameter` data object appears in the base workspace. The dual-scaled parameter property dialog box opens.

- 4 Configure the attributes of the dual-scaled parameter T1Rec. Execute the following MATLAB code. The code sets up a conversion from an internal calibration time value to a physical frequency (time reciprocal) value.

```

% Conversion from Time to Frequency
% F = 1/T
% In Other Words F = (0*T + 1)/(1*T+0);
T1Rec.CompuMethodName = 'CM3'; %Specify AUTOSAR CompuMethod name
T1Rec.DataType = 'fixdt(1,32,0.01,0)';
T1Rec.CalToMainCompuNumerator=1;
T1Rec.CalToMainCompuDenominator=[1 0];
T1Rec.CalibrationMin = 0.001;
T1Rec.CalibrationMax = 1.0;
T1Rec.CalibrationValue = 0.1500;
T1Rec.CoderInfo.StorageClass = 'Custom';
T1Rec.CoderInfo.Identifier = '';
T1Rec.CoderInfo.CustomStorageClass = 'InternalCalPrm';
T1Rec.CoderInfo.CustomAttributes.PerInstanceBehavior = ...
    'Parameter shared by all instances of the Software Component';
T1Rec.Description = '';
% T1Rec.Min = [];
% T1Rec.Max = [];

```

```
T1Rec.Unit = '';
T1Rec.CalibrationDocUnits = 'm/s^2';
```

- 5 Inspect the property dialog box for the dual-scaled parameter T1Rec. Here are the main attributes set by the MATLAB code.

- 6 Here are the calibration attributes set by the MATLAB code.

- 7 If CompuMethod direction is not already set to bidirectional in the AUTOSAR properties, use the AUTOSAR Dictionary, **XML Options** view, to set it.
- 8 Generate code from the model.

When you generate code from the model, the ARXML exporter generates a CompuMethod of category RAT_FUNC.

```
<COMPU-METHOD UUID="...">
  <SHORT-NAME>CM3</SHORT-NAME>
  <CATEGORY>RAT_FUNC</CATEGORY>
  <UNIT-REF DEST="UNIT">/myModel_pkg/myModel_dt/m_s_</UNIT-REF>
  <COMPU-INTERNAL-TO-PHYS>
    <COMPU-SCALES>
      <COMPU-SCALE>
        <COMPU-RATIONAL-COEFFS>
          <COMPU-NUMERATOR>
            <V>-100</V>
          </COMPU-NUMERATOR>
          <COMPU-DENOMINATOR>
            <V>0</V>
            <V>-1</V>
          </COMPU-DENOMINATOR>
        </COMPU-RATIONAL-COEFFS>
      </COMPU-SCALE>
    </COMPU-SCALES>
  </COMPU-INTERNAL-TO-PHYS>
  <COMPU-PHYS-TO-INTERNAL>
    <COMPU-SCALES>
```

```

    <COMPU-SCALE>
      <COMPU-RATIONAL-COEFFS>
        <COMPU-NUMERATOR>
          <V>100</V>
        </COMPU-NUMERATOR>
        <COMPU-DENOMINATOR>
          <V>0</V>
          <V>1</V>
        </COMPU-DENOMINATOR>
      </COMPU-RATIONAL-COEFFS>
    </COMPU-SCALE>
  </COMPU-SCALES>
</COMPU-PHYS-TO-INTERNAL>
</COMPU-METHOD>

```

The CompuMethod is referenced from the application data type generated for T1Rec.

```

<APPLICATION-PRIMITIVE-DATA-TYPE UUID="...">
  <SHORT-NAME>T1Rec_DualScaled</SHORT-NAME>
  <CATEGORY>VALUE</CATEGORY>
  <SW-DATA-DEF-PROPS>
    <SW-DATA-DEF-PROPS-VARIANTS>
      <SW-DATA-DEF-PROPS-CONDITIONAL>
        <SW-CALIBRATION-ACCESS>READ-WRITE</SW-CALIBRATION-ACCESS>
        <COMPU-METHOD-REF DEST="COMPU-METHOD">/mymodel_pkg/mymodel_dt/CM3</COMPU-METHOD-REF>
        <DATA-CONSTR-REF DEST="DATA-CONSTR">/mymodel_pkg/mymodel_dt/AppDataTypes/
          DataConstrs/DC_T1Rec_DualScaled</DATA-CONSTR-REF>
      </SW-DATA-DEF-PROPS-CONDITIONAL>
    </SW-DATA-DEF-PROPS-VARIANTS>
  </SW-DATA-DEF-PROPS>
</APPLICATION-PRIMITIVE-DATA-TYPE>

```

The application data type T1Rec_DualScaled is referenced from the parameter data prototype generated for T1Rec.

```

<PARAMETER-DATA-PROTOTYPE UUID="...">
  <SHORT-NAME>T1Rec</SHORT-NAME>
  <SW-DATA-DEF-PROPS>
    <SW-DATA-DEF-PROPS-VARIANTS>
      <SW-DATA-DEF-PROPS-CONDITIONAL>
        <SW-CALIBRATION-ACCESS>READ-WRITE</SW-CALIBRATION-ACCESS>
        <SW-IMPL-POLICY>STANDARD</SW-IMPL-POLICY>
      </SW-DATA-DEF-PROPS-CONDITIONAL>
    </SW-DATA-DEF-PROPS-VARIANTS>
  </SW-DATA-DEF-PROPS>
  <TYPE-TREF DEST="APPLICATION-PRIMITIVE-DATA-TYPE">/mymodel_pkg/mymodel_dt/AppDataTypes/
    T1Rec_DualScaled</TYPE-TREF>
  ...
</PARAMETER-DATA-PROTOTYPE>

```

See Also

Related Examples

- “Import AUTOSAR XML Descriptions Into Simulink” on page 3-13
- “Configure AUTOSAR Package for Component, Interface, CompuMethod, or SwAddrMethod” on page 4-94
- “Configure AUTOSAR XML Options” on page 4-43
- “Configure AUTOSAR Code Generation” on page 5-7

More About

- “CompuMethod Categories for Data Types” on page 2-48
- “AUTOSAR Component Configuration” on page 4-3

Configure AUTOSAR Data Types Export

The AUTOSAR standard defines an approach to AUTOSAR data types in which base data types are mapped to implementation data types and application data types. Application and implementation data types separate application-level physical attributes, such as real-world range of values, data structure, and physical semantics, from implementation-level attributes, such as stored-integer minimum and maximum and specification of a primitive-type (integer, Boolean, real, and so on). For information about modeling data types, see “Model AUTOSAR Data Types” on page 2-43.

The software supports AUTOSAR data types in Simulink originated and round-trip workflows:

- For AUTOSAR components originated in Simulink, the software generates AUTOSAR application, implementation, and base types to preserve the information contained within Simulink data types.
- For round-trip workflows involving AUTOSAR components originated outside MATLAB, the ARXML importer and exporter preserve data type information and mapping for each imported AUTOSAR data type.

For AUTOSAR data types originated in Simulink, you can control some aspects of data type export. For example, you can control when application data types are generated, specify the AUTOSAR package and short name exported for AUTOSAR data type mapping sets, or force ARXML export of internal data constraints for AUTOSAR implementation data types.

Control Application Data Type Generation

For AUTOSAR data types created in Simulink, by default, the software generates application base types only for fixed-point data types and enumerated data types with storage types. If you want to override the default behavior for generating application types, you can configure the ARXML exporter to generate an application type, along with the implementation type and base type, for each exported AUTOSAR data type. Use the XML options parameter **ImplementationDataReference** (XMLOptions property `ImplementationDataReference`), for which you can specify the following values:

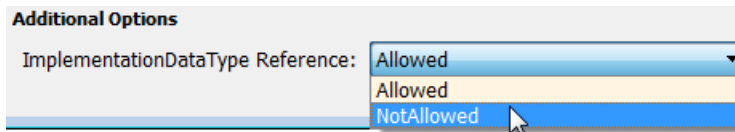
- `Allowed` (default) — Allow direct reference of implementation types in the generated ARXML code. If an application data type is not strictly required to describe an AUTOSAR data type, use an implementation data type reference.
- `NotAllowed` — Do not allow direct reference of implementation data types in the generated ARXML code. Generate an application data type for each AUTOSAR data type.

Note The software always generates an implementation type and an application type in the generated ARXML when exporting a `Simulink.ValueType` object.

To set the `ImplementationDataReference` property in the MATLAB Command Window, use an AUTOSAR property set function call similar to the following:

```
hModel = 'autosar_sw_c_expcns';
openExample(hModel);
arProps=autosar.api.getAUTOSARProperties(hModel);
set(arProps,'XmlOptions','ImplementationTypeReference','NotAllowed');
get(arProps,'XmlOptions','ImplementationTypeReference')
```

To set the `ImplementationDataReference` property in the AUTOSAR Dictionary, select **XML Options**. Select a value for parameter **ImplementationDataReference**. Click **Apply**.



Configure DataTypeMappingSet Package and Name

For AUTOSAR software components created in Simulink, you can control the AUTOSAR package and short name exported for AUTOSAR data type mapping sets. To configure the data type mapping set package for export, set the XMLOptions property DataTypeMappingPackage using the AUTOSAR Dictionary or the AUTOSAR property set function.



```
hModel = 'autosar_swc_expfncns';
openExample(hModel);
arProps=autosar.api.getAUTOSARProperties(hModel);
set(arProps,'XmlOptions','DataTypeMappingPackage','/pkg/dt/DataTypeMappings');
get(arProps,'XmlOptions','DataTypeMappingPackage')
```

The exported ARXML code uses the specified package. The default mapping set short-name is the component name ASWC prefixed to DataTypeMappingsSet.

```
<DATA-TYPE-MAPPING-REFS>
  <DATA-TYPE-MAPPING-REF DEST="DATA-TYPE-MAPPING-SET">
    /pkg/dt/DataTypeMappings/ASWCDataTypeMappingsSet</DATA-TYPE-MAPPING-REF>
</DATA-TYPE-MAPPING-REFS>
...
<AR-PACKAGE>
  <SHORT-NAME>DataTypeMappings</SHORT-NAME>
  <ELEMENTS>
    <DATA-TYPE-MAPPING-SET UUID="...">
      <SHORT-NAME>ASWCDataTypeMappingsSet</SHORT-NAME>
    ...
  </DATA-TYPE-MAPPING-SET>
</ELEMENTS>
</AR-PACKAGE>
```

You can specify a short name for a data type mapping set using the AUTOSAR property function `addPackageableElement`. The following example specifies a custom data type mapping set package and name using MATLAB commands.

```
% Add a new data type mapping set
modelName = 'autosar_swc_expfncns';
openExample(modelName);
propObj = autosar.api.getAUTOSARProperties(modelName);
newMappingSetPath = '/myPkg/mySubpkg/MyMappingSets';
newMappingSetName = 'MappingSetName';
newMappingSet = [newMappingSetPath '/' newMappingSetName];
addPackageableElement(propObj,'DataTypeMappingSet',newMappingSetPath,newMappingSetName);

% Configure the component behavior to use the new data type mapping set
swc = get(propObj,'XmlOptions','ComponentQualifiedNames');
ib = get(propObj,swc,'Behavior','PathType','FullyQualified');
set(propObj,ib,'DataTypeMapping',newMappingSet);
```

```
% Force generation of application data types
set(propObj,'XmlOptions','ImplementationTypeReference','NotAllowed');

% Build
slbuild(modelName);
```

The exported ARXML code uses the specified package and name, as shown below.

```
<INTERNAL-BEHAVIORS>
  <SWC-INTERNAL-BEHAVIOR UUID="...">
    <SHORT-NAME>IB</SHORT-NAME>
    <DATA-TYPE-MAPPING-REFS>
      <DATA-TYPE-MAPPING-REF DEST="DATA-TYPE-MAPPING-SET">
        /myPkg/mySubpkg/MyMappingSets/MappingSetName</DATA-TYPE-MAPPING-REF>
      </DATA-TYPE-MAPPING-REFS>
    ...
  </SWC-INTERNAL-BEHAVIOR>
</INTERNAL-BEHAVIORS>
```

Initialize Data with ApplicationValueSpecification

To initialize AUTOSAR data objects typed by application data type, the AUTOSAR standard (R4.1 or later) requires AUTOSAR application value specifications (ApplicationValueSpecifications). Embedded Coder provides the following support:

- The ARXML importer uses ApplicationValueSpecifications found in imported ARXML files to initialize the corresponding data objects in the Simulink model.
- Code generation exports ARXML code that uses ApplicationValueSpecifications to specify initial values for AUTOSAR data.

For AUTOSAR parameters typed by implementation data type, code generation exports ARXML code that uses NumericalValueSpecifications and (for enumerated types) TextValueSpecifications to specify initial values. If initial values for parameters specify multiple values, generated code uses ArrayValueSpecifications.

Configure AUTOSAR Internal Data Constraints Export

AUTOSAR applications use data constraints to implement limits on data types and provide a controlled range of possible values. Internal data constraints represent minimum and maximum values for implementation data types, reflecting the internal or machine view of the data.

By default, code generation does not export internal data constraint information for AUTOSAR implementation data types in ARXML code. If you want to force export of internal data constraints for implementation data types, select the XML option **Internal DataConstraints Export**.

If you select **Internal DataConstraints Export**, the exporter generates internal data constraints into an AUTOSAR package with a default name, `DataConstrs`, at a fixed location under the AUTOSAR data type package. Optionally, use the XML option **Internal DataConstraints Package** to specify a different AUTOSAR package name and path.

To configure export of AUTOSAR internal data constraint information in your model:

- 1 Open the AUTOSAR Dictionary. On the **AUTOSAR** tab, select **Code Interface > AUTOSAR Dictionary**.
- 2 Select **XML Options**. In the XML options view, under **Additional Options**, select **Internal DataConstraints Export**.

- 3 Optionally, under **Additional Packages**, enter a package path for **Internal DataConstraints Package**.

Internal DataConstraints Package:

Additional Options

ImplementationDataType Reference:

SwCalibrationAccess DefaultValue:

CompuMethod Direction:

Internal DataConstraints Export:

- 4 Build the model and inspect the generated code. Here is an example of an AUTOSAR internal data constraint exported to ARXML code.

```
<AR-PACKAGE>
  <SHORT-NAME>IDC</SHORT-NAME>
  <ELEMENTS>
    ...
    <DATA-CONSTR UUID="...">
      <SHORT-NAME>DC_SInt8</SHORT-NAME>
      <DATA-CONSTR-RULES>
        <DATA-CONSTR-RULE>
          <INTERNAL-CONSTRS>
            <LOWER-LIMIT INTERVAL-TYPE="CLOSED">-128</LOWER-LIMIT>
            <UPPER-LIMIT INTERVAL-TYPE="CLOSED">127</UPPER-LIMIT>
          </INTERNAL-CONSTRS>
        </DATA-CONSTR-RULE>
      </DATA-CONSTR-RULES>
    </DATA-CONSTR>
  </ELEMENTS>
</AR-PACKAGE>
```

Alternatively, you can programmatically configure the AUTOSAR XML options **Internal DataConstraints Export** and **Internal DataConstraints Package**. For example:

```
arProps = autosar.api.getAUTOSARProperties(hModel);
set(arProps, 'XmlOptions', 'InternalDataConstraintExport', true);
set(arProps, 'XmlOptions', 'InternalDataConstraintPackage', '/pkg/misc/IDC');
```

For more information, see “Configure AUTOSAR XML Options” on page 4-43.

See Also

Related Examples

- “Configure AUTOSAR XML Options” on page 4-43
- “Configure AUTOSAR Packages” on page 4-84
- “Configure AUTOSAR Code Generation” on page 5-7

More About

- “Model AUTOSAR Data Types” on page 2-43

Automatic AUTOSAR Data Type Generation

When you generate AUTOSAR-compliant C code for an AUTOSAR component model, Embedded Coder generates AUTOSAR platform data types in the code. AUTOSAR type generation allows you to generate AUTOSAR platform data types for top models, referenced models, and shared utilities without configuring Simulink data type replacement.

The AUTOSAR standard defines platform data types for use by AUTOSAR software components. In Simulink, you can model AUTOSAR data types used in elements such as data elements, operation arguments, calibration parameters, measurement variables, and inter-runnable variables. To model AUTOSAR data types, use corresponding Simulink built-in data types. For more information, see “Model AUTOSAR Data Types” on page 2-43.

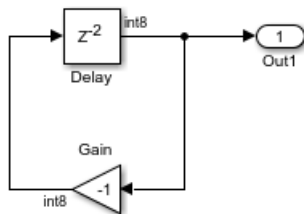
When you build your AUTOSAR model, C code generation replaces Simulink data types with corresponding AUTOSAR platform data types. You can configure the platform type names in the **XMLOptions** of the AUTOSAR Dictionary. For more information, see “AUTOSAR Platform Types” on page 4-47.

Simulink Data Type	AUTOSAR 3.x Platform Type	AUTOSAR 4.x Platform Type
boolean	Boolean	boolean
single	Float	float32
double	Double	float64
int8	SInt8	sint8
int16	SInt16	sint16
int32	SInt32	sint32
int64	SInt64	sint64
uint8	UInt8	uint8
uint16	UInt16	uint16
uint32	UInt32	uint32
uint64	UInt64	uint64

Not recommended starting in R2023a

Support for AUTOSAR 3.x platform names will be removed in a future release.

For example, suppose that you create a simple AUTOSAR model containing Gain and Delay blocks, and set the Gain block parameter **Output data type** to `int8`. When you generate code, in place of Simulink data type `int8`, the AUTOSAR-compliant C code references AUTOSAR data type `sint8`.



```
void Runnable_Step(void)
{
    sint8 rtb_Delay;
```

```
...  
simple_DW.Delay_DSTATE[1] = (sint8)-rtb_Delay;  
}
```

See Also

Related Examples

- “Configure AUTOSAR Code Generation” on page 5-7

More About

- “Model AUTOSAR Data Types” on page 2-43

Configure Parameters and Signals for AUTOSAR Calibration and Measurement

Configure Simulink® model workspace parameters and signals for AUTOSAR run-time calibration and measurement.

Map Model Workspace Parameters to AUTOSAR Parameters

Open the example model `autosar_sw_counter.slx`.

```
open_system('autosar_sw_counter')
```


From the **Apps** tab, open the AUTOSAR Component Designer app. Open the Code Mappings editor and select the **Parameters** tab. Expand the list of available model parameters and select **INC**. In the **Mapped To** drop-down list, select `ConstantMemory`.

The Simulink model 'autosar_sw_counter' consists of the following components and connections:

- INC** block: An incrementer block that takes an input and adds 1 to it.
- sum_out** block: A summing junction that adds the output of the **INC** block to the output of the **1/z** block.
- LIMIT** block: A limit block that takes the output of the summing junction and compares it against a threshold.
- equal_to_count** block: A comparison block that checks if the output of the summing junction is equal to a specified count.
- RESET** block: A reset block that takes the output of the **equal_to_count** block and sends a signal to the **switch_out** block.
- switch_out** block: A switch block that takes the output of the summing junction and the signal from the **RESET** block to control its output.
- 1/z** block: A discrete-time integrator block that takes the output of the switch and divides it by the complex variable z .
- Amplifier** block: An amplifier block that takes the output of the switch and the output of the **equal_to_count** block as inputs. It has an input port labeled 'In' and an output port labeled 'Out'. The output is connected to a constant value of 1.

This model maps Simulink model workspace parameters and signals to AUTOSAR parameters and variables for AUTOSAR run-time calibration and measurement.

Code Mappings - AUTOSAR SW Component	
Source	Mapped To
Model Parameter Arguments (0)	
Model Parameters (4)	
INC	ConstantMemory
K	SharedParameter
LIMIT	Auto
RESET	Auto

To view and modify AUTOSAR attributes for the constant memory, click the  icon. For more information about parameter code and calibration attributes, see “Map Model Workspace Parameters to AUTOSAR Component Parameters” on page 4-54.

Const
 Volatile
 AdditionalNativeTypeQualifier
 SwAddrMethod
 Calibration attributes
 SwCalibrationAccess
 DisplayFormat
 LongName

If you have Simulink Coder and Embedded Coder software, you can generate algorithmic C code and AUTOSAR XML (ARXML) component descriptions. You can test the generated code in Simulink or integrate the code and descriptions into an AUTOSAR run-time environment.

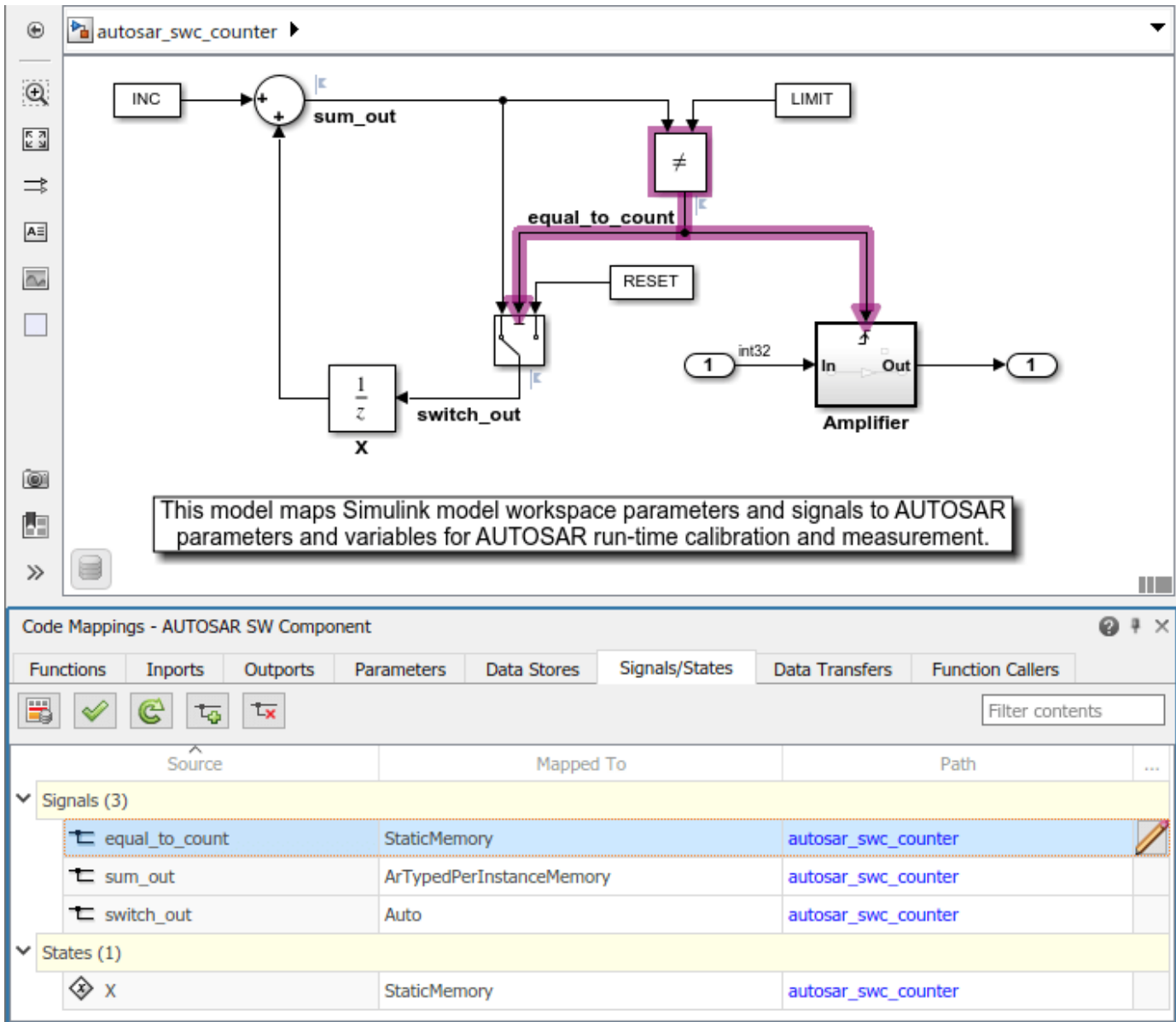
When you generate code:


- Exported ARXML files contain `CONSTANT-MEMORYS` descriptions for parameters that you configured as `ConstantMemory`. In the AUTOSAR run-time environment, calibration tools can access AUTOSAR `ConstantMemory` blocks for calibration and measurement.
- Generated C code declares and references the constant memory parameters.

Map Simulink Signals and States to AUTOSAR Variables

Open the example model `autosar_swc_counter.slx`, if it is not already open.

From the **Apps** tab, open the AUTOSAR Component Designer app. Open the Code Mappings editor and select the **Signals/States** tab. Expand the list of available signals and select `equal_to_count`. Selecting a signal highlights the signal in the model diagram. In the **Mapped To** drop-down list, select `StaticMemory`.



To view and modify AUTOSAR attributes for the static memory, click the  icon. For more information about signal code and calibration attributes, see “Map Block Signals and States to AUTOSAR Variables” on page 4-58.

The image shows a configuration dialog box for an AUTOSAR component. It contains the following fields and controls:

- ShortName:** A text box containing the value "SM_equal_to_count".
- Volatile:** A checked checkbox.
- AdditionalNativeTypeQualifier:** A text box containing the value "my_qualifier".
- SwAddrMethod:** A dropdown menu with "VAR" selected.
- Calibration attributes:** A section header for a group of fields.
 - SwCalibrationAccess:** A dropdown menu with "ReadOnly" selected.
 - DisplayFormat:** An empty text box.
 - LongName:** An empty text box.
- Open in Property Inspector:** A button at the bottom of the dialog.

If you have Simulink Coder and Embedded Coder software, you can generate algorithmic C code and AUTOSAR XML (ARXML) component descriptions. You can test the generated code in Simulink or integrate the code and descriptions into an AUTOSAR run-time environment.

When you generate code:

- Exported ARXML files contain **STATIC-MEMORYS** descriptions for signals and states that you configured as `StaticMemory`. In the AUTOSAR run-time environment, calibration tools can access AUTOSAR `StaticMemory` blocks for calibration and measurement.
- Generated C code declares and references the static memory variables.

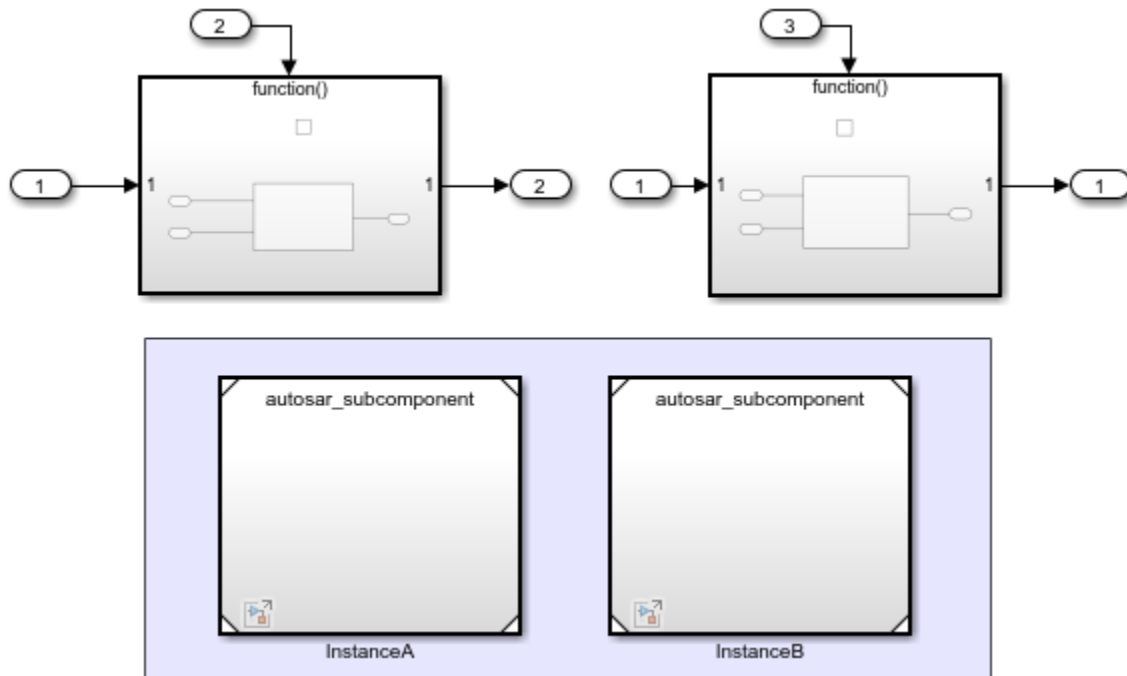
Related Links

- “Map Model Workspace Parameters to AUTOSAR Component Parameters” on page 4-54
- “Configure AUTOSAR Constant Memory” on page 4-210
- “Map Block Signals and States to AUTOSAR Variables” on page 4-58
- “Configure AUTOSAR Static Memory” on page 4-206
- “Configure AUTOSAR Per-Instance Memory” on page 4-201

Configure Subcomponent Data for AUTOSAR Calibration and Measurement

For any model in an AUTOSAR model reference hierarchy, you can configure the model data for run-time calibration and measurement. In submodels referenced from AUTOSAR software component models, you can map parameters, data stores, signals, and states to AUTOSAR parameters and variables. Submodel mapped internal data can be used in AUTOSAR memory sections, and is available for software-in-the-loop (SIL) and processor-in-the-loop (PIL) testing from the top model or calibration in the AUTOSAR run-time environment.

In this example, AUTOSAR component model `autosar_component` contains two instances of `autosar_subcomponent`.



Each instance of `autosar_subcomponent` receives a separate set of parameter values, which you can view in the **Instance parameters** tab of the Model block parameters dialog box.

To configure subcomponent data for run-time calibration and measurement, open the submodel standalone, that is, in a separate model window. Use the Code Mappings editor to:

- Map submodel parameters to AUTOSAR component `PerInstanceParameters`.
- Map submodel signals, states, and data stores to AUTOSAR `ArTypedPerInstanceMemory` variables.
- Set AUTOSAR code and calibration attributes for the submodel internal data.

To generate C code and AUTOSAR XML (ARXML) files that support run-time calibration of the submodel internal data, open and build the component model that references the submodel.

Map Submodel Parameters to AUTOSAR Component PerInstanceParameters

Open the example model `autosar_subcomponent`.

```
open_system('autosar_subcomponent');
```

At the top level is a Simulink Function, for which the top model provides per-instance parameters. Open the Simulink Function.


From the **Apps** tab, open the AUTOSAR Component Designer app. Open the Code Mappings editor and select the **Parameters** tab. The example submodel has four model workspace parameters, including a lookup table parameter. To map each Simulink parameter to an AUTOSAR per-instance parameter, select each parameter and, in the **Mapped To** drop-down list, select **PerInstanceParameter**.

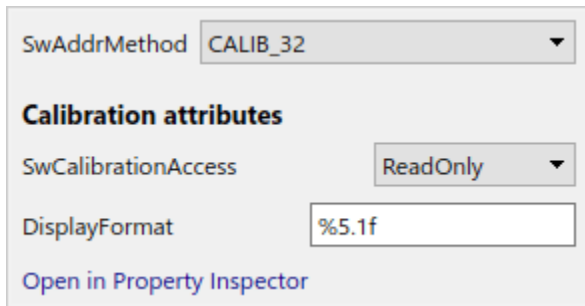
The screenshot displays the Simulink Function editor for the `autosar_subcomponent` model. The main workspace shows a Simulink Function block with the following components:

- Inputs: `max_press`, `engine_speed`, `u1`, `DSM_local`, `u2`, and `max_speed`.
- 2-D Lookup Table (LUT) block: Receives `u1` and `u2` as inputs and outputs `lutOutSig`.
- Summing Junction: Receives inputs from `max_press`, `engine_speed`, `lutOutSig`, `DSM_local`, `u2`, and `max_speed`. The output is `sumSignal`.
- Delay Block (z^{-1}): Receives `sumSignal` and outputs `sumSignal` (labeled as `sumSignal` in the diagram).
- Gain Block (1): Receives `sumSignal` and outputs `subSystemOut`.
- Output: `subSystemOut` is connected to a scope block `y`.

The Code Mappings editor (Submodel) is open at the bottom, showing the following table:

Source	Mapped To	...
Model Parameter Arguments (4)		
<code>engine_speed</code>	PerInstanceParameter	
<code>max_press</code>	PerInstanceParameter	
<code>max_speed</code>	PerInstanceParameter	
PressureEstimation	PerInstanceParameter	
Model Parameters (0)		

Select the parameter `engine_speed`. To view and modify additional AUTOSAR attributes for the per-instance parameter, click the  icon. A properties dialog box opens.



SwAddrMethod CALIB_32

Calibration attributes

SwCalibrationAccess ReadOnly

DisplayFormat %5.1f

[Open in Property Inspector](#)

For each AUTOSAR `PerInstanceParameter`, you can modify the `SwAddrMethod` (AUTOSAR memory section), the calibration data access, and the calibration data display format. For more information about parameter code and calibration attributes, see “Map Submodel Parameters to AUTOSAR Component Parameters” on page 4-68.

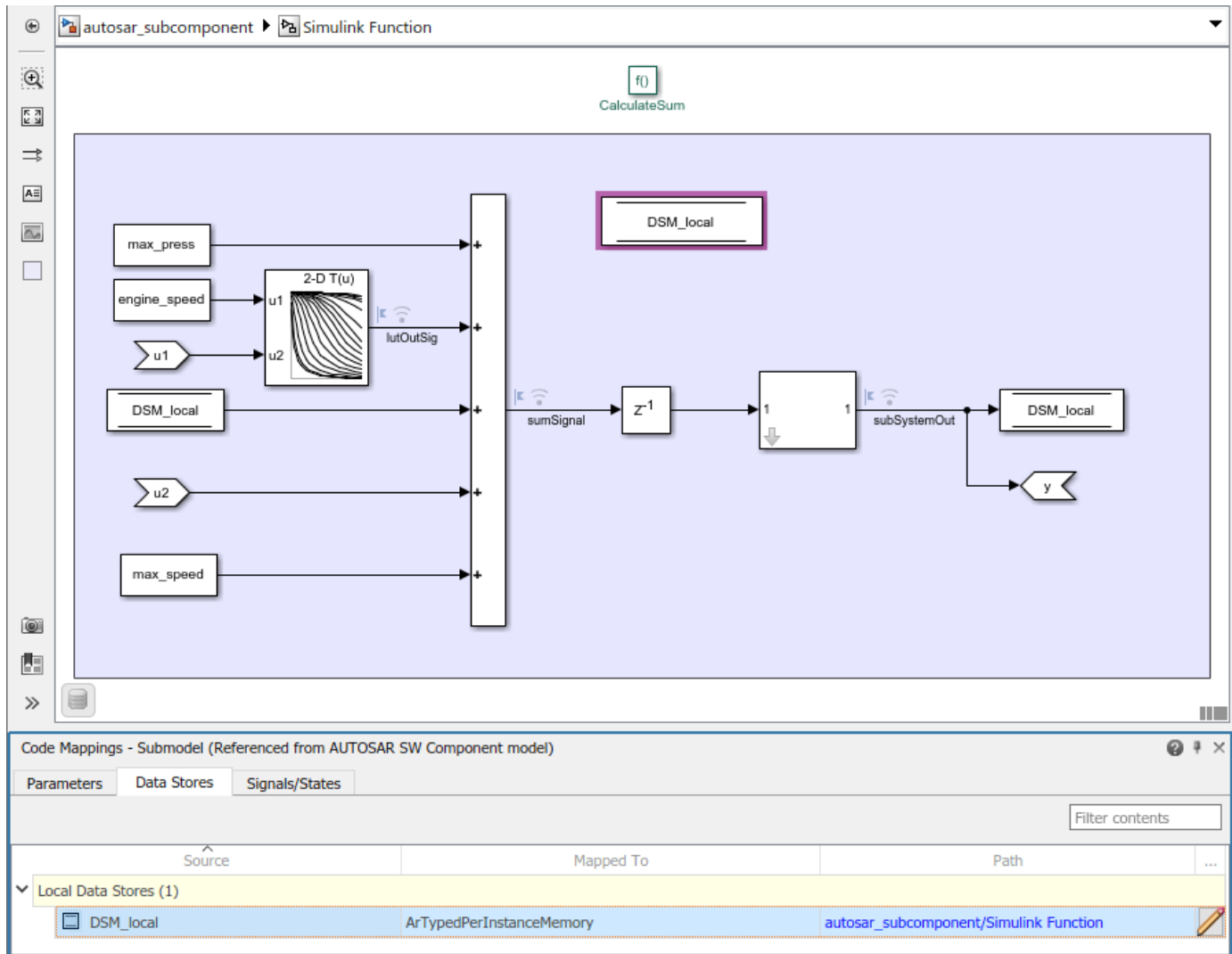
When you generate code from the component model that references the submodel:


- Exported ARXML files contain PER-INSTANCE-PARAMETERS descriptions for submodel parameters that you configured as AUTOSAR component `PerInstanceParameters`, and descriptions of the `SwAddrMethods` referenced in the submodel.
- Generated C code references the submodel AUTOSAR per-instance parameters.
- The model build generates macros that provide access to the submodel data for SIL and PIL testing and calibration in the AUTOSAR run-time environment.

Map Submodel Data Stores to AUTOSAR `ArTypedPerInstanceMemory` Variables

If they are not already open, open the example model `autosar_subcomponent`, the top-level Simulink Function, the AUTOSAR Component Designer app, and the Code Mappings editor.

In the Code Mappings editor, select the **Data Stores** tab. The example submodel has a Data Store Memory block named `DSM_local`. To map the Simulink data store to an AUTOSAR-typed per-instance memory variable, select `DSM_local`. Selecting a data store highlights the Data Store Memory block in the model diagram. In the **Mapped To** drop-down list, select `ArTypedPerInstanceMemory`.



To view and modify additional AUTOSAR attributes for the per-instance memory, click the  icon. A properties dialog box opens.

ShortName

SwAddrMethod

Calibration attributes

SwCalibrationAccess

DisplayFormat

[Open in Property Inspector](#)

For each AUTOSAR ArTypedPerInstanceMemory variable, you can modify the ARXML short name, the SwAddrMethod (AUTOSAR memory section), the calibration data access, and the calibration data

display format. For more information about data store code and calibration attributes, see “Map Submodel Data Stores to AUTOSAR Variables” on page 4-69.

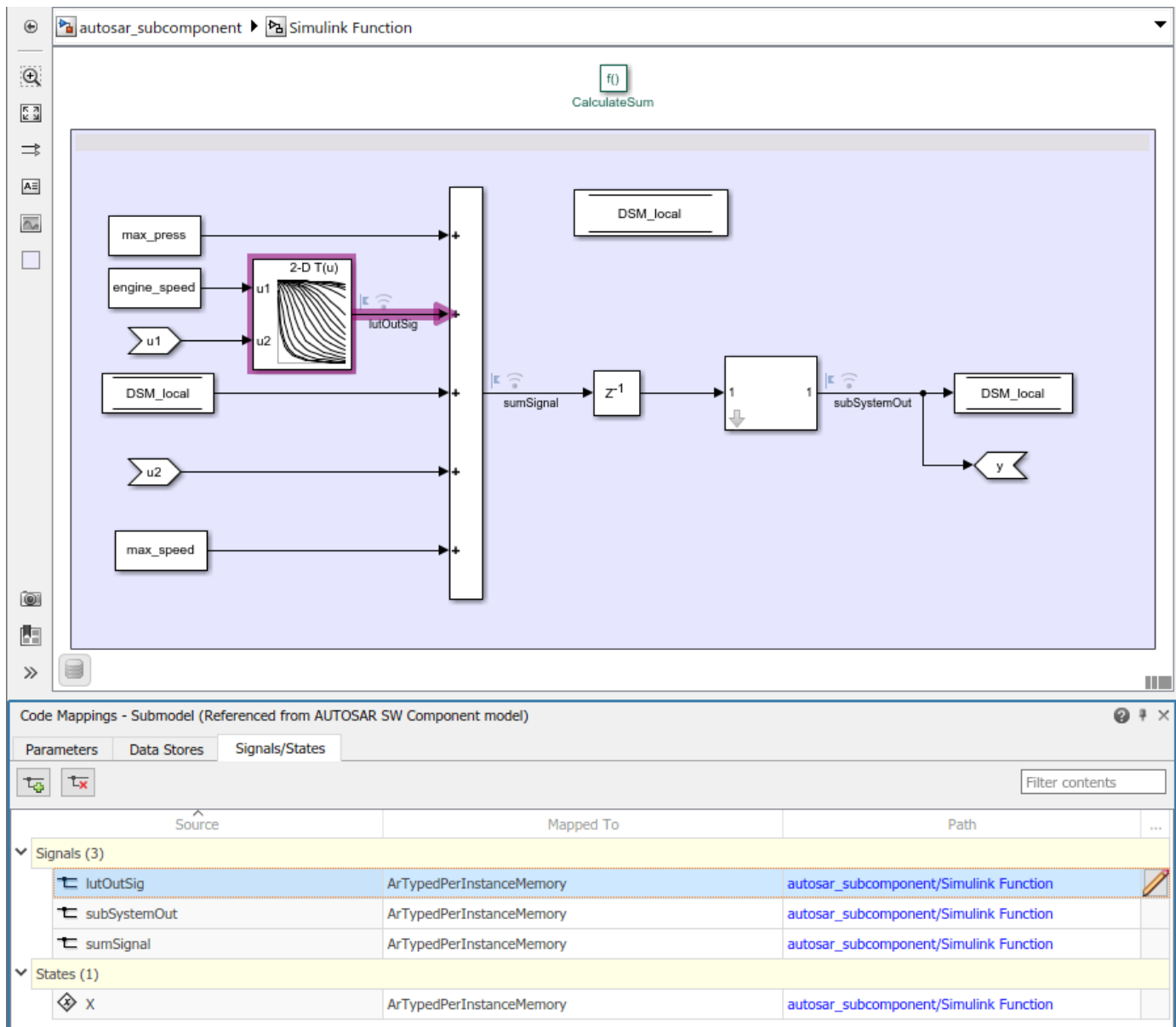
When you generate code from the component model that references the submodel:

- Exported ARXML files contain AR-TYPED-PER-INSTANCE-MEMORYS descriptions for submodel data stores that you configured as ArTypedPerInstanceMemory variables, and descriptions of the SwAddrMethods referenced in the submodel.
- Generated C code references the submodel AUTOSAR-typed per-instance memory variables.
- The model build generates macros that provide access to the submodel data for SIL and PIL testing and calibration in the AUTOSAR run-time environment.

Map Submodel Signals and States to AUTOSAR ArTypedPerInstanceMemory Variables


If they are not already open, open the example model `autosar_subcomponent`, the top-level Simulink Function, the AUTOSAR Component Designer app, and the Code Mappings editor.

In the Code Mappings editor, select the **Signals/States** tab. The **Signals/States** tab lists each Simulink block signal and state that you can map to an AUTOSAR variable. The example submodel has three mappable signals and one state. To map each Simulink signal and state to an AUTOSAR-typed per-instance memory variable, select each signal or state. Selecting a signal or state highlights the element in the model diagram. In the **Mapped To** drop-down list, select `ArTypedPerInstanceMemory`.



To make additional Simulink block signals available for mapping, use a Code Mappings editor button or a model cue:

- In the model canvas, select one or more signals. Open the Code Mappings editor, **Signals/States** tab, and click the **Add** button.
- In the model canvas, select a signal. Place your cursor over the displayed ellipsis and select model cue **Add selected signals to code mappings**.

In the Code Mappings editor, select the signal `lutOutSig`. To view and modify additional AUTOSAR attributes for the per-instance memory, click the  icon. A properties dialog box opens.

ShortName	<input type="text" value="lutsig"/>
SwAddrMethod	<input type="text" value="VAR_INIT_32"/>
Calibration attributes	
SwCalibrationAccess	<input type="text" value="ReadWrite"/>
DisplayFormat	<input type="text" value="%1d"/>
Open in Property Inspector	

For each AUTOSAR `ArTypedPerInstanceMemory` variable, you can modify the ARXML short name, the `SwAddrMethod` (AUTOSAR memory section), the calibration data access, and the calibration data display format. For more information about signal and state code and calibration attributes, see “Map Submodel Signals and States to AUTOSAR Variables” on page 4-70.

When you generate code from the component model that references the submodel:

- Exported ARXML files contain AR-TYPED-PER-INSTANCE-MEMORYS descriptions for submodel signals and states that you configured as `ArTypedPerInstanceMemory`, and descriptions of the `SwAddrMethods` referenced in the submodel.
- Generated C code references the submodel AUTOSAR-typed per-instance memory variables.
- The model build generates macros that provide access to the submodel data for SIL and PIL testing and calibration in the AUTOSAR run-time environment.

Related Links

- “Map Calibration Data for Submodels Referenced from AUTOSAR Component Models” on page 4-65
- “Generate Submodel Data Macros for Verification and Deployment” on page 4-73
- “Configure Model Workspace Parameters as AUTOSAR Per-Instance Parameters” on page 4-213
- “Configure AUTOSAR Per-Instance Memory” on page 4-201
- Code Mappings Editor

Configure AUTOSAR Data for Calibration and Measurement

In Simulink, you can import and export AUTOSAR software data definition properties and modify the properties for some forms of AUTOSAR data.

About Software Data Definition Properties (SwDataDefProps)

Embedded Coder supports ARXML import and export of the following AUTOSAR software data definition properties (SwDataDefProps):

- Software calibration access (SwCalibrationAccess) — Specifies calibration and measurement tool access to a data object.
- Display format (DisplayFormat) — Specifies calibration and measurement display format for a data object.
- Software address method (SwAddrMethod) — Specifies a method to access a data object (for example, a measurement or calibration parameter) according to a given address. Used to group data in memory for access by run-time calibration and measurement tools.
- Software alignment (SwAlignment) — Specifies the intended alignment of a data object within a memory section.
- Software implementation policy (SwImplPolicy) — Specifies the implementation policy for a data object, regarding consistency mechanisms of variables.
- Software record layout (SwRecordLayout) — Specifies how to serialize data in the memory of an AUTOSAR ECU.

In the Simulink environment, you can directly modify software data definition properties for some forms of AUTOSAR data. You cannot modify the SwImplPolicy or SwRecordLayout properties, but the properties are exported in ARXML code.

For more information, see “Configure SwCalibrationAccess” on page 4-262, “Configure DisplayFormat” on page 4-264, “Configure SwAddrMethod” on page 4-267, “Configure SwAlignment” on page 4-270, “Export SwImplPolicy” on page 4-271, and “Export SwRecordLayout for Lookup Table Data” on page 4-271.

Configure SwCalibrationAccess

You can specify the SwCalibrationAccess property for measurement variables, calibration parameters, and signal and parameter data objects. The valid values are:

- **ReadOnly** — Data element appears in the generated description file with read access only.
- **ReadWrite** — Data element appears in the generated description file with both read and write access.
- **NotAccessible** — Data element appears in the generated description file and is not accessible with calibration and measurement tools.

If you open a model with signals and parameters, you can specify the SwCalibrationAccess property in the following ways:

- “Specify SwCalibrationAccess for AUTOSAR Data Elements” on page 4-263
- “Specify Default SwCalibrationAccess for Application Data Types” on page 4-264

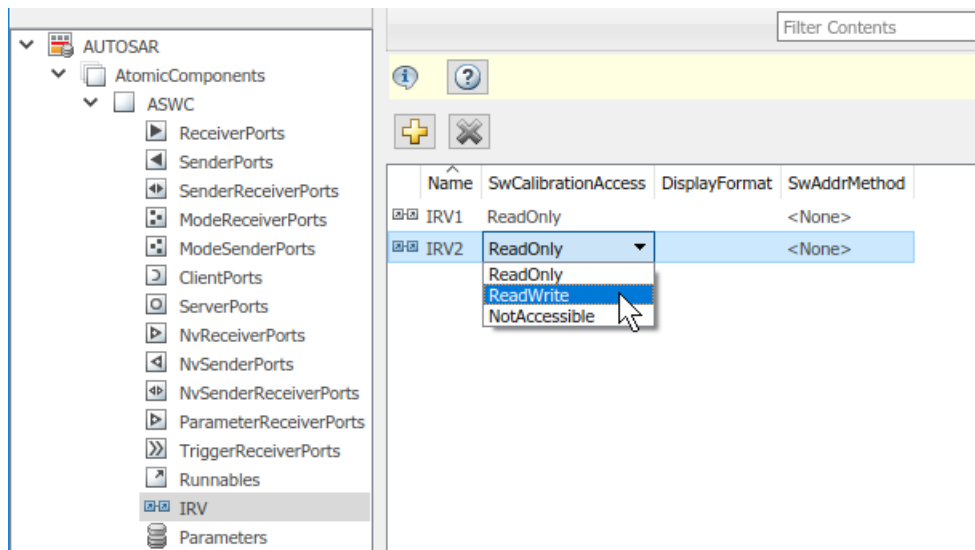
Specify SwCalibrationAccess for AUTOSAR Data Elements

You can use either the AUTOSAR Dictionary or MATLAB function calls to specify the `SwCalibrationAccess` property for the following AUTOSAR data elements:

- Sender-receiver interface data elements
- Nonvolatile interface data elements
- Client-server arguments
- Inter-runnable variables

For example:

- 1 Open a model that is configured for AUTOSAR.
- 2 Open the AUTOSAR Dictionary. Navigate to one of the following views:
 - S-R or NV interface, **DataElements** view
 - C-S interface, **Arguments** view
 - Atomic component, **IRV** view
- 3 Use the **SwCalibrationAccess** drop-down list to select the level of calibration and measurement tool access to allow for the data element.



Alternatively, you can use the AUTOSAR property functions to specify the `SwCalibrationAccess` property for AUTOSAR data elements. For example, the following code opens the `autosar_swc_fncalls` example model and sets calibration and measurement access to inter-runnable variable IRV2 to `ReadWrite`.

```
hModel = 'autosar_swc_fncalls';
openExample(hModel)
arProps = autosar.api.getAUTOSARProperties(hModel);
get(arProps, '/Company/Powertrain/Components/ASWC/ASWC_IB/IRV2', 'SwCalibrationAccess')

set(arProps, '/Company/Powertrain/Components/ASWC/ASWC_IB/IRV2', 'SwCalibrationAccess', 'ReadWrite');
get(arProps, '/Company/Powertrain/Components/ASWC/ASWC_IB/IRV2', 'SwCalibrationAccess')

ans =
    'ReadOnly'
```

```
ans =  
    'ReadWrite'
```

Here is a sample call to the AUTOSAR properties `set` function to set `SwCalibrationAccess` for an S-R interface data element in the same model.

```
set(arProps, '/Company/Powertrain/Interfaces/InIf/In1', 'SwCalibrationAccess', 'ReadWrite');  
get(arProps, '/Company/Powertrain/Interfaces/InIf/In1', 'SwCalibrationAccess')  
  
ans =  
    'ReadWrite'
```

Specify Default `SwCalibrationAccess` for Application Data Types

The AUTOSAR XML options include **`SwCalibrationAccess DefaultValue`** (property `SwCalibrationAccessDefault`), which defines the default `SwCalibrationAccess` value for AUTOSAR application data types in your model. You can use the AUTOSAR property functions to modify the default. For example, the following code opens the `autosar_swc_fcncalls` example model and changes the default calibration and measurement access for AUTOSAR application data types from `ReadOnly` to `ReadWrite`.

```
hModel = 'autosar_swc_fcncalls';  
openExample(hModel)  
arProps = autosar.api.getAUTOSARProperties(hModel);  
get(arProps, 'XmlOptions', 'SwCalibrationAccessDefault')  
  
set(arProps, 'XmlOptions', 'SwCalibrationAccessDefault', 'ReadWrite');  
get(arProps, 'XmlOptions', 'SwCalibrationAccessDefault')  
  
ans =  
    'ReadOnly'  
  
ans =  
    'ReadWrite'
```

Configure `DisplayFormat`

AUTOSAR display format specifications control the width and precision display for calibration and measurement data. You can import and export AUTOSAR display format specifications, and edit the specifications in Simulink. You can specify display format for the following AUTOSAR elements:

- Inter-runnable variables
- Sender-receiver interface data elements
- Client-server interface operation arguments
- `CompuMethods`

The display format specification is a subset of ANSI C `printf` specifiers, with the following form:

```
%[flags][width][.precision]type
```

Field	Description
flags (optional)	Characters specifying flags supported by AUTOSAR schemas: <ul style="list-style-type: none">• (' '): Insert a space before the value.• -: Left-justify.• +: Display plus or minus sign, even for positive numbers.• #:<ul style="list-style-type: none">• For types o, x, and X, display 0, 0x, or 0X prefix.• For types e, E, and f, display decimal point even if the precision is 0.• For types g and G, do not remove trailing zeros or decimal point.
width (optional)	Positive integer specifying the minimum number of characters to display.
precision (optional)	Positive integer specifying the precision to display: <ul style="list-style-type: none">• For integer type values (d, i, o, u, x, and X), specifies the minimum number of digits.• For types e, E, and f, specifies the number of digits to the right of the decimal point.• For types g and G, specifies the number of significant digits.

Field	Description
type	Characters specifying a numeric conversion type supported by AUTOSAR schemas: <ul style="list-style-type: none"> • d: Signed decimal integer. • i: Signed decimal integer. • o: Unsigned octal integer. • u: Unsigned decimal integer. • x: Unsigned hexadecimal integer, using characters "abcdef". • X: Unsigned hexadecimal integer, using characters "ABCDEF". • e: Signed floating-point value in exponential notation. The value has the form [-]d.dddd e [sign]ddd. <ul style="list-style-type: none"> • d is a single decimal digit. • dddd is one or more decimal digits. • ddd is exactly three decimal digits. • sign is + or -. • E: Identical to the e format except that E, rather than e, introduces the exponent. • f: Signed floating-point value in fixed-point notation. The value has the form [-]dddd.dddd. <ul style="list-style-type: none"> • dddd is one or more decimal digits. • The number of digits before the decimal point depends on the magnitude of the number. • The number of digits after the decimal point depends on the requested precision. • g: Signed value printed in f or e format, whichever is more compact for the given value and precision. Trailing zeros are truncated, and the decimal point appears only if one or more digits follow it. • G: Identical to the g format, except that E, rather than e, introduces the exponent (where required).



For example, the format specifier `%2.1d` specifies width 2, precision 1, and type signed decimal, producing a displayed value such as 12.2.

The **DisplayFormat** attribute appears in dialog boxes for AUTOSAR elements to which it applies. You can specify display format in a dialog box or with an element API that can modify attributes.

```
hModel = 'autosar_sw_c_counter';
openExample(hModel);
slMap = autosar.api.getSimuLinkMapping(hModel);
mapParameter(slMap, 'INC', 'ConstantMemory', 'DisplayFormat', '%2.6f')
```


- Model parameter or lookup table mapped to:
 - AUTOSAR constant memory
 - AUTOSAR internal calibration parameter
 - AUTOSAR port parameter
- Signal, state, or data store mapped to:
 - AUTOSAR static memory
 - AUTOSAR per-instance memory
- Model entry-point function mapped to AUTOSAR runnable
- Internal data inside a model entry-point function

To create and use SwAddrMethods in an AUTOSAR model:

- 1 Import SwAddrMethods from ARXML files or create SwAddrMethods in Simulink.
 - To import SwAddrMethods from ARXML files, use an `arxml.importer` function - `createComponentAsModel` or `createCompositionAsModel` for a new model, or `updateModel` or `updateAUTOSARProperties` for an existing model.
 - To create SwAddrMethods in an existing model, open the AUTOSAR Dictionary, SwAddrMethods view, and click the **Add** button . Alternatively, use equivalent AUTOSAR property functions. For more information, see “Create SwAddrMethods in Simulink” on page 4-269.
- 2 Associate SwAddrMethods with model data and functions. Open the Code Mappings editor and select the **Parameters**, **Data Stores**, **Signals/States**, or **Functions** tab. Select an element within that tab. To set the **SwAddrMethod** attribute, click the  icon. Alternatively, use the equivalent AUTOSAR map function. For more information, see “Associate SwAddrMethod with Model Data or Function” on page 4-270.
- 3 Generate code for your AUTOSAR model. (This example uses a signal mapped to AUTOSAR static memory in the model `autosar_swc_counter`.) In the generated files:

- Exported ARXML files contain SwAddrMethod descriptions and references.

```
<VARIABLE-DATA-PROTOTYPE UUID="...">
  <SHORT-NAME>SM_equal_to_count</SHORT-NAME>
  <CATEGORY>VALUE</CATEGORY>
  <SW-DATA-DEF-PROPS>
    <SW-DATA-DEF-PROPS-VARIANTS>
      <SW-DATA-DEF-PROPS-CONDITIONAL>
        <SW-ADDR-METHOD-REF DEST="SW-ADDR-METHOD">
          /Company/Powertrain/DataTypes/SwAddrMethods/VAR
        </SW-ADDR-METHOD-REF>
        <SW-CALIBRATION-ACCESS>READ-ONLY</SW-CALIBRATION-ACCESS>
        <SW-IMPL-POLICY>STANDARD</SW-IMPL-POLICY>
      </SW-DATA-DEF-PROPS-CONDITIONAL>
    </SW-DATA-DEF-PROPS-VARIANTS>
  </SW-DATA-DEF-PROPS>
  <TYPE-TREF DEST="IMPLEMENTATION-DATA-TYPE">
    /Company/Powertrain/DataTypes/Boolean_volatile_my_qualifier</TYPE-TREF>
</VARIABLE-DATA-PROTOTYPE>
```

- AUTOSAR-compliant C code contains comments and `#define` and `#include` statements that provide a wrapper around data or function definitions belonging to each SwAddrMethod memory section.

```
/* Static Memory for Internal Data */
/* SwAddrMethod VAR for Internal Data */
```


```
#define autosar_sw_counter_START_SEC_VAR
#include "autosar_sw_counter_MemMap.h"

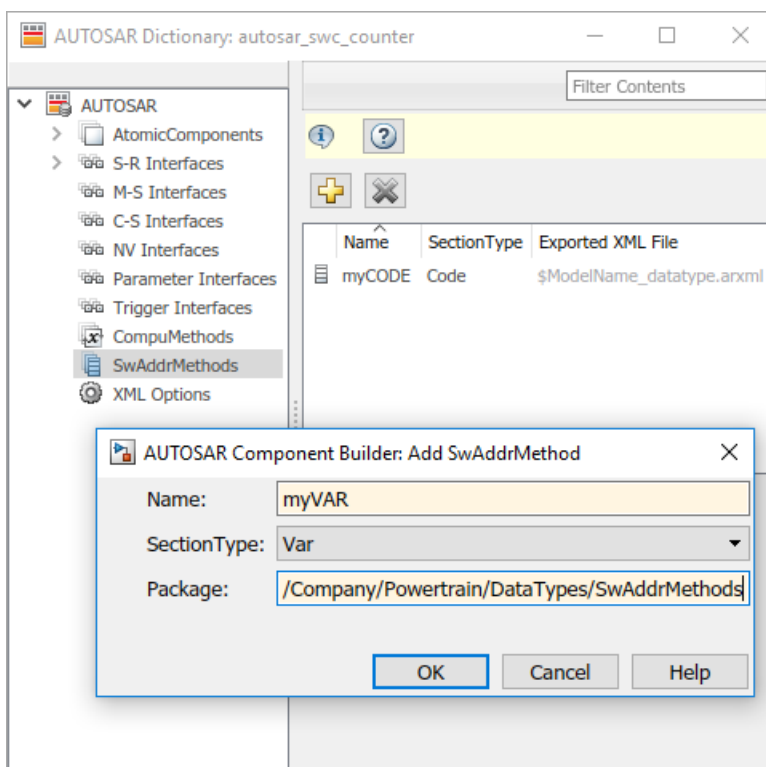
volatile my_qualifier boolean SM_equal_to_count;

#define autosar_sw_counter_STOP_SEC_VAR
#include "autosar_sw_counter_MemMap.h"
```

- “Create SwAddrMethods in Simulink” on page 4-269
- “Associate SwAddrMethod with Model Data or Function” on page 4-270

Create SwAddrMethods in Simulink

To create SwAddrMethods in an existing model, open the AUTOSAR Dictionary, SwAddrMethods view, and click the **Add** button .



Alternatively, use equivalent AUTOSAR property functions. This code adds SwAddrMethods myCODE and myVAR to an AUTOSAR component model.

```
hModel = 'autosar_sw_counter';
openExample(hModel)
arProps = autosar.api.getAUTOSARProperties(hModel);
addPackageableElement(arProps, 'SwAddrMethod', ...
    '/Company/Powertrain/DataTypes/SwAddrMethods', 'myCODE', ...
    'SectionType', 'Code')
swAddrPaths = find(arProps, [], 'SwAddrMethod', 'PathType', 'FullyQualified', ...
    'SectionType', 'Code')
addPackageableElement(arProps, 'SwAddrMethod', ...
    '/Company/Powertrain/DataTypes/SwAddrMethods', 'myVAR', ...
    'SectionType', 'Var')
swAddrPaths = find(arProps, [], 'SwAddrMethod', 'PathType', 'FullyQualified', ...
    'SectionType', 'Var')


swAddrPaths =
    {'/Company/Powertrain/DataTypes/SwAddrMethods/CODE'}
```

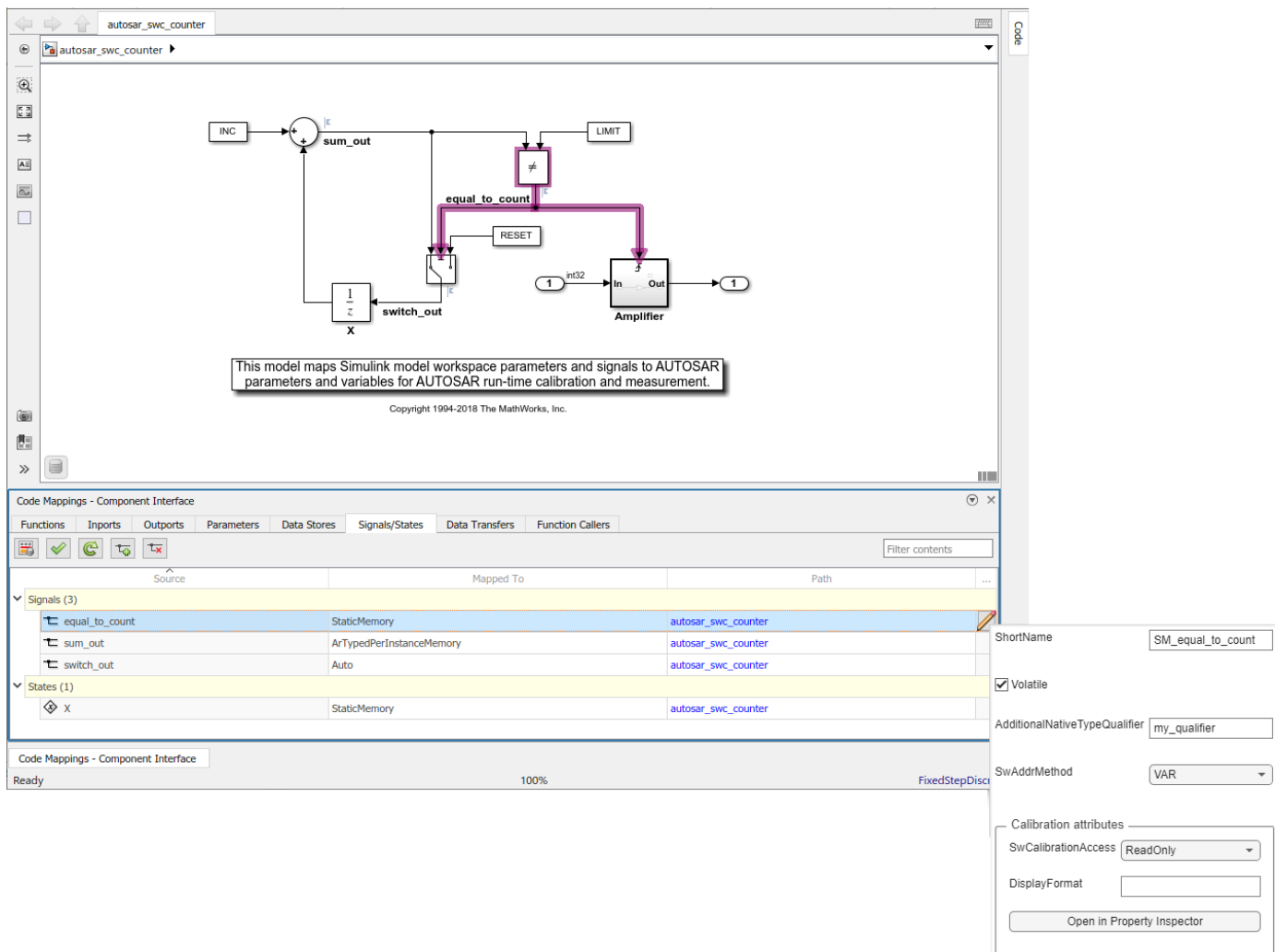
```

{/Company/Powertrain/DataTypes/SwAddrMethods/myCODE'}
swAddrPaths =
{/Company/Powertrain/DataTypes/SwAddrMethods/VAR'}
{/Company/Powertrain/DataTypes/SwAddrMethods/myVAR'}

```

Associate SwAddrMethod with Model Data or Function

To associate SwAddrMethods with model data and functions, open the Code Mappings editor and select the **Parameters**, **Data Stores**, **Signals/States**, or **Functions** tab. Select an element within that tab. To set the **SwAddrMethod** attribute, click the  icon. In this example, SwAddrMethod VAR is selected for signal equal_to_count, which is mapped to AUTOSAR static memory.



This model maps Simulink model workspace parameters and signals to AUTOSAR parameters and variables for AUTOSAR run-time calibration and measurement.

Copyright 1994-2018 The MathWorks, Inc.

Source	Mapped To	Path
equal_to_count	StaticMemory	autosar_swc_counter
sum_out	ArTypedPerInstanceMemory	autosar_swc_counter
switch_out	Auto	autosar_swc_counter
X	StaticMemory	autosar_swc_counter

ShortName: SM_equal_to_count

Volatile

AdditionalNativeTypeQualifier: my_qualifier

SwAddrMethod: VAR

Calibration attributes

SwCalibrationAccess: ReadOnly

DisplayFormat:

Open in Property Inspector

Alternatively, use the equivalent AUTOSAR map function. For more information, see the reference page for `mapFunction`, `mapParameter`, `mapSignal`, `mapState`, or `mapDataStore`.

Configure SwAlignment

The `SwAlignment` property describes the intended alignment of AUTOSAR data within a memory section. `SwAlignment` defines a quantity of bits. Valid values include 8, 12, 32, UNKNOWN

(deprecated), UNSPECIFIED, and BOOLEAN. For numeric data, typical `SwAlignment` values are 8, 16, and 32.

If you do not define the `SwAlignment` property, the `swBaseType` size and the `memoryAllocationKeywordPolicy` of the referenced `SwAlignment` determine the alignment.

You can use the AUTOSAR property function `set` to set `SwAlignment` for S-R interface data elements and inter-runnable variables. For example:

```
interfacePath = '/A/B/C/Interfaces/If1/';
dataElementName = 'E11';
swAlignmentValue = '32';
set(dataObj,[interfacePath dataElementName],'SwAlignment',swAlignmentValue);
```

To support the round-trip workflow, the ARXML importer imports and preserves the `SwAlignment` property for the following AUTOSAR data:

- Per-instance memory
- Software component parameters
- Parameter interface data elements
- Client-server interface operation arguments
- Static and constant memory

Export `SwImplPolicy`

The `SwImplPolicy` property specifies the implementation policy for a data element, regarding consistency mechanisms of variables. You cannot modify the `SwImplPolicy` property, but the property is set to `standard` or `queued` for AUTOSAR data in exported ARXML code. The value is set to:

- `standard` for
 - Per-instance memory
 - Inter-runnable variables
 - Software component parameters
 - Parameter interface data elements
 - Client-server interface operation arguments
 - Static and constant memory
- `standard` or `queued` for
 - Sender-receiver interface data elements

Export `SwRecordLayout` for Lookup Table Data

AUTOSAR software components use software record layouts (`SwRecordLayouts`) to specify how to serialize data in the memory of an AUTOSAR ECU. The ARXML importer imports and preserves the `SwRecordLayout` property for AUTOSAR data.

You can import `SwRecordLayouts` from ARXML files in either of two ways:

- If you create your AUTOSAR model from ARXML files using importer function `createComponentAsModel`, include an ARXML file that contains `SwRecordLayout` definitions in the import. The imported `SwRecordLayouts` are preserved and later exported in ARXML code.

- If you create your AUTOSAR model in Simulink, you can import shared definitions of SwRecordLayouts from ARXML files. Use importer function `updateAUTOSARProperties`. For example:

```
importerObj = arxml.importer(arxmlFileName);  
updateAUTOSARProperties(importerObj,modelName);
```

When you generate model code, the exported ARXML code contains references to the imported read-only SwRecordLayout elements, but not their definitions.

```
<APPLICATION-PRIMITIVE-DATA-TYPE>  
  <SHORT-NAME>App1_L_6_s16En4</SHORT-NAME>  
  <CATEGORY>CURVE</CATEGORY>  
  <SW-DATA-DEF-PROPS>  
    ...  
    <SW-RECORD-LAYOUT-REF DEST="SW-RECORD-LAYOUT">  
      /AUTOSAR/Ifx/SwRecordLayouts_Blueprint/IntCur_s16_s16  
    </SW-RECORD-LAYOUT-REF>  
  </SW-DATA-DEF-PROPS>  
</APPLICATION-PRIMITIVE-DATA-TYPE>
```

For more information, see “Import and Reference Shared AUTOSAR Element Definitions” on page 3-29.

See Also

Related Examples

- “Import AUTOSAR XML Descriptions Into Simulink” on page 3-13
- “Model AUTOSAR Calibration Parameters and Lookup Tables” on page 2-50
- “Configure AUTOSAR CompuMethods” on page 4-236
- “Configure AUTOSAR Code Generation” on page 5-7

More About

- “AUTOSAR Component Configuration” on page 4-3

Configure Lookup Tables for AUTOSAR Calibration and Measurement

In Simulink, you can implement standard axis (STD_AXIS), common axis (COM_AXIS), and fix axis (FIX_AXIS) lookup tables for AUTOSAR applications. AUTOSAR applications can use lookup tables in either or both of two ways:

- Implement fast search operations.
- Support tuning of the application with calibration and measurement tools.

To model lookup tables for automotive application tuning, use the classes `Simulink.LookupTable` and `Simulink.Breakpoint`. By creating `Simulink.LookupTable` and `Simulink.Breakpoint` objects in the model workspace, you can store and share lookup table and breakpoint data and configure the data for AUTOSAR code generation.

Configure STD_AXIS Lookup Tables by Using Lookup Table Objects

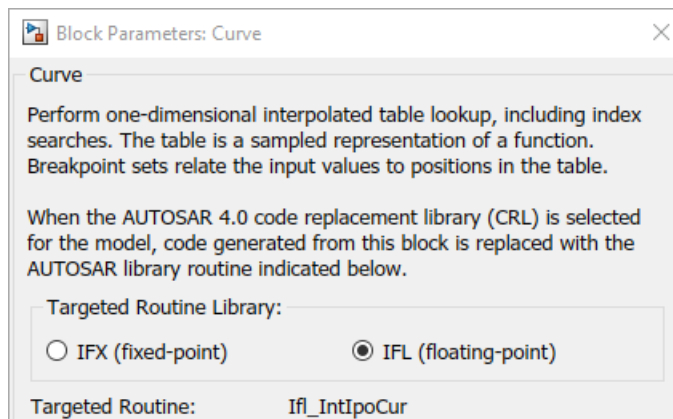
This example shows how to create STD_AXIS lookup tables in Simulink, using `Simulink.LookupTable` objects, and configure the lookup tables for AUTOSAR code generation. The example uses the model `mAutosarLutObjs.slx` from `matlabroot/help/toolbox/autosar/examples`. To copy the model file to your working folder, enter this MATLAB command:

```
copyfile(fullfile(matlabroot,'help/toolbox/autosar/examples/mAutosarLutObjs.slx'),'')
```

- 1 Model an AUTOSAR lookup table in a STD_AXIS configuration.
 - a In a mapped AUTOSAR software component model, add an AUTOSAR Blockset Curve or Map block. This example adds a Curve block.

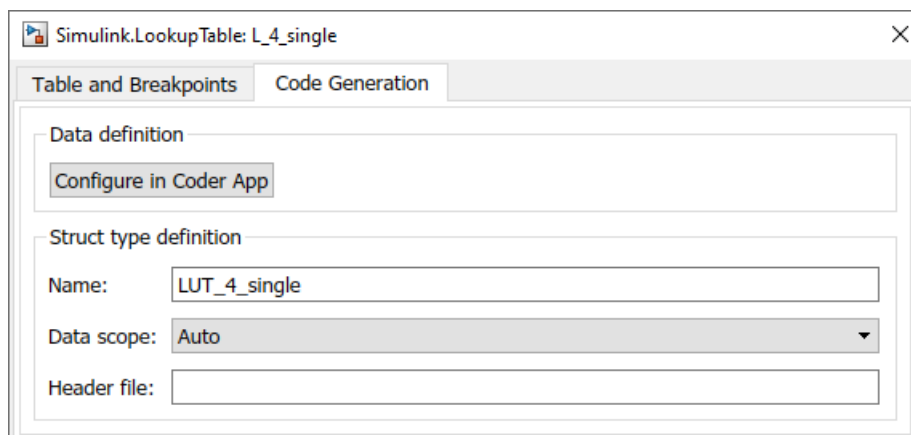
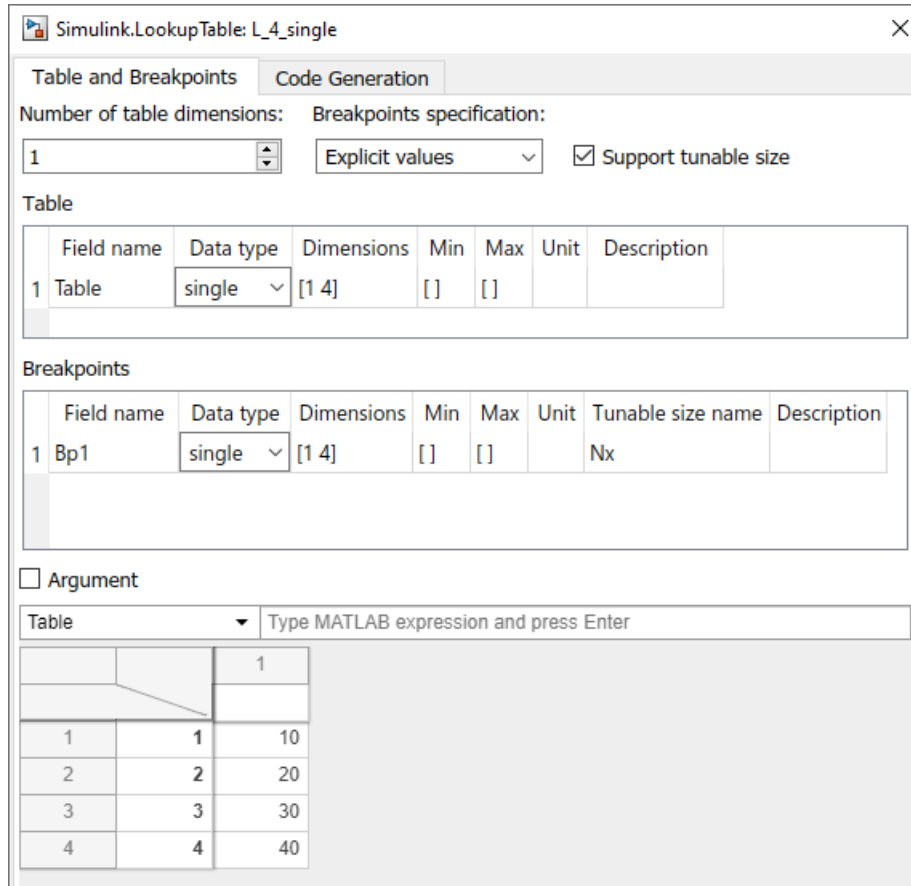


- b Open the Curve block and configure it to generate a routine from the AUTOSAR 4.0 code replacement library (CRL). As you modify block settings, the block dialog box updates the name of the targeted AUTOSAR routine.

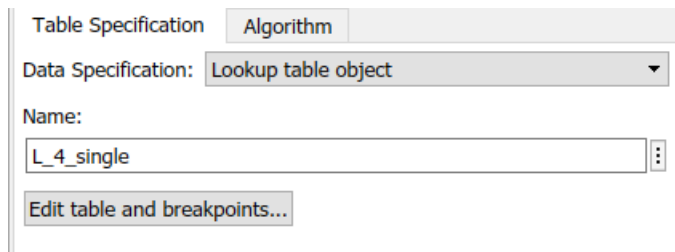


In the block dialog box, make these selections:

- To generate a floating-point routine, select **IFL (floating-point)**.
 - In the **Table Specification** tab, to specify table data using a lookup table object, set **Data Specification** to Lookup table object.
- c In the model workspace, create a Simulink.LookupTable object and configure it to store the lookup table data.



- d In the Curve block dialog box, **Table Specification** tab, enter the Simulink.LookupTable object name in the **Name** field.

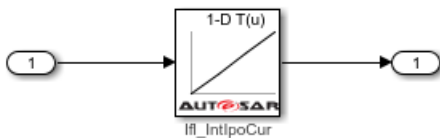


- e In the block dialog box, **Algorithm** tab, set **Integer Rounding Method** to Zero. Leave **Interpolation Method** set to Linear point-slope and **Index Search Method** set to Linear search.

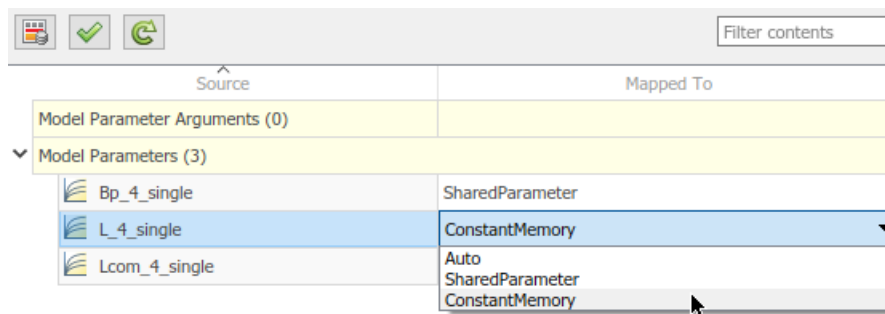
Table data appears in generated AUTOSAR C code as fields of a single structure. To control the characteristics of the structure type, such as its name, use the properties of the object.

2 Connect the Curve or Map block.

- Add AUTOSAR operating points to the lookup tables. Connect a root-level inport to the Curve or Map block. Alternatively, configure an input signal to the Curve or Map block with static global memory.
- Connect an output to the Curve or Map block.




- 3 In the AUTOSAR code perspective, use the Code Mappings editor to map Simulink.LookupTable objects to AUTOSAR internal calibration parameters. In the **Parameters** tab, select each Simulink.LookupTable object that you created. Map each object to AUTOSAR parameter type ConstantMemory, SharedParameter, or Auto. To accept software mapping defaults, specify Auto.



In this example, STD_AXIS lookup table object L_4_single is mapped to AUTOSAR ConstantMemory.

4

For each parameter, if you select a parameter type other than Auto, click the  icon to view or modify other code and calibration attributes. For more information on parameter properties, see “Map Model Workspace Parameters to AUTOSAR Component Parameters” on page 4-54.

- 5 Configure the model to generate C code based on the AUTOSAR 4.0 library. Open the Configuration Parameters dialog box and select **Code Generation > Interface**. Set the **Code replacement library** parameter to AUTOSAR 4.0. For more information, see “Code Generation with AUTOSAR Code Replacement Library” on page 5-12.
- 6 Build the model. The generated C code contains the expected Ifl and Ifx lookup function calls and Rte data access function calls. For example, you can search the HTML code generation report for the Ifl or Ifx routine prefix.

```

/* Model step function */
void Runnable_Step(void)
{
  /* Outport: '<Root>/Out1' incorporates:
   * Inport: '<Root>/In1'
   * Lookup_n-D: '<Root>/Curve'
   */
  Rte_IWrite_Runnable_Step_Out1_Out1(If1_IntIpoCur_f32_f32
  (Rte_IRead_Runnable_Step_In1_In1(), L_4_single.Nx, L_4_single.Bp1,
  L_4_single.Table));
}

```

The generated ARXML files contain data types of category CURVE (1-D table data) and MAP (2-D table data). The data types have the data calibration properties that you configured.

Configure COM_AXIS Lookup Tables by Using Lookup Table and Breakpoint Objects

This example shows how to create COM_AXIS lookup tables in Simulink, using Simulink.LookupTable and Simulink.Breakpoint objects, and configure the lookup tables for AUTOSAR code generation. The example uses the model mAutosarLutObjs.slx from matlabroot/help/toolbox/autosar/examples. To copy the model file to your working folder, enter this MATLAB command:

```
copyfile(fullfile(matlabroot,'help/toolbox/autosar/examples/mAutosarLutObjs.slx'),'')
```

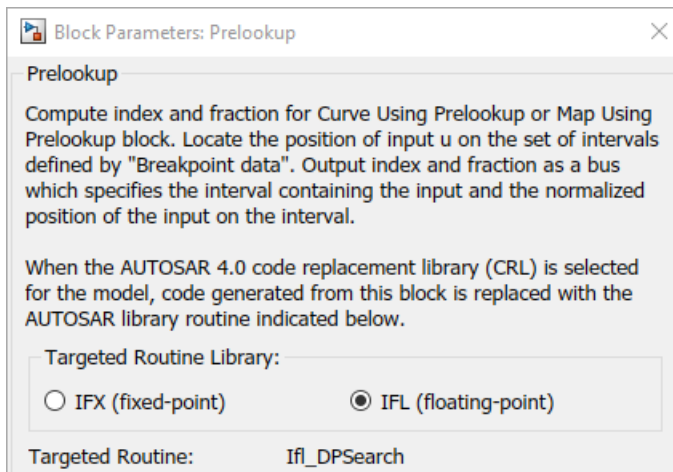
In this example, to model an AUTOSAR lookup table in a COM_AXIS configuration, you pair AUTOSAR Blockset Prelookup blocks with Curve Using Prelookup or Map Using Prelookup blocks.

1 Configure Prelookup blocks.

- a In a mapped AUTOSAR software component model, add one or more AUTOSAR Blockset Prelookup blocks. This example adds one Prelookup block.

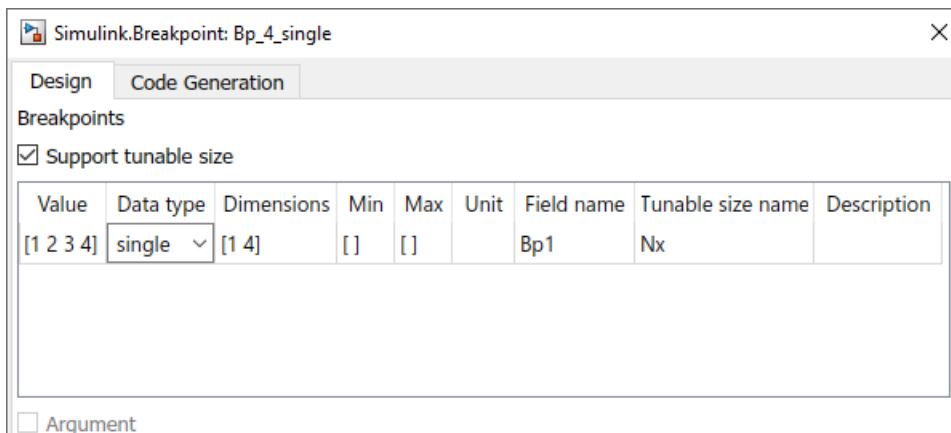


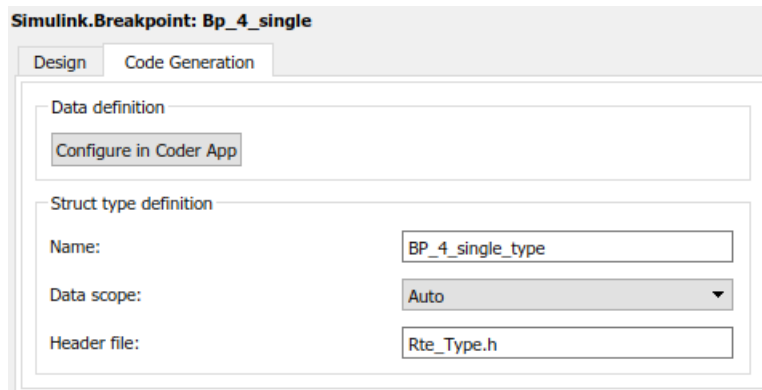
- b Open each block and configure it to generate a routine from the AUTOSAR 4.0 code replacement library (CRL). As you modify block settings, the block updates the name of the targeted AUTOSAR routine.



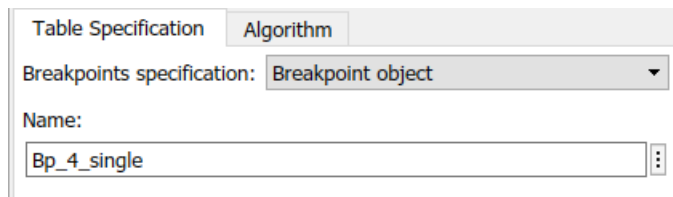
In the block dialog box, make these selections:

- To generate a floating-point routine, select **IFL (floating-point)**.
 - In the **Table Specification** tab, to specify breakpoint data using a breakpoint object, set **Breakpoints specification** to Breakpoint object.
- c For each breakpoint vector, in the model workspace, create and configure a Simulink.Breakpoint object.





- d In the Prelookup block dialog box, **Table Specification** tab, enter the Simulink.Breakpoint object name in the **Name** field. You can reduce memory consumption by sharing breakpoint data between lookup tables.



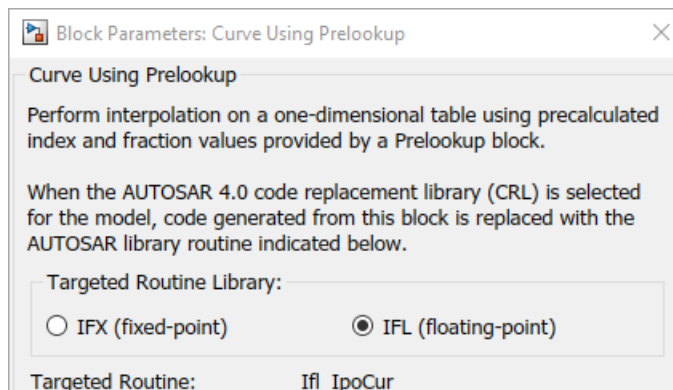
- e In the block dialog box, **Algorithm** tab, set **Integer Rounding Method** to Zero. Leave **Index Search Method** set to Linear search.

2 Configure Curve Using Prelookup and Map Using Prelookup blocks.

- a In the model, add one or more AUTOSAR Blockset Curve Using Prelookup or Map Using Prelookup blocks. Each block immediately follows a Prelookup block with which it is paired. This example adds one Curve Using Prelookup block.

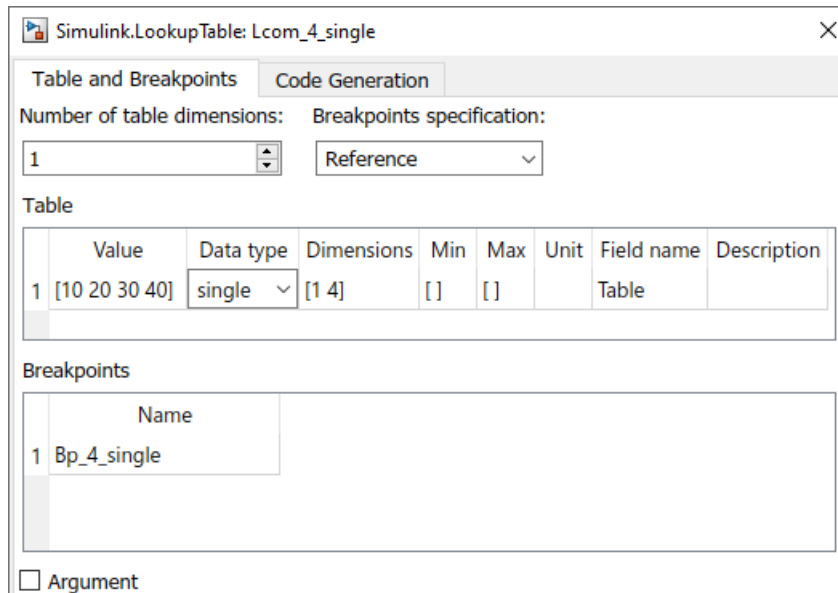


- b Open each Curve Using Prelookup or Map Using Prelookup block and configure it to generate a routine from the AUTOSAR 4.0 code replacement library (CRL). As you modify block settings, the block dialog box updates the name of the targeted AUTOSAR routine.

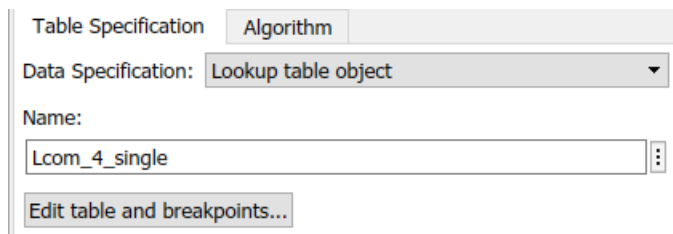


In the block dialog box, make these selections:

- To generate a floating-point routine, select **IFL (floating-point)**.
 - In the **Table Specification** tab, to specify table data using a lookup table object, set **Data Specification** to **Lookup table object**.
- c For each set of table data, in the model workspace, create and configure a `Simulink.LookupTable` object.



- d In each Curve Using Prelookup or Map Using Prelookup block dialog box, **Table Specification** tab, enter a `Simulink.LookupTable` object name in the **Name** field.

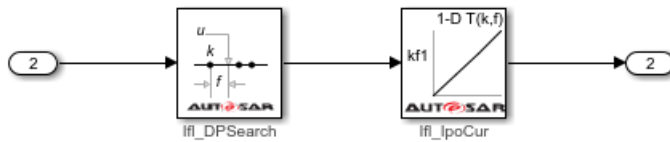


- e In the block dialog box, **Algorithm** tab, set **Integer Rounding Method** to Zero. Leave **Interpolation Method** set to Linear point-slope.

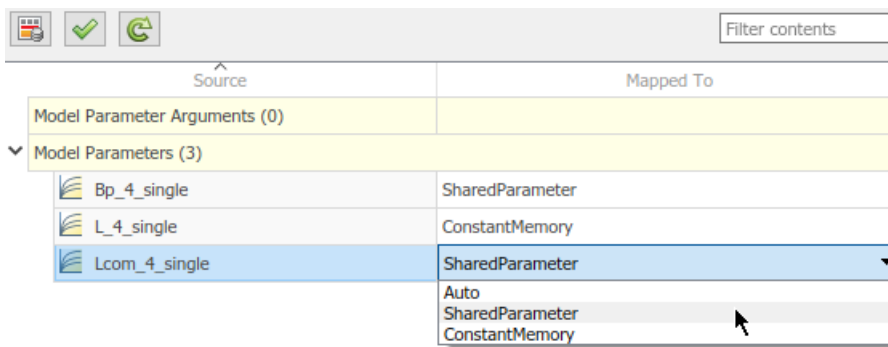
Each set of table data appears in the generated C code as a separate array variable. If the table size is tunable, each breakpoint vector appears as a structure. The structure contains a field to store the breakpoint data and, optionally, a field to store the length of the vector. The second field enables you to tune the effective size of the table. If the table size is not tunable, each breakpoint vector appears as an array.

- 3 Connect the Prelookup, Curve Using Prelookup, and Map Using Prelookup blocks.
- Add AUTOSAR operating points to the lookup tables. Connect root-level inports to the Prelookup blocks. Alternatively, configure input signals to the Prelookup blocks with static global memory.
 - Connect outputs to the Curve Using Prelookup and Map Using Prelookup blocks.


- Connect each Prelookup block to its matched Curve Using Prelookup or Map Using Prelookup block.

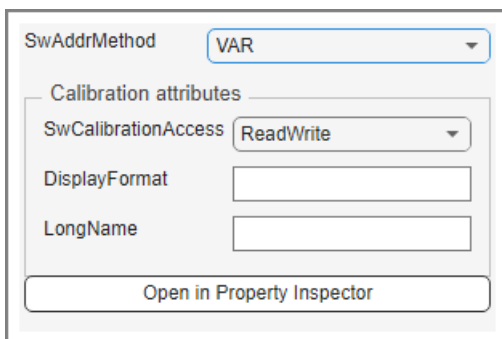


- 4 In the AUTOSAR code perspective, use the Code Mappings editor to map `Simulink.LookupTable` and `Simulink.Breakpoint` objects to AUTOSAR internal calibration parameters. In the **Parameters** tab, select each `Simulink.LookupTable` and `Simulink.Breakpoint` object that you created. Map each object to AUTOSAR parameter type `ConstantMemory`, `SharedParameter`, or `Auto`. To accept software mapping defaults, specify `Auto`.



In this example, COM_AXIS breakpoint object `Bp_4_single` and lookup table object `Lcom_4_single` are mapped to AUTOSAR SharedParameters. All instances of the AUTOSAR software component share the COM_AXIS parameters.

- 5 For each parameter, if you select a parameter type other than `Auto`, click the  icon to view or modify other code and calibration attributes. For more information on parameter properties, see “Map Model Workspace Parameters to AUTOSAR Component Parameters” on page 4-54.



- 6 Configure the model to generate C code based on the AUTOSAR 4.0 library. Open the Configuration Parameters dialog box and select **Code Generation > Interface**. Set the **Code replacement library** parameter to `AUTOSAR_4.0`. For more information, see “Code Generation with AUTOSAR Code Replacement Library” on page 5-12.

- 7 Build the model. The generated C code contains the expected Ifl and Ifx lookup function calls and Rte data access function calls. For example, you can search the HTML code generation report for the Ifl or Ifx routine prefix.

```

/* Model step function */
void Runnable_Step(void)
{
    Ifl_DPResultF32_Type rtb_Prelookup;

    /* PreLookup: '<Root>/PreLookup' incorporates:
     * Inport: '<Root>/In2'
     */
    Ifl_DPSearch_f32(&rtb_Prelookup, Rte_IRead_Runnable_Step_In2_In2(),
                    (Rte_CData_Bp_4_single()->Nx, (Rte_CData_Bp_4_single()->Bp1));

    /* Outport: '<Root>/Out2' incorporates:
     * Interpolation_n-D: '<Root>/Curve Using PreLookup'
     */
    Rte_IWrite_Runnable_Step_Out2_Out2(If1_IpoCur_f32(&rtb_Prelookup,
    Rte_CData_Lcom_4_single()));
}

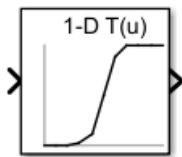
```

The generated ARXML files contain data types of category CURVE (1-D table data), MAP (2-D table data), and COM_AXIS (axis data). The data types have the data calibration properties that you configured.

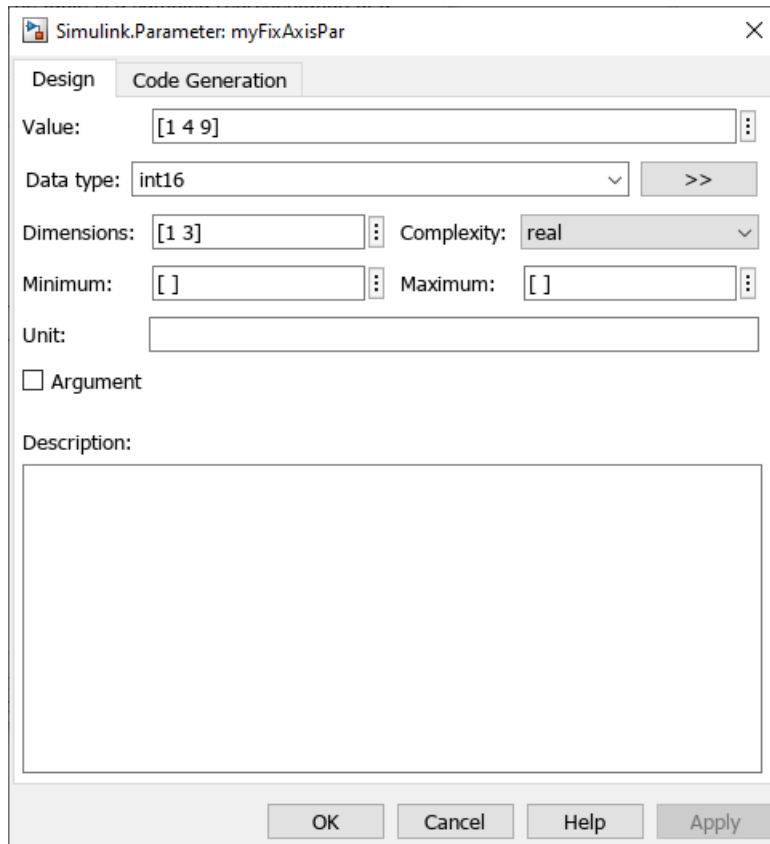
Configure FIX_AXIS Lookup Tables by Using Simulink Parameter Objects

This example shows how to create FIX_AXIS lookup tables in Simulink, using Simulink.Parameter objects, and configure the lookup tables for AUTOSAR code generation.

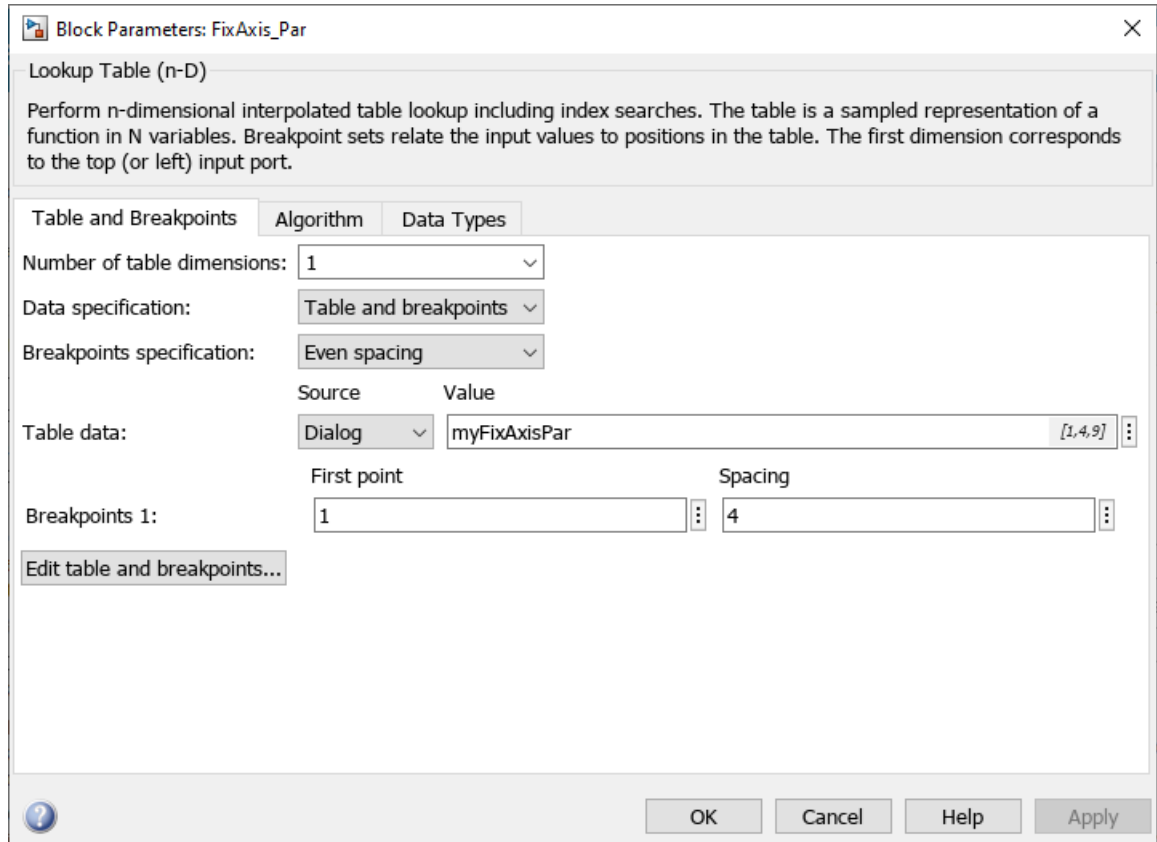
- 1 Model an AUTOSAR lookup table in a FIX_AXIS configuration.
 - a In a mapped AUTOSAR software component model, add a Simulink 1-D Lookup Table block.




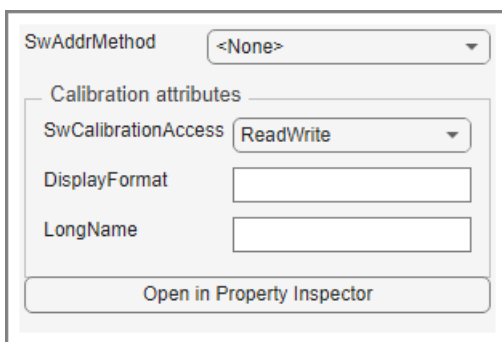
- b In the model workspace, create a Simulink.Parameter object and configure it to store the lookup table values.



- c Open the 1-D Lookup Table block. In the **Table and Breakpoints** tab, to configure the table to fix axis, set **Breakpoints specification** to Even spacing.
- d In the 1-D Lookup Table block dialog box, **Table and breakpoints** tab, enter the Simulink.Parameter object name in the **Value** field.



- e In the block dialog box, **Algorithm** tab, set **Extrapolation Method** to **Clip**. Enable **Use last table values for inputs at or above last breakpoint**.
- 2 Connect an inport and outport to the 1-D Lookup Table block.
- 3 In the AUTOSAR code perspective, use the Code Mappings editor to map `Simulink.Parameter` object to AUTOSAR internal calibration parameters. In the **Parameters** tab, select each `Simulink.Parameter` object that you created. Map the object to AUTOSAR parameter type `ConstantMemory`, `SharedParameter`, or `Auto`. To accept software mapping defaults, specify `Auto`.
- 4 For each parameter, if you select a parameter type other than `Auto`, click the  icon to view or modify other code and calibration attributes. For more information on parameter properties, see “Map Model Workspace Parameters to AUTOSAR Component Parameters” on page 4-54.



- 5 Configure the model to generate C code based on the AUTOSAR 4.0 library. Open the Configuration Parameters dialog box and select **Code Generation > Interface**. Set the **Code replacement library** parameter to AUTOSAR 4.0. For more information, see “Code Generation with AUTOSAR Code Replacement Library” on page 5-12.
- 6 Build the model. The generated C code contains the expected `Ifx` lookup function call and `Rte` data access function calls. For example, you can search the HTML code generation report for the `Ifx` routine prefix.

```

/* Outport: '<Root>/Out3' incorporates:
 * Inport: '<Root>/In3'
 * Lookup_n-D: '<Root>/FixAxis_Par'
 */
Rte_IWriteRunnableStep_Out3_Out3(Ifx_IntIpoFixCur_s16_s16
(Rte_IReadRunnableStep_In3_In3(), 3, &(Rte_CData_myFixAxisPar())[0], 1, 2));

```

The generated ARXML files contain data types of category CURVE (1-D table data). The data types have the data calibration properties that you configured.

Note For `FIX_AXIS` lookup table objects, the axes values are fixed and not tunable. The table data can be tuned.

Configure Array Layout for Multidimensional Lookup Tables

If an AUTOSAR model contains multidimensional lookup tables, you can configure the layout of lookup table array data for code generation as column-major or row-major. In the Simulink Configuration Parameters dialog box, **Interface** pane, set **Array layout** to `Column-major` (the default) or `Row-major`. The array layout selection affects code generation, including C code and exported ARXML descriptions.

If you select row-major layout, go to the **Math and Data Types** pane and select the configuration option **Use algorithms optimized for row-major array layout**. The algorithm selection affects simulation and code generation.

Exporting multidimensional lookup tables generates ARXML lookup table descriptions with the `SwRecordLayout` category set to either `COLUMN_DIR` or `ROW_DIR`. For example, this program listing shows the `SwRecordLayout` descriptions exported for an AUTOSAR model that contains a 2-dimensional row-major lookup table. The lookup table is implemented by using an AUTOSAR Map block.

```

<AR-PACKAGE>
  <SHORT-NAME>SwRecordLayouts</SHORT-NAME>
  <ELEMENTS>
    <SW-RECORD-LAYOUT UUID="...">
      <SHORT-NAME>Map_s16</SHORT-NAME>
      <SW-RECORD-LAYOUT-GROUP>
        <SHORT-LABEL>Val</SHORT-LABEL>
        <CATEGORY>ROW_DIR</CATEGORY>
        <SW-RECORD-LAYOUT-GROUP-AXIS>1</SW-RECORD-LAYOUT-GROUP-AXIS>
        <SW-RECORD-LAYOUT-GROUP-INDEX>X</SW-RECORD-LAYOUT-GROUP-INDEX>
        <SW-RECORD-LAYOUT-GROUP-FROM>1</SW-RECORD-LAYOUT-GROUP-FROM>
        <SW-RECORD-LAYOUT-GROUP-TO>-1</SW-RECORD-LAYOUT-GROUP-TO>
      <SW-RECORD-LAYOUT-GROUP>
        <SW-RECORD-LAYOUT-GROUP-AXIS>2</SW-RECORD-LAYOUT-GROUP-AXIS>
        <SW-RECORD-LAYOUT-GROUP-INDEX>Y</SW-RECORD-LAYOUT-GROUP-INDEX>
        <SW-RECORD-LAYOUT-GROUP-FROM>1</SW-RECORD-LAYOUT-GROUP-FROM>
        <SW-RECORD-LAYOUT-GROUP-TO>-1</SW-RECORD-LAYOUT-GROUP-TO>
    </SW-RECORD-LAYOUT>
  </ELEMENTS>
</AR-PACKAGE>

```

```

    <SW-RECORD-LAYOUT-V>
      <SHORT-LABEL>Val</SHORT-LABEL>
      <BASE-TYPE-REF DEST="SW-BASE-TYPE">
        /DataTypes/SwBaseTypes/sint32
      </BASE-TYPE-REF>
      <SW-RECORD-LAYOUT-V-AXIS>0</SW-RECORD-LAYOUT-V-AXIS>
      <SW-RECORD-LAYOUT-V-PROP>VALUE</SW-RECORD-LAYOUT-V-PROP>
      <SW-RECORD-LAYOUT-V-INDEX>X Y</SW-RECORD-LAYOUT-V-INDEX>
    </SW-RECORD-LAYOUT-V>
  </SW-RECORD-LAYOUT-GROUP>
</SW-RECORD-LAYOUT-GROUP>
</SW-RECORD-LAYOUT>
<SW-RECORD-LAYOUT UUID="...">
  <SHORT-NAME>Distr_s8_M</SHORT-NAME>
  <SW-RECORD-LAYOUT-GROUP>
    <SHORT-LABEL>Y</SHORT-LABEL>
    <CATEGORY>INDEX_INCR</CATEGORY>
    <SW-RECORD-LAYOUT-GROUP-AXIS>1</SW-RECORD-LAYOUT-GROUP-AXIS>
    <SW-RECORD-LAYOUT-GROUP-FROM>1</SW-RECORD-LAYOUT-GROUP-FROM>
    <SW-RECORD-LAYOUT-GROUP-TO>-1</SW-RECORD-LAYOUT-GROUP-TO>
  <SW-RECORD-LAYOUT-V>
    <SHORT-LABEL>VALUE</SHORT-LABEL>
    <BASE-TYPE-REF DEST="SW-BASE-TYPE">
      /DataTypes/SwBaseTypes/sint32
    </BASE-TYPE-REF>
    <SW-RECORD-LAYOUT-V-AXIS>1</SW-RECORD-LAYOUT-V-AXIS>
    <SW-RECORD-LAYOUT-V-PROP>VALUE</SW-RECORD-LAYOUT-V-PROP>
  </SW-RECORD-LAYOUT-V>
</SW-RECORD-LAYOUT-GROUP>
</SW-RECORD-LAYOUT>
</ELEMENTS>
</AR-PACKAGE>

```

Importing ARXML files with multidimensional lookup table descriptions creates Simulink lookup tables with **Array layout** set to **Column-major** or **Row-major**. If the ARXML files contain only row-major multidimensional lookup table descriptions, the ARXML importer creates Simulink lookup tables with **Array layout** set to **Row-major** and **Use algorithms optimized for row-major array layout** enabled.

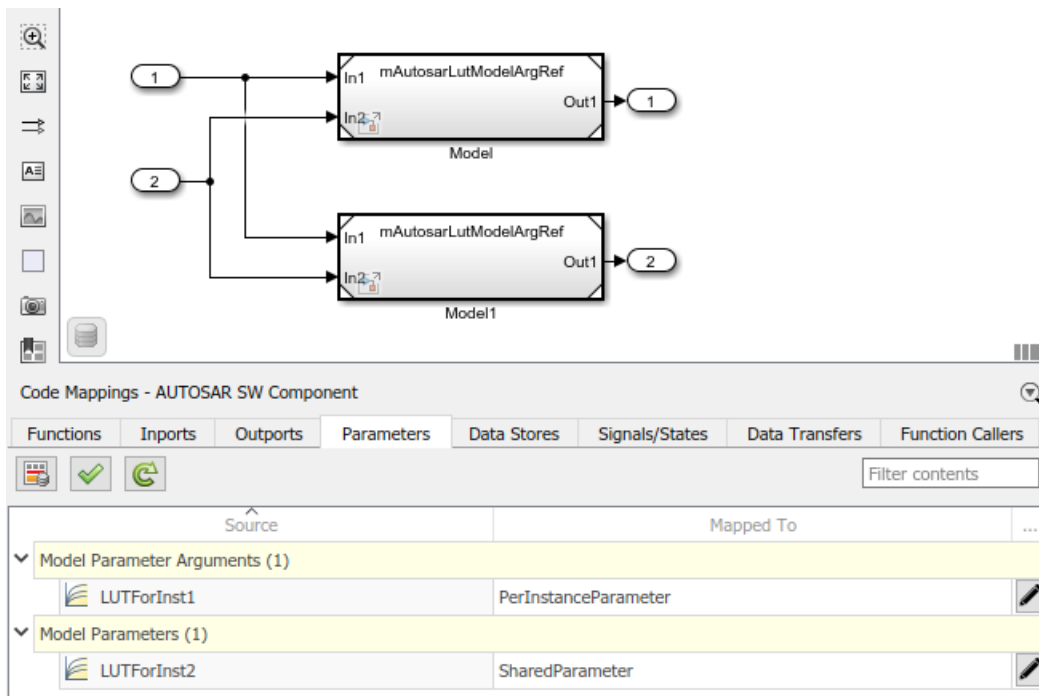
Parameterizing Instances of Reusable Referenced Model Lookup Tables and Breakpoints

In an AUTOSAR model hierarchy, you can parameterize a lookup table or breakpoint by placing it inside a referenced model and then parameterizing the referenced model. Parameterizing a referenced model involves configuring the referenced model to use model arguments, and then setting model argument values in the parent model.

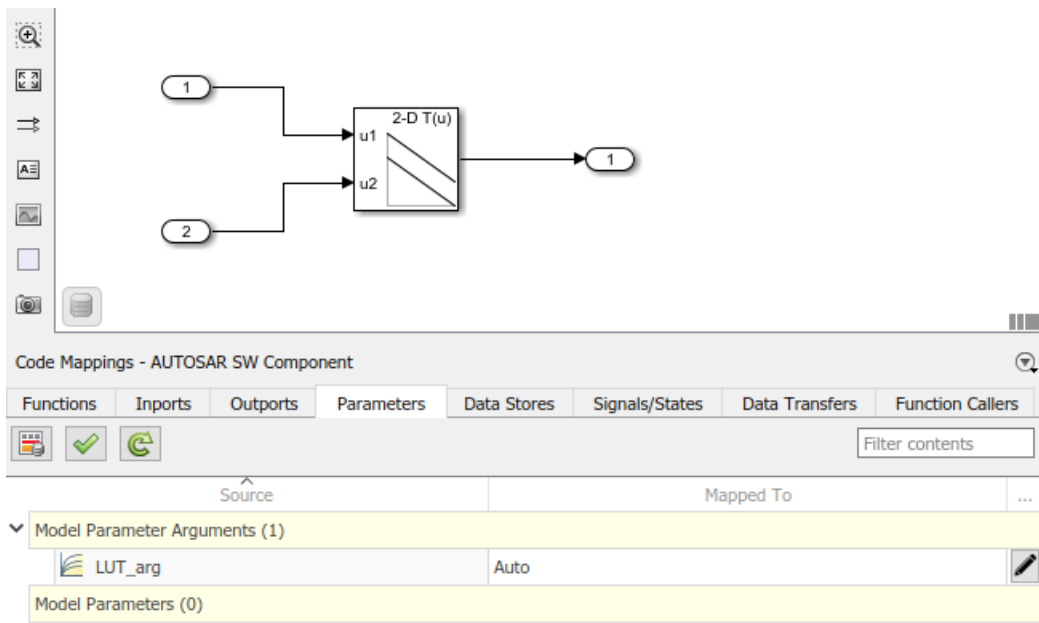
Parameterizing instances of reusable referenced model lookup tables allows you to place multiple instances of lookup table sub-units in an AUTOSAR model hierarchy. You can use sub-unit level testing with the lookup tables.

When a referenced model contains a lookup table or breakpoint, and the containing top model passes lookup table parameter values or breakpoint object to the model arguments of the referenced model, top-model export generates application data types for the lookup table parameters or breakpoints.

Consider this top model, which parameterizes two instances of a referenced model 2-D lookup table. Top-model parameter `LUTForInst1` is mapped to an AUTOSAR `PerInstanceParameter` and its values are passed to a model argument of the first lookup table instance. Top-model parameter `LUTForInst2` is mapped to an AUTOSAR `SharedParameter` and its values are passed to a model argument of the second lookup table instance.



The referenced model contains the 2-D lookup table and defines instance parameter LUT_arg. For more information about configuring instance parameters in a referenced model and specifying instance-specific values at the Model block, see “Parameterize Instances of a Reusable Referenced Model”.



When you build the top model, the exported ARXML defines application primitive data types Appl_LUTForInst1 and Appl_LUTForInst2 and maps them to implementation data type LUT_arg_Type.


```

<APPLICATION-PRIMITIVE-DATA-TYPE UUID="...">
  <SHORT-NAME>Appl_LUTForInst1</SHORT-NAME>
  <CATEGORY>MAP</CATEGORY>
  ...
</APPLICATION-PRIMITIVE-DATA-TYPE>
<APPLICATION-PRIMITIVE-DATA-TYPE UUID="...">
  <SHORT-NAME>Appl_LUTForInst2</SHORT-NAME>
  <CATEGORY>MAP</CATEGORY>
  ...
</APPLICATION-PRIMITIVE-DATA-TYPE>

<DATA-TYPE-MAP>
  <APPLICATION-DATA-TYPE-REF DEST="APPLICATION-PRIMITIVE-DATA-TYPE">
    /DataTypes/ApplDataTypes/Appl_LUTForInst1
  </APPLICATION-DATA-TYPE-REF>
  <IMPLEMENTATION-DATA-TYPE-REF DEST="IMPLEMENTATION-DATA-TYPE">
    /DataTypes/LUT_arg_Type
  </IMPLEMENTATION-DATA-TYPE-REF>
</DATA-TYPE-MAP>
<DATA-TYPE-MAP>
  <APPLICATION-DATA-TYPE-REF DEST="APPLICATION-PRIMITIVE-DATA-TYPE">
    /DataTypes/ApplDataTypes/Appl_LUTForInst2
  </APPLICATION-DATA-TYPE-REF>
  <IMPLEMENTATION-DATA-TYPE-REF DEST="IMPLEMENTATION-DATA-TYPE">
    /DataTypes/LUT_arg_Type
  </IMPLEMENTATION-DATA-TYPE-REF>
</DATA-TYPE-MAP>

```

The application primitive data types are then referenced by the AUTOSAR per-instance parameter LUTForInst1 and the AUTOSAR shared parameter LUTForInst2.

```

<PER-INSTANCE-PARAMETERS>
  <PARAMETER-DATA-PROTOTYPE UUID="...">
    <SHORT-NAME>LUTForInst1</SHORT-NAME>
    <CATEGORY>MAP</CATEGORY>
    ...
    <TYPE-TREF DEST="APPLICATION-PRIMITIVE-DATA-TYPE">
      /DataTypes/ApplDataTypes/Appl_LUTForInst1
    </TYPE-TREF>
    <INIT-VALUE>
      <CONSTANT-REFERENCE>
        <SHORT-LABEL>LUTForInst1</SHORT-LABEL>
        <CONSTANT-REF DEST="CONSTANT-SPECIFICATION">
          /DataTypes/Constants/LUTForInst1
        </CONSTANT-REF>
      </CONSTANT-REFERENCE>
    </INIT-VALUE>
  </PARAMETER-DATA-PROTOTYPE>
</PER-INSTANCE-PARAMETERS>

<SHARED-PARAMETERS>
  <PARAMETER-DATA-PROTOTYPE UUID="...">
    <SHORT-NAME>LUTForInst2</SHORT-NAME>
    <CATEGORY>MAP</CATEGORY>
    ...
    <TYPE-TREF DEST="APPLICATION-PRIMITIVE-DATA-TYPE">
      /DataTypes/ApplDataTypes/Appl_LUTForInst2
    </TYPE-TREF>
    <INIT-VALUE>
      <CONSTANT-REFERENCE>
        <SHORT-LABEL>LUTForInst2</SHORT-LABEL>
        <CONSTANT-REF DEST="CONSTANT-SPECIFICATION">
          /DataTypes/Constants/LUTForInst2
        </CONSTANT-REF>
      </CONSTANT-REFERENCE>
    </INIT-VALUE>
  </PARAMETER-DATA-PROTOTYPE>
</SHARED-PARAMETERS>

```

The exported ARXML lookup table descriptions can be round-tripped between Simulink and AUTOSAR authoring tools.

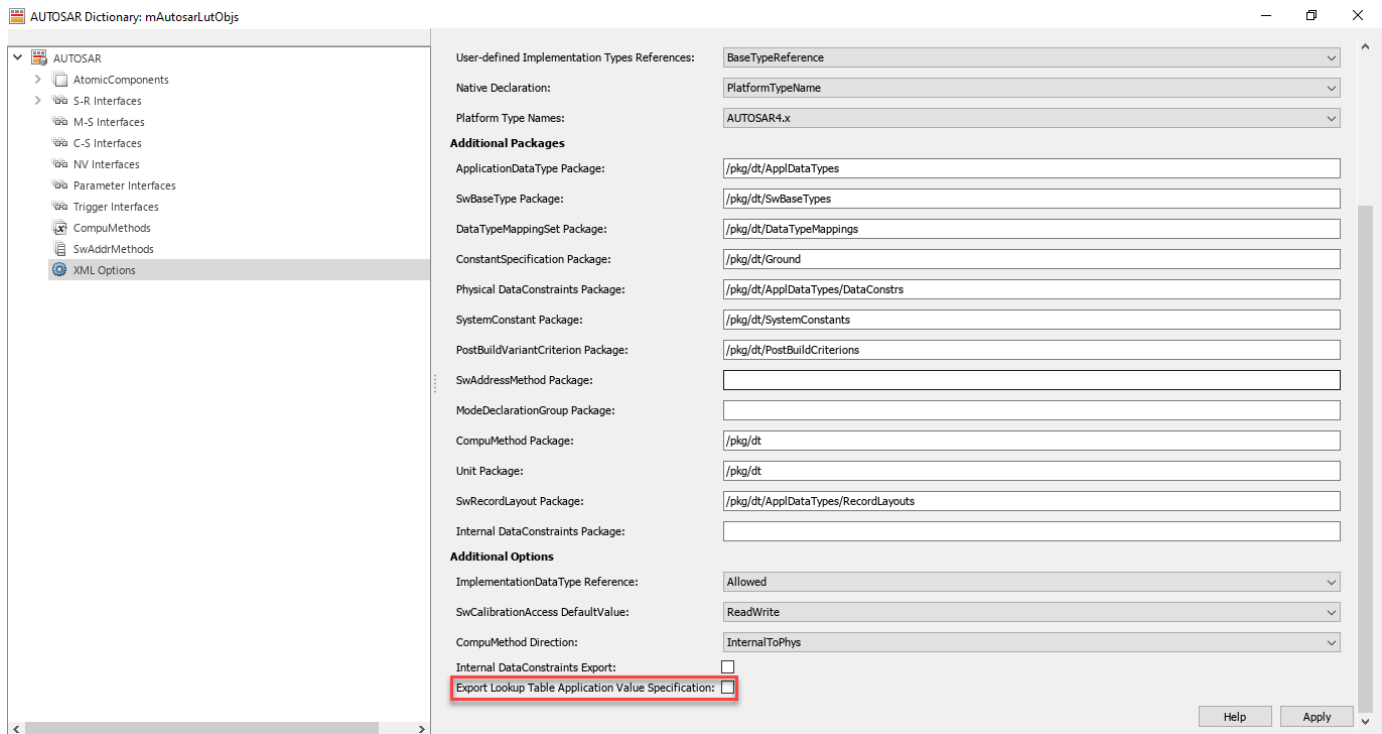
Exporting Lookup Table Constants as Record Value Specification

You can configure a model to export the lookup table constants as Application value specification or Record value specification. Refer to the example model from “Configure STD_AXIS Lookup Tables by Using Lookup Table Objects” on page 4-273.

By default the lookup table constants are exported as Application value specification. The generated ARXML file contains Application value specification as shown below:

```
<CONSTANT-SPECIFICATION UUID="...">
  <SHORT-NAME>L_4_single</SHORT-NAME>
  <VALUE-SPEC>
    <APPLICATION-VALUE-SPECIFICATION>
      <SHORT-LABEL>L_4_single</SHORT-LABEL>
      <CATEGORY>CURVE</CATEGORY>
      <SW-AXIS-CONTS>
        <SW-AXIS-CONT>
          <CATEGORY>STD_AXIS</CATEGORY>
          <UNIT-REF DEST="UNIT">/pkg/dt/NoUnit</UNIT-REF>
          <SW-AXIS-INDEX>1</SW-AXIS-INDEX>
          <SW-ARRAYSIZE>
            <V>4</V>
          </SW-ARRAYSIZE>
          <SW-VALUES-PHYS>
            <V>1</V>
            <V>2</V>
            <V>3</V>
            <V>4</V>
          </SW-VALUES-PHYS>
        </SW-AXIS-CONT>
      </SW-AXIS-CONTS>
      <SW-VALUE-CONT>
        <UNIT-REF DEST="UNIT">/pkg/dt/NoUnit</UNIT-REF>
        <SW-ARRAYSIZE>
          <V>4</V>
        </SW-ARRAYSIZE>
        <SW-VALUES-PHYS>
          <V>10</V>
          <V>20</V>
          <V>30</V>
          <V>40</V>
        </SW-VALUES-PHYS>
      </SW-VALUE-CONT>
    </APPLICATION-VALUE-SPECIFICATION>
  </VALUE-SPEC>
</CONSTANT-SPECIFICATION>
```

To export the constants as Record value specification, open the **Autosar > Autosar Dictionary** and disable the **Export Lookup Table Application Value Specification** option.



Alternatively use the command:

```
modelProperties = autosar.api.getAUTOSARProperties(<modelName>);
modelProperties.set('XmlOptions', 'ExportLookupTableApplicationValueSpecification', false);
```

Build the model. The generated ARXML contains the Record Value Specification as shown below.

```
<CONSTANT-SPECIFICATION UUID="...">
  <SHORT-NAME>L_4_single</SHORT-NAME>
  <VALUE-SPEC>
    <RECORD-VALUE-SPECIFICATION>
      <SHORT-LABEL>L_4_single</SHORT-LABEL>
      <FIELDS>
        <NUMERICAL-VALUE-SPECIFICATION>
          <SHORT-LABEL>Nx</SHORT-LABEL>
          <VALUE>4</VALUE>
        </NUMERICAL-VALUE-SPECIFICATION>
        <ARRAY-VALUE-SPECIFICATION>
          <SHORT-LABEL>Bp1</SHORT-LABEL>
          <ELEMENTS>
            <NUMERICAL-VALUE-SPECIFICATION>
              <SHORT-LABEL>Bp1_rt_Array_Float_4_1</SHORT-LABEL>
              <VALUE>1</VALUE>
            </NUMERICAL-VALUE-SPECIFICATION>
            <NUMERICAL-VALUE-SPECIFICATION>
              <SHORT-LABEL>Bp1_rt_Array_Float_4_2</SHORT-LABEL>
              <VALUE>2</VALUE>
            </NUMERICAL-VALUE-SPECIFICATION>
            <NUMERICAL-VALUE-SPECIFICATION>
              <SHORT-LABEL>Bp1_rt_Array_Float_4_3</SHORT-LABEL>
              <VALUE>3</VALUE>
            </NUMERICAL-VALUE-SPECIFICATION>
            <NUMERICAL-VALUE-SPECIFICATION>
              <SHORT-LABEL>Bp1_rt_Array_Float_4_4</SHORT-LABEL>
              <VALUE>4</VALUE>
            </NUMERICAL-VALUE-SPECIFICATION>
          </ELEMENTS>
        </ARRAY-VALUE-SPECIFICATION>
      </FIELDS>
    </RECORD-VALUE-SPECIFICATION>
  </VALUE-SPEC>
</CONSTANT-SPECIFICATION>
```

```

        </NUMERICAL-VALUE-SPECIFICATION>
    </ELEMENTS>
</ARRAY-VALUE-SPECIFICATION>
<ARRAY-VALUE-SPECIFICATION>
    <SHORT-LABEL>Table</SHORT-LABEL>
    <ELEMENTS>
        <NUMERICAL-VALUE-SPECIFICATION>
            <SHORT-LABEL>Table_rt_Array_Float_4_1</SHORT-LABEL>
            <VALUE>10</VALUE>
        </NUMERICAL-VALUE-SPECIFICATION>
        <NUMERICAL-VALUE-SPECIFICATION>
            <SHORT-LABEL>Table_rt_Array_Float_4_2</SHORT-LABEL>
            <VALUE>20</VALUE>
        </NUMERICAL-VALUE-SPECIFICATION>
        <NUMERICAL-VALUE-SPECIFICATION>
            <SHORT-LABEL>Table_rt_Array_Float_4_3</SHORT-LABEL>
            <VALUE>30</VALUE>
        </NUMERICAL-VALUE-SPECIFICATION>
        <NUMERICAL-VALUE-SPECIFICATION>
            <SHORT-LABEL>Table_rt_Array_Float_4_4</SHORT-LABEL>
            <VALUE>40</VALUE>
        </NUMERICAL-VALUE-SPECIFICATION>
    </ELEMENTS>
</ARRAY-VALUE-SPECIFICATION>
</FIELDS>
</RECORD-VALUE-SPECIFICATION>
</VALUE-SPEC>
</CONSTANT-SPECIFICATION>

```

Exporting AdminData Record Layout Annotations

AUTOSAR Blockset supports AdminData record layout annotations in ARXML lookup table descriptions.

Importing ARXML lookup table and axis descriptions that contain AdminData record layout annotations creates Simulink® lookup tables and breakpoints in which the AdminData annotations determine the order of structure elements.

Exporting lookup table AdminData is disabled by default. To enable export of AdminData, set the API-only XML option 'ExportSwRecordLayoutAnnotationsOnAdminData' to true. For example:

```

hModel = 'mAutosarLutObjs';
open_system(hModel);
arProps = autosar.api.getAUTOSARProperties(hModel);
set(arProps, 'XmlOptions', 'ExportSwRecordLayoutAnnotationsOnAdminData', true);
slbuild(hModel)

```

```

### Starting build procedure for: mAutosarLutObjs

```

```

Model "mAutosarLutObjs" contains deprecated AUTOSAR 3.x platform types that will
be automatically converted to AUTOSAR 4.x platform type names in future
releases.

```

```

To convert from AUTOSAR 3.x names to AUTOSAR 4.x platform type names directly,
use the 'PlatformTypeNames' XML Option.

```

```

### Generating code and artifacts to 'Model specific' folder structure
### Generating code into build folder: C:\TEMP\Bdoc23a_2213998_3568\ib570499\8\tp91c006b8\autosar
### Invoking Target Language Compiler on mAutosarLutObjs.rtw
### Using System Target File: B:\matlab\toolbox\rtw\targets\AUTOSAR\AUTOSAR\autosar.tlc
### Loading TLC function libraries
.....

```

```

### Initial pass through model to cache user defined code
### Caching model source code
.....
### Writing header file mAutosarLutObjs_types.h
### Writing header file mAutosarLutObjs.h
### Writing header file rtwtypes.h
### Writing source file mAutosarLutObjs.c
.
### Writing header file mAutosarLutObjs_private.h
### Writing source file mAutosarLutObjs_data.c
### TLC code generation complete (took 7.259s).
### Saving binary information cache.
### Generating XML files description for: mAutosarLutObjs
### Using toolchain: Microsoft Visual C++ 2019 v16.0 | nmake (64-bit Windows)
### Creating 'C:\TEMP\Bdoc23a_2213998_3568\ib570499\8\tp91c006b8\autosarblockset-ex86401155\mAutosarLutObjs'
### Creating HTML report file index.html
### Successful completion of code generation for: mAutosarLutObjs
### Simulink cache artifacts for 'mAutosarLutObjs' were created in 'C:\TEMP\Bdoc23a_2213998_3568'

```

Build Summary

Top model targets built:

Model	Action	Rebuild Reason
mAutosarLutObjs	Code generated.	Code generation information file does not exist.

1 of 1 models built (0 models already up to date)
 Build duration: 0h 1m 1.8141s

When AdminData export is enabled, exporting Simulink® lookup tables and breakpoints with structure elements generates lookup table and axis ImplementationDataTypes that include structure element AdminData annotations. For example:

```

<IMPLEMENTATION-DATA-TYPE UUID="...">
  <SHORT-NAME>LUT_4_single</SHORT-NAME>
  <CATEGORY>STRUCTURE</CATEGORY>
  <SUB-ELEMENTS>
    <IMPLEMENTATION-DATA-TYPE-ELEMENT UUID="...">
      <SHORT-NAME>Nx</SHORT-NAME>
      <CATEGORY>TYPE_REFERENCE</CATEGORY>
      <ADMIN-DATA>
        <SDGS>
          <SDG GID="DV:RecLayoutAnnotation">
            <SD GID="DV:Type">NO_AXIS_PTS_X</SD>
          </SDG>
        </SDGS>
      </ADMIN-DATA>
    </IMPLEMENTATION-DATA-TYPE-ELEMENT>
    <IMPLEMENTATION-DATA-TYPE-ELEMENT UUID="...">
      <SHORT-NAME>Bp1</SHORT-NAME>
      <CATEGORY>TYPE_REFERENCE</CATEGORY>
      <ADMIN-DATA>
        <SDGS>
          <SDG GID="DV:RecLayoutAnnotation">
            <SD GID="DV:Type">AXIS_PTS_X</SD>
          </SDG>
        </SDGS>
      </ADMIN-DATA>
    </IMPLEMENTATION-DATA-TYPE-ELEMENT>
  </SUB-ELEMENTS>
</IMPLEMENTATION-DATA-TYPE>

```

```
        </SDGS>
    </ADMIN-DATA>
    ..
</IMPLEMENTATION-DATA-TYPE-ELEMENT>
    ..
</SUB-ELEMENTS>
</IMPLEMENTATION-DATA-TYPE>
```

AdminData record layout annotations can be used with third-party AUTOSAR tools.

See Also

[Simulink.LookupTable](#) | [Simulink.Breakpoint](#) | [Curve](#) | [Curve Using Prelookup](#) | [Map](#) | [Map Using Prelookup](#) | [Prelookup](#) | [getParameter](#) | [mapParameter](#)

Related Examples

- “Map Model Workspace Parameters to AUTOSAR Component Parameters” on page 4-54

More About

- “Code Generation with AUTOSAR Code Replacement Library” on page 5-12
- “Model AUTOSAR Calibration Parameters and Lookup Tables” on page 2-50

Configure and Map AUTOSAR Component Programmatically

In Simulink, as an alternative to graphical configuration, you can programmatically configure an AUTOSAR software component. The AUTOSAR property and map functions allow you to get, set, add, and remove the same component properties and mapping information displayed in the AUTOSAR Dictionary and Code Mappings editor views of the AUTOSAR component model.

AUTOSAR Property and Map Functions

You can use AUTOSAR property and map functions to programmatically configure the Simulink representation of an AUTOSAR software component. For example:

- Use the AUTOSAR property functions to add AUTOSAR elements, find elements, get and set properties of elements, delete elements, and define ARXML packaging of elements.
- Use the AUTOSAR map functions to map Simulink model elements to AUTOSAR elements and return AUTOSAR mapping information for model elements.

The AUTOSAR property and map functions also validate syntax and semantics for requested AUTOSAR property and mapping changes.

For a complete list of property and map functions, see the functions listed for “Component Development”.

For example scripts, see “AUTOSAR Property and Map Function Examples” on page 4-299.

Note For information about functions for creating or importing an AUTOSAR software component, see “Component Creation”.

Tree View of AUTOSAR Configuration

The following tree view of an AUTOSAR configuration shows the types of AUTOSAR elements to which you can apply AUTOSAR property and map functions. This view corresponds with the AUTOSAR Dictionary tree display, but includes elements that might not be present in every configuration. Names shown in *italics* are user-selected.

- AUTOSAR
 - AtomicComponents
 - *MyComponent*
 - ReceiverPorts
 - SenderPorts
 - SenderReceiverPorts
 - ModeReceiverPorts
 - ModeSenderPorts
 - ClientPorts
 - ServerPorts

- NvReceiverPorts
- NvSenderPorts
- NvSenderReceiverPorts
- ParameterReceiverPorts
- TriggerReceiverPorts
- Runnables
- IRV
- Parameters
- S-R Interfaces
 - *SRInterface1*
 - DataElements
- M-S Interfaces
 - *MSInterface1*
- C-S Interfaces
 - *CSInterface1*
 - Operations
 - *operation1*
 - Arguments
- NV Interfaces
 - *NVInterface1*
 - DataElements
- Parameter Interfaces
 - *ParameterInterface1*
 - DataElements
- Trigger Interfaces
 - *TriggerInterface1*
 - Triggers
- CompuMethods
- XML Options

Properties of AUTOSAR Elements

The following table lists properties that are associated with AUTOSAR elements.

AUTOSAR Element Class	Properties
AtomicComponent	<ul style="list-style-type: none"> • ReceiverPorts (add/delete) • SenderPorts (add/delete) • SenderReceiverPorts (add/delete) • ModeReceiverPorts (add/delete) • ClientPorts (add/delete) • ServerPorts (add/delete) • NvReceiverPorts (add/delete) • NvSenderPorts (add/delete) • NvSenderReceiverPorts (add/delete) • ParameterReceiverPorts (add/delete) • TriggerReceiverPorts (add/delete) • Behavior (add/delete) • Kind • Name
ApplicationComponentBehavior	<ul style="list-style-type: none"> • Runnables (add/delete) • Events (add/delete) • PIM (add/delete) • IRV (add/delete) • Parameters (add/delete) • IncludedDataTypeSets • DataTypeMapping • Name
DataReceiverPort DataSenderPort DataSenderReceiverPort ClientPort ServerPort ModeReceiverPort NvDataReceiverPort NvDataSenderPort NvDataSenderReceiverPort ParameterReceiverPort TriggerReceiverPort	<ul style="list-style-type: none"> • Interface • Name
Runnable	<ul style="list-style-type: none"> • symbol • canBeInvokedConcurrently • SwAddrMethod • Name

AUTOSAR Element Class	Properties
TimingEvent	<ul style="list-style-type: none"> • Period • StartOnEvent • DisabledMode • Name
DataReceivedEvent DataReceiveErrorEvent OperationInvokedEvent	<ul style="list-style-type: none"> • Trigger • StartOnEvent • DisabledMode • Name
ModeSwitchEvent	<ul style="list-style-type: none"> • Trigger • Activation • StartOnEvent • DisabledMode • Name
InitEvent	<ul style="list-style-type: none"> • StartOnEvent • Name
IrvData	<ul style="list-style-type: none"> • Type • SwAddrMethod • SwCalibrationAccess • DisplayFormat • SwAlignment • Name
ParameterData	<ul style="list-style-type: none"> • Type • SwAddrMethod • SwCalibrationAccess • DisplayFormat • SwAlignment • Kind • Name
SenderReceiverInterface NvDataInterface ParameterInterface	<ul style="list-style-type: none"> • DataElements (add/delete) • IsService • Name
FlowData	<ul style="list-style-type: none"> • Type • SwAddrMethod • SwCalibrationAccess • DisplayFormat • SwAlignment • Name

AUTOSAR Element Class	Properties
ModeSwitchInterface	<ul style="list-style-type: none"> ModeGroup (add/delete) IsService Name
ModeDeclarationGroupElement	<ul style="list-style-type: none"> ModeGroup SwCalibrationAccess Name
ClientServerInterface	<ul style="list-style-type: none"> Operations (add/delete) IsService Name
TriggerInterface	<ul style="list-style-type: none"> Triggers (add/delete) IsService Name

Specify AUTOSAR Element Location

The AUTOSAR property functions typically require you to specify the name and location of an element. The location of an AUTOSAR element within a hierarchy of AUTOSAR packages and objects can be uniquely specified using a fully qualified path. A fully qualified path might include a package hierarchy and the element location within the object hierarchy, for example:

```
/pkgLevel1/pkgLevel2/pkgLevel3/grandParentName/parentName/childName
```

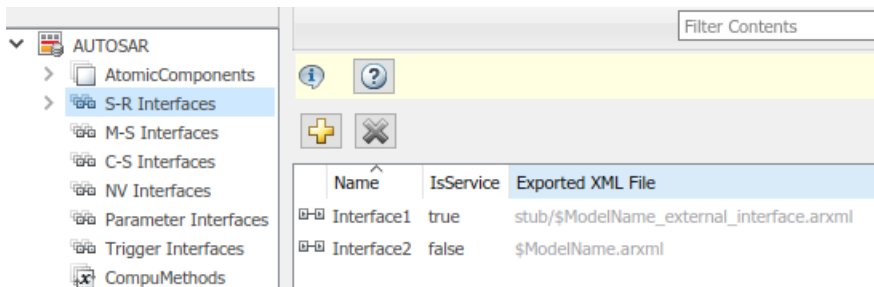
For AUTOSAR property functions other than `addPackageableElement`, you can specify a partially-qualified path that does not include the package hierarchy, for example:

```
grandParentName/parentName/childName
```

The following code sets the `IsService` property for the Sender-Receiver Interface located at path `Interface1` in the example model `autosar_swc_expfncs` to `true`. In this case, specifying the name `Interface1` is enough to locate the element.

```
hModel = 'autosar_swc_expfncs';
openExample(hModel);
arProps = autosar.api.getAUTOSARProperties(hModel);
set(arProps, 'Interface1', 'IsService', true);
```

Here is the resulting display in the **S-R Interfaces** view in the AUTOSAR Dictionary.



If you added a Sender-Receiver Interface to a component package, you would specify a fully qualified path, for example:

```
hModel = 'autosar_sw_c_expfncs';
openExample(hModel);
arProps = autosar.api.getAUTOSARProperties(hModel);
addPackageableElement(arProps, 'SenderReceiverInterface', '/pkg/if', 'Interface3', ...
    'IsService', true);
```

A potential advantage of using a partially qualified path rather than a fully-qualified path is that it is easier to construct a partially qualified path from looking at the AUTOSAR Dictionary view of the AUTOSAR component. A potential disadvantage is that a partially qualified path could refer to more than one element in the AUTOSAR configuration. For example, the path `s/r` conceivably might designate both a data element of a Sender-Receiver Interface and a runnable of a component. When a conflict occurs, the software displays an error and lists the fully-qualified paths.

Most AUTOSAR elements have properties that are made up of multiple parts (composite). For example, an atomic software component has composite properties such as `ReceiverPorts`, `SenderPorts`, and `InternalBehavior`. For elements that have composite properties that you can manipulate, such as property `ReceiverPorts` of a component, child elements are named and are uniquely defined within the parent element. To locate a child element within a composite property, use the parent element path and the child name, without the property name. For example, if the qualified path of a parent atomic software component is `/A/B/SWC`, and a child receiver port is named `RPort1`, the location of the receiver port is `/A/B/SWC/RPort1`.

AUTOSAR Property and Map Function Examples

After creating a Simulink model representation of an AUTOSAR software component, you refine the AUTOSAR configuration. You can refine the AUTOSAR configuration graphically, using the AUTOSAR Dictionary and the Code Mappings editor, or programmatically, using the AUTOSAR property and map functions.

This topic provides examples of using AUTOSAR property and map functions to programmatically refine an AUTOSAR configuration. The examples assume that you have created a Simulink model with an initial AUTOSAR configuration, as described in “Component Creation”. (To graphically refine an AUTOSAR configuration, see “AUTOSAR Component Configuration” on page 4-3.)

Here is representative ordering of programmatic configuration tasks.

- 1 “Configure AUTOSAR Software Component” on page 4-300
 - a “Configure AUTOSAR Software Component Name and Type” on page 4-300
 - b “Configure AUTOSAR Ports” on page 4-300
 - c “Configure AUTOSAR Runnables and Events” on page 4-304
 - d “Configure AUTOSAR Inter-Runnable Variables” on page 4-309
- 2 “Configure AUTOSAR Interfaces” on page 4-311
 - a “Configure AUTOSAR Sender-Receiver Interfaces” on page 4-312
 - b “Configure AUTOSAR Client-Server Interfaces” on page 4-314
 - c “Configure AUTOSAR Mode-Switch Interfaces” on page 4-317
- 3 “Configure AUTOSAR XML Export” on page 4-319

For a list of AUTOSAR property and map functions, see the **Functions** list on the “AUTOSAR Programmatic Interfaces” page.

The examples use a function call format in which a handle to AUTOSAR properties or mapping information is passed as the first call argument:

```
arProps = autosar.api.getAUTOSARProperties(hModel);
swc = get(arProps, 'XmlOptions', 'ComponentQualifiedNames');
```

The same calls can be coded in a method call format. The formats are interchangeable. For example:

```
arProps = autosar.api.getAUTOSARProperties(hModel);
swc = arProps.get('XmlOptions', 'ComponentQualifiedNames');
```

While configuring a model for AUTOSAR code generation, use the following functions to update and validate AUTOSAR model configurations:

- `autosar.api.syncModel` — Update Simulink to AUTOSAR mapping of specified model with modifications to Simulink entry-point functions, data transfers, and function callers.
- `autosar.api.validateModel` — Validate AUTOSAR properties and Simulink to AUTOSAR mapping of specified model.

The functions are equivalent to using the **Update**  and **Validate**  buttons in the Code Mappings editor.

Configure AUTOSAR Software Component

- “Configure AUTOSAR Software Component Name and Type” on page 4-300
- “Configure AUTOSAR Ports” on page 4-300
- “Configure AUTOSAR Runnables and Events” on page 4-304
- “Configure AUTOSAR Inter-Runnable Variables” on page 4-309

Configure AUTOSAR Software Component Name and Type

This example:

- 1 Opens a model.
- 2 Finds AUTOSAR software components.
- 3 Loops through components and lists property values.
- 4 Modifies the name and kind properties for a component.

```
% Open model
hModel = 'autosar_sw_c_expfncs';
openExample(hModel);

% Use AUTOSAR property functions
arProps = autosar.api.getAUTOSARProperties(hModel);

% Find AUTOSAR software components
aswcPaths = find(arProps,[],'AtomicComponent','PathType','FullyQualified');

% Loop through components and list Name and Kind property values
for ii=1:length(aswcPaths)
    aswcPath = aswcPaths{ii};
    swcName = get(arProps,aswcPath,'Name');
    swcKind = get(arProps,aswcPath,'Kind'); % Application, SensorActuator, etc.
    fprintf('Component %s: Name %s, Kind %s\n',aswcPath,swcName,swcKind);
end

Component /pkg/swc/ASWC: Name ASWC, Kind Application

% Modify component Name and Kind
aswcName = 'mySwc';
aswcKind = 'SensorActuator';
set(arProps,aswcPaths{1},'Name',aswcName);
aswcPaths = find(arProps,[],'AtomicComponent','PathType','FullyQualified');
set(arProps,aswcPaths{1},'Kind',aswcKind);
swcName = get(arProps,aswcPaths{1},'Name');
swcKind = get(arProps,aswcPaths{1},'Kind');
fprintf('Component %s: Name %s, Kind %s\n',aswcPaths{1},swcName,swcKind);

Component /pkg/swc/mySwc: Name mySwc, Kind SensorActuator
```

Configure AUTOSAR Ports

There are three types of AUTOSAR ports:

- Require (In)
- Provide (Out)
- Combined Provide-Require (InOut)

AUTOSAR ports can reference the following kinds of AUTOSAR communication interfaces:

- Sender-Receiver
- Client-Server
- Mode-Switch

The properties and mapping that you can set for an AUTOSAR port vary according to the type of interface it references. These examples show how to use the AUTOSAR property and map functions to configure AUTOSAR ports for each type of interface.

- “Configure and Map Sender-Receiver Ports” on page 4-301
- “Configure Client-Server Ports” on page 4-302
- “Configure and Map Mode Receiver Ports” on page 4-303

Configure and Map Sender-Receiver Ports

This example:

- 1 Opens a model.
- 2 Finds AUTOSAR sender or receiver ports.
- 3 Loops through the ports and lists associated sender-receiver interfaces.
- 4 Modifies the associated interface for a port.
- 5 Maps a Simulink inport to an AUTOSAR receiver port.

See also “Configure AUTOSAR Sender-Receiver Interfaces” on page 4-312.

```
% Open model
hModel = 'autosar_sw_c_expfncs';
openExample(hModel);

% Use AUTOSAR property functions
arProps = autosar.api.getAUTOSARProperties(hModel);

% Find AUTOSAR ports - specify DataReceiverPort, DataSenderPort, or DataSenderReceiverPort
arPortType = 'DataReceiverPort';
aswcPath = find(arProps, [], 'AtomicComponent', 'PathType', 'FullyQualified');
rPorts=find(arProps,aswcPath{1},arPortType, 'PathType', 'FullyQualified')

rPorts =
    {'/pkg/swc/ASWC/RPort'}

% Loop through ports and list their associated interfaces
for ii=1:length(rPorts)
    rPort = rPorts{ii};
    portIf = get(arProps, rPort, 'Interface');
    fprintf('Port %s has S-R interface %s\n', rPort, portIf);
end

Port /pkg/swc/ASWC/RPort has S-R interface Interface1

% Set Interface property for AUTOSAR port
rPort = '/pkg/swc/ASWC/RPort';
set(arProps, rPort, 'Interface', 'Interface2')
portIf = get(arProps, rPort, 'Interface');
fprintf('Port %s has S-R interface %s\n', rPort, portIf);

Port /pkg/swc/ASWC/RPort has S-R interface Interface2

% Use AUTOSAR map functions
sLMap=autosar.api.getSimulinkMapping(hModel);

% Get AUTOSAR mapping info for entry point functions
functions = find(sLMap, "Functions")

functions =

    4x1 string array

    "Initialize"
    "ExportedFunction:Runnable1"
    "ExportedFunction:Runnable2"
    "ExportedFunction:Runnable3"
```

```

% Get AUTOSAR mapping info for a specific Simulink inport
[arPortName,arDataElementName,arDataAccessMode]=getInport(sLMap,'RPort_DE2')

arPortName =
    'RPort'

arDataElementName =
    0x0 empty char array

arDataAccessMode =
    'ImplicitReceive'

% Map Simulink inport to AUTOSAR port, data element, and data access mode
mapInport(sLMap,'RPort_DE2','RPort','DE2','ExplicitReceive')
[arPortName,arDataElementName,arDataAccessMode]=getInport(sLMap,'RPort_DE2')

arPortName =
    RPort

arDataElementName =
    DE2

arDataAccessMode =
    ExplicitReceive

```

Configure Client-Server Ports

Find either client or server ports of a model. List the associated client-server interfaces, and modify the associated interface for a port of that model.

See also: “Configure AUTOSAR Client-Server Interfaces” on page 4-314.

Create and open model 'mControllerWithInterface_server'.

```

hModel = 'mControllerWithInterface_server';
open_system(hModel);

```

Retrieve AUTOSAR property functions.

```

arProps = autosar.api.getAUTOSARProperties(hModel);

```

Find AUTOSAR ports - specify either ServerPort or ClientPort.

```

arPortType = 'ServerPort';
aswcPath = find(arProps,[],'AtomicComponent','PathType','FullyQualified');
sPorts=find(arProps,aswcPath{1},arPortType,'PathType','FullyQualified');

```

Loop through ports and list the associated interfaces of the found AUTOSAR ports.

```

for ii=1:length(sPorts)
    sPort = sPorts{ii};
    portIf = get(arProps,sPort,'Interface');
    fprintf('Port %s has C-S interface %s\n',sPort,portIf);
end

```

```

Port /pkg/swc/SWC_Controller/sPort has C-S interface CsIf1

```

Set 'Interface' property for the found AUTOSAR port.

```

set(arProps,sPorts{1},'Interface','CsIf2')
portIf = get(arProps,sPorts{1},'Interface');
fprintf('Port %s has C-S interface %s\n',sPorts{1},portIf);

```

```

Port /pkg/swc/SWC_Controller/sPort has C-S interface CsIf2

```


Configure and Map Mode Receiver Ports

Using model 'mAutosarMsConfigAfter' find AUTOSAR mode receiver ports, loop through and list the associated mode-switch interfaces for these ports. Modify the associated interface for a port. Map a Simulink® inport to an AUTOSAR model receiver port.

See also: “Configure AUTOSAR Mode-Switch Interfaces” on page 4-317.

Open model 'mAutosarMsConfigAfter'.

```
hModel = 'mAutosarMsConfigAfter';
open_system(hModel);
```

Retrieve AUTOSAR property functions.

```
arProps = autosar.api.getAUTOSARProperties(hModel);
```

Find AUTOSAR mode receiver ports.

```
arPortType = 'ModeReceiverPort';
aswcPath = find(arProps,[],'AtomicComponent','PathType','FullyQualified');
mrPorts=find(arProps,aswcPath{1},arPortType,'PathType','FullyQualified');
```

Loop through ports and list their associated interfaces.

```
for ii=1:length(mrPorts)
    mrPort = mrPorts{ii};
    portIf = get(arProps,mrPort,'Interface');
    fprintf('Port %s has M-S interface %s\n',mrPort,portIf);
end
```

```
Port /pkg/swc/ASWC/myMRPort has M-S interface myMsIf
```

Set 'Interface' property for AUTOSAR port.

```
set(arProps,mrPorts{1},'Interface','MsIf2')
portIf = get(arProps,mrPort,'Interface');
fprintf('Port %s has M-S interface %s\n',mrPorts{1},portIf);
```

```
Port /pkg/swc/ASWC/myMRPort has M-S interface MsIf2
```

Use AUTOSAR map functions.

```
slMap=autosar.api.getSimulinkMapping(hModel);
```

Get AUTOSAR mapping info for Simulink inport

```
[arPortName,arDataElementName,arDataAccessMode]=getInport(slMap,'MRPort')
```

```
arPortName =
'myMRPort'
```

```
arDataElementName =
```

```
    0x0 empty char array
```

```
arDataAccessMode =
'ModeReceive'
```

Map Simulink inport to AUTOSAR port, mode group, and data access mode.

```
mapInport(slMap, 'MRPort', 'myMRPort', 'mdgModes', 'ModeReceive')
[arPortName, arDataElementName, arDataAccessMode]=getInport(slMap, 'MRPort')

arPortName =
'myMRPort'

arDataElementName =
'mdgModes'

arDataAccessMode =
'ModeReceive'
```

Configure AUTOSAR Runnables and Events

The behavior of an AUTOSAR software component is implemented by one or more runnables. An AUTOSAR runnable is a schedulable entity that is directly or indirectly scheduled by the underlying AUTOSAR operating system. Each runnable is triggered by RTEEvents, events generated by the AUTOSAR run-time environment (RTE). For each runnable, you configure an event to which it responds. Here are examples of AUTOSAR events to which runnables respond.

- **TimingEvent** — Triggers a periodic runnable.
- **DataReceivedEvent** or **DataReceiveErrorEvent** — Triggers a runnable with a receiver port that is participating in sender-receiver communication.
- **OperationInvokedEvent** — Triggers a runnable with a server port that is participating in client-server communication.
- **ModeSwitchEvent** — Triggers a runnable with a mode receiver port that is participating in mode-switch communication.
- **InitEvent** (AUTOSAR schema 4.1 or higher) — Triggers a runnable that performs component initialization.
- **ExternalTriggerOccurredEvent** — Triggers a runnable with a trigger receiver port that is participating in external trigger event communication.
- “Configure AUTOSAR TimingEvent for Periodic Runnable” on page 4-304
- “Configure and Map Runnables” on page 4-305
- “Retrieve and Map Signals and States” on page 4-306
- “Configure Events for Runnable Activation” on page 4-307
- “Gather Information for AUTOSAR Custom Scheduler Script” on page 4-308

Configure AUTOSAR TimingEvent for Periodic Runnable

This example:

- 1 Opens a model.
- 2 Finds AUTOSAR runnables.
- 3 Loops through runnables and lists properties.
- 4 Modifies the name and symbol for an AUTOSAR periodic runnable.
- 5 Loops through AUTOSAR timing events and lists associated runnables.
- 6 Renames an AUTOSAR timing event.

7 Maps a Simulink entry-point function to an AUTOSAR periodic runnable.

```

% Open model
hModel = 'autosar_swc_expfncs';
openExample(hModel);

% Use AUTOSAR property functions
arProps = autosar.api.getAUTOSARProperties(hModel);

% Find AUTOSAR runnables
swc = get(arProps,'XmlOptions','ComponentQualified');
ib = get(arProps,swc,'Behavior');
runnables = find(arProps,ib,'Runnable','PathType','FullyQualified');
runnables{2}

ans =
    '/pkg/swc/ASWC/IB/Runnable1'

% Loop through runnables and list property values
for ii=1:length(runnables)
    runnable = runnables{ii};
    rnName = get(arProps,runnable,'Name');
    rnSymbol = get(arProps,runnable,'symbol');
    rnCBIC = get(arProps,runnable,'canBeInvokedConcurrently');
    fprintf('Runnable %s: symbol %s, canBeInvokedConcurrently %u\n',...
        rnName, rnSymbol, rnCBIC);
end

Runnable Runnable_Init: symbol Runnable_Init, canBeInvokedConcurrently 0
Runnable Runnable1: symbol Runnable1, canBeInvokedConcurrently 0
Runnable Runnable2: symbol Runnable2, canBeInvokedConcurrently 0
Runnable Runnable3: symbol Runnable3, canBeInvokedConcurrently 0

% Modify Runnable1 name and symbol
set(arProps,runnables{2},'Name','myRunnable','symbol','myAlgorithm');
runnables = find(arProps,ib,'Runnable','PathType','FullyQualified');
rnName = get(arProps,runnables{2},'Name');
rnSymbol = get(arProps,runnables{2},'symbol');
rnCBIC = get(arProps,runnables{2},'canBeInvokedConcurrently');
fprintf('Runnable %s: symbol %s, canBeInvokedConcurrently %u\n',...
    rnName, rnSymbol, rnCBIC);

Runnable myRunnable: symbol myAlgorithm, canBeInvokedConcurrently 0

% Loop through AUTOSAR timing events and list runnable associations
events = find(arProps,ib,'TimingEvent','PathType','FullyQualified');
for ii=1:length(events)
    event = events{ii};
    eventStart0n = get(arProps,event,'Start0nEvent');
    fprintf('AUTOSAR event %s triggers %s\n',event,eventStart0n);
end

AUTOSAR event /pkg/swc/ASWC/IB/Event_t_ltic_A triggers ASWC/IB/myRunnable
AUTOSAR event /pkg/swc/ASWC/IB/Event_t_ltic_B triggers ASWC/IB/Runnable2
AUTOSAR event /pkg/swc/ASWC/IB/Event_t_l0tic triggers ASWC/IB/Runnable3

% Modify AUTOSAR event name
set(arProps,events{1},'Name','myEvent');
events = find(arProps,ib,'TimingEvent','PathType','FullyQualified');
eventStart0n = get(arProps,events{1},'Start0nEvent');
fprintf('AUTOSAR event %s triggers %s\n',events{1},eventStart0n);

AUTOSAR event /pkg/swc/ASWC/IB/myEvent triggers ASWC/IB/myRunnable

% Use AUTOSAR map functions
slMap=autosar.api.getSimulinkMapping(hModel);

% Map Simulink exported function Runnable1 to renamed AUTOSAR runnable
mapFunction(slMap,'Runnable1','myRunnable');
arRunnableName = getFunction(slMap,'Runnable1')

arRunnableName =
    'myRunnable'

```

Configure and Map Runnables

This example:

- 1 Opens a model.

- 2 Adds AUTOSAR initialization and periodic runnables to the model.
- 3 Adds a timing event to the periodic runnable.
- 4 Maps Simulink initialization and step functions to the AUTOSAR runnables.

See also “Configure Events for Runnable Activation” on page 4-307.

```
% Open model
hModel = 'autosar_swc_counter';
openExample(hModel);

% Use AUTOSAR property functions
arProps = autosar.api.getAUTOSARProperties(hModel);

% Add AUTOSAR initialization and periodic runnables
initRunnable = 'myInitRunnable';
periodicRunnable = 'myPeriodicRunnable';
swc = get(arProps, 'XmlOptions', 'ComponentQualifiedNames');
ib = get(arProps, swc, 'Behavior');
add(arProps, ib, 'Runnables', initRunnable);
add(arProps, ib, 'Runnables', periodicRunnable);

% Add AUTOSAR timing event
eventName = 'myPeriodicEvent';
add(arProps, ib, 'Events', eventName, 'Category', 'TimingEvent', 'Period', 1, ...
    'StartOnEvent', [ib '/' periodicRunnable]);

% Use AUTOSAR map functions
slMap=autosar.api.getSimulinkMapping(hModel);

% Map AUTOSAR runnables to Simulink initialize and step functions
mapFunction(slMap, 'InitializeFunction', initRunnable);
mapFunction(slMap, 'StepFunction', periodicRunnable);

% To pass validation, remove redundant initialize and step runnables in AUTOSAR configuration
runnables = get(arProps, ib, 'Runnables');
delete(arProps, [ib, '/Runnable_Init']);
delete(arProps, [ib, '/Runnable_Step']);
runnables = get(arProps, ib, 'Runnables')

swc =
    '/Company/Powertrain/Components/autosar_swc_counter'

ib =
    'autosar_swc_counter/ASWC_IB'

runnables =
    {'autosar_swc_counter/ASWC_IB/myInitRunnable'}
    {'autosar_swc_counter/ASWC_IB/myPeriodicRunnable'}
```

Retrieve and Map Signals and States

This example:

- 1 Opens a model.
- 2 Creates a Simulink Mapping object.
- 3 Retrieve Signals and States.
- 4 Map Signals and States.

```
% Open model
hModel = "autosar_swc_counter";
openExample(hModel);

% Use AUTOSAR map functions
slMap=autosar.api.getSimulinkMapping(hModel);

% Use AUTOSAR mapping find function to retrieve signals
listOfSignals = find(slMap, "Signals")
namesOfSignals = get_param(listOfSignals, "Name")

listOfSignals =
    36.0001  37.0001  38.0001
```

```

namesOfSignals =
    3x1 cell array

    {'equal_to_count'}
    {'sum_out'       }
    {'switch_out'   }

% Use AUTOSAR mapping find function to retrieve States
listOfStates = find(sLMMap,"States")

ans =

    "autosar_sw_c_counter/X"

% Map Signals
mapSignal(sLMMap, listOfSignals(1),'StaticMemory')
mapSignal(sLMMap, listOfSignals(2),'ArTypedPerInstanceMemory')
mapSignal(sLMMap, listOfSignals(3),'Auto')

% Map States
mapState(sLMMap, 'autosar_sw_c_counter/X', '', 'StaticMemory')

```

Configure Events for Runnable Activation

This example shows the property function syntax for adding an AUTOSAR TimingEvent, DataReceivedEvent, and DataReceiveErrorEvent to a runnable in a model. For a DataReceivedEvent or DataReceiveErrorEvent, you specify a trigger. The trigger name includes the name of the AUTOSAR receiver port and data element that receives the event, for example, 'RPort.DE1'.

For OperationInvokedEvent syntax, see “Configure AUTOSAR Client-Server Interfaces” on page 4-314.

For ModeSwitchEvent syntax, see “Configure AUTOSAR Mode-Switch Interfaces” on page 4-317.

```

% Open model
hModel = 'autosar_sw_c_expfncs';
openExample(hModel);

% Use AUTOSAR property functions
arProps = autosar.api.getAUTOSARProperties(hModel);

% Specify AUTOSAR runnable to which to add event
swc = get(arProps,'XmlOptions','ComponentQualifiedName')
ib = get(arProps,swc, 'Behavior')
runnables = get(arProps,ib, 'Runnables')
runnable = 'Runnable1';

% Add AUTOSAR timing event
timingEventName = 'myTimingEvent';
add(arProps,ib, 'Events', timingEventName, 'Category', 'TimingEvent', ...
    'Period', 1, 'StartOnEvent', [ib '/' runnable]);

% Add AUTOSAR data received event
drEventName = 'myDREvent';
add(arProps,ib, 'Events', drEventName, 'Category', 'DataReceivedEvent', ...
    'Trigger', 'RPort.DE1', 'StartOnEvent', [ib '/' runnable]);

% Add AUTOSAR data receive error event
dreEventName = 'myDREEvent';
add(arProps,ib, 'Events', dreEventName, 'Category', 'DataReceiveErrorEvent', ...
    'Trigger', 'RPort.DE1', 'StartOnEvent', [ib '/' runnable]);

% To pass validation, remove redundant timing event in AUTOSAR configuration
events = get(arProps,ib, 'Events');
delete(arProps, [ib, '/Event_t_1tic_A'])
events = get(arProps,ib, 'Events')

swc =
    '/pkg/swc/ASWC'

ib =
    'ASWC/IB'

runnables =

```

```

1x4 cell array
{'ASWC/IB/Runnable_Init'} {'ASWC/IB/Runnable1'}
{'ASWC/IB/Runnable2'} {'ASWC/IB/Runnable3'}

events =
1x5 cell array
{'ASWC/IB/Event_t_ltic_B'} {'ASWC/IB/Event_t_l0tic'} {'ASWC/IB/myTimingEvent'}
{'ASWC/IB/myDREvent'} {'ASWC/IB/myDREvent'}

```

Gather Information for AUTOSAR Custom Scheduler Script

This example shows:

- 1 Loops through events and runnables in an open model.
- 2 For each event or runnable, extracts information to use with a custom scheduler.

`hModel` specifies the name of an open AUTOSAR model, in this case `'autosar_swc_expfcns'`.

The following code is an example of how to extract timing information for runnables to prepare for hooking up a custom scheduler.

First use AUTOSAR property functions to retrieve property values for the software component.

```

hModel = 'autosar_swc_expfcns';
open_system(hModel);
arProps = autosar.api.getAUTOSARProperties(hModel);
swc = get(arProps, 'XmlOptions', 'ComponentQualifiedName');

```

As well as to determine AUTOSAR internal behavior, runnables and events.

```

ib = get(arProps, swc, 'Behavior');
events = get(arProps, ib, 'Events');
runnables = get(arProps, ib, 'Runnables');

```

Loop through these events and runnables using for loops.

```

for ii=1:length(events)
    event = events{ii};
    category = get(arProps, event, 'Category');

    switch category
        case 'TimingEvent'
            runnablePath = get(arProps, event, 'StartOnEvent');
            period = get(arProps, event, 'Period');
            eventName = get(arProps, event, 'Name');
            runnableName = get(arProps, runnablePath, 'Name');
            fprintf('Event %s triggers runnable %s with period %g\n', eventName, runnableName, period);
        otherwise
            % Not interested in other events
    end
end

```

```

Event Event_t_ltic_A triggers runnable Runnable1 with period 1
Event Event_t_ltic_B triggers runnable Runnable2 with period 1
Event Event_t_l0tic triggers runnable Runnable3 with period 10

```

```

for ii=1:length(runnables)
    runnable = runnables{ii};
    runnableName = get(arProps, runnable, 'Name');

```

```

    runnableSymbol = get(arProps,runnable,'symbol');
    fprintf('Runnable %s has symbol %s\n',runnableName,runnableSymbol);
end

Runnable Runnable_Init has symbol Runnable_Init
Runnable Runnable1 has symbol Runnable1
Runnable Runnable2 has symbol Runnable2
Runnable Runnable3 has symbol Runnable3

```

Running the example code on the example model `autosar_swc_expfncs` generates the following output:

```

Event Event_t_ltic_A triggers runnable Runnable1 with period 1
Event Event_t_ltic_B triggers runnable Runnable2 with period 1
Event Event_t_l0tic triggers runnable Runnable3 with period 10
Runnable Runnable_Init has symbol Runnable_Init
Runnable Runnable1 has symbol Runnable1
Runnable Runnable2 has symbol Runnable2
Runnable Runnable3 has symbol Runnable3

```

Running the example code on the example model `mMultitasking_4rates` generates the following output:

```

Event Event_Runnable_Step triggers runnable Runnable_Step with period 1
Event Event_Runnable_Step1 triggers runnable Runnable_Step1 with period 2
Event Event_Runnable_Step2 triggers runnable Runnable_Step2 with period 4
Event Event_Runnable_Step3 triggers runnable Runnable_Step3 with period 8
Runnable Runnable_Init has symbol Runnable_Init
Runnable Runnable_Step has symbol Runnable_Step
Runnable Runnable_Step1 has symbol Runnable_Step1
Runnable Runnable_Step2 has symbol Runnable_Step2
Runnable Runnable_Step3 has symbol Runnable_Step3

```

Configure AUTOSAR Inter-Runnable Variables

In an AUTOSAR software component with multiple runnables, inter-runnable variables (IRVs) are used to communicate data between runnables. In Simulink, you model IRVs using data transfer lines that connect subsystems. In an application with multiple rates, the data transfer lines might include Rate Transition blocks to handle transitions between differing rates.

These examples show how to use the AUTOSAR property and map functions to configure AUTOSAR IRVs without or with rate transitions.

- “Configure Inter-Runnable Variable for Data Transfer Line” on page 4-309
- “Configure Inter-Runnable Variable for Data Transfer with Rate Transition.” on page 4-310

Configure Inter-Runnable Variable for Data Transfer Line

This example:

- 1 Opens a model.
- 2 Adds an AUTOSAR inter-runnable variable (IRV) to the model.
- 3 Maps a Simulink data transfer to the IRV.

```

% Open model
hModel = 'autosar_swc_expfncs';
openExample(hModel);

```

```

% Use AUTOSAR property functions
arProps = autosar.api.getAUTOSARProperties(hModel);

% Get AUTOSAR internal behavior and add IRV myIrv with SwCalibrationAccess ReadWrite
irvName = 'myIrv';
swCalibValue = 'ReadWrite';
swc = get(arProps,'XmlOptions','ComponentQualifiedNames');
ib = get(arProps,swc,'Behavior');
irvs = get(arProps,ib,'IRV');
add(arProps,ib,'IRV',irvName,'SwCalibrationAccess',swCalibValue);
irvs = get(arProps,ib,'IRV');

% Use AUTOSAR map functions
slMap=autosar.api.getSimulinkMapping(hModel);

% Retrieve list of DataTransfers
listOfDataTransfers = find(slMap, "DataTransfers")

listOfDataTransfers =

    1x4 string array

    "irv3"    "irv1"    "irv2"    "irv4"

% Map Simulink signal irv1 to AUTOSAR IRV myIrv with access mode Explicit
irvAccess = 'Explicit';
[arIrvName,arDataAccessMode] = getDataTransfer(slMap,'irv1');
mapDataTransfer(slMap,'irv1',irvName,irvAccess);
[arIrvName,arDataAccessMode] = getDataTransfer(slMap,'irv1')

% To pass validation, remove redundant IRV in AUTOSAR configuration
irvs = get(arProps,ib,'IRV');
delete(arProps,[ib,'/IRV1'])
irvs = get(arProps,ib,'IRV')

swc =
    '/pkg/swc/ASWC'

ib =
    'ASWC/IB'

irvs =
    {'ASWC/IB/IRV1'}    {'ASWC/IB/IRV2'}
    {'ASWC/IB/IRV3'}    {'ASWC/IB/IRV4'}

arIrvName =
    'myIrv'

arDataAccessMode =
    'Explicit'

irvs =
    {'ASWC/IB/IRV2'}    {'ASWC/IB/IRV3'}
    {'ASWC/IB/IRV4'}    {'ASWC/IB/myIrv'}

```

Configure Inter-Runnable Variable for Data Transfer with Rate Transition.

This example:

- 1 Opens a model with multiple rates.
- 2 Adds an AUTOSAR inter-runnable variable (IRV) to the model.
- 3 Maps a Simulink® Rate Transition block to the IRV.

Open the model 'mMultitasking_4rates'.

```

hModel = 'mMultitasking_4rates';
open_system(hModel);

```

Use AUTOSAR property functions.


```
arProps = autosar.api.getAUTOSARProperties(hModel);
```

Retrieve AUTOSAR internal behavior and add IRV 'myIrv' with SwCalibrationAccess ReadWrite.

```
irvName = 'myIrv';
swCalibValue = 'ReadWrite';
swc = get(arProps, 'XmlOptions', 'ComponentQualifiedName')

swc =
'/mMultitasking_4rates_pkg/mMultitasking_4rates_sw/mMultitasking_4rates'

ib = get(arProps, swc, 'Behavior')

ib =
'mMultitasking_4rates/Behavior'

irvs = get(arProps, ib, 'IRV')

irvs = 1x3 cell
    {'mMultitasking_4rates/Behavior/IRV1'}    {'mMultitasking_4rates/Behavior/IRV2'}    {'mMulti

add(arProps, ib, 'IRV', irvName, 'SwCalibrationAccess', swCalibValue);
irvs = get(arProps, ib, 'IRV');
```

Use AUTOSAR map functions:

```
slMap=autosar.api.getSimulinkMapping(hModel);
```

Map the Simulink Real-Time block 'RateTransition2' to AUTOSAR IRV 'myIrv' with access mode Explicit.

```
irvAccess = 'Explicit';
[arIrvName, arDataAccessMode] = getDataTransfer(slMap, 'mMultitasking_4rates/RateTransition2');
mapDataTransfer(slMap, 'mMultitasking_4rates/RateTransition2', irvName, irvAccess);
[arIrvName, arDataAccessMode] = getDataTransfer(slMap, 'mMultitasking_4rates/RateTransition2')

arIrvName =
'myIrv'

arDataAccessMode =
'Explicit'
```

To pass validation, remove any redundant IRV in the AUTOSAR configuration.

```
irvs = get(arProps, ib, 'IRV');
delete(arProps, [ib, '/IRV3'])
irvs = get(arProps, ib, 'IRV')

irvs = 1x3 cell
    {'mMultitasking_4rates/Behavior/IRV1'}    {'mMultitasking_4rates/Behavior/IRV2'}    {'mMulti
```

Configure AUTOSAR Interfaces

AUTOSAR software components can use ports and interfaces to implement the following forms of communication:

- Sender-receiver (S-R)
- Client-server (C-S)
- Mode-switch (M-S)
- Nonvolatile (NV) data

These examples show how to use AUTOSAR property and map functions to configure AUTOSAR ports, interfaces, and related elements for S-R, C-S, and M-S communication. The techniques shown for configuring S-R ports and interfaces also broadly apply to NV communication.

- “Configure AUTOSAR Sender-Receiver Interfaces” on page 4-312
- “Configure AUTOSAR Client-Server Interfaces” on page 4-314
- “Configure AUTOSAR Mode-Switch Interfaces” on page 4-317

Configure AUTOSAR Sender-Receiver Interfaces

- “Configure and Map Sender-Receiver Interface” on page 4-312
- “Configure Sender-Receiver Data Element Properties” on page 4-313

Configure and Map Sender-Receiver Interface

This example:

- 1 Opens a model.
- 2 Adds an AUTOSAR sender-receiver interface to the model.
- 3 Adds data elements.
- 4 Creates sender and receiver ports.
- 5 Maps Simulink inports and outports to AUTOSAR receiver and sender ports.

See also “Configure AUTOSAR Runnables and Events” on page 4-304.

```
% Open model
hModel = 'autosar_swc_expfncns';
openExample(hModel);

% Use AUTOSAR property functions
arProps = autosar.api.getAUTOSARProperties(hModel);

% Add AUTOSAR S-R interface
ifName = 'mySrIf';
ifPkg = get(arProps,'XmlOptions','InterfacePackage')
addPackageableElement(arProps,'SenderReceiverInterface',ifPkg,ifName,'IsService',false);
ifPaths=find(arProps,[],'SenderReceiverInterface','PathType','FullyQualified')

% Add AUTOSAR S-R data elements with ReadWrite calibration access
de1 = 'myDE1';
de2 = 'myDE2';
swCalibValue= 'ReadWrite';
add(arProps, [ifPkg '/' ifName], 'DataElements', de1, 'SwCalibrationAccess', swCalibValue);
add(arProps, [ifPkg '/' ifName], 'DataElements', de2, 'SwCalibrationAccess', swCalibValue);

% Add AUTOSAR receiver and sender ports with S-R interface name
rPortName = 'myRPort';
pPortName = 'myPPort';
aswcPath = find(arProps,[],'AtomicComponent','PathType','FullyQualified');
add(arProps,aswcPath{1}, 'ReceiverPorts', rPortName, 'Interface', ifName);
add(arProps,aswcPath{1}, 'SenderPorts', pPortName, 'Interface', ifName);

% Use AUTOSAR map functions
slMap=autosar.api.getSimulinkMapping(hModel);

% Map Simulink inport RPort_DE2 to AUTOSAR receiver port myRPort and data element myDE2
rDataAccessMode = 'ImplicitReceive';
[arPortName,arDataElementName,arDataAccessMode]=getInport(slMap, 'RPort_DE2')
```

```

mapInport(s1Map, 'RPort_DE2', rPortName, de2, rDataAccessMode);
[arPortName, arDataElementName, arDataAccessMode]=getInport(s1Map, 'RPort_DE2')

% Map Simulink outpost PPort_DE1 to AUTOSAR sender port myPPort and data element myDE1
sDataAccessMode = 'ImplicitSend';
[arPortName, arDataElementName, arDataAccessMode]=getOutport(s1Map, 'PPort_DE1')
mapOutport(s1Map, 'PPort_DE1', pPortName, de1, sDataAccessMode);
[arPortName, arDataElementName, arDataAccessMode]=getOutport(s1Map, 'PPort_DE1')

ifPkg =
    '/pkg/if'

ifPaths =
    {'/pkg/if/Interface1'}    {'/pkg/if/Interface2'}    {'/pkg/if/mySrIf'}

arPortName =
    'RPort'
arDataElementName =
    'DE2'
arDataAccessMode =
    'ImplicitReceive'

arPortName =
    'myRPort'
arDataElementName =
    'myDE2'
arDataAccessMode =
    'ImplicitReceive'

arPortName =
    'PPort'
arDataElementName =
    'DE1'
arDataAccessMode =
    'ImplicitSend'

arPortName =
    'myPPort'
arDataElementName =
    'myDE1'
arDataAccessMode =
    'ImplicitSend'

```

Configure Sender-Receiver Data Element Properties

This example loops through AUTOSAR sender-receiver (S-R) interfaces and data elements to configure calibration properties for S-R data elements.

```

% Open model
hModel = 'autosar_swc_expfncns';
openExample(hModel);

% Use AUTOSAR property functions
arProps = autosar.api.getAUTOSARProperties(hModel);

% Configure SwCalibrationAccess for AUTOSAR data elements in S-R interfaces
srIfs = find(arProps, [], 'SenderReceiverInterface', 'PathType', 'FullyQualified')

% Loop through S-R interfaces and get data elements
for i=1:length(srIfs)
    srIf = srIfs{i};
    dataElements = get(arProps, srIf, 'DataElements', 'PathType', 'FullyQualified')

% Loop through data elements for each S-R interface and set SwCalibrationAccess
    swCalibValue = 'ReadWrite';
    for ii=1:length(dataElements)
        dataElement = dataElements{ii};
        set(arProps, dataElement, 'SwCalibrationAccess', swCalibValue)
        get(arProps, dataElement, 'SwCalibrationAccess');
    end
end

srIfs =
    {'/pkg/if/Interface1'}    {'/pkg/if/Interface2'}

dataElements =
    {'/pkg/if/Interface1/DE1'}    {'/pkg/if/Interface1/DE2'}

dataElements =
    {'/pkg/if/Interface2/DE1'}    {'/pkg/if/Interface2/DE2'}
    {'/pkg/if/Interface2/DE3'}    {'/pkg/if/Interface2/DE4'}

```

Configure AUTOSAR Client-Server Interfaces

- “Configure Server Properties” on page 4-314
- “Configure Client Properties” on page 4-315

Configure Server Properties

Using example model 'mControllerWithInterface_server' add and configure an AUTOSAR client-server (C-S) interface.

Open example model 'mControllerWithInterface_server'.

```
hModel = 'mControllerWithInterface_server';
open_system(hModel);
```

Use AUTOSAR property functions.

```
arProps = autosar.api.getAUTOSARProperties(hModel);
```

Add AUTOSAR C-S interface.

```
ifName = 'myCsIf';
ifPkg = get(arProps, 'XmlOptions', 'InterfacePackage')

ifPkg =
'/ControllerWithInterface_ar_pkg/ControllerWithInterface_ar_if'

addPackageableElement(arProps, 'ClientServerInterface', ifPkg, ifName, 'IsService', false);
ifPaths=find(arProps, [], 'ClientServerInterface', 'PathType', 'FullyQualified');
```

Add AUTOSAR operation to C-S interface.

```
csOp = 'readData';
add(arProps, [ifPkg '/' ifName], 'Operations', csOp);
```

Add AUTOSAR arguments to C-S operation with Direction and SwCalibrationAccess properties.

```
args = {'Op', 'In'; 'Data', 'Out'; 'ERR', 'Out'; 'NegCode', 'Out'}

args = 4x2 cell
    {'Op'   }    {'In'  }
    {'Data' }    {'Out' }
    {'ERR'  }    {'Out' }
    {'NegCode'}    {'Out' }

swCalibValue = 'ReadOnly';
for i=1:length(args)
    add(arProps, [ifPkg '/' ifName '/' csOp], 'Arguments', args{i,1}, 'Direction', args{i,2}, ...
        'SwCalibrationAccess', swCalibValue);
end
get(arProps, [ifPkg '/' ifName '/' csOp], 'Arguments')

ans = 1x4 cell
    {'myCsIf/readData/Op' }    {'myCsIf/readData/Data' }    {'myCsIf/readData/ERR' }    {'myCsIf/readData/NegCode' }
```

Add AUTOSAR server port with C-S interface name.

```
sPortName = 'mySPort';
aswcPath = find(arProps,[],'AtomicComponent','PathType','FullyQualified');
add(arProps,aswcPath{1},'ServerPorts',sPortName,'Interface',ifName);
```

Add AUTOSAR server runnable with symbol name that matches Simulink function name.

```
serverRunnable = 'Runnable_myReadData';
serverRunnableSymbol = 'readData';
swc = get(arProps,'XmlOptions','ComponentQualifiedName')
```

```
swc =
'/pkg/swc/SWC_Controller'
```

```
ib = get(arProps,swc,'Behavior')
```

```
ib =
'SWC_Controller/ControllerWithInterface_ar'
```

```
runnables = get(arProps,ib,'Runnables');
```

To avoid symbol conflict, remove existing runnable with symbol name readData

```
delete(arProps,'SWC_Controller/ControllerWithInterface_ar/Runnable_readData')
add(arProps,ib,'Runnables',serverRunnable,'symbol',serverRunnableSymbol);
runnables = get(arProps,ib,'Runnables');
```

Add AUTOSAR operation invoked event

```
oiEventName = 'Event_myReadData';
add(arProps,ib,'Events',oiEventName,'Category','OperationInvokedEvent',...
    'Trigger','mySPort.readData','StartOnEvent',[ib '/' serverRunnable]);
```

Use AUTOSAR map functions.

```
slMap=autosar.api.getSimulinkMapping(hModel);
```

Map Simulink function readData to AUTOSAR runnable Runnable_myReadData.

```
mapFunction(slMap,'readData',serverRunnable);
```

The function name value 'readData' is obsolete and will be removed in a future release. For valid function name values, use `autosar.api.getSimulinkMapping(modelName).find("Functions")`.
The function name value 'readData' is obsolete and will be removed in a future release. For valid function name values, use `autosar.api.getSimulinkMapping(modelName).find("Functions")`.

```
arRunnableName=getFunction(slMap,'readData')
```

The function name value 'readData' is obsolete and will be removed in a future release. For valid function name values, use `autosar.api.getSimulinkMapping(modelName).find("Functions")`.

```
arRunnableName =
'Runnable_myReadData'
```

Configure Client Properties

This example:

- 1 Opens a model.
- 2 Adds an AUTOSAR client-server (C-S) interface to the model.
- 3 Adds an operation.
- 4 Creates a client port.
- 5 Maps a Simulink® function caller to the AUTOSAR client port and operation.

Open the model 'mControllerWithInterface_client'.

```
hModel = 'mControllerWithInterface_client';
open_system(hModel);
```

Retrieve the AUTOSAR property functions.

```
arProps = autosar.api.getAUTOSARProperties(hModel);
```

Add AUTOSAR C-S interface.

```
ifName = 'myCsIf';
ifPkg = get(arProps, 'XmlOptions', 'InterfacePackage');
addPackageableElement(arProps, 'ClientServerInterface', ifPkg, ifName, 'IsService', false);
ifPaths=find(arProps, [], 'ClientServerInterface', 'PathType', 'FullyQualified')
```

```
ifPaths = 1x2 cell
    {'/pkg/if/csInterface'}    {'/pkg/if/myCsIf'}
```

Add AUTOSAR operation to C-S interface.

```
csOp = 'readData';
add(arProps, [ifPkg '/' ifName], 'Operations', csOp);
```

Add AUTOSAR arguments to C-S operation with Direction and SwCalibrationAccess properties.

```
args = {'Op', 'In'; 'Data', 'Out'; 'ERR', 'Out'; 'NegCode', 'Out'}
```

```
args = 4x2 cell
    {'Op'   }    {'In'  }
    {'Data' }    {'Out' }
    {'ERR'  }    {'Out' }
    {'NegCode'}    {'Out' }
```

```
swCalibValue = 'ReadOnly';
```

```
for i=1:length(args)
    add(arProps, [ifPkg '/' ifName '/' csOp], 'Arguments', args{i,1}, 'Direction', args{i,2}, ...
        'SwCalibrationAccess', swCalibValue);
end
```

```
get(arProps, [ifPkg '/' ifName '/' csOp], 'Arguments')
```

```
ans = 1x4 cell
    {'myCsIf/readData/Op'}    {'myCsIf/readData/Data'}    {'myCsIf/readData/ERR'}    {'myCsIf/readData/NegCode'}
```

Add AUTOSAR client port with C-S interface name

```
cPortName = 'myCPort';
aswcPath = find(arProps, [], 'AtomicComponent', 'PathType', 'FullyQualified');
add(arProps, aswcPath{1}, 'ClientPorts', cPortName, 'Interface', ifName);
```

Use AUTOSAR map functions

```
slMap=autosar.api.getSimulinkMapping(hModel);
```

Map Simulink function caller readData to AUTOSAR client port and operation

```
[arPort,arOp] = getFunctionCaller(slMap,'readData');
mapFunctionCaller(slMap,'readData',cPortName,csOp);
[arPort,arOp] = getFunctionCaller(slMap,'readData')
```

```
arPort =
'myCPort'
```

```
arOp =
'readData'
```

Configure AUTOSAR Mode-Switch Interfaces

This example:

- 1 Opens a model.
- 2 Declares an AUTOSAR mode declaration group.
- 3 Adds a mode-switch (M-S) interface to the model.
- 4 Adds a mode receiver port.
- 5 Adds a ModeSwitchEvent to a runnable.
- 6 Maps a Simulink® inport to the AUTOSAR mode receiver port and mode group.

Open model 'mAutosarMsConfig'.

```
hModel = 'mAutosarMsConfig';
open_system(hModel);
```

Retrieve AUTOSAR property functions.

```
arProps = autosar.api.getAUTOSARProperties(hModel);
```

The file `mdgModes.m` declares AUTOSAR mode declaration group `mdgModes` for use with the M-S interface.

The enumerated mode values are:

- STARTUP(0)
- RUN(1)
- SHUTDOWN(2)

Separate code, below, defines mode declaration group information for XML export.

Apply data type `mdgModes` to Simulink inport `MRPort`

```
set_param([hModel, '/MRPort'], 'OutDataTypeStr', 'Enum: mdgModes')
get_param([hModel, '/MRPort'], 'OutDataTypeStr');
```

Apply data type `mdgModes` and value `STARTUP` to `Runnable1_subsystem/Enumerated Constant`.

```
set_param([hModel, '/Runnable1_subsystem/Enumerated Constant'], 'OutDataTypeStr', 'Enum: mdgModes')
set_param([hModel, '/Runnable1_subsystem/Enumerated Constant'], 'Value', 'mdgModes.STARTUP')
```

Add AUTOSAR M-S interface and set its ModeGroup to mdgModes.

```
ifName = 'myMsIf';
modeGroup = 'mdgModes';
ifPkg = get(arProps, 'XmlOptions', 'InterfacePackage');
addPackageableElement(arProps, 'ModeSwitchInterface', ifPkg, ifName, 'IsService', true);
add(arProps, [ifPkg '/' ifName], 'ModeGroup', modeGroup)
ifPaths=find(arProps, [], 'ModeSwitchInterface', 'PathType', 'FullyQualified')
```

```
ifPaths = 1x1 cell array
    {'/pkg/if/myMsIf'}
```

Add AUTOSAR mode-receiver port with M-S interface name.

```
mrPortName = 'myMRPort';
aswcPath = find(arProps, [], 'AtomicComponent', 'PathType', 'FullyQualified');
add(arProps, aswcPath{1}, 'ModeReceiverPorts', mrPortName, 'Interface', ifName);
```

Define AUTOSAR ModeSwitchEvent for runnable.

```
msRunnable = 'Runnable1';
msEventName = 'myMSEvent';
swc = get(arProps, 'XmlOptions', 'ComponentQualified_name');
ib = get(arProps, swc, 'Behavior');
runnables = get(arProps, ib, 'Runnables')
```

```
runnables = 1x4 cell
    {'ASWC/Behavior/Runnable_Init'}    {'ASWC/Behavior/Runnable1'}    {'ASWC/Behavior/Runnable2'}
```

```
add(arProps, ib, 'Events', msEventName, 'Category', 'ModeSwitchEvent', ...
    'Activation', 'OnTransition', ...
    'StartOnEvent', [ib '/' msRunnable]);
```

Separate the following code. The code below sets ModeSwitchEvent port and trigger values.

To pass validation, remove the redundant timing event in AUTOSAR configuration.

```
events = get(arProps, ib, 'Events');
delete(arProps, [ib, '/Event_t_ltic_A'])
events = get(arProps, ib, 'Events')
```

```
events = 1x3 cell
    {'ASWC/Behavior/Event_t_ltic_B'}    {'ASWC/Behavior/Event_t_l0tic'}    {'ASWC/Behavior/myMSEvent'}
```

Export the mode declaration group information to the AUTOSAR data type package in XML.

```
mdgPkg = get(arProps, 'XmlOptions', 'DataTypePackage');
mdgPath = [mdgPkg '/' modeGroup]
```

```
mdgPath =
    '/pkg/dt/mdgModes'
```

```
initMode = [mdgPath '/STARTUP']
```



```

initMode =
'/pkg/dt/mdgModes/STARTUP'

addPackageableElement(arProps, 'ModeDeclarationGroup', mdgPkg, modeGroup, 'OnTransitionValue', 100)

```

Add modes to ModeDeclarationGroup and set InitialMode.

```

add(arProps, mdgPath, 'Mode', 'STARTUP', 'Value', 0)
add(arProps, mdgPath, 'Mode', 'RUN', 'Value', 1)
add(arProps, mdgPath, 'Mode', 'SHUTDOWN', 'Value', 2)
set(arProps, mdgPath, 'InitialMode', initMode)

```

Set ModeGroup for the M-S interface.

```

set(arProps, [ifPkg '/' ifName '/' modeGroup], 'ModeGroup', mdgPath)

```

Set the port and trigger for AUTOSAR property ModeSwitchEvent.

```

expTrigger = {[mrPortName '.STARTUP'], [mrPortName '.SHUTDOWN']}

expTrigger = 1x2 cell
             {'myMRPort.STARTUP'}     {'myMRPort.SHUTDOWN'}

```

```

set(arProps, [ib '/' msEventName], 'Trigger', expTrigger)

```

Retrieve the AUTOSAR map functions.

```

slMap=autosar.api.getSimulinkMapping(hModel);

```

Map the Simulink® inport MRPort to the AUTOSAR mode receiver port myMRPort and the mode group mdgModes.

```

msDataAccessMode = 'ModeReceive';
[arPortName, arDataElementName, arDataAccessMode]=getInport(slMap, 'MRPort');
mapInport(slMap, 'MRPort', mrPortName, modeGroup, msDataAccessMode);
[arPortName, arDataElementName, arDataAccessMode]=getInport(slMap, 'MRPort')

```

```

arPortName =
'myMRPort'

```

```

arDataElementName =
'mdgModes'

```

```

arDataAccessMode =
'ModeReceive'

```

To pass validation, set inport Runnable1 sample time to -1 (inherited).

```

set_param([hModel, '/Runnable1'], 'SampleTime', '-1')

```

Configure AUTOSAR XML Export

- “Configure XML Export Options” on page 4-320
- “Configure AUTOSAR Package Paths” on page 4-320

Configure XML Export Options

This example configures AUTOSAR XML export parameter **Exported XML file packaging** (`ArxmlFilePackaging`).

To configure AUTOSAR package paths, see “Configure AUTOSAR Package Paths” on page 4-320.

```
% Open model
hModel = 'autosar_sw_counter';
openExample(hModel);

% Use AUTOSAR property functions
arProps = autosar.api.getAUTOSARProperties(hModel);

% Set exported AUTOSAR XML file packaging to Single file
get(arProps, 'XmlOptions', 'ArxmlFilePackaging')
set(arProps, 'XmlOptions', 'ArxmlFilePackaging', 'SingleFile');
get(arProps, 'XmlOptions', 'ArxmlFilePackaging')

ans =
    'Modular'

ans =
    'SingleFile'
```

Configure AUTOSAR Package Paths

This example configures an AUTOSAR package path for XML export. For other AUTOSAR package path property names, see “Configure AUTOSAR Packages and Paths” on page 4-85.

To configure other XML export options, see “Configure XML Export Options” on page 4-320.

```
% Open model
hModel = 'autosar_sw_counter';
openExample(hModel);

% Use AUTOSAR property functions
arProps = autosar.api.getAUTOSARProperties(hModel);

% Specify AUTOSAR application data type package path for XML export
get(arProps, 'XmlOptions', 'ApplicationDataTypePackage')
set(arProps, 'XmlOptions', 'ApplicationDataTypePackage', '/Company/Powertrain/DataTypes/ADTs');
get(arProps, 'XmlOptions', 'ApplicationDataTypePackage')

ans =
    '/Company/Powertrain/DataTypes/ApplDataTypes'

ans =
    '/Company/Powertrain/DataTypes/ADTs'
```

See Also

get | set

Related Examples

- “Configure and Map AUTOSAR Component Programmatically” on page 4-293

More About

- “AUTOSAR Component Configuration” on page 4-3

Limitations and Tips

The following limitation applies to AUTOSAR component development.

AUTOSAR Client Block in Referenced Model

The software does not support the use of an AUTOSAR client block, such as Function Caller or Invoke AUTOSAR Server Operation, in a referenced model.

AUTOSAR Code Generation

- “Generate AUTOSAR C Code and XML Descriptions” on page 5-2
- “Configure AUTOSAR Code Generation” on page 5-7
- “Code Generation with AUTOSAR Code Replacement Library” on page 5-12
- “Verify AUTOSAR C Code with SIL and PIL” on page 5-29
- “Integrate Generated Code for Multi-Instance Software Components” on page 5-31
- “Import and Simulate AUTOSAR Code from Previous Releases” on page 5-32
- “Limitations and Tips” on page 5-33

Generate AUTOSAR C Code and XML Descriptions

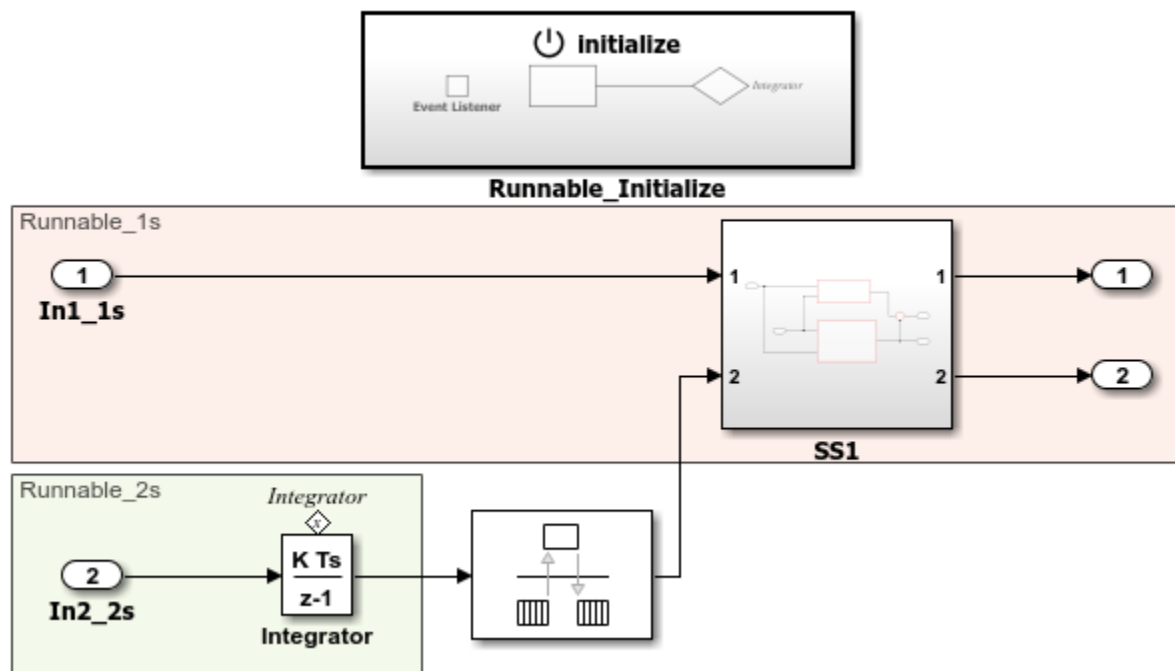
Generate AUTOSAR-compliant C code and export AUTOSAR XML (ARXML) descriptions from AUTOSAR component model.

If you have Simulink Coder and Embedded Coder software, you can build AUTOSAR component models. Building a classic component model generates algorithmic C code and exports ARXML descriptions that comply with AUTOSAR Classic Platform specifications. Use the generated C code and ARXML descriptions for testing in Simulink or integration into an AUTOSAR run-time environment.

Prepare AUTOSAR Component Model for Code Generation

Open a component model from which you want to generate AUTOSAR C code and ARXML descriptions. This example uses AUTOSAR example model `autosar_sw_c`.

```
open_system('autosar_sw_c');
```



Optionally, to refine model configuration settings for code generation, you can use the Embedded Coder Quick Start (recommended). This example uses the Embedded Coder Quick Start. From the **Apps** tab, open the AUTOSAR Component Designer app. On the **AUTOSAR** tab, click **Quick Start**.

Work through the quick-start procedure. In the Output window, select output option **C code compliant with AUTOSAR**.

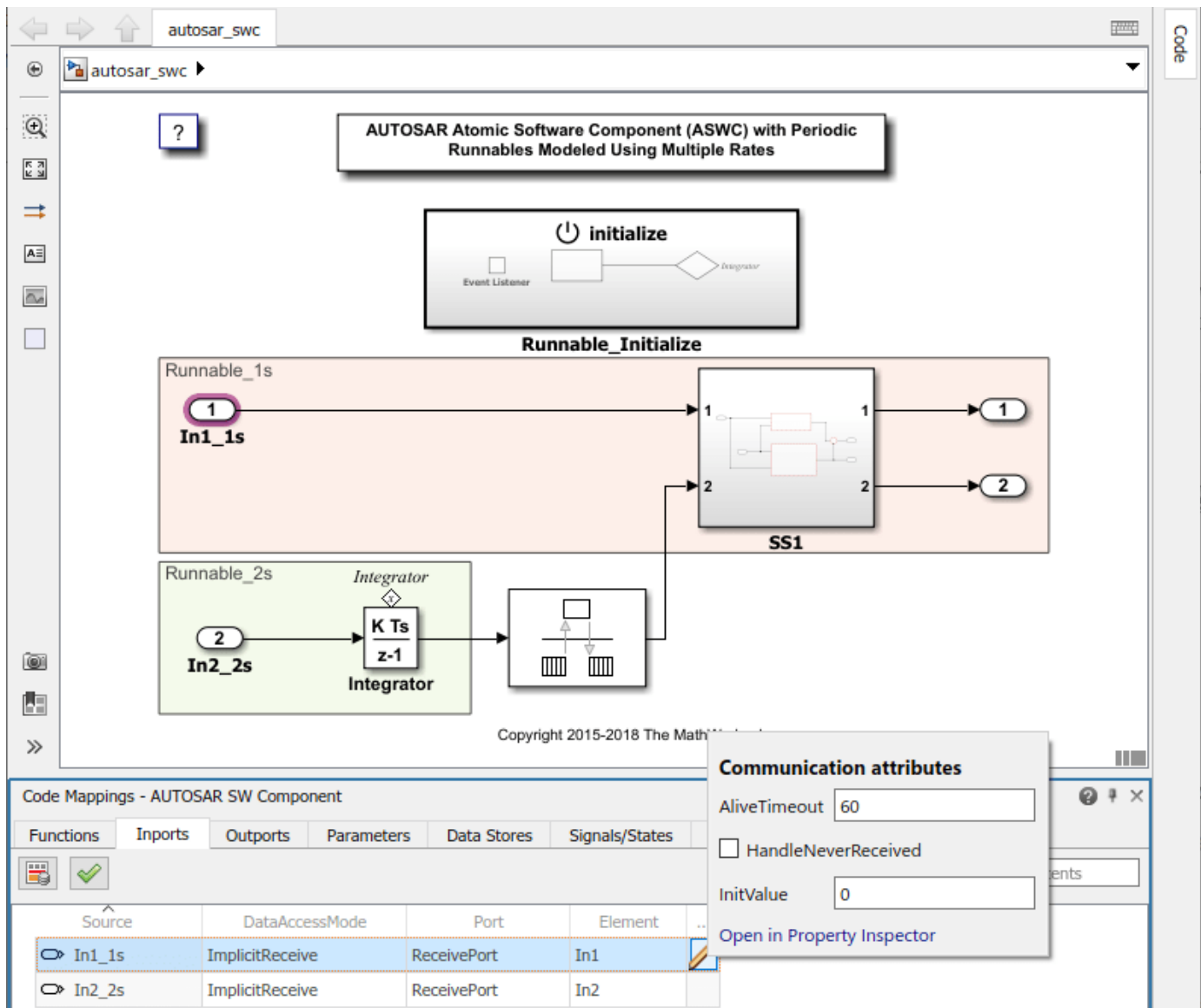
Select the output for your generated code.

- C code
- C code compliant with AUTOSAR
- C++ code
- C++ code compliant with AUTOSAR Adaptive Platform

The quick-start software takes the following steps to configure an AUTOSAR software component model:

- 1** Configures code generation settings for the model. If the AUTOSAR target is not selected, the software sets model configuration parameter **System target file** to `autosar.tlc` and **Generate XML for schema version** to a default value.
- 2** If no AUTOSAR mapping exists, the software creates a mapped AUTOSAR software component for the model.
- 3** Performs a model build.

In the last window, when you click **Finish**, your model opens in the AUTOSAR code perspective.



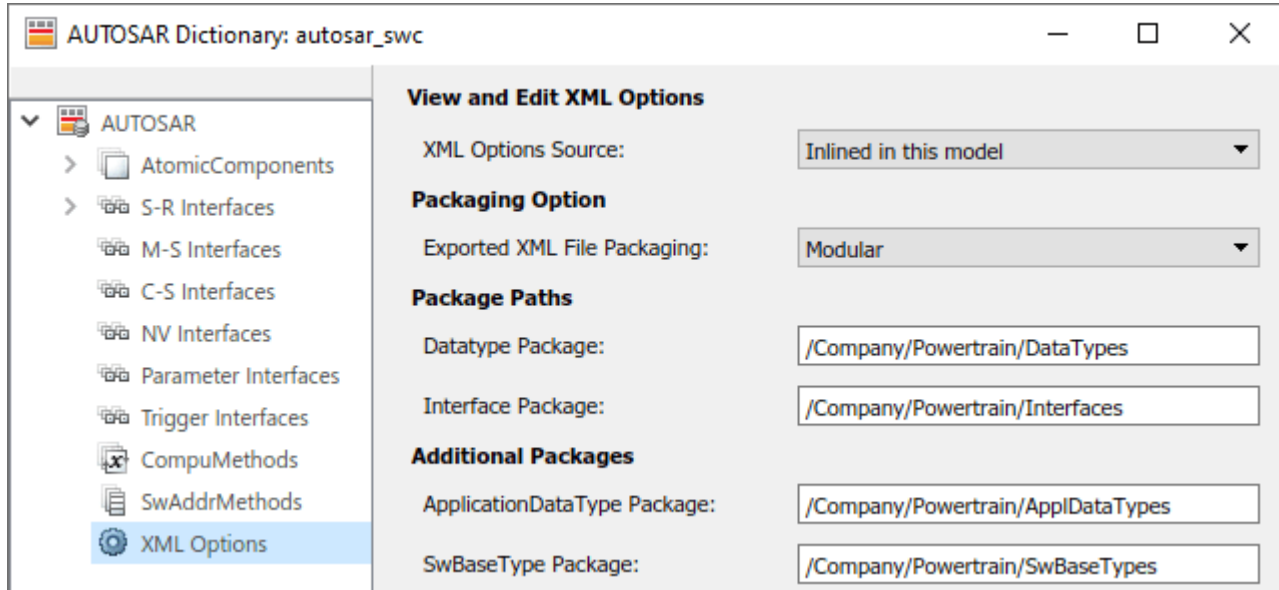
Inspect XML Options in AUTOSAR Dictionary

Before generating code, open the AUTOSAR Dictionary and examine the settings of AUTOSAR XML export parameters. On the **AUTOSAR** tab, select **Code Interface > AUTOSAR Dictionary**. In the AUTOSAR Dictionary, select **XML Options**.

The XML options view in the AUTOSAR Dictionary displays XML export parameters and their values. You can configure:

- XML file packaging for AUTOSAR elements created in Simulink
- AUTOSAR package paths
- Aspects of exported AUTOSAR XML content

This example sets **Exported XML file packaging** to Modular, so that ARXML is exported into modular files, including *modelName_component.arxml*, *modelName_datatype.arxml*, and *modelName_interface.arxml*.



Generate AUTOSAR C Code and XML Descriptions

To generate AUTOSAR C code and XML software descriptions that comply with Classic Platform specifications, build the model. In the model window, press **Ctrl+B**. The build process generates C code and ARXML descriptions to the model build folder, *autosar_sw_autosar_rtw*. Data types and related elements that are not used in the model are removed from the exported ARXML files. When the build completes, a code generation report opens.

The screenshot shows a window titled "Code Generation Report" with a search bar and a "Match Case" button. The main content area is divided into two panes. The left pane, titled "Content", lists various reports and code files. The right pane, titled "autosar_swk", displays the source code for "autosar_swk.c".

Content Pane:

- Summary
- Subsystem Report
- Code Interface Report
- Traceability Report
- Static Code Metrics Report
- Code Replacements Report
- Coder Assumptions

Code Pane:

```

autosar_swk.c
├── autosar_swk.h
├── autosar_swk_private.h
├── autosar_swk_types.h
├── Shared files
│   └── rtwtypes.h
├── Interface files
│   ├── autosar_swk_component.arxml
│   ├── autosar_swk_datatype.arxml
│   ├── autosar_swk_implementation.arxml
│   ├── autosar_swk_interface.arxml
│   └── autosar_swk_timing.arxml
├── RTE files
│   ├── Compiler.h
│   ├── Platform_Types.h
│   ├── Rte_ASWC.h
│   ├── Rte_Type.h
│   └── Std_Types.h

```

```

25  /* Model step function for TID0 */
26  void Runnable_1s(void)          /* Sample time: [1.0s, 0.0s] */
27  {
28      float64 rtb_RateTransition;
29      float64 rtb_Sum_n;
30
31      /* RateTransition: '<Root>/RateTransition' */
32      rtb_RateTransition = Rte_IrvIRead_Runnable_1s_IRV1();
33
34      /* Outputs for Atomic SubSystem: '<S2>/SS2' */
35      /* Sum: '<S4>/Sum' incorporates:
36       * Gain: '<S4>/Gain'
37       * Inport: '<Root>/In1_1s'
38       */
39      rtb_Sum_n = 2.0 * rtb_RateTransition + Rte_IRead_Runnable_1s_ReceivePort_In1();
40
41      /* End of Outputs for SubSystem: '<S2>/SS2' */
42
43      /* Outputs for Atomic SubSystem: '<S2>/SS1' */
44      /* Output: '<Root>/Out1' incorporates:
45       * Gain: '<S3>/Gain1'
46       * Gain: '<S3>/Gain2'
47       * Inport: '<Root>/In1_1s'
48       * Sum: '<S2>/Sum'
49       * Sum: '<S3>/Sum'
50       */
51      Rte_IWrite_Runnable_1s_SenderPort_Out1(5.0 *
52          (Rte_IRead_Runnable_1s_ReceivePort_In1() + 3.0 * rtb_RateTransition) +
53          rtb_Sum_n);
54
55      /* End of Outputs for SubSystem: '<S2>/SS1' */
56
57      /* Output: '<Root>/Out2' */
58      Rte_IWrite_Runnable_1s_SenderPort_Out2(rtb_Sum_n);
59  }

```

Related Links

- “Code Generation”
- “AUTOSAR Component Configuration” on page 4-3
- “AUTOSAR Blockset”

Configure AUTOSAR Code Generation

To generate AUTOSAR-compliant C code and ARXML component descriptions from a model configured for the AUTOSAR Classic Platform:

- 1 In the Configuration Parameters dialog box, on the **Code Generation > AUTOSAR Code Generation Options** pane, configure AUTOSAR code generation parameters.
- 2 Configure AUTOSAR XML export options by using the AUTOSAR Dictionary or AUTOSAR property functions.
- 3 Build the model.

Select AUTOSAR Classic Schema

For import and export of ARXML files and generation of AUTOSAR-compliant C code, the software supports the following AUTOSAR Classic Platform schema versions.

Schema Version Value	Schema Revisions Supported for Import	Export Schema Revision
R21-11 (default)	R21-11	R21-11
R20-11	R20-11	R20-11
R19-11	R19-11	R19-11
4.4	4.4.0	4.4.0
4.3	4.3.0, 4.3.1	4.3.1
4.2	4.2.1, 4.2.2	4.2.2
4.1	4.1.1, 4.1.2, 4.1.3	4.1.3
4.0	4.0.1, 4.0.2, 4.0.3	4.0.3

Selecting the AUTOSAR system target file for your model for the first time sets the schema version parameter to the default value, R21-11.

If you import ARXML files into Simulink, the ARXML importer detects the schema version and sets the schema version parameter in the model. For example, if you import ARXML files based on schema 4.3 revision 4.3.0 or 4.3.1, the importer sets the schema version parameter to 4.3.

When you build an AUTOSAR model, the code generator exports ARXML descriptions and generates C code that comply with the current schema version. For example, if **Generate XML file for schema version** equals 4.3, export uses the export schema revision listed above for schema 4.3, that is, revision 4.3.1.

Before exporting your AUTOSAR software component, check the selected schema version. If you need to change the selected schema version, use the model configuration parameter **Generate XML file for schema version**.

Note Set the AUTOSAR model configuration parameters to the same values for top and referenced models. This guideline applies to **Generate XML file for schema version**, **Maximum SHORT-NAME length**, **Use AUTOSAR compiler abstraction macros**, and **Support root-level matrix I/O using one-dimensional arrays**.

Specify Maximum SHORT-NAME Length

The AUTOSAR standard specifies that the maximum length of SHORT-NAME XML elements is 128 characters.

To specify a maximum length for SHORT-NAME elements exported by the code generator, set the model configuration parameter **Maximum SHORT-NAME length** to an integer value between 32 and 128, inclusive. The default is 128 characters.

Configure AUTOSAR Compiler Abstraction Macros

Compilers for 16-bit platforms (for example, Cosmic and Metrowerks for S12X or Tasking for ST10) use special keywords to deal with the limited 16-bit addressing range. The location of data and code beyond the 64k border is selected explicitly by special keywords. However, if such keywords are used directly within the source code, then software must be ported separately for each microcontroller family. That is, the software is not platform-independent.

AUTOSAR specifies C macros to abstract compiler directives (near/far memory calls) in a platform-independent manner. These compiler directives, derived from the 16-bit platforms, enable better code efficiencies for 16-bit micro-controllers without separate porting of source code for each compiler. This approach allows your system integrator, rather than your software component implementer, to choose the location of data and code for each software component.

For more information on AUTOSAR compiler abstraction, see www.autosar.org.

To enable AUTOSAR compiler macro generation, select the model configuration parameter **Use AUTOSAR compiler abstraction macros**.

When you build the model, the software applies compiler abstraction macros to global data and function definitions in the generated code.

For data, the macros are in the following form:

- `CONST(consttype, memclass) varname;`
- `VAR(type, memclass) varname;`

where

- `consttype` and `type` are data types
- `memclass` is a macro string `SWC_VAR` (`SWC` is the software component identifier)
- `varname` is the variable identifier

For functions (model and subsystem), the macros are in the following form:

- `FUNC(type, memclass) funcname(void)`

where

- `type` is the data type of the return argument
- `memclass` is a macro string. This string can be either `SWC_CODE` for runnables (external functions), or `SWC_CODE_LOCAL` for internal functions (`SWC` is the software component identifier).

If you do *not* select **Use AUTOSAR compiler abstraction macros**, the code generator produces the following code:

```
/* Block signals (auto storage) */
BlockIO rtB;

/* Block states (auto storage) */
D_Work rtDWork;

/* Model step function */
void Runnable_Step(void)
```

However, if you select **Use AUTOSAR compiler abstraction macros**, the code generator produces macros in the code:

```
/* Block signals (auto storage) */
VAR(BlockIO, SWC1_VAR) rtB;

/* Block states (auto storage) */
VAR(D_Work, SWC1_VAR) rtDWork;

/* Model step function */
FUNC(void, SWC1_CODE) Runnable_Step(void)
```

Root-Level Matrix I/O

For an AUTOSAR component model with multidimensional arrays, if you set the model configuration parameter **Array layout** to `Row-major`, you can preserve dimensions of multidimensional arrays in the generated C code. Preserving array dimensions in the generated code can enhance code integration.

If your application design requires `Column-major` array layout, you can configure ARXML export to support root-level matrix I/O. The software can export ARXML descriptions that implement matrices as one-dimensional arrays.

By default, for `Column-major` array layout, the software does not allow matrix I/O at the root level. Building the model generates an error. To enable root-level matrix I/O, select the model configuration parameter **Support root-level matrix I/O using one-dimensional arrays**.

When **Array layout** is set to `Row-major`, **Support root-level matrix I/O using one-dimensional arrays** has no effect.

Inspect AUTOSAR XML Options

Examine the XML options that you configured by using the AUTOSAR Dictionary. If you have not yet configured them, see “Configure AUTOSAR XML Options” on page 4-43.

Generate AUTOSAR C and XML Files

After configuring AUTOSAR code generation and XML options, generate code. To generate C code and export XML descriptions, build the component model.

The build process generates AUTOSAR-compliant C code and AUTOSAR XML descriptions to the model build folder. The exported XML files include:

- One or more *modelName*.arxml* files, based on whether you set **Exported XML file packaging** to `Single file` or `Modular`.

- If you imported ARXML files into Simulink, updated versions of the same files.

This table lists *modelname*.arxml* files that are generated based on the value of the **Exported XML file packaging** option configured in the AUTOSAR Dictionary.

Exported XML File Packaging Value	Exported File Name	By Default Contains...
Single file	<i>modelname.arxml</i>	AUTOSAR elements for software components, data types, implementation, interfaces, and timing.
Modular	<i>modelname_component.arxml</i>	<p>Software components, including:</p> <ul style="list-style-type: none"> • Ports • Events • Runnables • Inter-runnable variables (IRVs) • Included data type sets • Component-scoped parameters and variables <p>This is the main ARXML file exported for the Simulink model. In addition to software components, the component file contains packageable elements that the exporter does not move to data type, implementation, interface, or timing files based on AUTOSAR element category.</p>
	<i>modelname_datatype.arxml</i>	<p>Data types and related elements, including:</p> <ul style="list-style-type: none"> • Application data types • Software base types • Data type mapping sets • Constant specifications • Physical data constraints • System constants • Software address methods • Mode declaration groups • Computation methods • Units and unit groups • Software record layouts • Internal data constraints
	<i>modelname_implementation.arxml</i>	Software component implementation, including code descriptors.
	<i>modelname_interface.arxml</i>	Interfaces, including S-R, C-S, M-S, NV, parameter, and trigger interfaces. The interfaces include type-specific elements, such as S-R data elements, C-S operations, port-based parameters, or triggers.

Exported XML File Packaging Value	Exported File Name	By Default Contains...
	<i>modelname_timing.arxml</i>	Timing model, including runnable execution order constraints.

You can merge the AUTOSAR XML component descriptions back into an AUTOSAR authoring tool. The AUTOSAR component information is partitioned into separate files to facilitate merging. The partitioning attempts to minimize the number of merges that you must do. You do not need to merge the data type file into the authoring tool because data types are usually defined early in the design process. You must, however, merge the internal behavior file because this information is part of the model implementation.

To help support the round trip of AUTOSAR elements between an AUTOSAR authoring tool (AAT) and the Simulink model-based design environment, the code generator preserves AUTOSAR elements and their universal unique identifiers (UUIDs) across ARXML import and export. For more information, see “Round-Trip Preservation of AUTOSAR XML File Structure and Element Information” on page 3-37.

For an example of how to generate AUTOSAR-compliant C code and export AUTOSAR XML component descriptions from a Simulink model, see “Generate AUTOSAR C Code and XML Descriptions” on page 5-2.

See Also

Generate XML file for schema version | Maximum SHORT-NAME length | Use AUTOSAR compiler abstraction macros | Support root-level matrix I/O using one-dimensional arrays

Related Examples

- “Configure AUTOSAR XML Options” on page 4-43
- “Generate AUTOSAR C Code and XML Descriptions” on page 5-2

More About

- “Round-Trip Preservation of AUTOSAR XML File Structure and Element Information” on page 3-37

Code Generation with AUTOSAR Code Replacement Library

If your model is configured for AUTOSAR code generation, you can use the AUTOSAR 4.0 code replacement library to produce functions that closely align with the AUTOSAR standard.

In this section...

“Code Replacement Library for AUTOSAR Code Generation” on page 5-12

“Find Supported AUTOSAR Library Routines” on page 5-12

“Configure Code Generator to Use AUTOSAR 4.0 Code Replacement Library” on page 5-13

“AUTOSAR 4.0 Library Host Code Verification” on page 5-13

“Code Replacement Library Checks” on page 5-14

“AUTOSAR Code Replacement Library Example for IFL/IFL Function Replacement” on page 5-14

“Required Algorithm Property Settings for IFL/IFX Function and Block Mappings” on page 5-16

Code Replacement Library for AUTOSAR Code Generation

The AUTOSAR 4.0 code replacement library enables you to customize the code generator to produce C code that closely aligns with the AUTOSAR standard. Consider using the code replacement library if:

- You want to use service routines provided in the library.
- You have replacement code for the service routines.
- The replacement code follows the AUTOSAR file naming convention, that is, routines for any given specification are in one header file (for example, `Mfl.h` or `Mfx.h`)
- You have a build harness setup that can compile and link the AUTOSAR library with the generated code. For more information about building code for AUTOSAR, see “Code Generation”.

Note MATLAB and Simulink lookup table indexing differs from AUTOSAR MAP indexing. MATLAB takes the linear algebra approach—row ($u1$) and column ($u2$). AUTOSAR (and ASAM) takes the Cartesian coordinate approach—x-axis ($u2$) and y-axis ($u1$), where $u1$ and $u2$ are input arguments to Simulink 2-D lookup table blocks. Due to the difference, the code replacement software transposes the input arguments for AUTOSAR MAP routines.

For more information on code replacement and code replacement libraries, see “What Is Code Replacement?” (Embedded Coder) and “Code Replacement Libraries” (Embedded Coder).

Find Supported AUTOSAR Library Routines

To explore the AUTOSAR library routines supported by the AUTOSAR code replacement library, use the **Code Replacement Viewer**. To open the viewer, at the command prompt, enter `crviewer('AUTOSAR 4.0')`.

For more information, see “Choose a Code Replacement Library” (Simulink Coder).

Configure Code Generator to Use AUTOSAR 4.0 Code Replacement Library

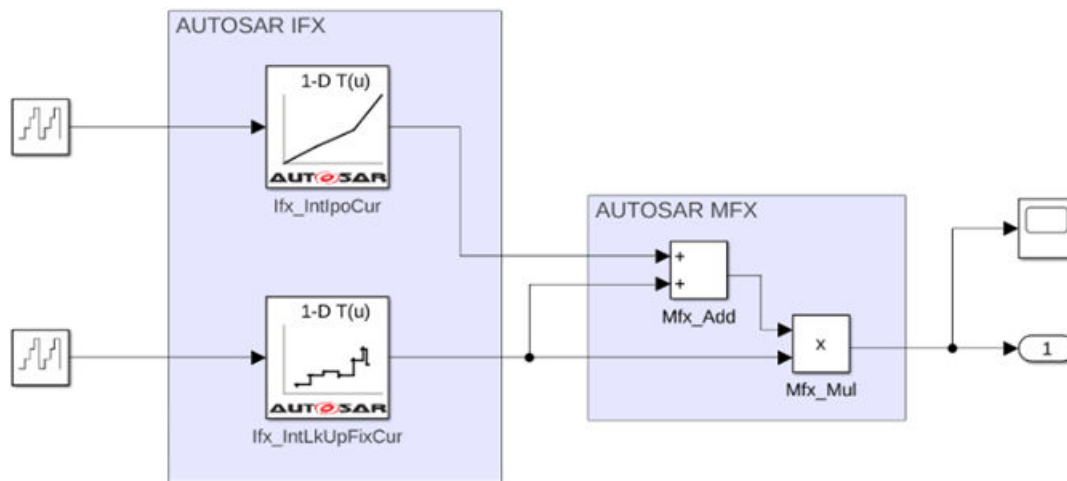
To configure the code generator to use the AUTOSAR code replacement library for your model, open the Configuration Parameters dialog box. Select **Code Generation > Interface > Code replacement libraries > AUTOSAR 4.0**.

For more information on code replacement and code replacement libraries, see “What Is Code Replacement?” (Embedded Coder) and “Code Replacement Libraries” (Embedded Coder).

AUTOSAR 4.0 Library Host Code Verification

To help support MATLAB host code verification for AUTOSAR models, AUTOSAR Blockset provides host implementations of IFX, IFL, MFX, and MFL routines in the AUTOSAR 4.0 library. The host library implementations enable software-in-the-loop (SIL) validation for models that trigger code replacements from the AUTOSAR 4.0 library.

Consider the following AUTOSAR model, which contains interpolation and math blocks that have been tuned to trigger AUTOSAR IFX and MFX routine code replacements. In the model configuration parameters, **System target file** is set to `autosar.tlc` and **Code replacement libraries** is set to `AUTOSAR 4.0`.



Configure and run a SIL simulation of the model. The SIL simulation:

- 1 Generates model code. MathWorks host library implementations are used in IFX, IFL, MFX, and MFL routine code replacements.
- 2 Builds the SIL application. The host library is linked to the SIL executable.
- 3 Runs the model and produces simulation output, based on your SIL settings.

If you prefer to use your own host library or custom code for SIL simulations, you can disable the MathWorks host library by using the following command:

```
set_param(modelName, 'DisableAUTOSARRoutinesHostLibrary', 'on');
```

Code Replacement Library Checks

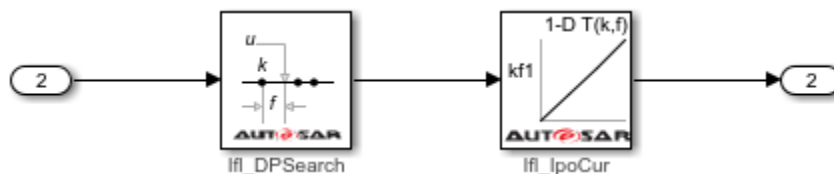
Code replacement requires that the combination of types for input, breakpoint, table, and output types are compatible with the AUTOSAR specification. Floating-point (IFL) replacement only supports single types while fixed-point (IFX) replacement supports uint8, uint16, int8, int16 and associated fixed-point types. When using these routine blocks, the type combination requirements vary and are enforced as required.

AUTOSAR Code Replacement Library Example for IFX/IFL Function Replacement

The **Code Replacement Viewer** lists AUTOSAR floating-point interpolation (IFL) and fixed-point interpolation (IFX) library routines that you can generate in lookup table C code. For replacing lookup table C code with IFL or IFX library routines, AUTOSAR Blocks provides lookup table blocks that are preconfigured for AUTOSAR code generation. You insert a block such as Curve or Map in your model, then open the block dialog box and configure the block to generate a specific interpolation routine required by your design. For more information, see “Configure Lookup Tables for AUTOSAR Calibration and Measurement” on page 4-273.

This example shows how to replace code generated for AUTOSAR lookup table blocks with functions that are compatible with AUTOSAR IFL library routines. If you want to replace code with IFX library routines, you can edit the lookup table block dialog boxes to change the targeted routine library.

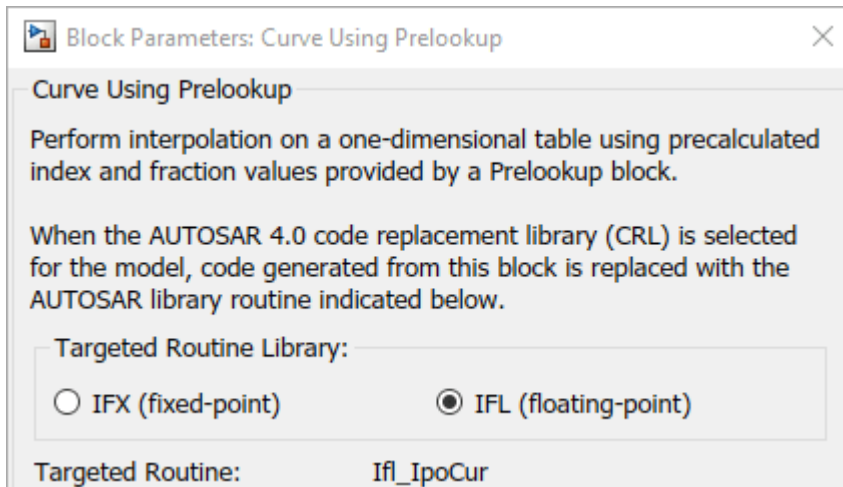
You can create your Simulink model by using any of these AUTOSAR lookup table blocks: Prelookup, Curve Using Prelookup, Map Using Prelookup, Curve, or Map. For example, here is a Prelookup block connected to a Curve Using Prelookup block.



You can also open the model file by entering the following command at the MATLAB® command line:

```
open_system('mAutosarLutObjs');
```

Open each lookup table block and configure it to generate a routine from the AUTOSAR 4.0 code replacement library (CRL). As you modify block settings, the block dialog box updates the name of the targeted AUTOSAR routine.



For details about configuring the blocks in this example, see “Configure COM_AXIS Lookup Tables by Using Lookup Table and Breakpoint Objects” on page 4-276.

- 1 Configure the code generator to use the AUTOSAR 4.0 CRL for your model. In the Configuration Parameters dialog box, select **Code Generation > Interface > Code replacement libraries > AUTOSAR 4.0**. Alternatively, from the command line or programmatically, use `set_param` to set the `CodeReplacementLibrary` parameter to 'AUTOSAR 4.0'.
- 2 Optionally, you can configure the model to produce a code generation report that summarizes which blocks trigger code replacements. In the Configuration Parameters dialog box, in the **Code Generation > Report** pane, select the option **Summarize which blocks triggered code replacements**. Alternatively, from the command line or programmatically, use `set_param` to set the `GenerateCodeReplacementReport` parameter to 'on'.
- 3 Build the model and review the generated code for expected code replacements. For example, search the generated code for the routine prefix `Ifl`.

```

/* Model step function */
void Runnable_Step(void)
{
    Ifl_DPResultF32_Type rtb_Prelookup;

    /* PreLookup: '<Root>/PreLookup' incorporates:
     * Inport: '<Root>/In2'
     */
    Ifl_DPSearch_f32(&rtb_Prelookup, Rte_IRead_Runnable_Step_In2_In2(),
        (Rte_CData_Bp_4_single()->Nx, (Rte_CData_Bp_4_single()->Bp1));

    /* Outport: '<Root>/Out2' incorporates:
     * Interpolation_n-D: '<Root>/Curve Using PreLookup'
     */
    Rte_IWrite_Runnable_Step_Out2_Out2(IfI_IpoCur_f32(&rtb_Prelookup,
        Rte_CData_Lcom_4_single()));
}

```

Required Algorithm Property Settings for IFL/IFX Function and Block Mappings

IFL/IFX Function and Block Mapping	Algorithm Property Parameters	Value	
Ifl_DPSearch Prelookup	Extrapolation method ExtrapMethod	Clip	
	Index search method IndexSearchMethod	Linear search or Binary search	
	Use last breakpoint for input at or above upper limit UseLastBreakPoint	On	
	Remove protection against out-of-range input in generated code RemoveProtectionInput	Off	
	Integer rounding mode RndMeth	Round or Zero	
	Ifl_IpoCur Interpolation Using Prelookup	Interpolation method InterpMethod	Linear
		Extrapolation method ExtrapMethod	Clip
Valid index input may reach last index ValidIndexMayReachLast		On	
Remove protection against out-of-range index in generated code RemoveProtectionIndex		Off	
Integer rounding mode RndMeth		Round or Zero	
Ifl_IpoMap Interpolation Using Prelookup		Interpolation method InterpMethod	Linear
	Extrapolation method ExtrapMethod	Clip	

IFL/IFX Function and Block Mapping	Algorithm Property Parameters	Value
	Valid index input may reach last index ValidIndexMayReachLast	On
	Remove protection against out-of-range index in generated code RemoveProtectionIndex	Off
	Integer rounding mode RndMeth	Round or Zero
IfI_IntIpoCur n-D Lookup Table	Interpolation method InterpMethod	Linear
	Extrapolation method ExtrapMethod	Clip
	Index search method IndexSearchMethod	Linear search or Binary search
	Use last table value for inputs at or above last breakpoint UseLastTableValue	On
	Remove protection against out-of-range input in generated code RemoveProtectionInput	Off
	Integer rounding mode RndMeth	Round or Zero
	IfI_IntIpoMap n-D Lookup Table	Interpolation method InterpMethod
Extrapolation method ExtrapMethod		Clip
Index search method IndexSearchMethod		Linear search or Binary search
Use last table value for inputs at or above last breakpoint UseLastTableValue		On

IFL/IFX Function and Block Mapping	Algorithm Property Parameters	Value
	Remove protection against out-of-range input in generated code RemoveProtectionInput	Off
	Integer rounding mode RndMeth	Round or Zero
Ifx_DPSearch Prelookup	Extrapolation method ExtrapMethod	Clip
	Index search method IndexSearchMethod	Linear search or Binary search
	Use last breakpoint for input at or above upper limit UseLastBreakPoint	On
	Remove protection against out-of-range input in generated code RemoveProtectionInput	Off
	Integer rounding mode RndMeth	Round or Zero
Ifx_IpoCur Interpolation Using Prelookup	Interpolation method InterpMethod	Linear
	Extrapolation method ExtrapMethod	Clip
	Valid index input may reach last index ValidIndexMayReachLast	On
	Remove protection against out-of-range index in generated code RemoveProtectionIndex	Off
	Integer rounding mode RndMeth	Round or Zero
Ifx_LkUpCur Interpolation Using Prelookup	Interpolation method InterpMethod	Flat

IFL/IFX Function and Block Mapping	Algorithm Property Parameters	Value
	Extrapolation method	Clip
	ExtrapMethod	
	Remove protection against out-of-range index in generated code	Off
	RemoveProtectionIndex	
	Integer rounding mode	Round or Zero
	RndMeth	
	Valid index input may reach last index	On
	ValidIndexMayReachLast	
Ifx_IpoMap	Interpolation method	Linear
Interpolation Using Prelookup	InterpMethod	
	Extrapolation method	Clip
	ExtrapMethod	
	Valid index input may reach last index	On
	ValidIndexMayReachLast	
	Remove protection against out-of-range index in generated code	Off
	RemoveProtectionIndex	
	Integer rounding mode	Round or Zero
	RndMeth	
Ifx_LkUpMap	Interpolation method	Nearest
Interpolation Using Prelookup	InterpMethod	
	Extrapolation method	Clip
	ExtrapMethod	
	Remove protection against out-of-range index in generated code	Off
	RemoveProtectionIndex	
	Integer rounding mode	Round or Zero
	RndMeth	

IFL/IFX Function and Block Mapping	Algorithm Property Parameters	Value
	Valid index input may reach last index ValidIndexMayReachLast	On
Ifx_LkUpBaseMap Interpolation Using Prelookup	Interpolation method InterpMethod Extrapolation method ExtrapMethod Remove protection against out-of-range index in generated code RemoveProtectionIndex Integer rounding mode RndMeth Valid index input may reach last index ValidIndexMayReachLast	Flat Clip Off Round or Zero On
Ifx_IntIpoCur n-D Lookup Table	Interpolation method InterpMethod Extrapolation method ExtrapMethod Index search method IndexSearchMethod Use last table value for inputs at or above last breakpoint UseLastTableValue Remove protection against out-of-range input in generated code RemoveProtectionInput Integer rounding mode RndMeth	Linear Clip Linear search or Binary search On Off Round or Zero
Ifx_IntLkUpCur n-D Lookup Table	Interpolation method InterpMethod	Flat

IFL/IFX Function and Block Mapping	Algorithm Property Parameters	Value
	Extrapolation method ExtrapMethod	Clip
	Index search method IndexSearchMethod	Linear search or Binary search
	Remove protection against out-of-range input in generated code RemoveProtectionInput	Off
	Integer rounding mode RndMeth	Round or Zero
	Use last table value for inputs at or above last breakpoint UseLastTableValue	On
	Ifx_IntIpoFixCur n-D Lookup Table	Interpolation method InterpMethod
Extrapolation method ExtrapMethod	Clip	
Index search method IndexSearchMethod	Evenly spaced points	
Use last table value for inputs at or above last breakpoint UseLastTableValue	On	
Remove protection against out-of-range input in generated code RemoveProtectionInput	Off	
Integer rounding mode RndMeth	Round or Zero	
Use last table value for inputs at or above last breakpoint UseLastTableValue	On	
Model configuration parameter Optimization > Default parameter behavior DefaultParameterBehavior	Inlined	

IFL/IFX Function and Block Mapping	Algorithm Property Parameters	Value
	Breakpoint data should match power 2 spacing.	
Ifx_IntLkUpFixCur n-D Lookup Table	Interpolation method	Flat
	InterpMethod	
	Extrapolation method	Clip
	ExtrapMethod	
	Index search method	Evenly spaced points
	IndexSearchMethod	
	Remove protection against out-of-range input in generated code	Off
	RemoveProtectionInput	
	Integer rounding mode	Round or Zero
RndMeth		
Model configuration parameter		
Optimization > Signals and Parameters > Default parameter behavior	Inlined	
DefaultParameterBehavior		
Breakpoint data must match power 2 spacing.		
Ifx_IntIpoFixICur n-D Lookup Table	Interpolation method	Linear
	InterpMethod	
	Extrapolation method	Clip
	ExtrapMethod	
	Index search method	Evenly spaced points
	IndexSearchMethod	
	Use last table value for inputs at or above last breakpoint	On
	UseLastTableValue	
	Remove protection against out-of-range input in generated code	Off
RemoveProtectionInput		
Integer rounding mode	Round or Zero	
RndMeth		

IFL/IFX Function and Block Mapping	Algorithm Property Parameters	Value
	Breakpoint data must not match power 2 spacing.	
Ifx_IntLkUpFixICur n-D Lookup Table	Interpolation method	Flat
	InterpMethod	
	Extrapolation method	Clip
	ExtrapMethod	
	Index search method	Evenly spaced points
	IndexSearchMethod	
	Remove protection against out-of-range input in generated code	Off
	RemoveProtectionInput	
	Integer rounding mode	Round or Zero
RndMeth		
Use last table value for inputs at or above last breakpoint	On	
UseLastTableValue		
	Breakpoint data must not match power 2 spacing.	
Ifx_IntIpoMap n-D Lookup Table	Interpolation method	Linear
	InterpMethod	
	Extrapolation method	Clip
	ExtrapMethod	
	Index search method	Linear search or Binary search
	IndexSearchMethod	
	Use last table value for inputs at or above last breakpoint	On
	UseLastTableValue	
Remove protection against out-of-range input in generated code	Off	
RemoveProtectionInput		
Integer rounding mode	Round or Zero	
RndMeth		

IFL/IFX Function and Block Mapping	Algorithm Property Parameters	Value	
Ifx_IntLkUpMap n-D Lookup Table	Interpolation method InterpMethod	Nearest	
	Extrapolation method ExtrapMethod	Clip	
	Index search method IndexSearchMethod	Linear search or Binary search	
	Remove protection against out-of-range input in generated code RemoveProtectionInput	Off	
	Integer rounding mode RndMeth	Round or Zero	
	Use last table value for inputs at or above last breakpoint UseLastTableValue	On	
	Ifx_IntLkUpBaseMap n-D Lookup Table	Interpolation method InterpMethod	Flat
		Extrapolation method ExtrapMethod	Clip
Index search method IndexSearchMethod		Linear search or Binary search	
Remove protection against out-of-range input in generated code RemoveProtectionInput		Off	
Integer rounding mode RndMeth		Round or Zero	
Use last table value for inputs at or above last breakpoint UseLastTableValue		On	
Ifx_IntIpoFixMap n-D Lookup Table		Interpolation method InterpMethod	Linear
		Extrapolation method ExtrapMethod	Clip

IFL/IFX Function and Block Mapping	Algorithm Property Parameters	Value
	Index search method IndexSearchMethod	Evenly spaced points
	Use last table value for inputs at or above last breakpoint UseLastTableValue	On
	Remove protection against out-of-range input in generated code RemoveProtectionInput	Off
	Integer rounding mode RndMeth	Round or Zero
	Model configuration parameter Optimization > Signals and Parameters > Default parameter behavior DefaultParameterBehavior	Inlined
	Breakpoint data must match power 2 spacing.	
	Ifx_IntLkUpFixMap n-D Lookup Table	Interpolation method InterpMethod
Extrapolation method ExtrapMethod	Clip	
Index search method IndexSearchMethod	Evenly spaced points	
Remove protection against out-of-range input in generated code RemoveProtectionInput	Off	
Integer rounding mode RndMeth	Round or Zero	
Use last table value for inputs at or above last breakpoint UseLastTableValue	On	

IFL/IFX Function and Block Mapping	Algorithm Property Parameters	Value
	Model configuration parameter Optimization > Signals and Parameters > Default parameter behavior DefaultParameterBehavior Breakpoint data must match power 2 spacing.	Inlined
Ifx_IntLkUpFixBaseMap n-D Lookup Table	Interpolation method InterpMethod	Flat
	Extrapolation method ExtrapMethod	Clip
	Index search method IndexSearchMethod	Evenly spaced points
	Remove protection against out-of-range input in generated code RemoveProtectionInput	Off
	Integer rounding mode RndMeth	Round or Zero
	Use last table value for inputs at or above last breakpoint UseLastTableValue	On
	Model configuration parameter Optimization > Signals and Parameters > Default parameter behavior DefaultParameterBehavior Breakpoint data must match power 2 spacing.	Inlined
Ifx_IntIpoFixIMap n-D Lookup Table	Interpolation method InterpMethod	Linear
	Extrapolation method ExtrapMethod	Linear
	Index search method IndexSearchMethod	Evenly spaced points

IFL/IFX Function and Block Mapping	Algorithm Property Parameters	Value
	Use last table value for inputs at or above last breakpoint UseLastTableValue	On
	Remove protection against out-of-range input in generated code RemoveProtectionInput	Off
	Integer rounding mode RndMeth	Round or Zero
	Breakpoint data must not match power 2 spacing.	
Ifx_IntLkUpFixIMap n-D Lookup Table	Interpolation method InterpMethod	Nearest
	Extrapolation method ExtrapMethod	Clip
	Index search method IndexSearchMethod	Evenly spaced points
	Remove protection against out-of-range input in generated code RemoveProtectionInput	Off
	Integer rounding mode RndMeth	Round or Zero
	Use last table value for inputs at or above last breakpoint UseLastTableValue	On
	Breakpoint data must not match power 2 spacing.	
Ifx_IntLkUpFixIBaseMap n-D Lookup Table	Interpolation method InterpMethod	Flat
	Extrapolation method ExtrapMethod	Clip
	Index search method IndexSearchMethod	Evenly spaced points

IFL/IFX Function and Block Mapping	Algorithm Property Parameters	Value
	Remove protection against out-of-range input in generated code RemoveProtectionInput	Off
	Integer rounding mode RndMeth	Round or Zero
	Use last table value for inputs at or above last breakpoint UseLastTableValue	On
	Breakpoint data must not match power 2 spacing.	

See Also

Related Examples

- “Configure Lookup Tables for AUTOSAR Calibration and Measurement” on page 4-273

More About

- “What Is Code Replacement?” (Embedded Coder)
- “Code Generation”

Verify AUTOSAR C Code with SIL and PIL

As part of developing AUTOSAR software for the Classic Platform, you can carry out code verification of AUTOSAR software components by using software-in-the-loop (SIL) and processor-in-the-loop (PIL) simulations. Use SIL for verification of generated source code on your development computer, and PIL for verification of object code on your production target hardware.

Through behavioral and structural comparisons, code verification demonstrates the equivalence between a component model and its generated code. You can:

- Test numerical equivalence between your component model and generated code by comparing normal mode simulation results against SIL or PIL simulation results.
- Show the absence of unintended functionality by comparing model coverage against code coverage or performing a traceability analysis.
 - Configure SIL and PIL simulations to generate code coverage metrics.
 - Generate reports that provide bidirectional traceability between model objects and generated code.

With AUTOSAR models, you run SIL and PIL testing by configuring either the top model or Model blocks.

- For unit-level testing of an AUTOSAR software component, use top model SIL or PIL. You can test a top model that is configured for the AUTOSAR system target file (`autosar.tlc`) by setting the simulation mode to **Software-in-the-Loop (SIL)** or **Processor-in-the-Loop (PIL)**.
- For unit-level testing of a subcomponent referenced from an AUTOSAR software component, use Model block SIL or PIL. In the Model block for the submodel, set **Simulation mode** to SIL or PIL and set **Code interface** to Model reference.
- For composition-level testing of multiple AUTOSAR software components, reference the component models in a composition, architecture, or test harness model. In the Model block for each component under test, set **Simulation mode** to SIL or PIL and set **Code interface** to Top model.

For more information, see “Simulation with Top Model” (Embedded Coder) and “Simulation with Model Blocks” (Embedded Coder).

If you have Simulink Test software, you can use test harnesses to:

- Perform composition-level testing of AUTOSAR software components. For more information, see “Testing AUTOSAR Compositions” (Simulink Test).
- Perform unit-level testing of atomic subsystems in AUTOSAR software components. For more information, see “Unit Test Subsystem Code with SIL/PIL Manager” (Embedded Coder).

See Also

Related Examples

- “Simulation with Top Model” (Embedded Coder)
- “Simulation with Model Blocks” (Embedded Coder)
- “Testing AUTOSAR Compositions” (Simulink Test)

- “Unit Test Subsystem Code with SIL/PIL Manager” (Embedded Coder)

More About

- “SIL and PIL Simulations” (Embedded Coder)
- “Choose a SIL or PIL Approach” (Embedded Coder)

Integrate Generated Code for Multi-Instance Software Components

When you build an AUTOSAR software component model that is configured for multiple instantiation:

- The generated ARXML describes internal data such as `BlockIO` and `DWork` as C-typed per-instance memory (PIM).
- The generated model header file `model.h` contains type definitions for the PIMs.

When you integrate the generated ARXML files and code into the AUTOSAR run-time environment (RTE), the RTE generator does not automatically generate the PIM type definitions. To make the type definitions available for component instances, the RTE must include the generated model header file.

The method for including the model header file varies according to the integration tooling. For example:

- In Vector tooling, file `Rte.h` includes an optional user types file, `Rte_UserTypes.h`. Update `Rte_UserTypes.h` to include `model.h`.
- In ETAS® tooling, `Rte_UserCfg.h` is an optional user configuration file. Update `Rte_UserCfg.h` to include `model.h`.

See Also

Related Examples

- “Map Calibration Data for Submodels Referenced from AUTOSAR Component Models” on page 4-65
- “Configure Subcomponent Data for AUTOSAR Calibration and Measurement” on page 4-255
- “Configure AUTOSAR Shared or Per-Instance Parameters” on page 4-212
- “Configure AUTOSAR Per-Instance Memory” on page 4-201
- “Multi-Instance Components” on page 2-8

Import and Simulate AUTOSAR Code from Previous Releases

You can import into the current release AUTOSAR component code that you generated in a previous release. To observe the interaction of code from a previous release with components implemented in the current release, run software-in-the-loop (SIL) or processor-in-the-loop (PIL) simulations with the imported code.

For more information, see:

- “Cross-Release Code Integration” (Embedded Coder)
- “Import AUTOSAR Code from Previous Releases” (Embedded Coder)

See Also

More About

- “Code Generation”

Limitations and Tips

The following limitations apply to AUTOSAR code generation.

Generate Code Only Check Box

If you do not select the **Generate code only** check box, the software produces an error message when you build the model. The message states that you can build an executable with the AUTOSAR system target file only if you:

- Configure the model to create a software-in-the-loop (SIL) or processor-in-the-loop (PIL) block
- Run the model in SIL or PIL simulation mode
- Provide a custom template makefile

AUTOSAR Compiler Abstraction Macros (Classic Platform)

The software does not generate AUTOSAR compiler abstraction macros for data or functions arising from the following:

- Model blocks
- Stateflow
- MATLAB Coder
- Shared utility functions
- Custom storage classes
- Local or temporary variables

Preservation of Bus Element Dimensions in Exported ARXML and Code

Bus element dimensions are preserved in the exported ARXML and generated code when a model is configured with the property **Array layout** set to Row-major. Previously, if a model contained a `Simulink.Bus` data type that had a multidimensional array `Simulink.BusElement`, the exported ARXML and generated code flattened the bus element to a one-dimensional array. Now, the generated code and exported ARXML preserves the dimensionality as expected.

C++11 Style Scoped Enum Classes Generated for AUTOSAR Adaptive Applications

To facilitate easier integration, by default the generated C++ code for AUTOSAR adaptive models emit C++ 11 style scoped enum class data types in the generated code. You can view this data type definition in the header file for enumerations located in the `aragen/stub` folder of the build folder. This data type definition is standardized and validated prior to code generation.

The following table shows a comparison of a scoped enum class definition versus the previously generated code behavior for a dynamic enumeration:

```
Simulink.defineIntEnumType('BasicColors', ...
{'Red', 'Green', 'Blue'}, ...
[0;1;2], ...
```

```
'DataScope', 'Auto', ...
'StorageType', 'uint8')
```

Generated Enumeration Definition in Header File

Previous Behavior (C++03)	Current Default Behavior (C++11)
<pre>#ifndef IMPL_TYPE_BASICCOLORS_H_ #define IMPL_TYPE_BASICCOLORS_H_ #include <cstdint> using BasicColors = uint8_t; const BasicColors Red = 0; const BasicColors Green = 1; const BasicColors Blue = 2; #endif //IMPL_TYPE_BASICCOLORS_H_</pre>	<pre>#ifndef IMPL_TYPE_BASICCOLORS_H_ #define IMPL_TYPE_BASICCOLORS_H_ #include <cstdint> enum class BasicColors : uint8_t { Red = 0, Green = 1, Blue = 2 }; #endif //IMPL_TYPE_BASICCOLORS_H_</pre>

The default behavior is determined by the default **Language standard** for a model set to C++11 (ISO). If you configure this setting so that a model generates C++ 03, then the generated code emits the previous code definition behavior and may not compile if used with a third-party generator.

AUTOSAR Adaptive Software Component Modeling

- “Model AUTOSAR Adaptive Software Components” on page 6-2
- “Create and Configure AUTOSAR Adaptive Software Component” on page 6-6
- “Import AUTOSAR Adaptive Software Descriptions” on page 6-12
- “Import AUTOSAR Adaptive Components to Simulink” on page 6-13
- “Import AUTOSAR Package into Adaptive Component Model” on page 6-17
- “Configure AUTOSAR Adaptive Elements and Properties” on page 6-21
- “Map AUTOSAR Adaptive Elements for Code Generation” on page 6-37
- “Configure AUTOSAR Adaptive Software Components” on page 6-41
- “Model AUTOSAR Adaptive Service Communication” on page 6-50
- “Configure Memory Allocation for AUTOSAR Adaptive Service Data” on page 6-60
- “Configure AUTOSAR Adaptive Service Discovery Modes” on page 6-62
- “Configure AUTOSAR Adaptive Service Instance Identification” on page 6-64
- “Model AUTOSAR Adaptive Persistent Memory” on page 6-66
- “Generate AUTOSAR Adaptive C++ Code and XML Descriptions” on page 6-68
- “Configure AUTOSAR Adaptive Code Generation” on page 6-73
- “Configure AUTOSAR Adaptive Data for Run-Time Calibration and Measurement” on page 6-80
- “Configure AUTOSAR Adaptive Model for External Mode Simulation” on page 6-82
- “Build Library or Executable from AUTOSAR Adaptive Model” on page 6-83
- “Build Out of the Box Linux Executable from AUTOSAR Adaptive Model” on page 6-86
- “Configure Run-Time Logging for AUTOSAR Adaptive Executables” on page 6-89
- “Get Started with Embedded Coder Support Package for Linux Applications” on page 6-92
- “Event Communication Between AUTOSAR Adaptive Applications Using Message Polling” on page 6-96
- “Event Communication Between AUTOSAR Adaptive Applications Using Message Triggering” on page 6-100

Model AUTOSAR Adaptive Software Components

In Simulink, you can flexibly model the structure and behavior of software components for the AUTOSAR Adaptive Platform.

The AUTOSAR Adaptive Platform defines a service-oriented architecture for automotive components that must flexibly adapt to external events and conditions. Compared to the AUTOSAR Classic Platform, the Adaptive Platform requires:

- High-performance computing, potentially with multiple cores and heterogeneous processor types.
- Fast communication, potentially with Ethernet or networks on chips.
- Strong service-based interaction among components.
- Ability to adapt running automotive applications to external events and information sources (potentially for highly automated driving), as well as external communication, monitoring, and live software updates.

An AUTOSAR adaptive system potentially contains multiple interconnected adaptive software components. You deploy adaptive software components in the run-time environment defined by the Adaptive Platform, AUTOSAR Runtime for Adaptive Applications (ARA).

An AUTOSAR adaptive software component provides and consumes services. The adaptive service architecture is flexible, scalable, and distributed. Services can be discovered dynamically and can run on local or remote Electronic Control Units (ECUs). Each software component contains:

- An automotive algorithm, which performs tasks in response to received events.
- Required and provided ports, each associated with a service interface, through which events are received and sent.
- Service interfaces, which provide the framework for event-based communication, and their associated events and namespaces.

To model an AUTOSAR adaptive software component in Simulink, you start with a model that contains an automotive algorithm. From that model, you generate an AUTOSAR Dictionary that defines service interfaces, and an AUTOSAR code perspective that maps Simulink model elements to AUTOSAR component elements. As you further develop and refine the adaptive component in Simulink, you can iteratively simulate and build the model.

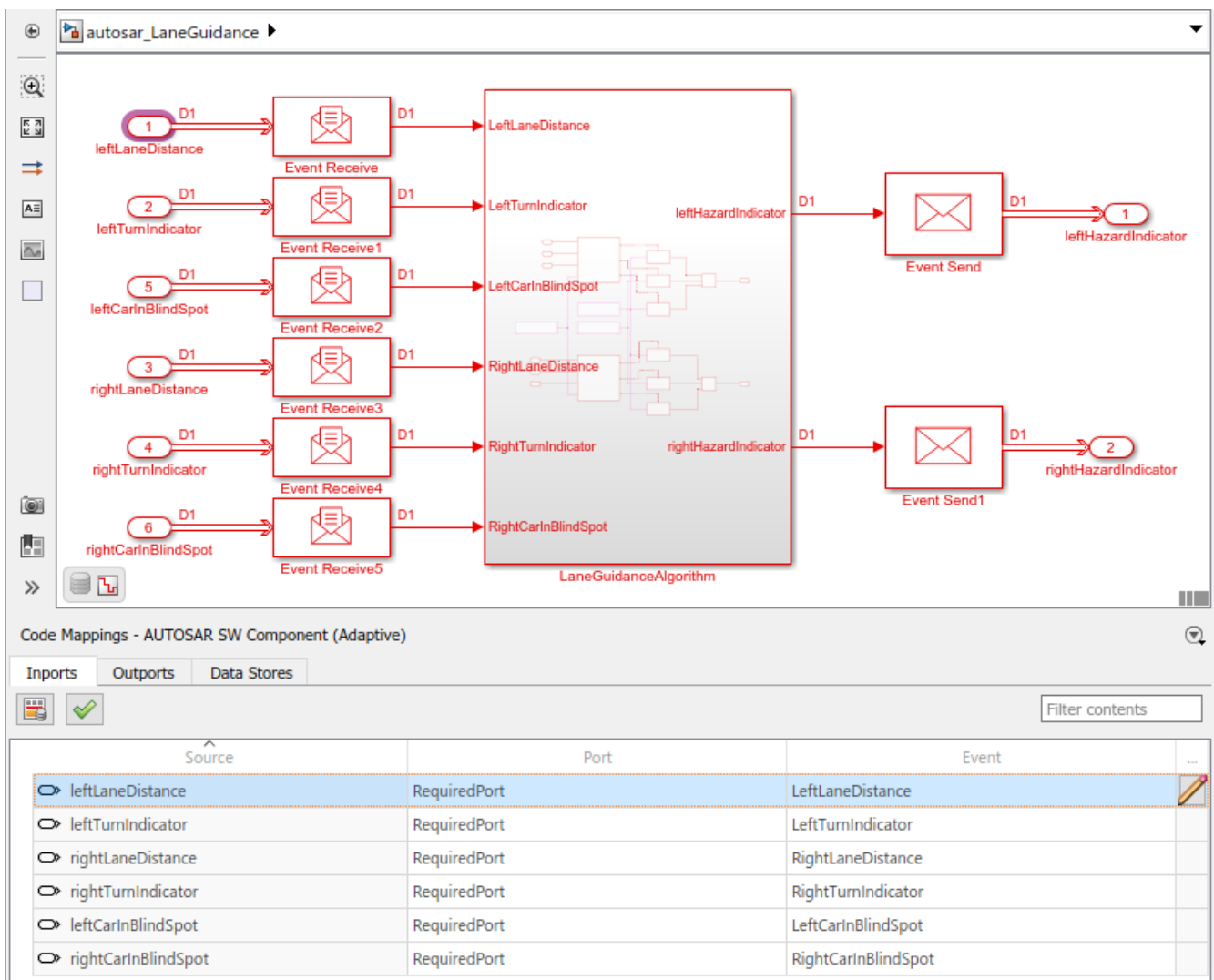
When you complete the component implementation, you can combine the adaptive software component model with other component models in an application-level simulation container model. The end goal is to deploy the component as part of an application in the ARA environment.

Here is the high-level workflow for modeling software components based on the AUTOSAR Adaptive Platform.

- 1 Open a Simulink model that either is empty or contains a functional algorithm.
- 2 Using the Model Configuration Parameters dialog box, configure the model for adaptive AUTOSAR code generation. Set **System target file** to `autosar_adaptive.tlc`.
- 3 Develop the model algorithmic content for use in an AUTOSAR adaptive software component. If the model is empty, construct or copy in an algorithm. Possible sources for algorithms include algorithmic elements in other Simulink models. Examples include subsystems, referenced models, MATLAB Function blocks, and C Caller blocks.

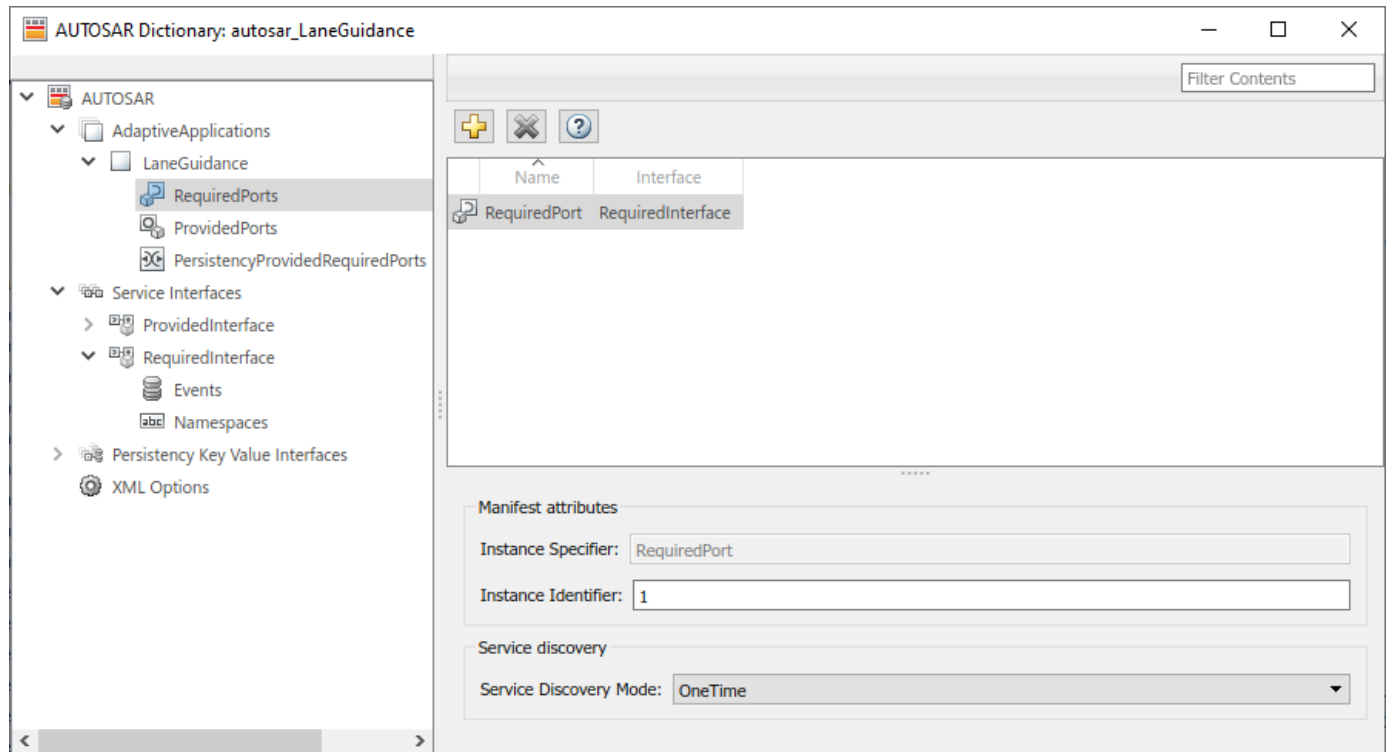
- 4 At the top level of the model, set up event-based communication.
 - After each root inport, add an Event Receive block, which converts an input event to a signal while preserving the signal values and data type.
 - Before each root outport, add an Event Send block, which converts an input signal to an event while preserving the signal values and data type.
- 5 Map the algorithm model to an AUTOSAR adaptive software component. For example, in the **Apps** tab, click **AUTOSAR Component Designer**. Because the model is unmapped, the AUTOSAR Component Quick Start opens.

Work through the quick-start procedure. Click **Finish** to map the model. The model opens in the AUTOSAR code perspective.



- 6 Using the AUTOSAR code perspective and the AUTOSAR Dictionary (or equivalent AUTOSAR map and property functions), further refine the AUTOSAR adaptive model configuration.
 - In the AUTOSAR code perspective, examine the mapping of Simulink inports and outports to AUTOSAR required and provided ports and events.

- In the AUTOSAR Dictionary, examine the AUTOSAR properties for RequiredPorts, ProvidedPorts, and Service Interfaces.



You can expand service interface nodes to examine their associated AUTOSAR events and define namespaces for interface C++ code.

- 7 Build the AUTOSAR adaptive software component model. Building the model generates:
 - C++ files that implement the model algorithms for the AUTOSAR Adaptive Platform and provide shared data type definitions.
 - AUTOSAR XML descriptions of the AUTOSAR adaptive software component and manifest information for application deployment and service configuration.
 - C++ files that implement a main program module.
 - AUTOSAR Runtime Adaptive (ARA) environment header files.
 - CMakeLists.txt file that supports CMake generation of executables.

For more information, see “Configure AUTOSAR Adaptive Software Components” on page 6-41.

See Also

Event Receive | Event Send

Related Examples

- “Configure AUTOSAR Adaptive Software Components” on page 6-41
- “Create and Configure AUTOSAR Adaptive Software Component” on page 6-6
- “Map AUTOSAR Adaptive Elements for Code Generation” on page 6-37

- “Configure AUTOSAR Adaptive Elements and Properties” on page 6-21
- “Model AUTOSAR Adaptive Service Communication” on page 6-50
- “Configure AUTOSAR Adaptive Code Generation” on page 6-73

More About

- “Comparison of AUTOSAR Classic and Adaptive Platforms” on page 1-5

Create and Configure AUTOSAR Adaptive Software Component

Create an AUTOSAR adaptive software component model from an algorithm model.

AUTOSAR Blockset software supports AUTomotive Open System ARchitecture (AUTOSAR), an open and standardized automotive software architecture. Automobile manufacturers, suppliers, and tool developers jointly develop AUTOSAR components. To develop AUTOSAR adaptive components in Simulink, follow this general workflow:

- 1 Create a Simulink representation of an AUTOSAR adaptive component.
- 2 Develop the component by refining the AUTOSAR configuration and creating algorithmic model content.
- 3 Generate ARXML descriptions and algorithmic C++ code for testing in Simulink or integration into an AUTOSAR run-time environment. (AUTOSAR code generation requires Simulink Coder and Embedded Coder.)

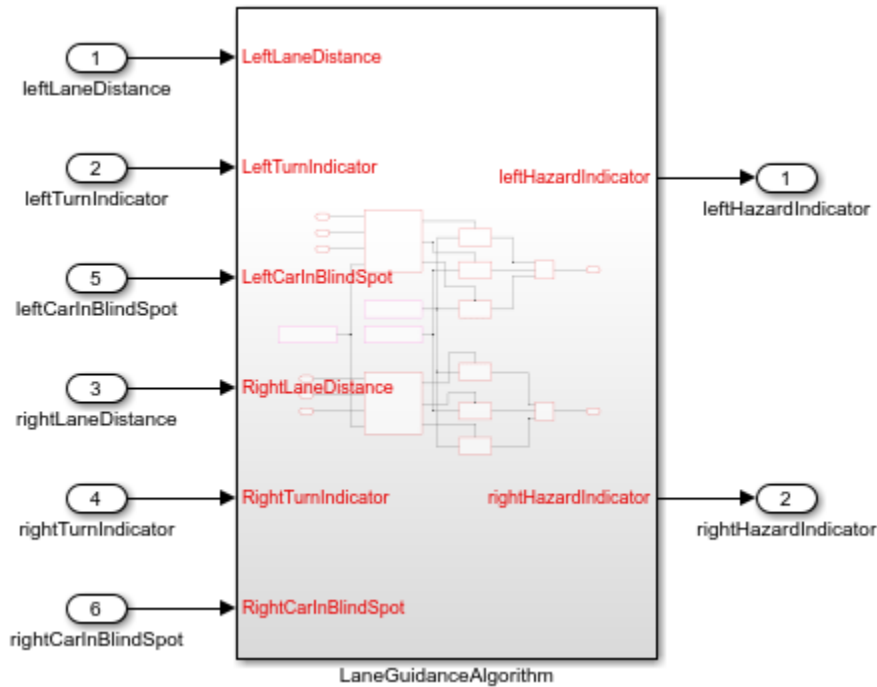
Create AUTOSAR Adaptive Software Component in Simulink

To create an initial Simulink representation of an AUTOSAR adaptive software component, you take one of these actions:

- Create an AUTOSAR adaptive software component using an existing Simulink model.
- Import an AUTOSAR adaptive software component description from ARXML files into a new Simulink model. (See example “Import AUTOSAR Adaptive Components to Simulink” on page 6-13.)

To create an AUTOSAR adaptive software component using an existing model, first open a Simulink component model for which an AUTOSAR software component is not mapped. This example uses AUTOSAR example model LaneGuidance.

```
open_system('LaneGuidance');
```



In the model window, on the **Modeling** tab, select **Model Settings**. In the Configuration Parameters dialog box, **Code Generation** pane, set the system target file to `autosar_adaptive.tlc`. Click **OK**.

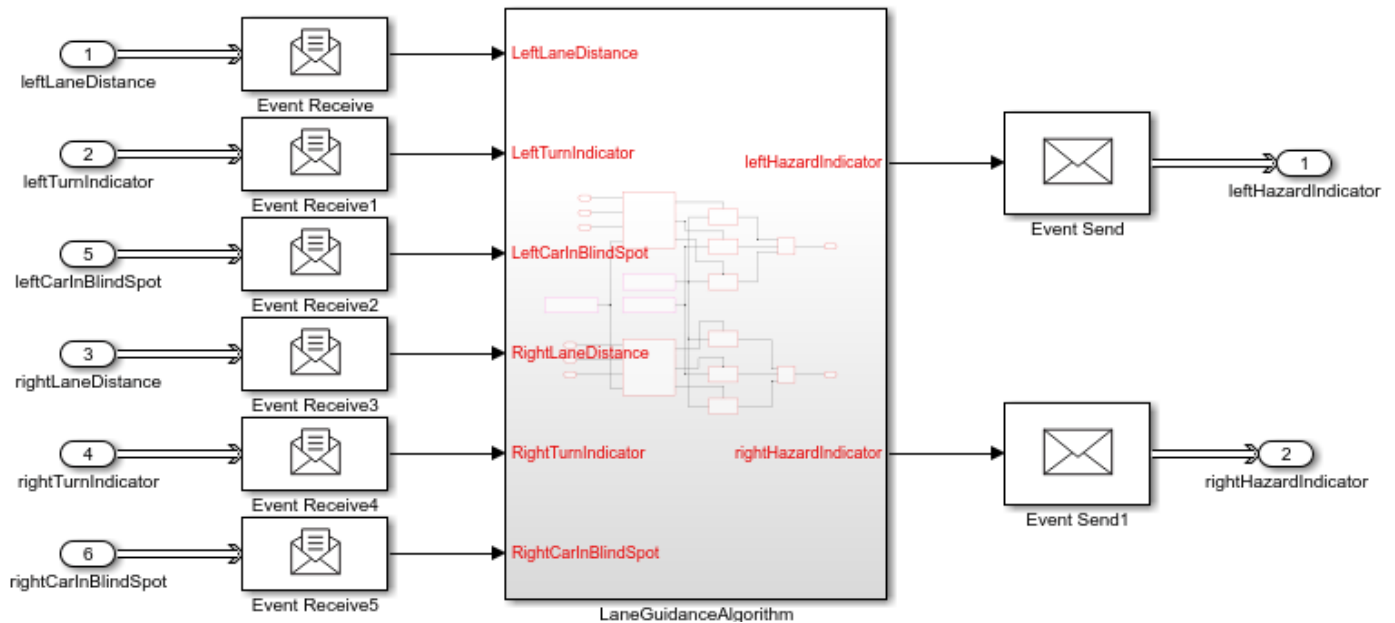
At the top level of the model, set up event-based communication. An AUTOSAR adaptive software component provides and consumes services. Each component contains:

- An algorithm that performs tasks in response to received events
- Required and provided ports, each associated with a service interface
- Service interfaces, with associated events and associated namespaces

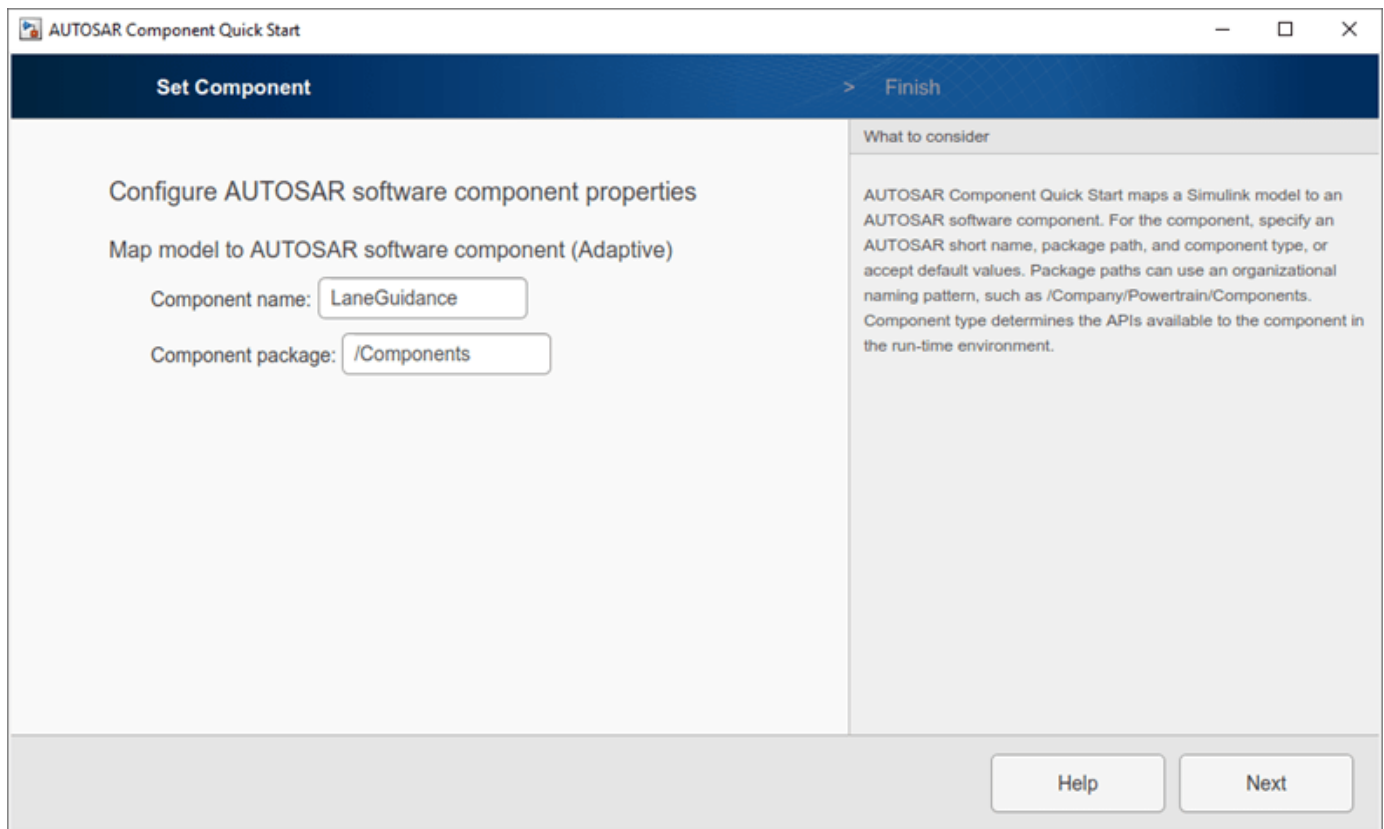
AUTOSAR Blockset provides Event Receive and Event Send blocks to make the necessary event and signal connections.

- After each root input, add an Event Receive block, which converts an input event to a signal while preserving the signal values and data type.
- Before each root output, add an Event Send block, which converts an input signal to an event while preserving the signal values and data type.

(To expedite the block insertion, you can copy the event blocks from AUTOSAR example model `autosar_LaneGuidance`.)



To configure the model as a mapped AUTOSAR adaptive software component, open the AUTOSAR Component Quick Start. On the **Apps** tab, click **AUTOSAR Component Designer**. The AUTOSAR Component Quick Start opens.

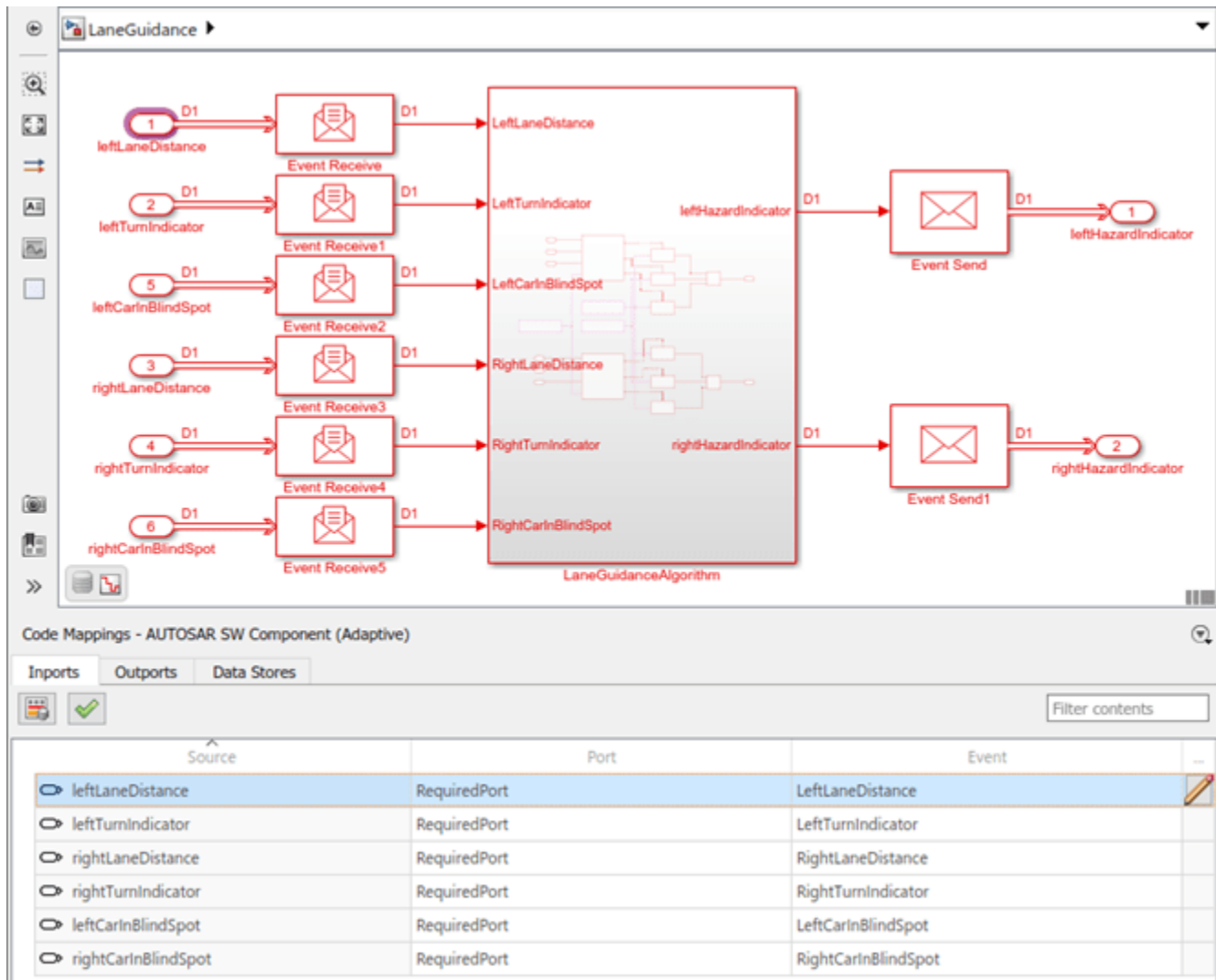


To configure the model for AUTOSAR adaptive software component development, work through the quick-start procedure. This example accepts default settings for the options in the Quick Start **Set Component** pane.

In the **Finish** pane, when you click **Finish**, your model opens in the AUTOSAR code perspective.

Configure AUTOSAR Adaptive Software Component in Simulink

The AUTOSAR code perspective displays your model, and directly below the model, the Code Mappings editor.

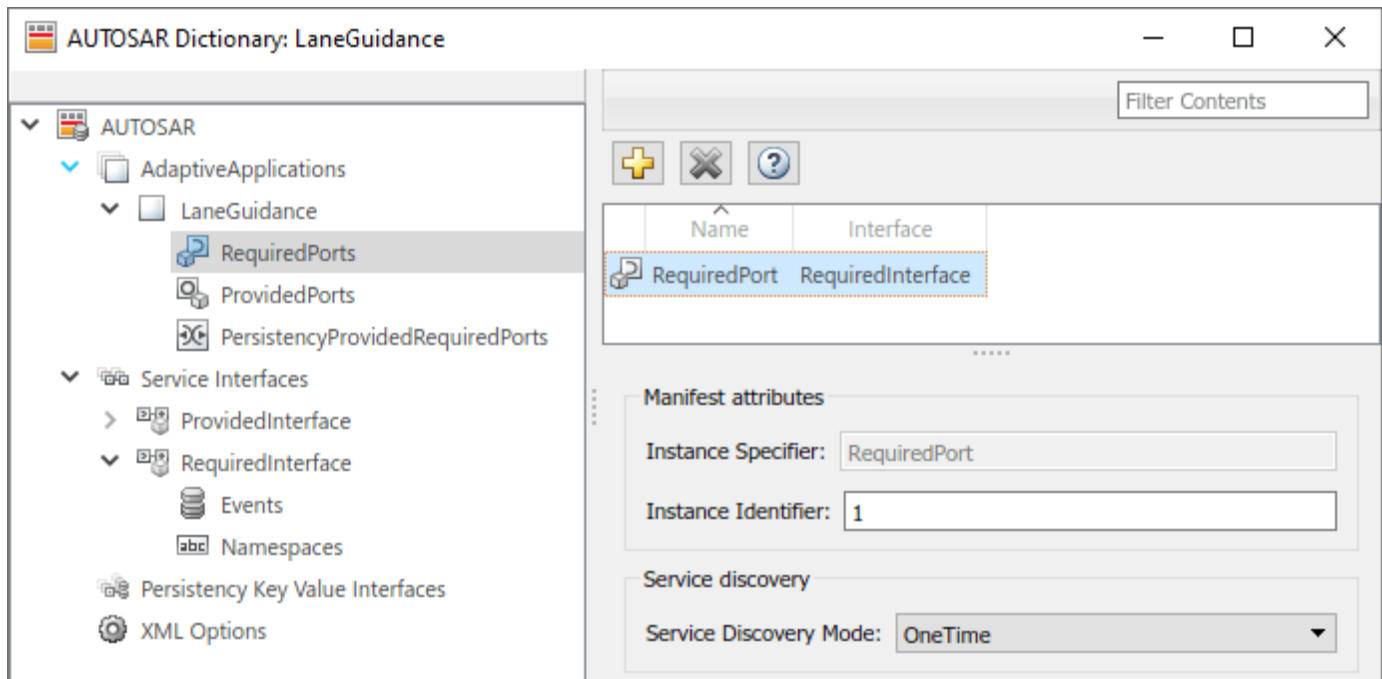


Next you use the Code Mappings editor and the AUTOSAR Dictionary to further develop the AUTOSAR adaptive component.

The Code Mappings editor displays model inports and outports. Use the editor to map Simulink inports and outports to AUTOSAR required ports and provided ports (defined in the AUTOSAR standard) from a Simulink model perspective.

Open each Code Mapping tab and examine the mapped model elements. To modify the AUTOSAR mapping for an element, select an element and modify its associated properties. When you select an element, it is highlighted in the model.

To configure the AUTOSAR properties of the mapped AUTOSAR adaptive software component, open the AUTOSAR Dictionary. In the Code Mappings editor, click the AUTOSAR Dictionary button, which is the leftmost icon. The AUTOSAR Dictionary opens in the AUTOSAR view that corresponds to the Simulink element that you last selected and mapped in the Code Mappings editor. If you selected and mapped a Simulink inport, the dictionary opens in RequiredPorts view and displays the AUTOSAR port to which you mapped the inport.



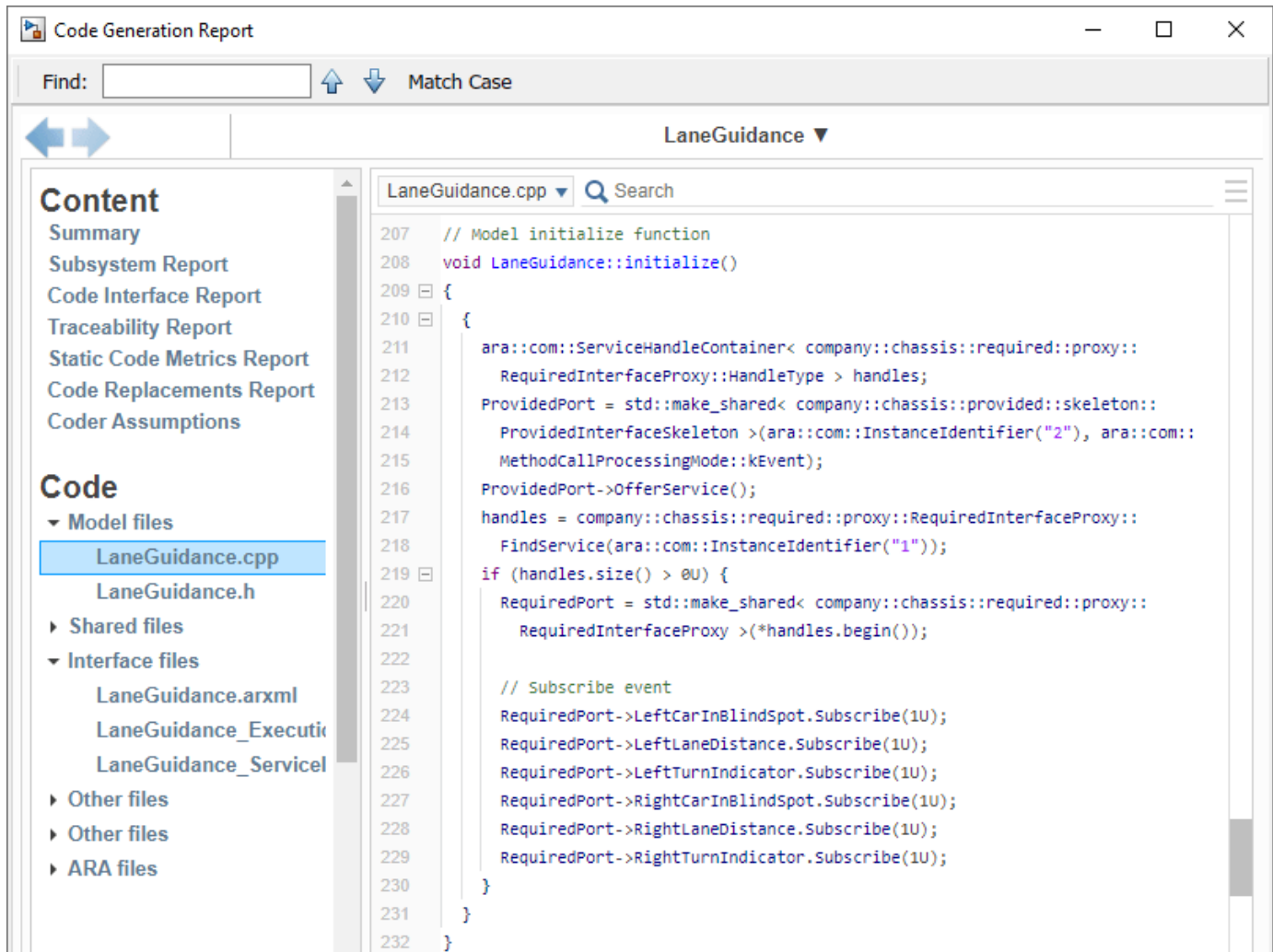
The AUTOSAR Dictionary displays the mapped AUTOSAR adaptive component and its elements, communication interfaces, and XML options. Use the dictionary to configure AUTOSAR elements and properties from an AUTOSAR component perspective.

Open each node and examine its AUTOSAR elements. To modify an AUTOSAR element, select an element and modify its associated properties. AUTOSAR XML and AUTOSAR-compliant C code generated from the model reflect your modifications.

Generate C++ Code and ARXML Descriptions (Embedded Coder)

If you have Simulink Coder and Embedded Coder software, you can build the AUTOSAR adaptive model. Building the AUTOSAR model generates AUTOSAR-compliant C++ code and exports AUTOSAR XML (ARXML) descriptions. In the model window, press **Ctrl+B** or, on the **AUTOSAR** tab, click **Generate Code**.

When the build completes, a code generation report opens. Examine the report. Verify that your Code Mappings editor and AUTOSAR Dictionary changes are reflected in the C++ code and ARXML descriptions. For example, use the **Find** field to search for the names of the Simulink model elements and AUTOSAR component elements that you modified.



Related Links

- “AUTOSAR Component Configuration” on page 4-3
- “Code Generation” (Adaptive Platform)
- “AUTOSAR Blockset”

Import AUTOSAR Adaptive Software Descriptions

You can import AUTOSAR XML (ARXML) descriptions of adaptive software components, service interfaces, and data types into Simulink. Use the ARXML importer to:

- Create an initial Simulink representation of an AUTOSAR adaptive software component.
- Update a mapped AUTOSAR adaptive component model with shared ARXML definitions of service interfaces and data types.

You can participate in round-trip exchanges of adaptive component ARXML descriptions between Simulink and other development environments.

To create an initial Simulink representation of AUTOSAR adaptive software component from an ARXML component description, use the ARXML importer function `createComponentAsModel`. For example:

```
ar = arxml.importer('myAdaptiveSWC.arxml')
createComponentAsModel(ar, '/Company/Components/Swc')
```

For a detailed example, see “Import AUTOSAR Adaptive Components to Simulink” on page 6-13.

To update a mapped AUTOSAR adaptive component model with shared ARXML definitions, use the ARXML importer function `updateAUTOSARProperties`. For example:

```
modelName = 'my_adaptive_swc';
open_system(modelName);
ar = arxml.importer('ServiceInterfaces.arxml');
updateAUTOSARProperties(ar,modelName);
```

For a detailed example, see “Import AUTOSAR Package into Adaptive Component Model” on page 6-17.

See Also

`createComponentAsModel` | `updateAUTOSARProperties`

Related Examples

- “Import AUTOSAR Adaptive Components to Simulink” on page 6-13
- “Import AUTOSAR Package into Adaptive Component Model” on page 6-17
- “Configure AUTOSAR Adaptive XML Options” on page 6-33
- “Round-Trip Preservation of AUTOSAR XML File Structure and Element Information” on page 3-37

More About

- “AUTOSAR ARXML Importer” on page 3-35
- “Workflows for AUTOSAR” on page 1-13

Import AUTOSAR Adaptive Components to Simulink

Create Simulink® models from XML descriptions of AUTOSAR adaptive software components.

Import AUTOSAR Adaptive Components from ARXML Files to Simulink

Use the MATLAB function `createComponentAsModel` to import AUTOSAR XML (ARXML) adaptive software component descriptions and create Simulink models.

First, parse the ARXML description files and list the components they contain.

```
ar = arxml.importer({'fusion_app.arxml','radarService_app_mod.arxml','radar_svc_mod.arxml','stdt...
names = getComponentNames(ar)

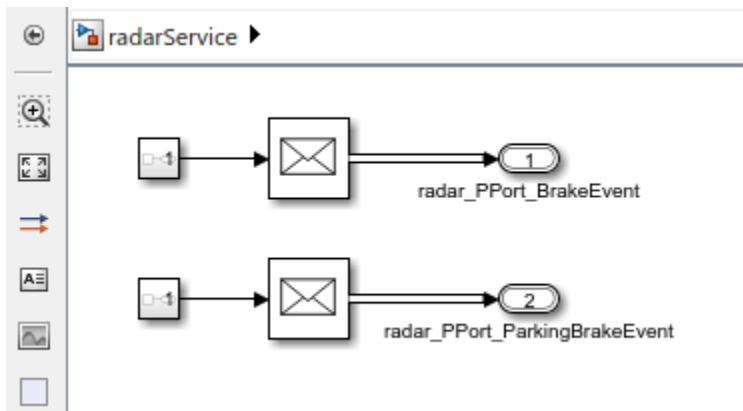
names = 2x1 cell
    {'/RadarFusion/fusion'      }
    {'/RadarFusion/radarService'}
```

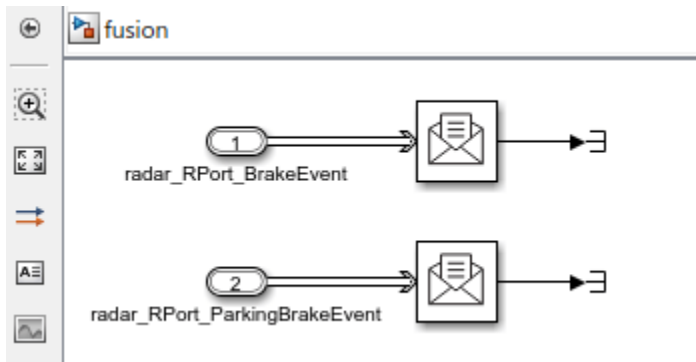
For each listed adaptive software component, use `createComponentAsModel` to create a Simulink representation. These commands create models named `fusion` and `radarService`.

```
createComponentAsModel(ar, '/RadarFusion/fusion');
createComponentAsModel(ar, '/RadarFusion/radarService');
```

Each created model contains:

- Simulink elements configured to model AUTOSAR adaptive component elements.
- An AUTOSAR dictionary, which stores the imported AUTOSAR adaptive element definitions.
- A mapping of the Simulink model elements to the AUTOSAR adaptive component elements.





In each model:

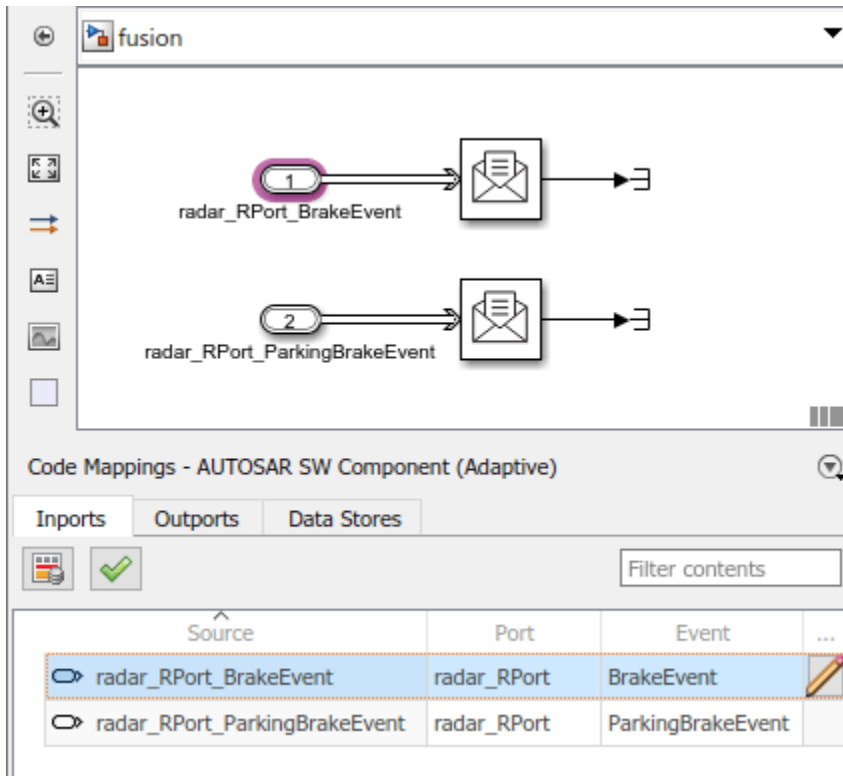
- Simulink ports represent AUTOSAR adaptive component provide and require ports.
- After each root inport, an Event Receive block converts an input event to a signal while preserving the signal values and data type.
- Before each root outport, an Event Send block converts an input signal to an event while preserving the signal values and data type.
- The ports are stubbed with Ground and Terminator blocks so that the model can immediately be updated and simulated.

Configure AUTOSAR Adaptive Software Component in Simulink

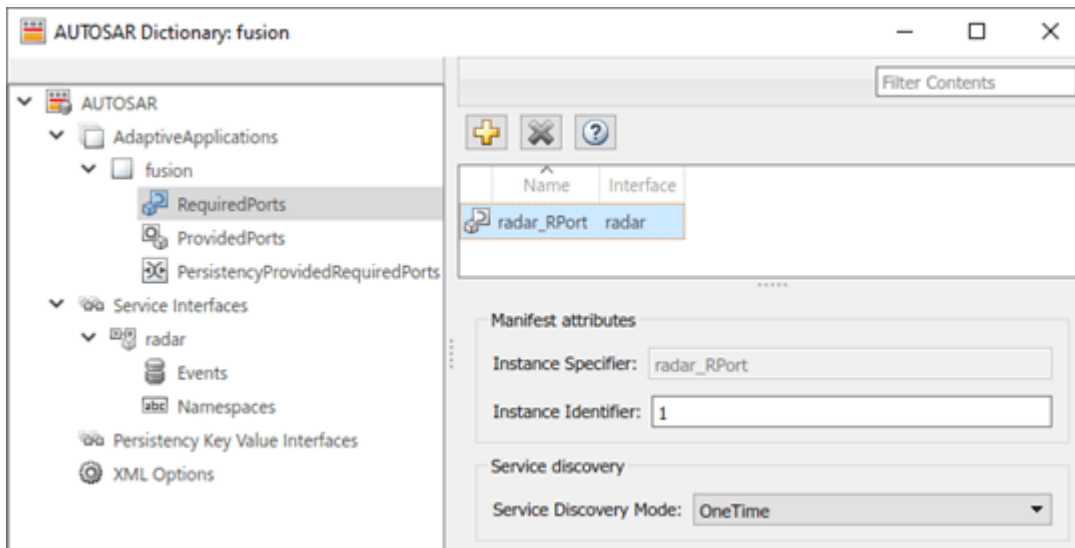
After you create an AUTOSAR adaptive software component model, use the **AUTOSAR Component Designer** app to refine the configuration of the AUTOSAR adaptive component.

Open an adaptive component model. On the **Apps** tab, select **AUTOSAR Component Designer**. The **AUTOSAR** tab opens.

To view the mapping of Simulink model elements to AUTOSAR adaptive component elements, open the Code Mappings pane. Use this view to map model elements to AUTOSAR component elements from a Simulink model perspective.



To view AUTOSAR adaptive element definitions, on the **AUTOSAR** tab, select **Code Interface > AUTOSAR Dictionary**. The dictionary opens. Use this view to configure AUTOSAR elements from an AUTOSAR component perspective.

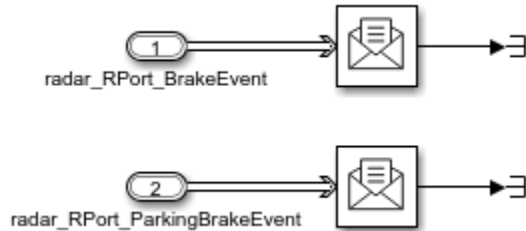


For more information, see “AUTOSAR Component Configuration” on page 4-3.

Develop AUTOSAR Adaptive Component Algorithms, Simulate, and Generate Code

After you create an AUTOSAR adaptive software component model and refine the configuration, you develop the component. Create algorithmic model content that implements the component requirements.

For example, the `fusion` component model that you created contains an initial stub implementation of the component behavior.



To implement the component requirements, replace the Terminator blocks with blocks that implement Simulink algorithms.

As you develop AUTOSAR adaptive components, you can:

- Simulate the component model individually or in a containing composition or test harness.
- Generate ARXML component description files and algorithmic C++ code for testing in Simulink or integration into an AUTOSAR run-time environment. (AUTOSAR code generation requires Simulink Coder and Embedded Coder.)

For more information, see “Component Development” and “Code Generation”.

Related Links

- `createComponentAsModel`
- “Component Creation”
- “AUTOSAR Component Configuration” on page 4-3
- “Component Development”
- “Code Generation”

Import AUTOSAR Package into Adaptive Component Model

Import and reference shared ARXML element definitions for the Adaptive Platform.

Add AUTOSAR Adaptive Element Definitions to Model

When developing an AUTOSAR adaptive software component in Simulink, you can import AUTOSAR element definitions that are common to many components. After you create an AUTOSAR adaptive component model, you import the definitions from AUTOSAR XML (ARXML) files that contain packages of AUTOSAR shared elements. To help implement the component behavior, you want to reference predefined elements such as service interfaces, with their associated events and namespaces, and data types.

Suppose that you are developing an AUTOSAR adaptive software component model. You want to import predefined adaptive platform type elements that are shared by multiple product lines and teams. This example uses AUTOSAR importer function `updateAUTOSARProperties` to import definitions from shared descriptions file `Adaptive_PlatformTypes.arxml` into example model `autosar_LaneGuidance`.

```
modelName = 'autosar_LaneGuidance';
open_system(modelName);
ar = arxml.importer('Adaptive_PlatformTypes.arxml');
updateAUTOSARProperties(ar,modelName);

### Updating model autosar_LaneGuidance
### Saving original model as autosar_LaneGuidance_backup.slx
### Creating HTML report autosar_LaneGuidance_update_report.html
```

The function copies the elements in the specified ARXML files to the AUTOSAR Dictionary of the specified model. If you import data types, the function also creates data objects, in a data dictionary (if available) or in the base workspace, for the imported types.

The function generates an HTML report listing the workspace changes and the element additions. Here are the Simulink workspace changes, reflecting creation of data objects to represent previously undefined adaptive platform types.

Automatic Workspace Changes

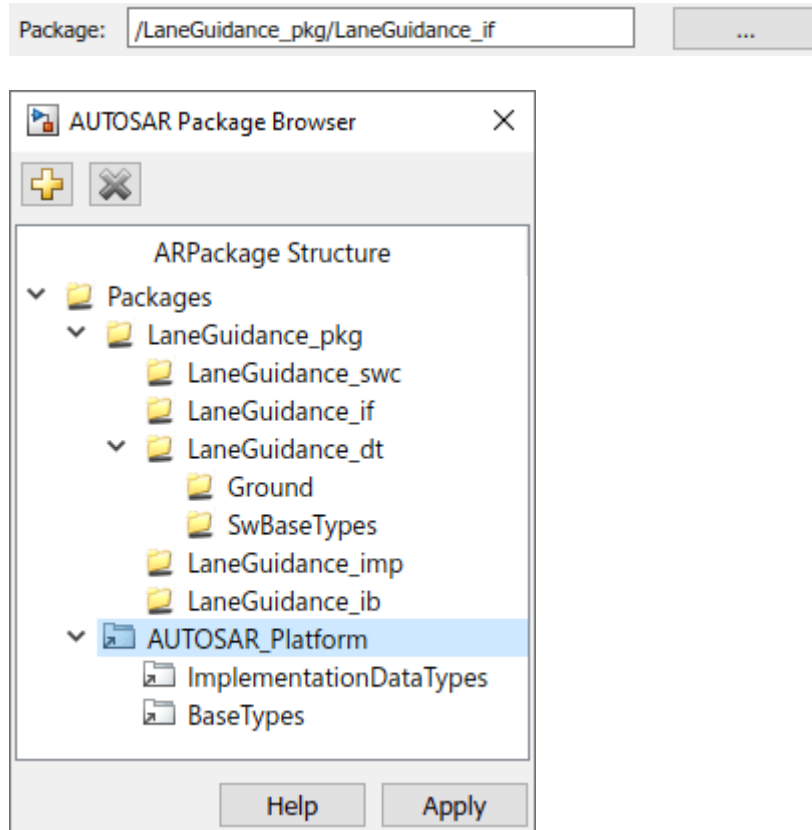
Added	Simulink.AliasType sint8_least
Added	Simulink.AliasType sint16_least
Added	Simulink.AliasType sint32_least
Added	Simulink.AliasType uint8_least
Added	Simulink.AliasType uint16_least
Added	Simulink.AliasType uint32_least

Here are the AUTOSAR element additions. Notice that the function created a new AUTOSAR package named `AUTOSAR_Platform`. Based on the imported adaptive platform types, the function populated the package with AUTOSAR software base types and AUTOSAR implementation data types.

Automatic AUTOSAR Element Changes

Added	SwBaseType /AUTOSAR_Platform/BaseTypes/sint32_least
Added	SwBaseType /AUTOSAR_Platform/BaseTypes/sint16_least
Added	SwBaseType /AUTOSAR_Platform/BaseTypes/uint8_least
Added	SwBaseType /AUTOSAR_Platform/BaseTypes/uint16_least
Added	SwBaseType /AUTOSAR_Platform/BaseTypes/uint32_least
Added	SwBaseType /AUTOSAR_Platform/BaseTypes/float32
Added	SwBaseType /AUTOSAR_Platform/BaseTypes/float64
Added	SwBaseType /AUTOSAR_Platform/BaseTypes/void
Added	SwBaseType /AUTOSAR_Platform/BaseTypes/boolean
Added	SwBaseType /AUTOSAR_Platform/BaseTypes/sint8_least
Added	SwBaseType /AUTOSAR_Platform/BaseTypes/sint16
Added	SwBaseType /AUTOSAR_Platform/BaseTypes/sint8
Added	SwBaseType /AUTOSAR_Platform/BaseTypes/sint64
Added	SwBaseType /AUTOSAR_Platform/BaseTypes/sint32
Added	SwBaseType /AUTOSAR_Platform/BaseTypes/uint8
Added	SwBaseType /AUTOSAR_Platform/BaseTypes/uint32
Added	SwBaseType /AUTOSAR_Platform/BaseTypes/uint16
Added	SwBaseType /AUTOSAR_Platform/BaseTypes/uint64
Added	Boolean /AUTOSAR_Platform/ImplementationDataTypes/boolean
Added	Integer /AUTOSAR_Platform/ImplementationDataTypes/sint32
Added	Integer /AUTOSAR_Platform/ImplementationDataTypes/sint8
Added	Integer /AUTOSAR_Platform/ImplementationDataTypes/sint16
Added	Integer /AUTOSAR_Platform/ImplementationDataTypes/uint8_least
Added	Integer /AUTOSAR_Platform/ImplementationDataTypes/sint16_least
Added	Integer /AUTOSAR_Platform/ImplementationDataTypes/sint8_least
Added	FloatingPoint /AUTOSAR_Platform/ImplementationDataTypes/float32
Added	Integer /AUTOSAR_Platform/ImplementationDataTypes/uint16
Added	Integer /AUTOSAR_Platform/ImplementationDataTypes/sint64
Added	Integer /AUTOSAR_Platform/ImplementationDataTypes/uint8
Added	FloatingPoint /AUTOSAR_Platform/ImplementationDataTypes/float64
Added	Integer /AUTOSAR_Platform/ImplementationDataTypes/sint32_least
Added	Integer /AUTOSAR_Platform/ImplementationDataTypes/uint32_least
Added	Integer /AUTOSAR_Platform/ImplementationDataTypes/uint32
Added	Integer /AUTOSAR_Platform/ImplementationDataTypes/uint16_least
Added	Integer /AUTOSAR_Platform/ImplementationDataTypes/uint64
Added	Package /AUTOSAR_Platform
Added	Package /AUTOSAR_Platform/ImplementationDataTypes
Added	Package /AUTOSAR_Platform/BaseTypes

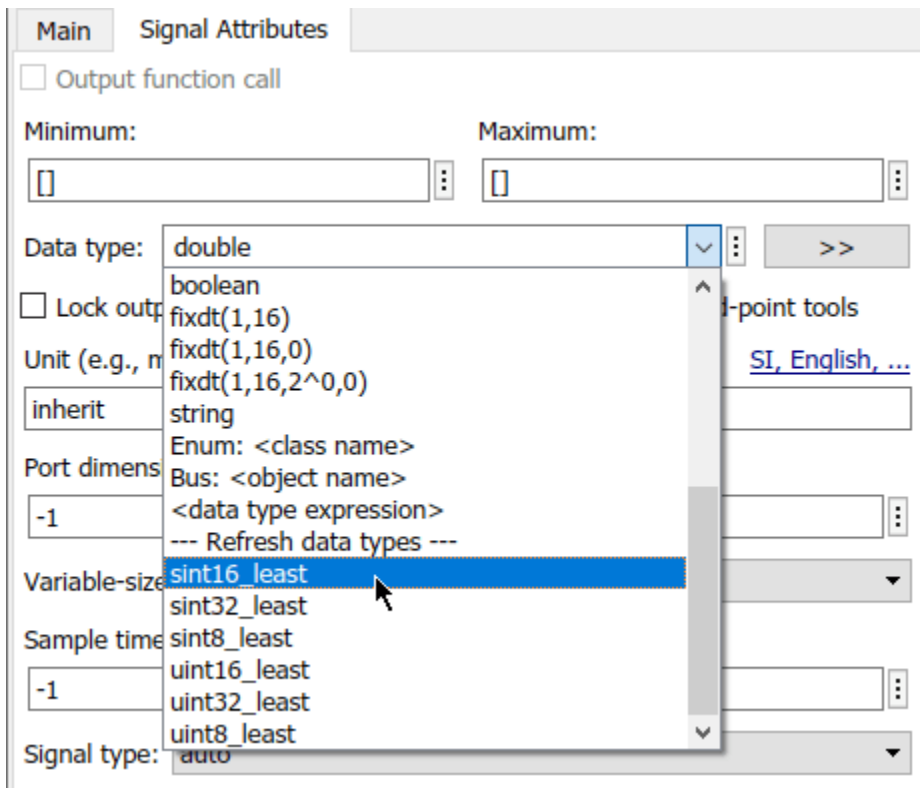
The package changes are reflected in the AUTOSAR Dictionary views of the package tree. If you open the AUTOSAR Dictionary and navigate to an individual service interface, you can click the horizontal ellipsis to the right of the **Package** field to view the current package tree.



Reference and Configure Imported AUTOSAR Adaptive Elements

After importing AUTOSAR elements into the adaptive software component model, you can reference and configure the elements in the same manner as any AUTOSAR Dictionary element.

If you imported data types, you can reference the types from your model blocks. For example, open a Simulink port block in your model and select the **Signal Attributes** tab. Expand the **Data type** list of values and notice that the imported data types are available for selection.



If you have Simulink Coder and Embedded Coder software, you can generate AUTOSAR-compliant C++ code and export ARXML descriptions from the adaptive component model. The C++ code reflects references from model blocks to the imported adaptive elements. Export preserves the file structure and content of the shared descriptions files from which you imported definitions. In ARXML files other than the shared description files, ARXML descriptions reference the shared element definitions where required.

Related Links

- `updateAUTOSARProperties`
- “Import AUTOSAR Adaptive Software Descriptions” on page 6-12
- “AUTOSAR ARXML Importer” on page 3-35

Configure AUTOSAR Adaptive Elements and Properties

In Simulink, you can use the AUTOSAR Dictionary and the Code Mappings editor separately or together to graphically configure an AUTOSAR adaptive software component and map Simulink model elements to AUTOSAR component elements. For more information, see “AUTOSAR Component Configuration” on page 4-3.

Use the AUTOSAR Dictionary to configure AUTOSAR elements from an AUTOSAR perspective. Using a tree format, the AUTOSAR Dictionary displays a mapped AUTOSAR adaptive component and its elements, communication interfaces, and XML options. Use the tree to select AUTOSAR elements and configure their properties. The properties that you modify are reflected in exported ARXML descriptions and potentially in generated AUTOSAR-compliant C++ code.

In this section...

- “AUTOSAR Elements Configuration Workflow” on page 6-21
- “Configure AUTOSAR Adaptive Software Components” on page 6-22
- “Configure AUTOSAR Adaptive Service Interfaces and Ports” on page 6-25
- “Configure AUTOSAR Adaptive Persistent Memory Interfaces and Ports” on page 6-30
- “Configure AUTOSAR Adaptive XML Options” on page 6-33


AUTOSAR Elements Configuration Workflow

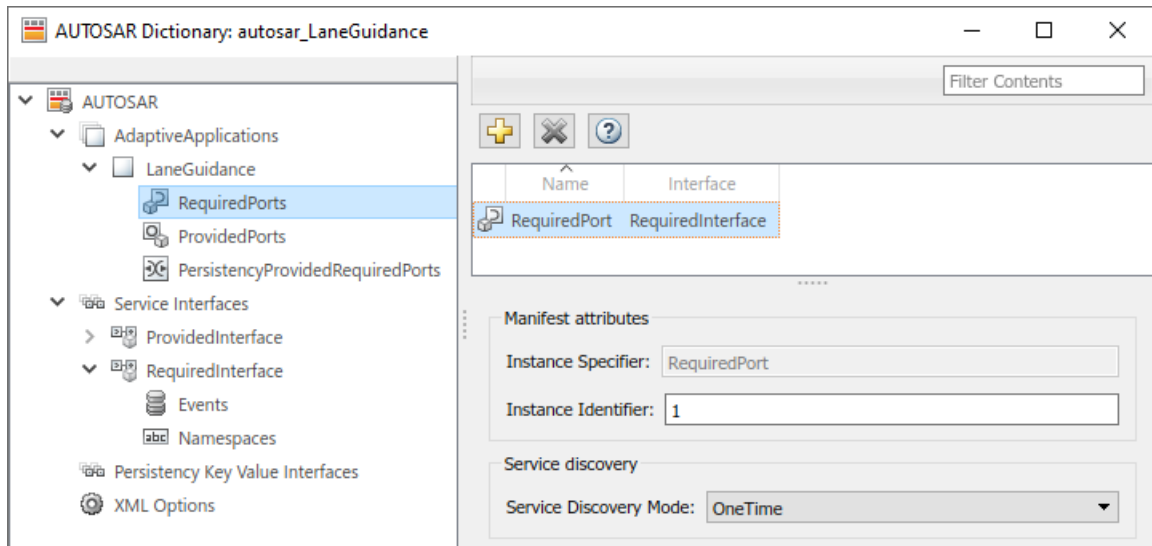
To configure AUTOSAR component elements for the Adaptive Platform in Simulink:


- 1 Open a model for which the AUTOSAR system target file `autosar_adaptive.tlc` is selected.
- 2 Create or open a mapped view of the AUTOSAR model. In the model window, do one of the following:
 - From the **Apps** tab, open the AUTOSAR Component Designer app.
 - Click the perspective control in the lower-right corner and select **Code**.

If the model has not yet been mapped to an AUTOSAR software component, the AUTOSAR Component Quick Start opens. Work through the quick-start procedure and click **Finish**. For more information, see “Create Mapped AUTOSAR Component with Quick Start” on page 3-2.

The model opens in the AUTOSAR Code perspective. This perspective displays the model and directly below the model, the Code Mappings editor.

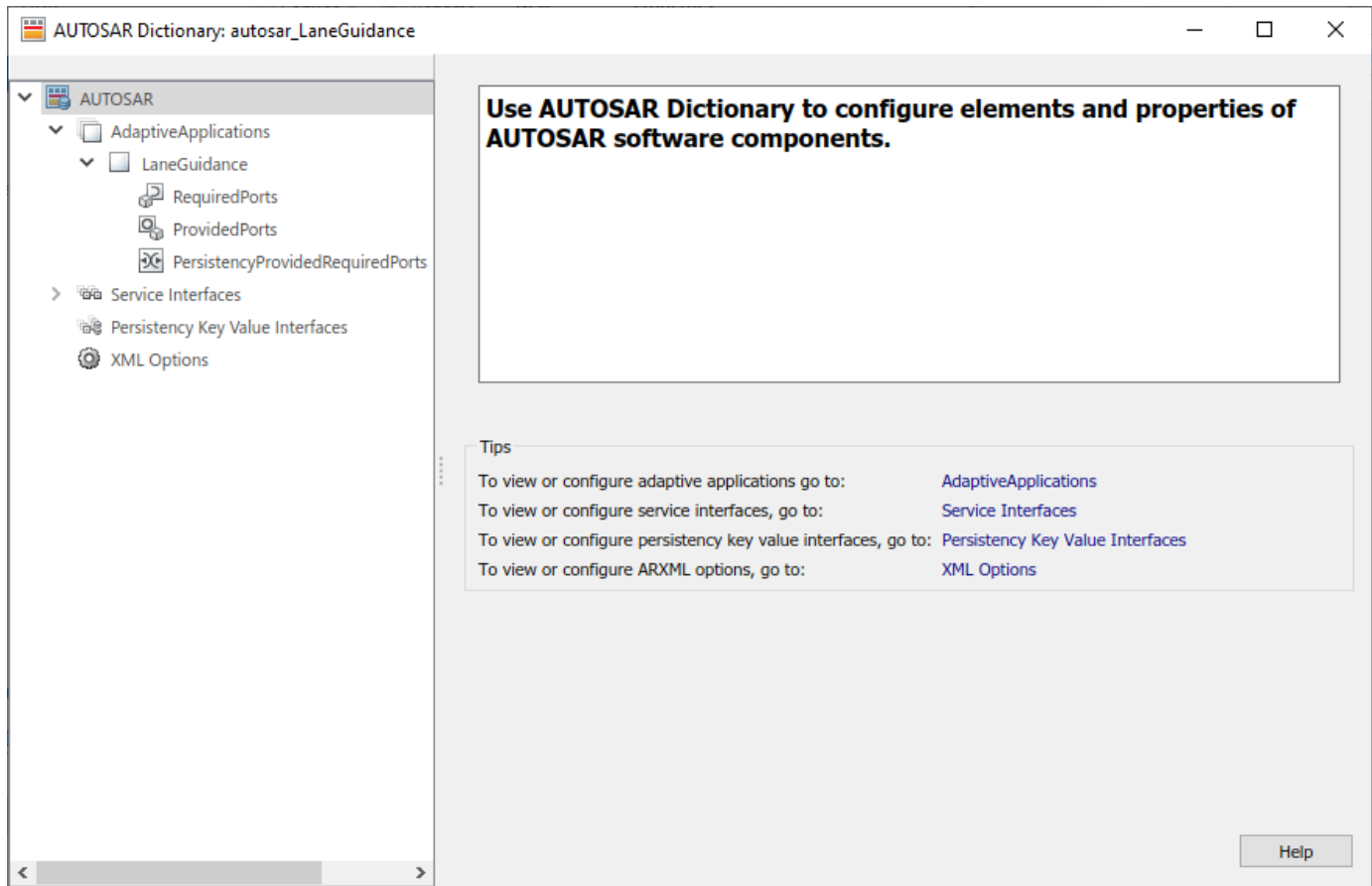
- 3 Open the AUTOSAR Dictionary. Either click the **AUTOSAR Dictionary** button  in the Code Mappings editor or, on the **AUTOSAR** tab, select **Code Interface > AUTOSAR Dictionary**.
- 4 To configure AUTOSAR elements and properties, navigate the AUTOSAR Dictionary tree. You can add elements, remove elements, or select elements to view and modify their properties. Use the **Filter Contents** field (where available) to selectively display some elements, while omitting others, in the current view.




- 5 After configuring AUTOSAR adaptive elements and properties, open the Code Mappings editor. Use Code Mapping tabs to map Simulink elements to new or modified AUTOSAR elements.
- 6 To validate the AUTOSAR component configuration, click the **Validate** button . If errors are reported, address them, and then retry validation.

Configure AUTOSAR Adaptive Software Components

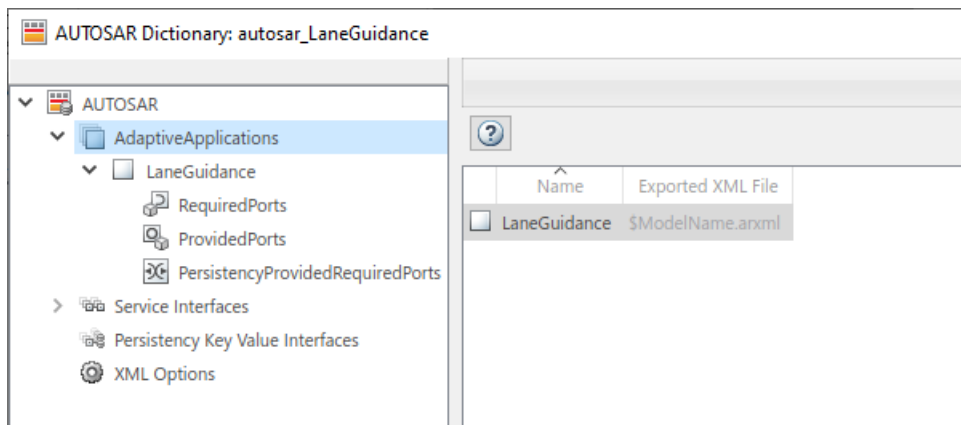
AUTOSAR adaptive software components contain AUTOSAR elements defined in the AUTOSAR standard, such as required ports and provided ports. In the AUTOSAR Dictionary, component elements appear in a tree format under the component that owns them. To access component elements and their properties, expand the component name.



To configure AUTOSAR adaptive software component elements and properties:

- 1 Open a model for which a mapped AUTOSAR adaptive software component has been created. For more information, see “Component Creation”.
- 2 From the **Apps** tab, open the AUTOSAR Component Designer app.
- 3 Open the AUTOSAR Dictionary. Either click the **AUTOSAR Dictionary** button  in the Code Mappings editor or, on the **AUTOSAR** tab, select **Code Interface > AUTOSAR Dictionary**.
- 4 In the leftmost pane of the AUTOSAR Dictionary, under **AUTOSAR**, select **AdaptiveApplications**.

The adaptive applications view in the AUTOSAR Dictionary displays adaptive software components. You can rename an AUTOSAR adaptive component by editing its name text.



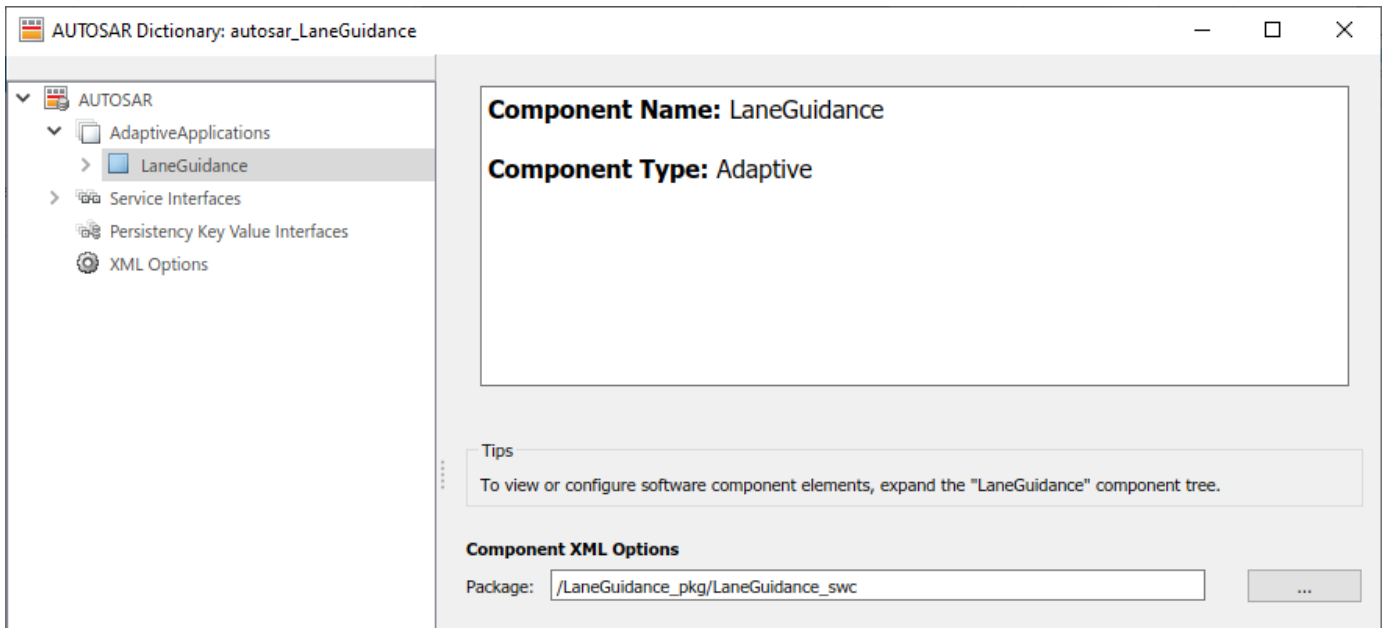
- 5 In the leftmost pane of the AUTOSAR Dictionary, expand **AdaptiveApplications** and select an AUTOSAR adaptive component.

The component view in the AUTOSAR Dictionary displays the name and type of the selected component, and component options for ARXML file export. You can modify the AUTOSAR package to be generated for the component.

To specify the AUTOSAR package path, you can do either of the following:

- Enter a package path in the **Package** parameter field. Package paths can use an organizational naming pattern, such as `/CompanyName/Powertrain`.
- To open the AUTOSAR Package Browser, click the button to the right of the **Package** field. Use the browser to navigate to an existing package or create a package. When you select a package in the browser and click **Apply**, the component **Package** parameter value is updated with your selection. For more information about the AUTOSAR Package Browser, see “Configure AUTOSAR Package for Component, Interface, CompuMethod, or SwAddrMethod” on page 4-94.

For more information about component XML options, see “Configure AUTOSAR Packages” on page 4-84.



Configure AUTOSAR Adaptive Service Interfaces and Ports

An AUTOSAR adaptive software component uses communication interfaces and ports defined in the AUTOSAR standard, including adaptive service interfaces and required and provided ports. In the AUTOSAR Dictionary, communication interfaces appear in a tree format.



- To access service interfaces and their properties, expand the **Service Interfaces** node and select an interface.
- To access required and provided ports and their properties, expand an application node and select either **RequiredPorts** or **ProvidedPorts**.

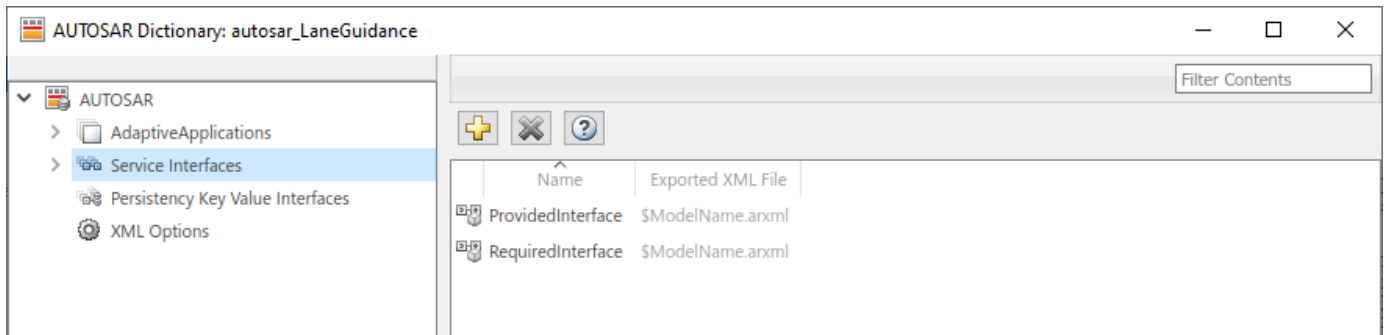
The interface and port views in the AUTOSAR Dictionary support modeling AUTOSAR adaptive service communication in Simulink. You use the AUTOSAR Dictionary to first configure AUTOSAR service interfaces, events, and C++ namespaces, and then configure required and provided ports. For more information, see “Model AUTOSAR Adaptive Service Communication” on page 6-50.

To configure AUTOSAR service interface elements and properties, open a model for which a mapped AUTOSAR adaptive software component has been created and open the AUTOSAR Dictionary.

- 1 In the leftmost pane of the AUTOSAR Dictionary, select **Service Interfaces**.

The service interfaces view in the AUTOSAR Dictionary lists AUTOSAR service interfaces and their properties. You can:

- Select a service interface and rename it by editing its name text.
- To add a service interface, click the **Add** button  and use the Add Interfaces dialog box. Specify an interface name, the number of events it contains, and the path of the Interface package.
- To remove a service interface, select the interface and then click the **Delete** button .

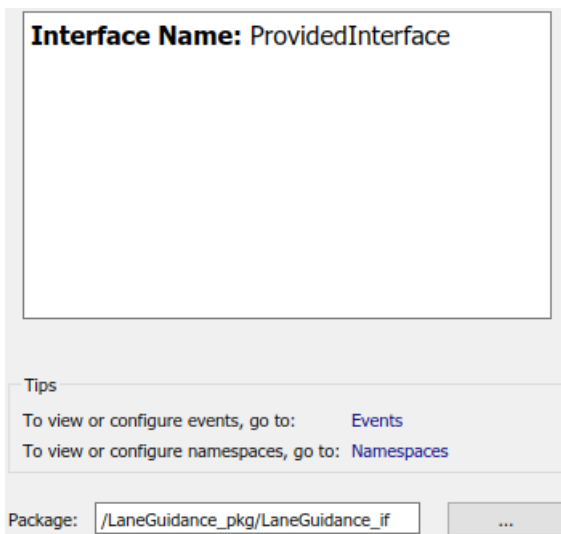


- 2 In the leftmost pane of the AUTOSAR Dictionary, expand **Service Interfaces** and select a service interface from the list.

The service interface view in the AUTOSAR Dictionary displays the name of the selected service interface and the AUTOSAR package to be generated for the interface.

To modify the AUTOSAR package for the interface, you can do either of the following:



- Enter a package path in the **Package** parameter field.
- To open the AUTOSAR Package Browser, click the button to the right of the **Package** field. Use the browser to navigate to an existing package or create a package. When you select a package in the browser and click **Apply**, the interface **Package** parameter value is updated with your selection. For more information about the AUTOSAR Package Browser, see “Configure AUTOSAR Package for Component, Interface, CompuMethod, or SwAddrMethod” on page 4-94.

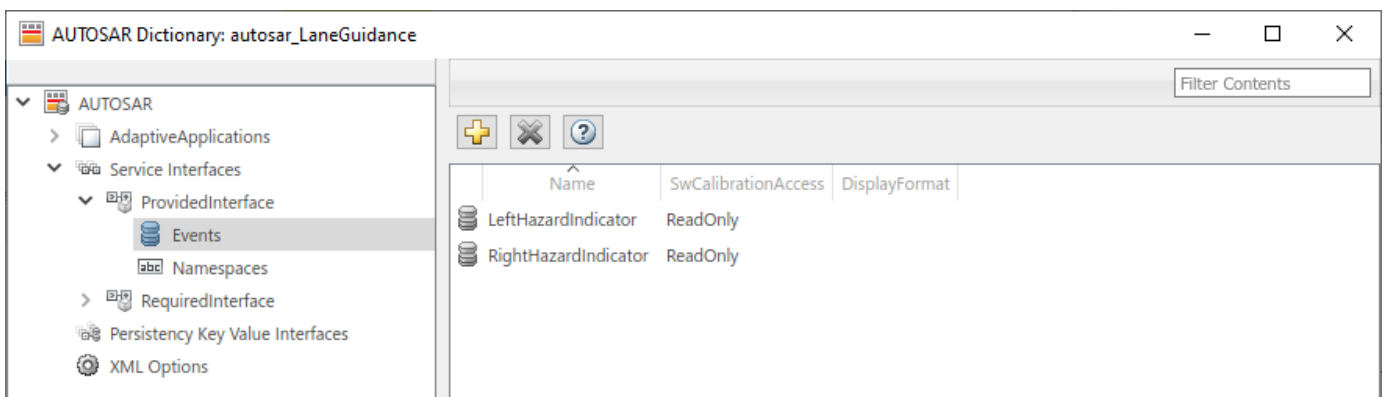


- 3 In the leftmost pane of the AUTOSAR Dictionary, expand the selected interface and select **Events**.

The events view in the AUTOSAR Dictionary lists AUTOSAR service interface events and their properties. You can:



- Select a service interface event and edit the name value.

- Specify the level of calibration and measurement tool access to service interface events. Select an event and set its **SwCalibrationAccess** value to `ReadOnly`, `ReadWrite`, or `NotAccessible`.
- Optionally specify the format to be used by calibration and measurement tools to display the event. In the **DisplayFormat** field, enter an ANSI C `printf` format specifier string. For example, `%2.1d` specifies a signed decimal number. The number has a minimum width of two characters and a maximum precision of one digit, producing a displayed value such as 12.2. For more information about constructing a format specifier string, see “Configure DisplayFormat” on page 4-264.
- To add an event, click the **Add** button .
- To remove an event, select the event and then click the **Delete** button .

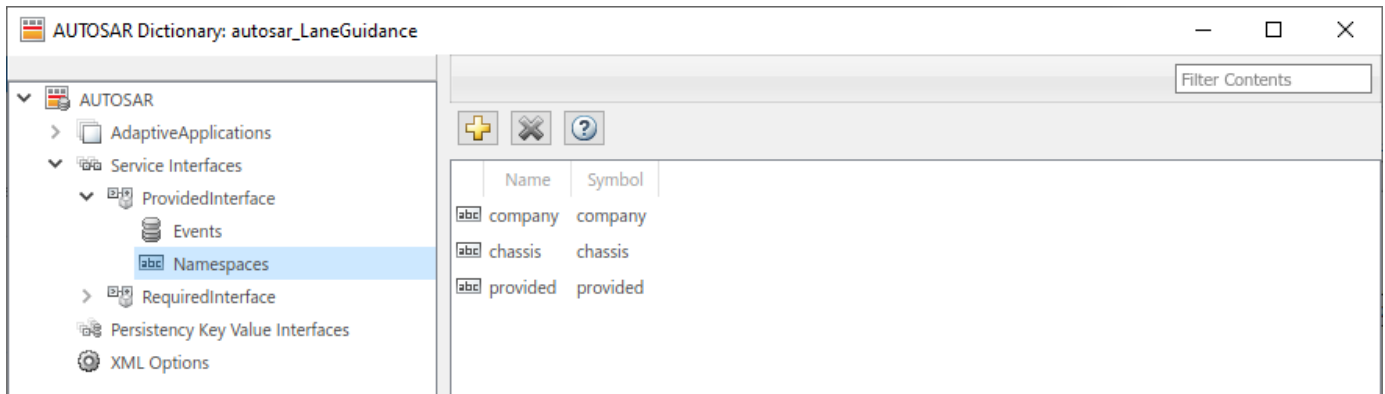


- 4 In the leftmost pane of the AUTOSAR Dictionary, below **Events**, select **Namespaces**.

The namespaces view in the AUTOSAR Dictionary enables you to define a unique namespace for each service interface. The code generator uses the defined namespace when producing C++ code for the interface. To modify or construct a namespace specification, you can:

- Select a namespace element and edit the name value.
- To add a namespace element to the namespace specification, click the **Add** button .
- To remove a namespace element, select the element and then click the **Delete** button .



For example, this namespaces view defines namespace `company::chassis::provided` for service interface `ProvidedInterface`.

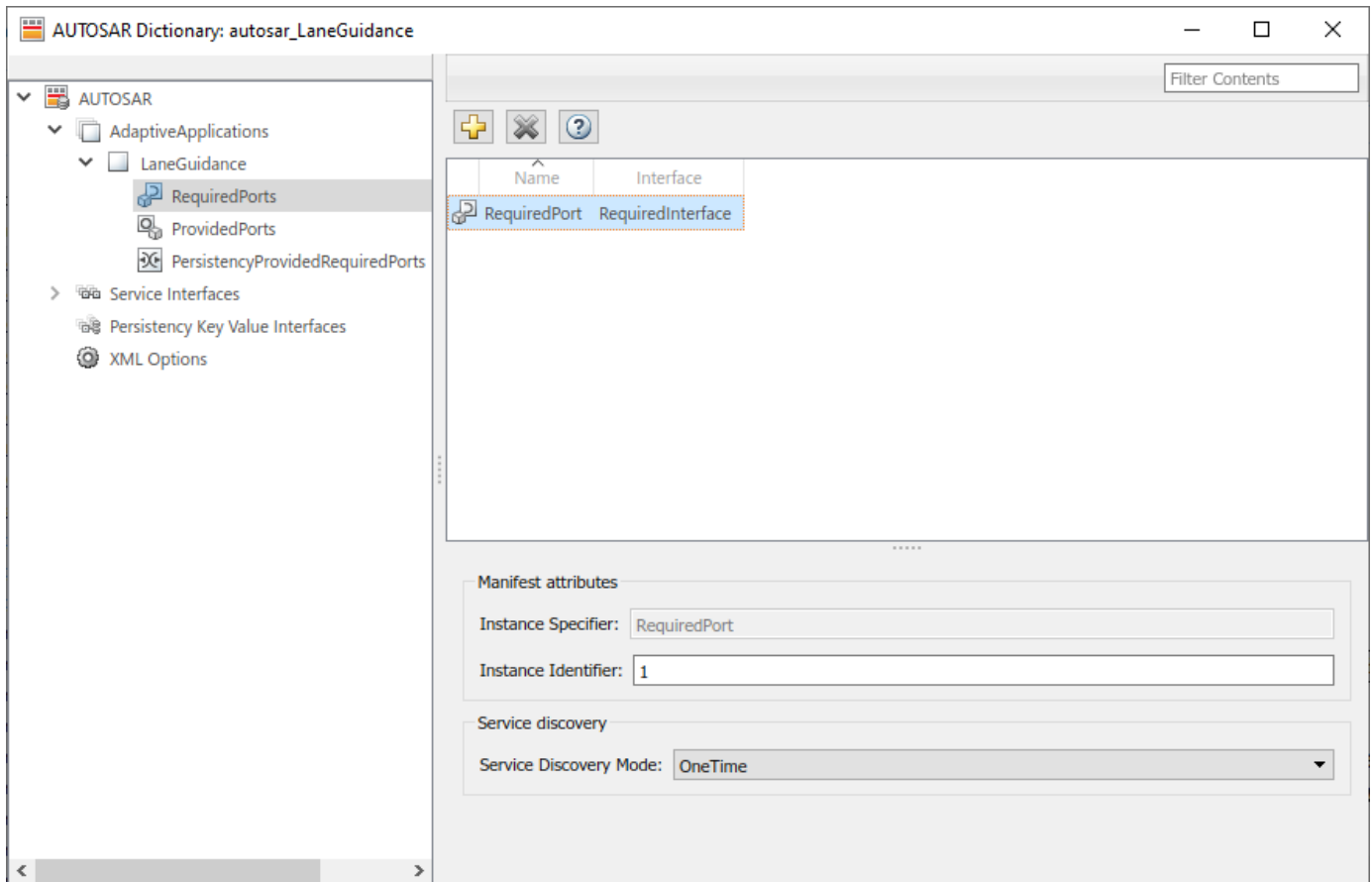


To configure AUTOSAR required and provided port elements, open a model for which a mapped AUTOSAR adaptive software component has been created and open the AUTOSAR Dictionary.

- 1 In the leftmost pane of the AUTOSAR Dictionary, expand the component name and select **RequiredPorts**.



The required ports view in the AUTOSAR Dictionary lists required ports and their properties. You can:

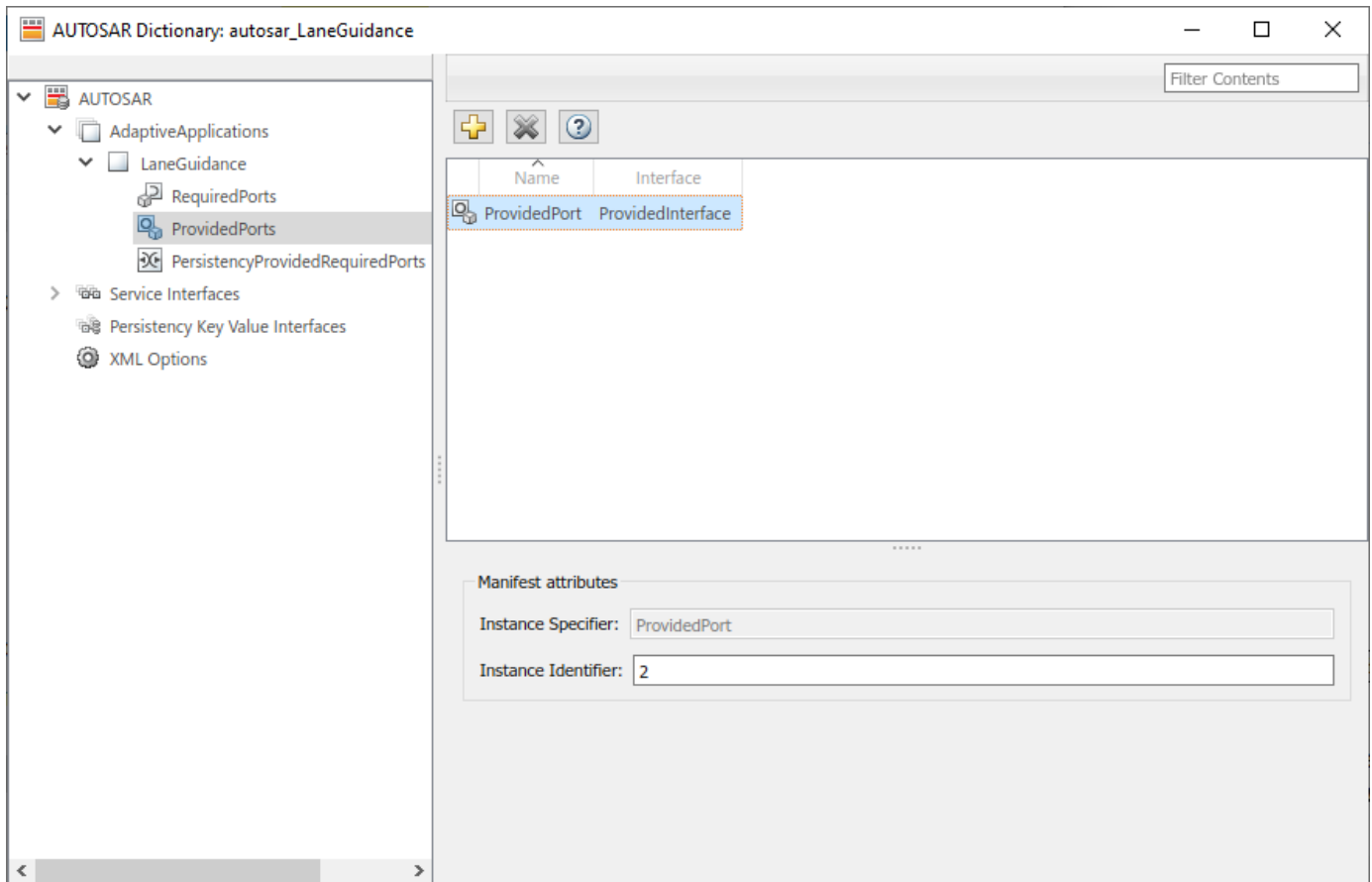
- Select an AUTOSAR required port, and view and optionally reselect its associated service interface.
- Rename a required port by editing its name text.
- To configure adaptive service instance identification for a required port, select the port and view its **Manifest attributes**. Based on the service instance form selected in XML options, examine the value for **Instance Specifier** or **Instance Identifier**. You can enter a value or accept an existing value. For more information, see “Configure AUTOSAR Adaptive Service Instance Identification” on page 6-64.
- To configure the adaptive service discovery behavior for a required port, select the port and view its **Service Discovery Mode**. You can select mode **OneTime** or **DynamicDiscovery**. For more information, see “Configure AUTOSAR Adaptive Service Discovery Modes” on page 6-62.
- To add a required port, click the **Add** button  and use the Add Ports dialog box. Specify a port name and associate it with an existing service interface.
- To remove a required port, select the port and then click the **Delete** button .



- 2 In the leftmost pane of the AUTOSAR Dictionary, select **ProvidedPorts**.

The provided ports view in the AUTOSAR Dictionary lists provided ports and their properties. You can:

- Select an AUTOSAR provided port, and view and optionally reselect its associated service interface.
- Rename a provided port by editing its name text.
- To configure adaptive service instance identification for a provided port, select the port and view its **Manifest attributes**. Based on the service instance form selected in XML options, examine the value for **Instance Specifier** or **Instance Identifier**. You can enter a value or accept an existing value. For more information, see “Configure AUTOSAR Adaptive Service Instance Identification” on page 6-64.
- To add a provided port, click the **Add** button  and use the Add Ports dialog box. Specify a port name and associate it with an existing service interface.
- To remove a provided port, select the port and then click the **Delete** button .



To map Simulink root inports and outports to AUTOSAR required and provided service ports and service interface events, see “Map Inports and Outports to AUTOSAR Service Ports and Events” on page 6-39.

Configure AUTOSAR Adaptive Persistent Memory Interfaces and Ports

An AUTOSAR adaptive software component uses communication interfaces and ports defined in the AUTOSAR standard, including adaptive persistency key value interfaces and persistency provided-required ports. In the AUTOSAR Dictionary, interfaces and ports appear in a tree format.



- To access persistent memory interfaces and their properties, expand the **Persistency Key Value Interfaces** node and select an interface.
- To access persistent memory ports and their properties, expand an application node and select **PersistencyProvidedRequiredPorts**.

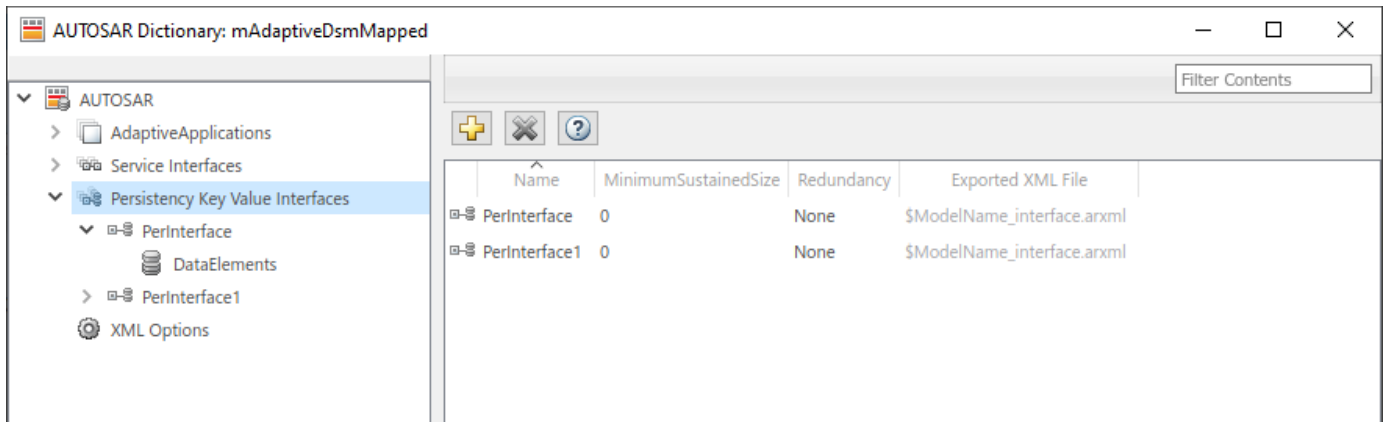
The interface and port views in the AUTOSAR Dictionary support modeling AUTOSAR adaptive persistent memory in Simulink. You use the AUTOSAR Dictionary to first configure AUTOSAR persistency key value interfaces and data elements, and then configure persistency provided-required ports. For more information, see “Model AUTOSAR Adaptive Persistent Memory” on page 6-66.

To configure AUTOSAR adaptive persistency key value interfaces, open a model for which a mapped AUTOSAR adaptive software component has been created and open the AUTOSAR Dictionary.

- 1 In the leftmost pane of the AUTOSAR Dictionary, select **Persistency Key Value Interfaces**.

This view in the AUTOSAR Dictionary lists AUTOSAR persistency key value interfaces and their properties. You can:

- Select a persistency interface and rename it by editing its name text.
- To add a persistency interface, click the **Add** button  and use the Add Interfaces dialog box. Specify an interface name, the number of data elements it contains, and the path of the Interface package.
- To remove a persistency interface, select the interface and then click the **Delete** button .

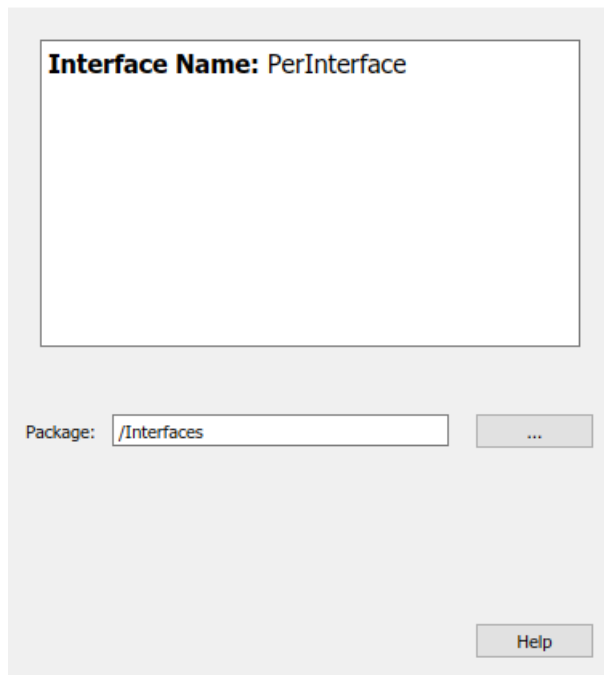


- 2 In the leftmost pane of the AUTOSAR Dictionary, expand **Persistency Key Value Interfaces** and select a persistency interface from the list.

The persistency interface view in the AUTOSAR Dictionary displays the name of the selected persistency interface and the AUTOSAR package to be generated for the interface.



To modify the AUTOSAR package for the interface, you can do either of the following:

- Enter a package path in the **Package** parameter field.
- To open the AUTOSAR Package Browser, click the button to the right of the **Package** field. Use the browser to navigate to an existing package or create a package. When you select a package in the browser and click **Apply**, the interface **Package** parameter value is updated with your selection. For more information about the AUTOSAR Package Browser, see “Configure AUTOSAR Package for Component, Interface, CompuMethod, or SwAddrMethod” on page 4-94.





- 3 In the leftmost pane of the AUTOSAR Dictionary, expand the selected interface and select **DataElements**.

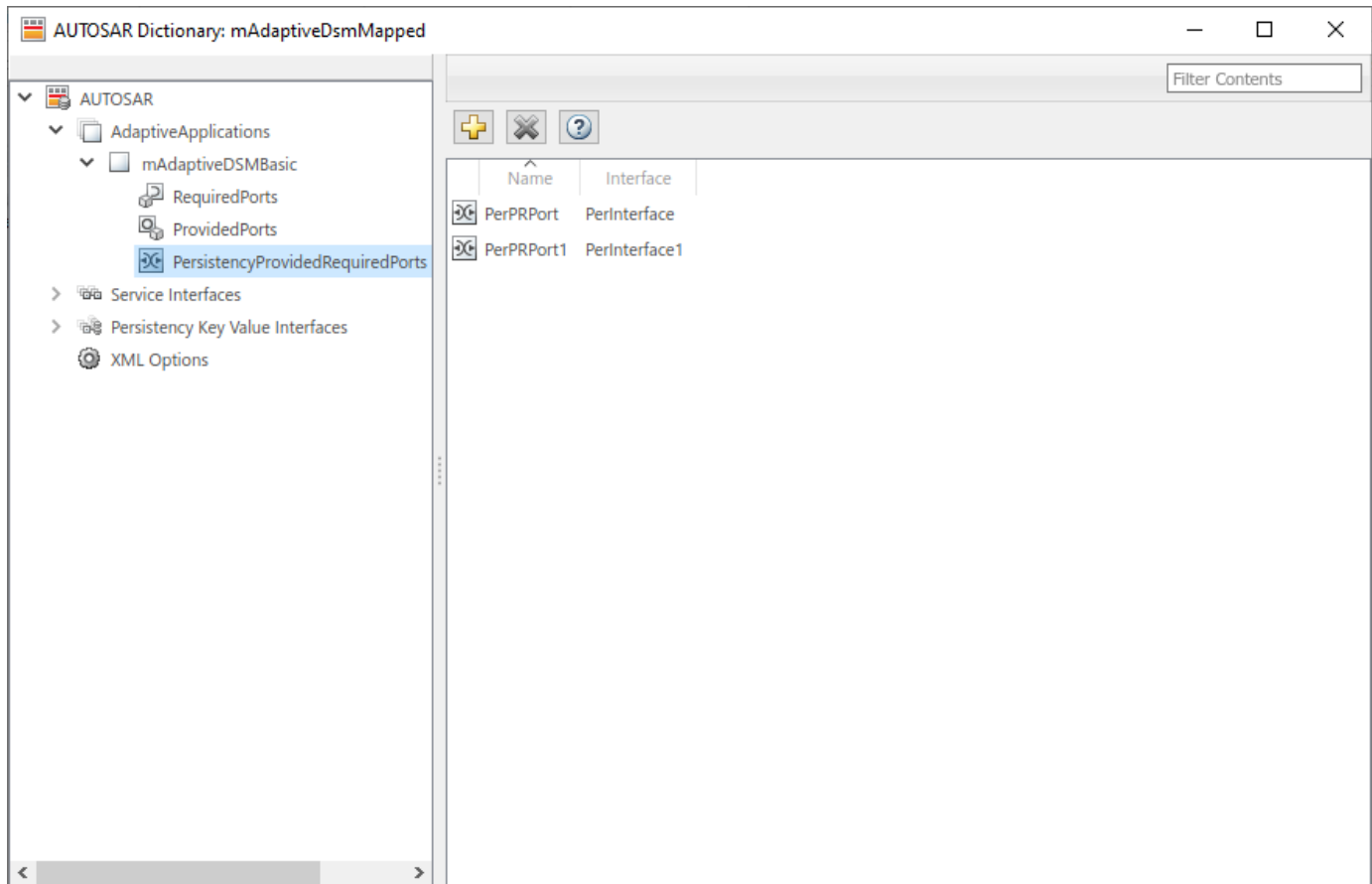
The data elements view in the AUTOSAR Dictionary lists AUTOSAR persistency interface data elements and their properties. You can:

- Select a persistency interface data element and edit the name value.
- To add a data element, click the **Add** button .
- To remove a data element, select the data element and then click the **Delete** button .

To configure AUTOSAR adaptive persistency provided-required port elements, open a model for which a mapped AUTOSAR adaptive software component has been created and open the AUTOSAR Dictionary.

In the leftmost pane of the AUTOSAR Dictionary, select **PersistencyProvidedRequiredPorts**. This view in the AUTOSAR Dictionary lists AUTOSAR persistency provided-required ports and their properties. You can:

- Select an AUTOSAR persistency provided-required port and select or modify its associated persistency key value interface.
- Rename a persistency port by editing its name text.
- To add a persistency port, click the **Add** button  and use the Add Ports dialog box. Specify a port name and associate it with an existing persistency key value interface.
- To remove a persistency port, select the port and then click the **Delete** button .



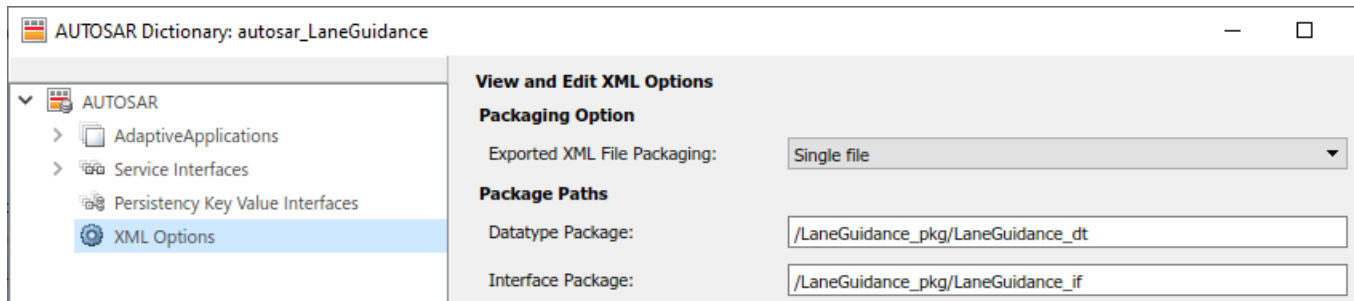
To map Simulink data stores to AUTOSAR persistency provided-required ports and key value interface data elements, see “Map Data Stores to AUTOSAR Persistent Memory Ports and Data Elements” on page 6-39.

Configure AUTOSAR Adaptive XML Options

To configure AUTOSAR adaptive XML options for ARXML export, open a model for which a mapped AUTOSAR adaptive software component has been created and open the AUTOSAR Dictionary. Select **XML Options**.

The XML options view in the AUTOSAR Dictionary displays XML export parameters and their values. You can configure:

- XML file packaging for AUTOSAR elements created in Simulink
- AUTOSAR package paths
- Aspects of exported AUTOSAR XML content



- “Exported XML File Packaging” on page 6-34
- “AUTOSAR Package Paths” on page 6-35
- “Additional XML Options” on page 6-35

Exported XML File Packaging

In the XML options view, you can specify the granularity of XML file packaging for AUTOSAR elements created in Simulink. Imported AUTOSAR XML files retain their file structure, as described in “Round-Trip Preservation of AUTOSAR XML File Structure and Element Information” on page 3-37. Select one of the following values for **Exported XML file packaging**.

- **Single file** — Exports XML into a single file, *modelname.arxml*.
- **Modular** — Exports XML into multiple files, named according to the type of information contained.

Exported File Name	File Contents
<i>modelname_component.arxml</i>	Adaptive software components, including required and provided ports. This file is the main ARXML file exported for the Simulink model. In addition to software components, the component file contains packageable elements that the exporter does not move to data type or interface files based on AUTOSAR element category.
<i>modelname_datatype.arxml</i>	Data types and related elements, including: <ul style="list-style-type: none"> • Application data types • Standard Cpp implementation data types • Constant specifications • Physical data constraints • Units and unit groups • Software record layouts
<i>modelname_interface.arxml</i>	Adaptive interfaces, including required and provided service interfaces with namespaces and events.

Alternatively, you can programmatically configure exported XML file packaging by calling the AUTOSAR `set` function. For property `ArxmlFilePackaging`, specify either `SingleFile` or `Modular`. For example:

```
arProps = autosar.api.getAUTOSARProperties(hModel);
set(arProps, 'XmlOptions', 'ArxmlFilePackaging', 'SingleFile');
```


For the Adaptive Platform, model builds also generate XML manifests for AUTOSAR executables and service instances. For more information, see “Generate AUTOSAR Adaptive C++ and XML Files” on page 6-77.

AUTOSAR Package Paths

In the XML options view, you can configure AUTOSAR packages (AR-PACKAGEs), which contain groups of AUTOSAR elements and reside in a hierarchical AR-PACKAGE structure. The AR-PACKAGE structure for a component is logically distinct from the ARXML file partitioning selected with the XML option **Exported XML file packaging** or imported from AUTOSAR XML files. For more information about AUTOSAR packages, see “Configure AUTOSAR Packages” on page 4-84.

Inspect and modify the AUTOSAR package paths grouped under the headings **Package Paths** and **Additional Packages**.

Package Paths	
Datatype Package:	<input type="text" value="/LaneGuidance_pkg/LaneGuidance_dt"/>
Interface Package:	<input type="text" value="/LaneGuidance_pkg/LaneGuidance_if"/>
Additional Packages	
ApplicationDataType Package:	<input type="text"/>
SwBaseType Package:	<input type="text"/>
ConstantSpecification Package:	<input type="text" value="/LaneGuidance_pkg/LaneGuidance_dt/Ground"/>
Physical DataConstraints Package:	<input type="text" value="/LaneGuidance_pkg/LaneGuidance_dt/AppIDataTypes/DataConstrs"/>
Unit Package:	<input type="text" value="/LaneGuidance_pkg/LaneGuidance_dt"/>
SwRecordLayout Package:	<input type="text"/>
Internal DataConstraints Package:	<input type="text"/>

Alternatively, you can programmatically configure an AUTOSAR package path by calling the AUTOSAR `set` function. Specify a package property name and a package path. For example:

```
arProps = autosar.api.getAUTOSARProperties(hModel);
set(arProps, 'XmlOptions', 'ApplicationDataTypePackage', ...
    '/Company/Powertrain/DataTypes/AppIDataTypes');
```

For more information about AUTOSAR package property names and defaults, see “Configure AUTOSAR Packages and Paths” on page 4-85.

Additional XML Options

In the XML options view, under the heading **Additional Options**, you can configure aspects of exported AUTOSAR XML content.

Additional Options	
ImplementationDataType Reference:	<input type="text" value="Allowed"/>
SwCalibrationAccess DefaultValue:	<input type="text" value="ReadWrite"/>
Internal DataConstraints Export:	<input type="checkbox"/>
Identify Service Instance Using:	<input type="text" value="InstanceIdentifier"/>

You can:

- Optionally override the default behavior for generating AUTOSAR application data types in ARXML code. To force generation of an application data type for each AUTOSAR data type, change the value of **ImplementationDataType Reference** from **Allowed** to **NotAllowed**. For more information, see “Control Application Data Type Generation” on page 4-244.
- Control the default value of the **SwCalibrationAccess** property of generated AUTOSAR measurement variables, calibration parameters, and signal and parameter data objects. For **SwCalibrationAccess DefaultValue**, select one of the following values:
 - **ReadOnly** — Read access only.
 - **ReadWrite** (default) — Read and write access.
 - **NotAccessible** — Not accessible with calibration and measurement tools.

For more information, see “Configure SwCalibrationAccess” on page 4-262.

- Optionally override the default behavior for generating internal data constraint information for AUTOSAR implementation data types in ARXML code. To force export of internal data constraints for implementation data types, select the option **Internal DataConstraints Export**. For more information, see “Configure AUTOSAR Internal Data Constraints Export” on page 4-246.
- Specify the form in which to generate adaptive service instance information. Set **Identify Service Instance Using** to **InstanceIdentifier** or **InstanceSpecifier**. The form that you select is used to identify service instances in generated Proxy and Skeleton functions. For more information, see “Configure AUTOSAR Adaptive Service Instance Identification” on page 6-64.

Alternatively, you can programmatically configure the additional XML options by calling the AUTOSAR `set` function. Specify a property name and value. The valid property names are `ImplementationTypeReference`, `SwCalibrationAccessDefault`, `InternalDataConstraintExport`, and `IdentifyServiceInstance`. For example:

```
arProps = autosar.api.getAUTOSARProperties(hModel);
set(arProps, 'XmlOptions', 'ImplementationTypeReference', 'NotAllowed');
set(arProps, 'XmlOptions', 'SwCalibrationAccessDefault', 'ReadOnly');
set(arProps, 'XmlOptions', 'InternalDataConstraintExport', true);
set(arProps, 'XmlOptions', 'IdentifyServiceInstance', 'InstanceSpecifier')
```

See Also

Related Examples

- “Map AUTOSAR Adaptive Elements for Code Generation” on page 6-37
- “Configure and Map AUTOSAR Component Programmatically” on page 4-293
- “Configure AUTOSAR Adaptive Code Generation” on page 6-73

More About

- “AUTOSAR Component Configuration” on page 4-3

Map AUTOSAR Adaptive Elements for Code Generation

In Simulink, you can use the Code Mappings editor and the AUTOSAR Dictionary separately or together to graphically configure an AUTOSAR adaptive software component and map Simulink model elements to AUTOSAR component elements. For more information, see “AUTOSAR Component Configuration” on page 4-3.

Use the Code Mappings editor to map Simulink model elements to AUTOSAR component elements from a Simulink model perspective. The editor display consists of tabbed tables, including **Inports** and **Outports**. Use the tables to select Simulink elements and map them to corresponding AUTOSAR elements. The mappings that you configure are reflected in generated AUTOSAR-compliant C++ code and exported ARXML descriptions.

In this section...

“Simulink to AUTOSAR Mapping Workflow” on page 6-37

“Map Inports and Outports to AUTOSAR Service Ports and Events” on page 6-39

“Map Data Stores to AUTOSAR Persistent Memory Ports and Data Elements” on page 6-39

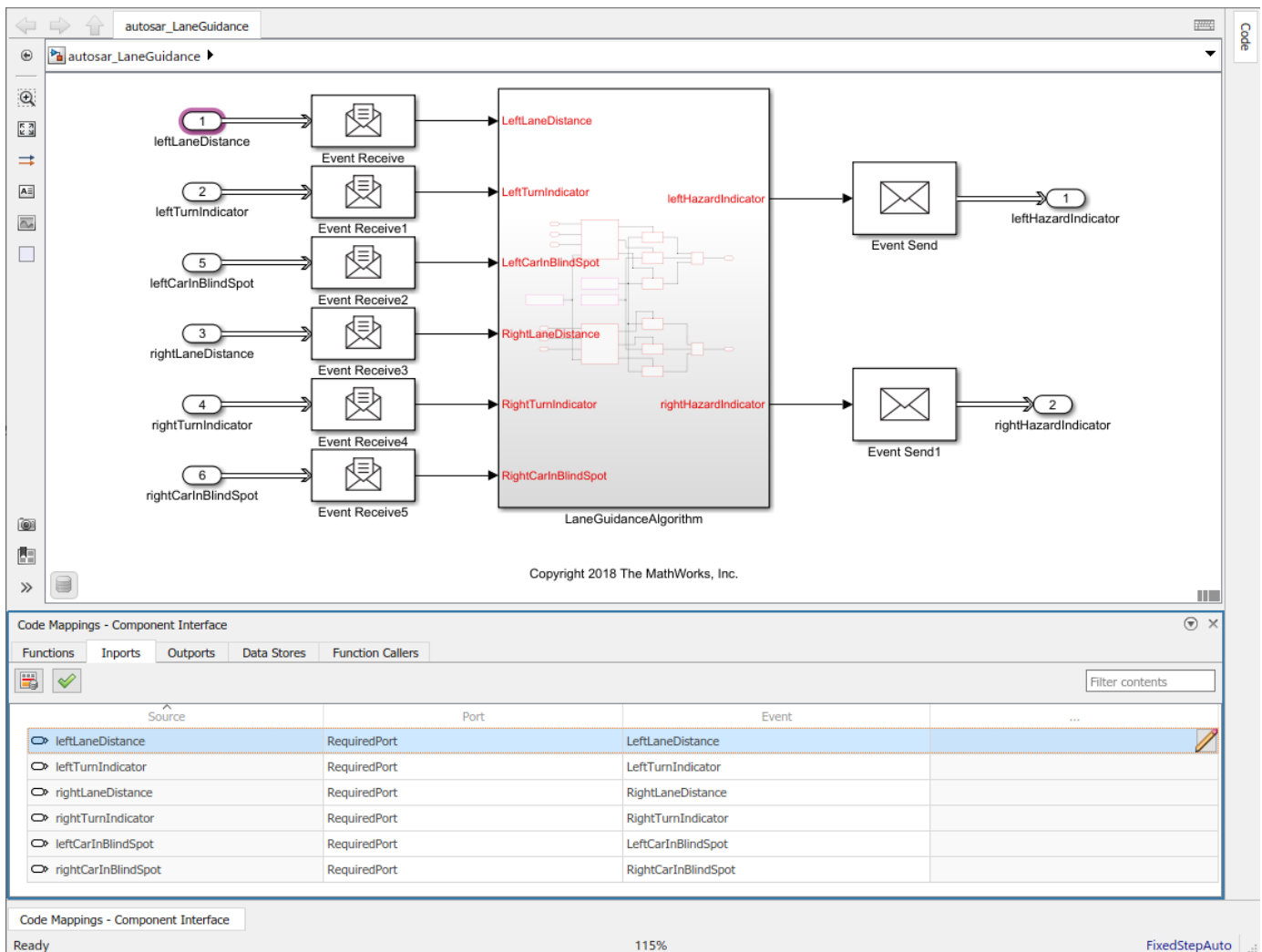
Simulink to AUTOSAR Mapping Workflow

To map Simulink model elements to AUTOSAR adaptive software component elements:

- 1 Open a model for which AUTOSAR system target file `autosar_adaptive.tlc` is selected.
- 2 Create or open a mapped view of the AUTOSAR model. In the model window, do one of the following:
 - From the **Apps** tab, open the AUTOSAR Component Designer app.
 - Click the perspective control in the lower-right corner and select **Code**.

If the model has not yet been mapped to an AUTOSAR software component, the AUTOSAR Component Quick Start opens. To configure the model for AUTOSAR component development, work through the quick-start procedure and click **Finish**. For more information, see “Create Mapped AUTOSAR Component with Quick Start” on page 3-2.

The model opens in the AUTOSAR Code perspective. This perspective displays the model and directly below the model, the Code Mappings editor.




The Code Mappings editor provides in-canvas access to AUTOSAR mapping information, with batch editing, element filtering, easy navigation to model elements and AUTOSAR properties, and model element traceability.

3 Navigate the Code Mappings editor tabs to perform these actions:

- Map a Simulink inport or outport to an AUTOSAR required or provided port and a service interface event.
- Map a Simulink data store to an AUTOSAR persistency provided-required port and a key value interface data element.

Use the **Filter contents** field (where available) to selectively display some elements, while omitting others, in the current view.

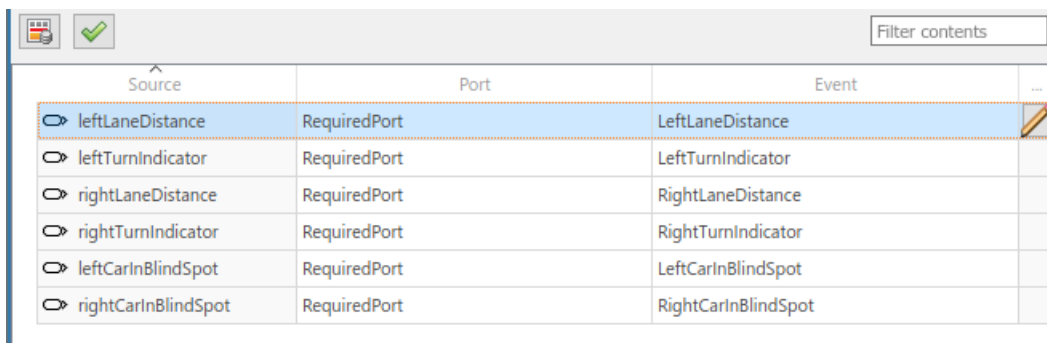
4 After mapping model elements, click the **Validate** button  to validate the AUTOSAR component configuration. If errors are reported, address them, and then retry validation.

Map Inports and Outports to AUTOSAR Service Ports and Events

The **Inports** and **Outports** tabs of the Code Mappings editor support modeling AUTOSAR service interface communication in Simulink. After using the AUTOSAR Dictionary to create AUTOSAR required and provided service ports, service interfaces, and service interface events in your model, open the Code Mappings editor. Use the **Inports** and **Outports** tabs to map Simulink root inports and outports to AUTOSAR required and provided service ports and service interface events.

For more information, see “Model AUTOSAR Adaptive Service Communication” on page 6-50.

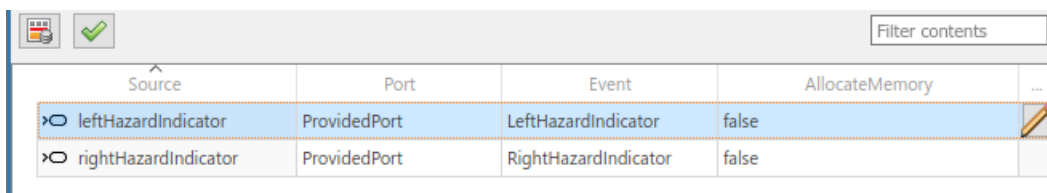
The **Inports** tab of the Code Mappings editor maps each Simulink root inport to an AUTOSAR required port and a service interface event. To map a Simulink inport, select the inport, and then select menu values for an AUTOSAR port and an AUTOSAR event among values listed for the component.



Source	Port	Event	...
leftLaneDistance	RequiredPort	LeftLaneDistance	
leftTurnIndicator	RequiredPort	LeftTurnIndicator	
rightLaneDistance	RequiredPort	RightLaneDistance	
rightTurnIndicator	RequiredPort	RightTurnIndicator	
leftCarInBlindSpot	RequiredPort	LeftCarInBlindSpot	
rightCarInBlindSpot	RequiredPort	RightCarInBlindSpot	

The **Outports** tab of the Code Mappings editor maps each Simulink root outport to an AUTOSAR provided port and a service interface event. In the **Outports** tab, you can:

- Map a Simulink outport by selecting the outport, and then selecting menu values for an AUTOSAR port and an AUTOSAR event among values listed for the component.
- Use the code attribute `AllocateMemory` to configure memory allocation for AUTOSAR adaptive service data. Specify whether to send event data by reference (the default) or by `ara::com` allocated memory. To send event data by `ara::com` allocated memory, select the value `true`. To send event data by reference, select `false`. For more information, see “Configure Memory Allocation for AUTOSAR Adaptive Service Data” on page 6-60.



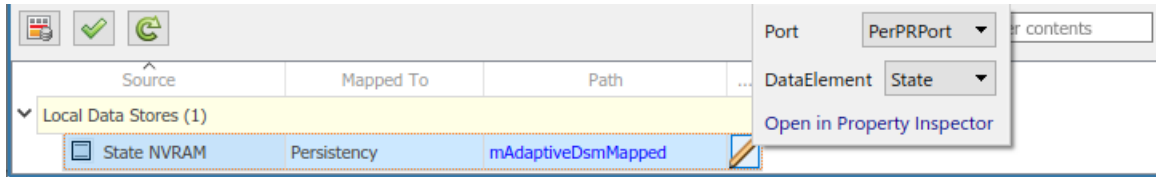
Source	Port	Event	AllocateMemory	...
leftHazardIndicator	ProvidedPort	LeftHazardIndicator	false	
rightHazardIndicator	ProvidedPort	RightHazardIndicator	false	


Map Data Stores to AUTOSAR Persistent Memory Ports and Data Elements

The **Data Stores** tab of the Code Mappings editor supports modeling AUTOSAR persistent memory in Simulink. After using the AUTOSAR Dictionary to create AUTOSAR persistency provided-required ports, persistency key value interfaces, and key value interface data elements, open the Code Mappings editor. Use the **Data Stores** tab to map Simulink data stores to AUTOSAR persistency provided-required ports and key value interface data elements.

For more information, see “Model AUTOSAR Adaptive Persistent Memory” on page 6-66.

To map a Simulink data store, select a data store in the **Data Stores** tab and, in the **Mapped To** menu, select **Persistency**. By default the data stores are mapped to **Auto**.



To configure the AUTOSAR persistency provided-required port and key value interface data element for the mapped data store, click the  icon. A properties dialog box opens. Select menu values for **Port** and **Data Element**.

Attribute	Purpose
Port	Select the name of a persistency provided-required port configured in the AUTOSAR Dictionary.
DataElement	Select the name of a persistency key value interface data element configured in the AUTOSAR Dictionary.

See Also

Related Examples

- “Model AUTOSAR Adaptive Service Communication” on page 6-50
- “Configure Memory Allocation for AUTOSAR Adaptive Service Data” on page 6-60
- “Model AUTOSAR Adaptive Persistent Memory” on page 6-66
- “Configure AUTOSAR Adaptive Elements and Properties” on page 6-21
- “Configure and Map AUTOSAR Component Programmatically” on page 4-293

More About

- “AUTOSAR Component Configuration” on page 4-3
- “Code Generation”

Configure AUTOSAR Adaptive Software Components

In Simulink, you can flexibly model the structure and behavior of software components for the AUTOSAR Adaptive Platform. The AUTOSAR Adaptive Platform defines a service-oriented architecture for automotive components that must flexibly adapt to external events and conditions.

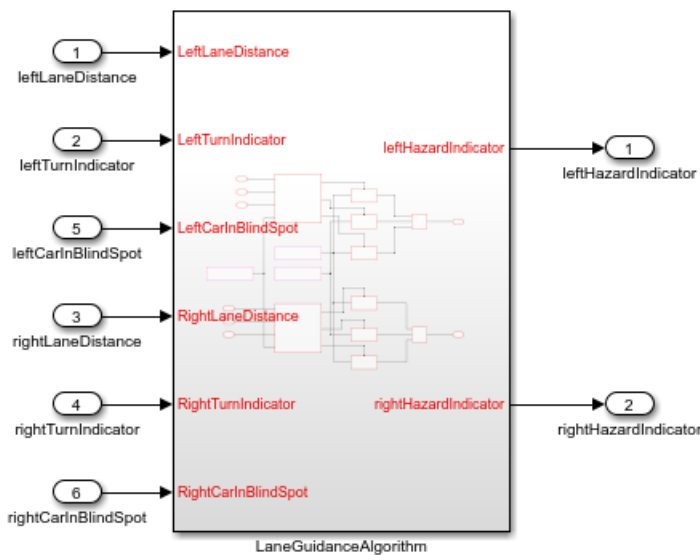
An AUTOSAR adaptive software component provides and consumes services. Each software component contains:

- An automotive algorithm, which performs tasks in response to received events.
- Required and provided ports, each associated with a service interface.
- Service interfaces, with associated events and associated namespaces.

For more information, see “Model AUTOSAR Adaptive Software Components” on page 6-2.

This example configures a Simulink representation of an automotive algorithm as an AUTOSAR adaptive software component. The configuration steps use example models LaneGuidance and `autosar_LaneGuidance`.

- 1 Open a Simulink model that either is empty or contains a functional algorithm. This example uses ERT algorithm model LaneGuidance.



- 2 Using the Model Configuration Parameters dialog box, **Code Generation** pane, configure the model for adaptive AUTOSAR code generation. Set **System target file** to `autosar_adaptive.tlc`. Apply the change.

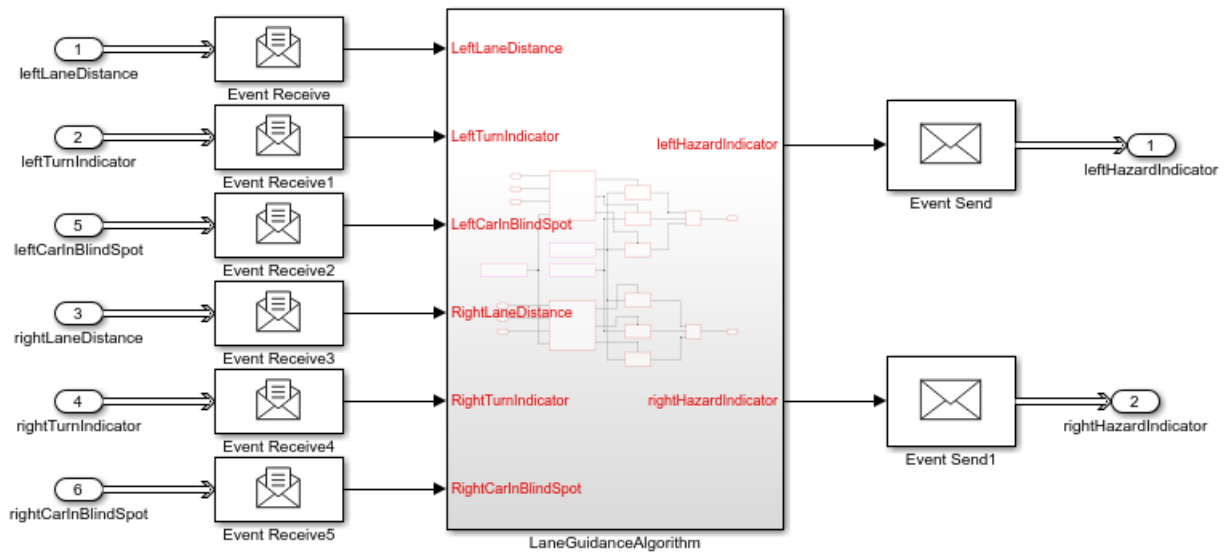
The image shows two configuration panels. The top panel, titled 'Target selection', contains the following fields: 'System target file' with the value 'autosar_adaptive.tlc' and a 'Browse...' button; 'Language' set to 'C++'; 'Language standard' set to 'C++11 (ISO)'; and 'Description' set to 'AUTOSAR Adaptive'. The bottom panel, titled 'Build process', contains: a checked checkbox for 'Generate code only'; an unchecked checkbox for 'Package code and artifacts' with a 'Zip file name' field containing 'mApp.zip'; and a 'Toolchain settings' section with a 'Toolchain' dropdown menu set to 'AUTOSAR Adaptive | CMake'.

The new setting affects other model settings. For example, the target file selection:

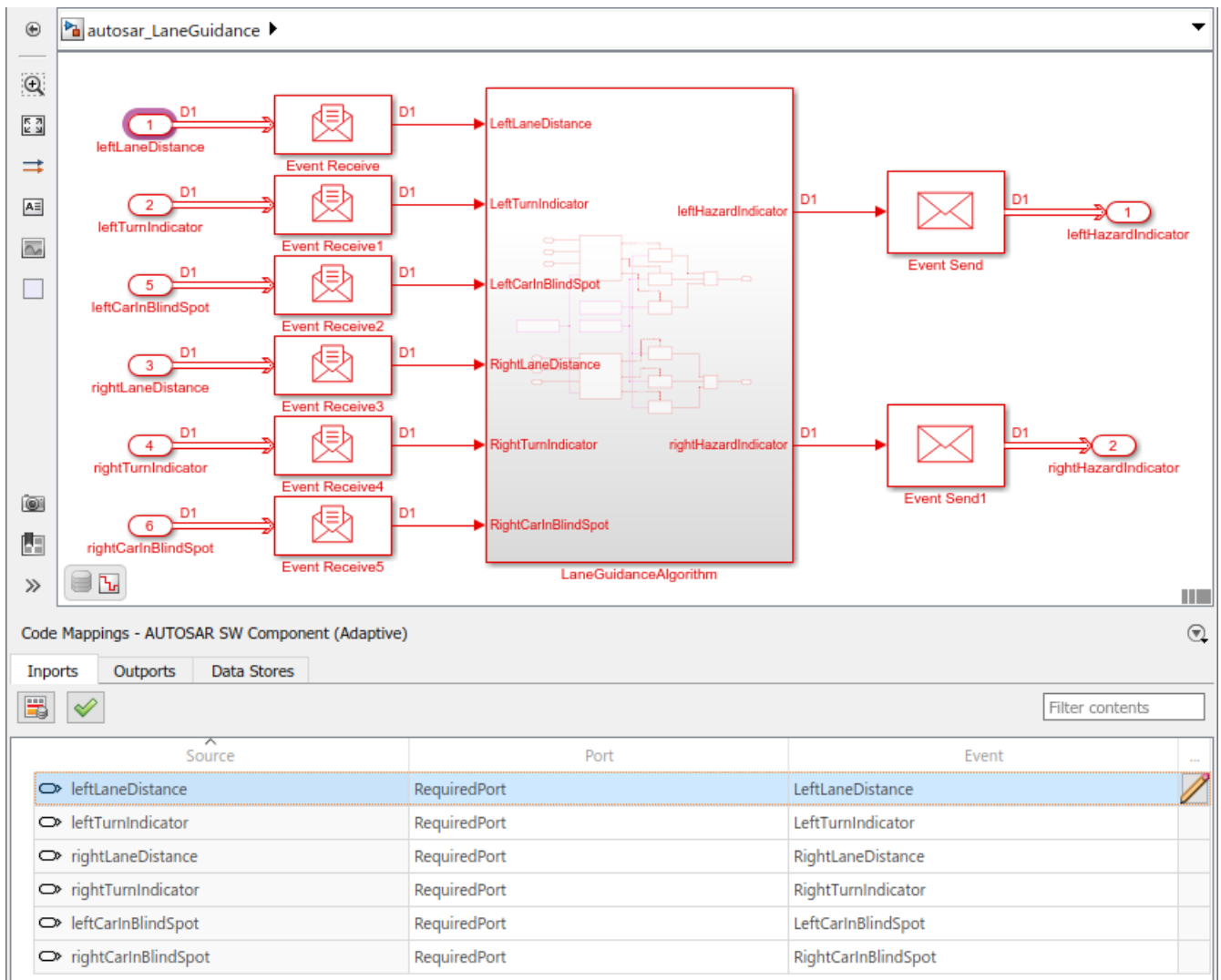
- Sets **Language** to C++.
 - Selects **Generate code only**.
 - Sets **Toolchain** to AUTOSAR Adaptive | CMake.
 - Sets **Interface > Code interface packaging** to C++ class.
- 3 Develop the model algorithmic content for use in an AUTOSAR adaptive software component. If the model is empty, construct or copy in an algorithm. Possible sources for algorithms include algorithmic elements in other Simulink models. Examples include subsystems, referenced models, MATLAB Function blocks, and C Caller blocks.
 - 4 At the top level of the model, set up event-based communication, which the AUTOSAR Adaptive Platform requires for AUTOSAR required and provided ports. AUTOSAR Blockset provides Event Receive and Event Send blocks to make the necessary event/signal connections.
 - After each root inport, add an Event Receive block, which converts an input event to a signal while preserving the signal values and data type.
 - Before each root outport, add an Event Send block, which converts an input signal to an event while preserving the signal values and data type.

Note Alternatively, you can skip this step. A later step provides a finished mapped model with event conversion blocks included.

Here is example model LaneGuidance with the event blocks added and connected.




- 5 Map the algorithm model to an AUTOSAR adaptive software component.
 - a To map the algorithm model, in the **Apps** tab, click **AUTOSAR Component Designer**. In this example the AUTOSAR Component Quick Start opens because the model is unmapped. Otherwise, to map algorithm information you can click the perspective control in the lower-right corner and select **Code**, or use the API to call MATLAB function `autosar.api.create(modelName)`.
 - b Work through the quick-start procedure. Click **Finish** to map the model.
 - c The model opens in the AUTOSAR Code perspective. The perspective displays the mapping of Simulink elements to AUTOSAR adaptive software component elements and an AUTOSAR Dictionary, which contains AUTOSAR adaptive component elements with default properties.

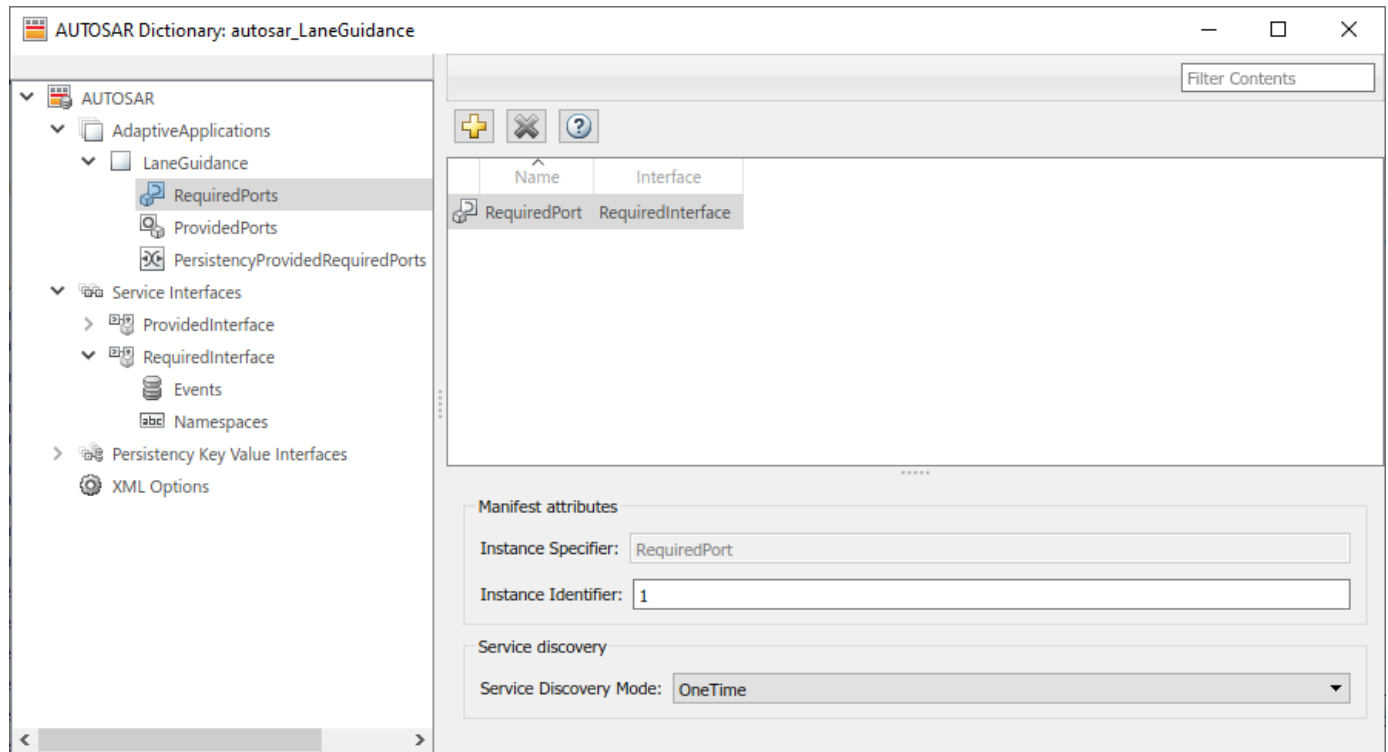


- 6 If you completed the adaptive configuration steps, save the AUTOSAR adaptive software component model with a unique name.

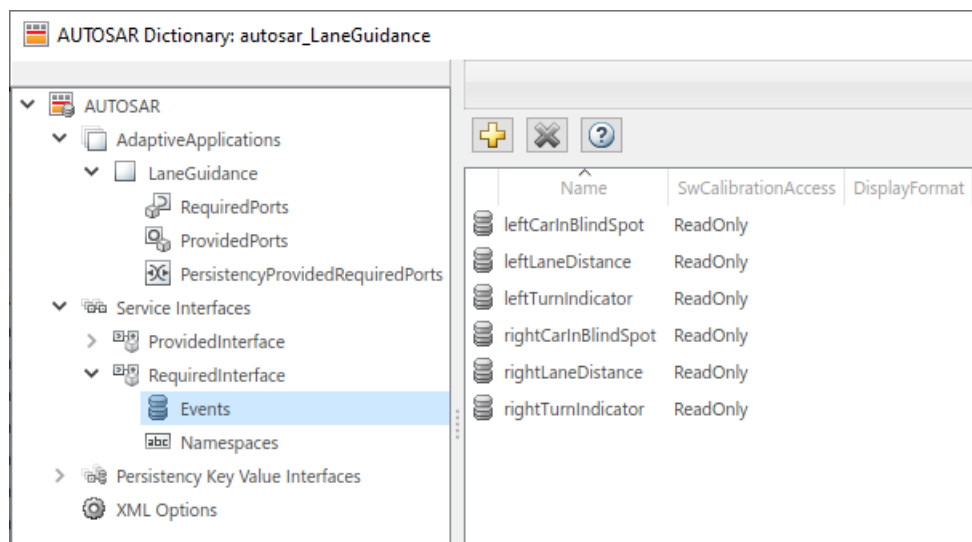
If you skipped any steps, open an example of a finished mapped AUTOSAR adaptive software component, example model `autosar_LaneGuidance`.

- 7 Using the AUTOSAR Code perspective and the AUTOSAR Dictionary (or equivalent AUTOSAR map and property functions), further refine the AUTOSAR adaptive model configuration.
- In the model window, check model data to see if you need to make post-mapping adjustments to types or other attributes. For example, verify that event data is configured correctly for your design.
 - In the AUTOSAR Code perspective, examine the mapping of Simulink inports and outports to AUTOSAR required and provided ports and events.
 - To open the AUTOSAR Dictionary, select an inport or outport and click the **AUTOSAR Dictionary** button . The dictionary opens in the view of the corresponding mapped AUTOSAR port.

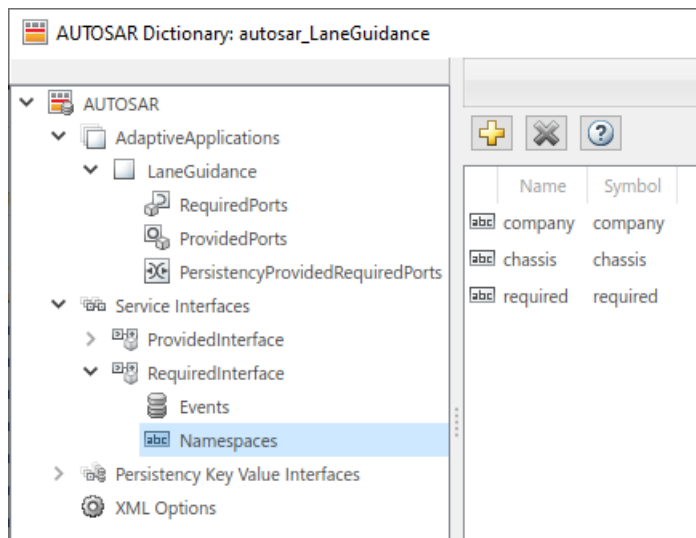
Select a port to configure its AUTOSAR attributes, such as manifest attributes or, for required ports, a service discovery mode.



- In the dictionary, you can expand service interface nodes to examine the AUTOSAR events created by the default component mapping.



- In the dictionary, you can define a unique namespace for each service interface. Example model `autosar_LaneGuidance` defines namespaces `company::chassis::provided` and `company::chassis::required` for the respective service interfaces. When you build the model, generated C++ code uses the service interface namespaces.



- In the dictionary, in the XML options view, you can configure characteristics of exported AUTOSAR XML. To generate compact code, example model `autosar_LaneGuidance` sets XML option **Exported XML file packaging** to `Single file`. In the Model Configuration Parameters dialog box, the example model sets **Code Placement > File packaging format** to `Compact`.
- 8 Build the AUTOSAR adaptive software component model. For example, in the model window, enter **Ctrl+B**. Building the model generates:

- C++ files that implement the model algorithms for the AUTOSAR Adaptive Platform and provide shared data type definitions.

[-] **Model files**

[autosar_LaneGuidance.cpp](#)
[autosar_LaneGuidance.h](#)

[-] **Shared files**

[rtwtypes.h](#)

- AUTOSAR XML descriptions of the AUTOSAR adaptive software component and manifest information for application deployment and service configuration.

[-] **Interface files**

[autosar_LaneGuidance.arxml](#)
[autosar_LaneGuidance_ExecutionManifest.arxml](#)
[autosar_LaneGuidance_ServiceInstanceManifest.arxml](#)

- C++ files that implement a main program module.

[-] **Other files**

[MainUtils.hpp](#)
[main.cpp](#)

- AUTOSAR Runtime Adaptive (ARA) environment header files.

```
[-] ARA files
  impl\_type\_double.h
  providedinterface\_common.h
  providedinterface\_skeleton.h
  requiredinterface\_common.h
  requiredinterface\_proxy.h
```

- CMakeLists.txt file that supports CMake generation of executables.

The generated C++ model files include model class definitions and AUTOSAR Runtime for Adaptive Applications (ARA) calls to implement the adaptive software component services. For example, model file `autosar_LaneGuidance.cpp` contains initialization code for each service interface and event. The code reflects the service interface namespaces and event names configured in the AUTOSAR Dictionary.

```
// Model initialize function
void autosar_LaneGuidanceModelClass::initialize()
{
    {
        ara::com::ServiceHandleContainer< company::chassis::required::proxy::
            RequiredInterfaceProxy::HandleType > handles;
        handles = company::chassis::required::proxy::RequiredInterfaceProxy::
            FindService(ara::com::InstanceIdentifier("1"));
        if (handles.size() > 0U) {
            RequiredPort = std::make_shared< company::chassis::required::proxy::
                RequiredInterfaceProxy >(*handles.begin());

            // Subscribe event
            RequiredPort->leftLaneDistance.Subscribe(1U);
            RequiredPort->leftTurnIndicator.Subscribe(1U);
            RequiredPort->leftCarInBlindSpot.Subscribe(1U);
            RequiredPort->rightLaneDistance.Subscribe(1U);
            RequiredPort->rightTurnIndicator.Subscribe(1U);
            RequiredPort->rightCarInBlindSpot.Subscribe(1U);
        }

        ProvidedPort = std::make_shared< company::chassis::provided::skeleton::
            ProvidedInterfaceSkeleton >(ara::com::InstanceIdentifier("2"), ara::com::
                MethodCallProcessingMode::kPoll);
        ProvidedPort->OfferService();
    }
}
```

Model file `autosar_LaneGuidance.cpp` also contains step code for each service interface event. For example, the step code for RequiredInterface, event `rightCarInBlindSpot`, calls a function to fetch and handle new `rightCarInBlindSpot` event data received by AUTOSAR Runtime Adaptive (ARA) environment middleware.

```
// Model step function
void autosar_LaneGuidanceModelClass::step()
{
    ...
    if (RequiredPort) {
        leftLaneDistanceResultSharedPtr = std::make_shared< ara::core::Result<size_t>
            >(RequiredPort->rightCarInBlindSpot.GetNewSamples(std::move(std::bind
                (&autosar_LaneGuidanceModelClass::
                    RequiredPortrightCarInBlindSpotReceive, this, std::placeholders::_1)));
        leftLaneDistanceResultSharedPtr->ValueOrThrow();
    }
    ...
}
```

The exported AUTOSAR XML code includes descriptions of AUTOSAR elements that you configured by using the AUTOSAR Dictionary. For example, component file `autosar_LaneGuidance.arxml` describes the namespaces and events specified for required and provided interfaces.

```

<SERVICE-INTERFACE UUID="...">
  <SHORT-NAME>RequiredInterface</SHORT-NAME>
  <NAMESPACES>
    <SYMBOL-PROPS>
      <SHORT-NAME>company</SHORT-NAME>
      <SYMBOL>company</SYMBOL>
    </SYMBOL-PROPS>
    <SYMBOL-PROPS>
      <SHORT-NAME>chassis</SHORT-NAME>
      <SYMBOL>chassis</SYMBOL>
    </SYMBOL-PROPS>
    <SYMBOL-PROPS>
      <SHORT-NAME>required</SHORT-NAME>
      <SYMBOL>required</SYMBOL>
    </SYMBOL-PROPS>
  </NAMESPACES>
  <EVENTS>
    ...
    <VARIABLE-DATA-PROTOTYPE UUID="...">
      <SHORT-NAME>rightCarInBlindSpot</SHORT-NAME>
      <CATEGORY>VALUE</CATEGORY>
      <SW-DATA-DEF-PROPS>
        <SW-DATA-DEF-PROPS-VARIANTS>
          <SW-DATA-DEF-PROPS-CONDITIONAL>
            <SW-CALIBRATION-ACCESS>READ-ONLY</SW-CALIBRATION-ACCESS>
            <SW-IMPL-POLICY>QUEUED</SW-IMPL-POLICY>
          </SW-DATA-DEF-PROPS-CONDITIONAL>
        </SW-DATA-DEF-PROPS-VARIANTS>
      </SW-DATA-DEF-PROPS>
      <TYPE-TREF DEST="IMPLEMENTATION-DATA-TYPE">
        /LaneGuidance_pkg/LaneGuidance_dt/Double</TYPE-TREF>
    </VARIABLE-DATA-PROTOTYPE>
  </EVENTS>
</SERVICE-INTERFACE>

```

The generated C++ main program file provides a framework for running adaptive software component service code. For the `autosar_LaneGuidance` model, the `main.cpp` file:

- Instantiates the adaptive software component model object.
- Reports the adaptive application state to ARA.
- Calls the model initialize and terminate functions.
- Sets up asynchronous function call objects for each task.
- Runs asynchronous function calls in response to base-rate tick semaphore posts.

See Also

Event Receive | Event Send

Related Examples

- “Model AUTOSAR Adaptive Software Components” on page 6-2
- “Create and Configure AUTOSAR Adaptive Software Component” on page 6-6
- “Map AUTOSAR Adaptive Elements for Code Generation” on page 6-37
- “Configure AUTOSAR Adaptive Elements and Properties” on page 6-21
- “Model AUTOSAR Adaptive Service Communication” on page 6-50
- “Configure AUTOSAR Adaptive Code Generation” on page 6-73

More About

- “Comparison of AUTOSAR Classic and Adaptive Platforms” on page 1-5

Model AUTOSAR Adaptive Service Communication

The AUTOSAR Adaptive Platform defines service-oriented, event-based communication between adaptive software components. Each adaptive software component provides and consumes services, and interconnected components send and receive service events. A component contains:


- An algorithm that performs tasks in response to received events.
- Required and provided ports, through which events are received and sent.
- Service interfaces, which provide the framework for event-based communication.

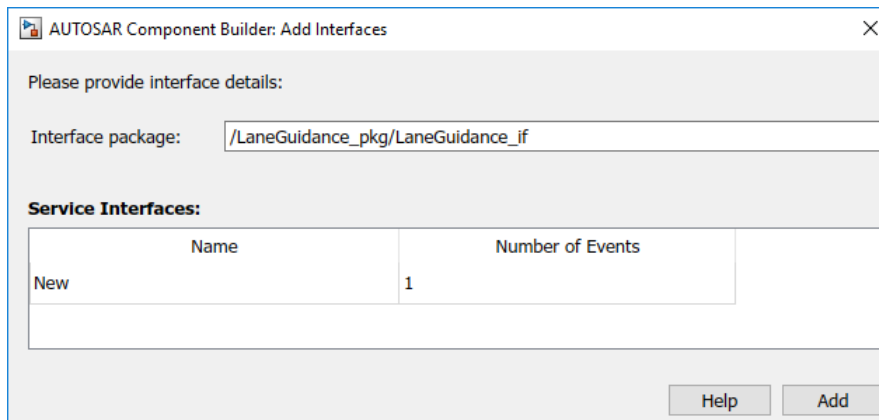
To model adaptive service communication in Simulink, you can:

- Create AUTOSAR required and provided ports, service interfaces, service interface events, and C++ namespaces.
- Create root-level inports and outports and map them to AUTOSAR required and provided ports and service interface events.

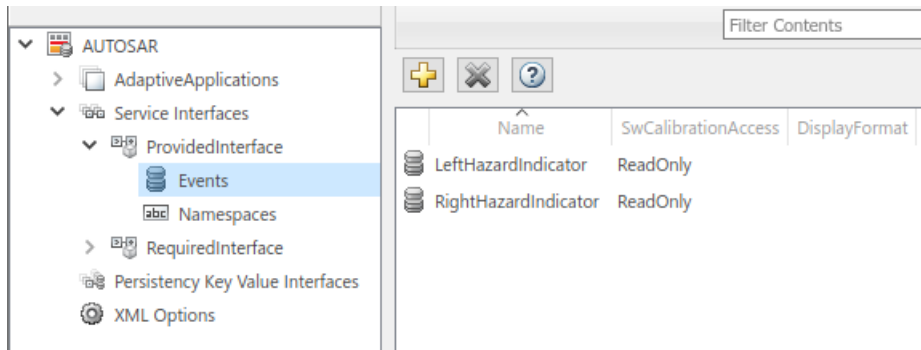
If you have Simulink Coder and Embedded Coder software, you can generate C++ code and ARXML descriptions for AUTOSAR service communication.

To implement adaptive service communication in Simulink:

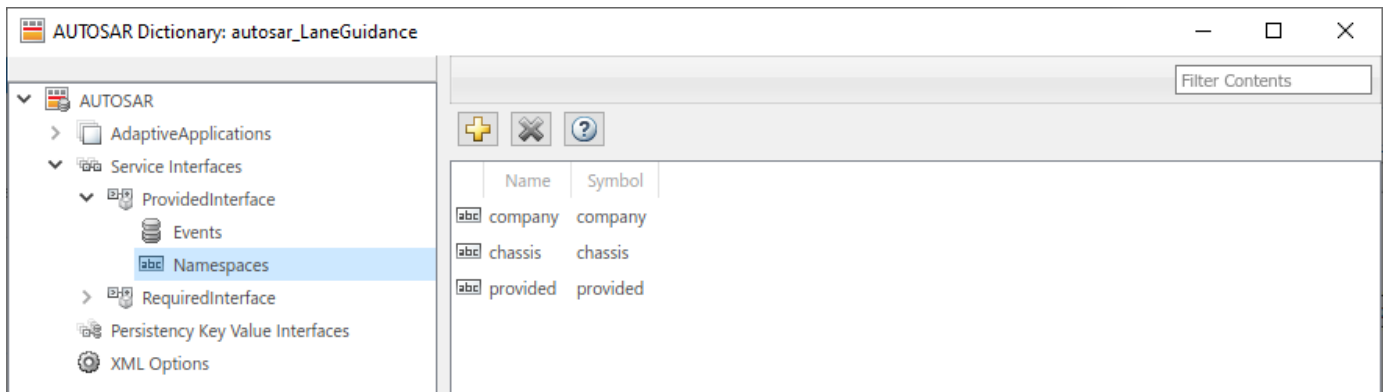
- 1 Open a model configured for the AUTOSAR Adaptive Platform. Displays in this example use model `autosar_LaneGuidance`.
- 2 Open the AUTOSAR Dictionary and select **Service Interfaces**. To create an AUTOSAR service interface, click the **Add** button . In the Add Interfaces dialog box, specify the interface name and the number of associated events.



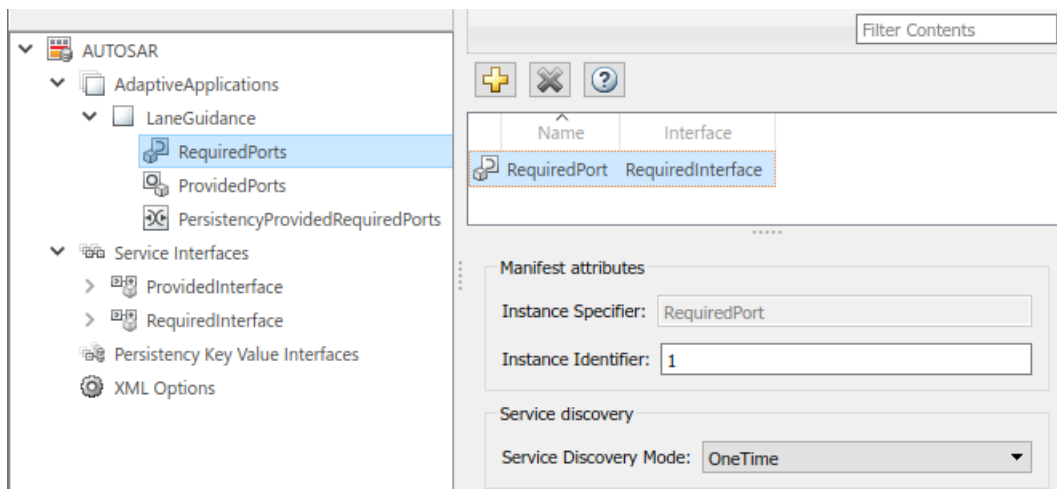
- 3 Expand the **Service Interfaces** node. Expand the new service interface and select **Events**. In the events view, select each service event and configure its attributes.



- 4 Select **Namespaces**. The namespaces view allows you to define a unique namespace for each service interface. The code generator uses the defined namespace when producing C++ code for the interface. To modify or construct a namespace specification, select a namespace element and edit the name value. For example, this namespaces view defines namespace `company::chassis::provided` for service interface `ProvidedInterface`.



- 5 At the top level of the AUTOSAR Dictionary, expand **AdaptiveApplications** and expand the adaptive software component. Use the **RequiredPorts** and **ProvidedPorts** views to add AUTOSAR required and provided ports that you want to associate with the new service interface. For each new service port, select the service interface you created.



- 6 Optionally, you can configure adaptive service instance identification for AUTOSAR ports. In the **RequiredPorts** or **ProvidedPorts** view, select a port and view its **Manifest attributes**. Based on the service instance form selected in XML options, examine the value for **Instance Specifier** or **Instance Identifier**. You can enter a value or accept an existing value. For more information, see “Configure AUTOSAR Adaptive Service Instance Identification” on page 6-64.
- 7 Optionally, for AUTOSAR required ports, you can configure service discovery, which affects how adaptive applications find dynamic services. In the **RequiredPorts** view, select a port and configure its **Service Discovery Mode**. Select **OneTime** or **DynamicDiscovery**. For more information, see “Configure AUTOSAR Adaptive Service Discovery Modes” on page 6-62.
- 8 In the model widow, to model AUTOSAR adaptive service ports, create root-level inports and outports.
- 9 Open the Code Mappings editor. Use the **Inports** and **Outports** tabs to map Simulink inports and outports to AUTOSAR required and provided ports. For each inport or outport, select an AUTOSAR required or provided port and an AUTOSAR service interface event.

Source	Port	Event	AllocateMemory	...
leftHazardIndicator	ProvidedPort	LeftHazardIndicator	false	
rightHazardIndicator	ProvidedPort	RightHazardIndicator	false	

- 10 Optionally, you can configure memory allocation for service data sent from AUTOSAR provided ports. In the **Outports** tab, select a port and use the code attribute `AllocateMemory` to configure memory allocation. Specify whether to send event data by reference (the default) or by `ara::com` allocated memory. To send event data by `ara::com` allocated memory, select the value `true`. To send event data by reference, select `false`. For more information, see “Configure Memory Allocation for AUTOSAR Adaptive Service Data” on page 6-60.
- 11 After validating the adaptive component model configuration, you can simulate or generate code for AUTOSAR service communication.

To programmatically configure AUTOSAR adaptive service communication, use the AUTOSAR property and mapping functions. For example, the following MATLAB code adds an AUTOSAR service interface, event, and required port to an open model. It then maps a Simulink inport to the AUTOSAR required port.

```
hModel = 'autosar_LaneGuidance';
openExample(hModel);

% Add AUTOSAR service interface mySvcInterface with event mySvcEvent
arProps = autosar.api.getAUTOSARProperties(hModel);
addPackageableElement(arProps, 'ServiceInterface', ...
    '/LaneGuidance_pkg/LaneGuidance_if', 'mySvcInterface');
add(arProps, 'mySvcInterface', 'Events', 'mySvcEvent');

% Add AUTOSAR required port myRPort, associated with mySvcInterface
add(arProps, 'LaneGuidance', 'RequiredPorts', 'myRPort', ...
    'Interface', 'mySvcInterface');

% Map Simulink inport to AUTOSAR port/event pair myRPort and mySvcEvent
slMap = autosar.api.getSimulinkMapping(hModel);
mapInport(slMap, 'rightCarInBlindSpot', 'myRPort', 'mySvcEvent');
```

Model Client-Server Communication

Adaptive AUTOSAR supports client-server communication between application software components. Methods on an AUTOSAR service interface define the interaction between a software component modeled as a server that *provides* an interface implementation and a software component modeled as a client that *requires* an interface.

In Simulink, you can model client-server communication with synchronous or asynchronous call behaviors. Synchronous client models produce blocked client execution, where the client sends requests to the server and waits for the response. Asynchronous client models produce non-blocking execution, which consists of clients sending requests, continuing execution after the request is sent, and processing the response upon method completion.

To model AUTOSAR clients and servers in the Simulink environment for simulation and code generation:

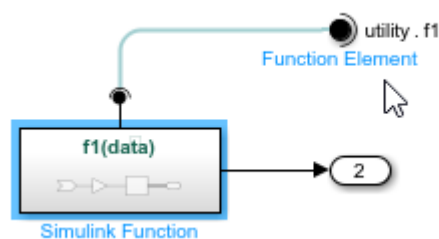
- Model AUTOSAR servers by using Simulink Function blocks at the root level of a model.
- Model synchronous or asynchronous AUTOSAR clients:
 - Model synchronous client calls by using Function Caller blocks.
 - Model asynchronous client calls by using Function Caller blocks and “Message Triggered Subsystem” blocks.

If you import ARXML definitions of clients or servers, AUTOSAR Blockset creates and configures the components in Simulink, after which you can directly simulate the component models and export generated C++ code and ARXML. ARXML definitions of clients are imported as synchronous client calls, which you can reconfigure in Simulink to use asynchronous behavior if needed.

Configure AUTOSAR Adaptive Servers

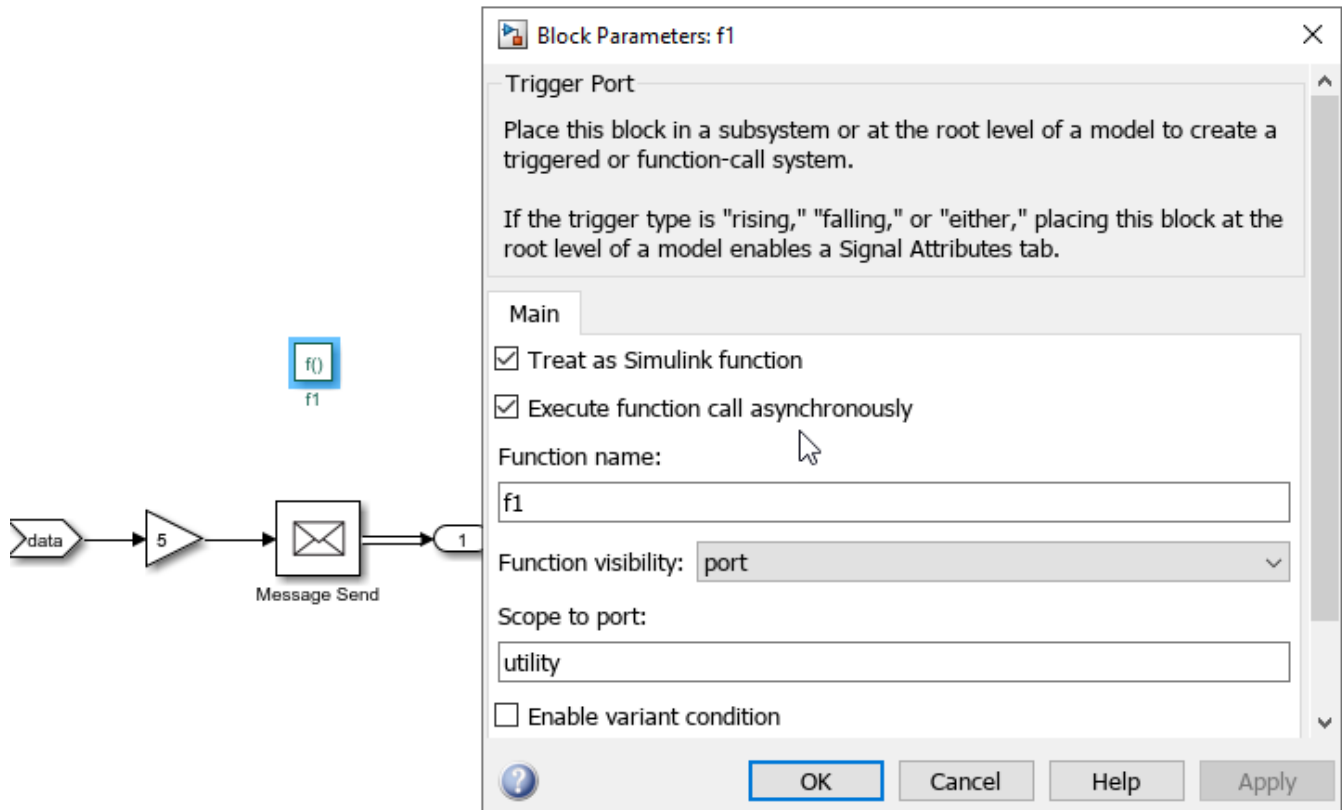
A server provides services to clients. To model and simulate an AUTOSAR adaptive server in the Simulink environment, use a Simulink Function block and Function Element block to provide the service to clients. Optionally, you can generate C++ code and export ARXML definitions.

- 1 Create or open an export-function model configured for the AUTOSAR Adaptive Platform.
- 2 Model the AUTOSAR adaptive server. Use Simulink Function blocks and Function Element ports to model provided services to clients.



- 3 Open the Simulink Function block, right-click the Trigger Port, and select **Block Parameters**. Verify **Function visibility** is set to **port** and **Scope to port** is set to the name of the exporting function port created by the root-level Function Element.


If you are modeling asynchronous client-server behavior, select **Execute function call asynchronously**.



- 4 On the **Apps** tab, click AUTOSAR Component Designer.
- 5 View the AUTOSAR properties of the service interface.

Open the AUTOSAR Dictionary and view the properties derived from the component model. Optionally, you can create additional properties:

- a Select **Service Interfaces** to view or create an interface. This interface defines the properties for the modeled server component.

To create an AUTOSAR service interface, click the add button .

- b Under **Service Interfaces**, view or create the methods. The AUTOSAR adaptive standard defines request-response methods that expect a response to a method call and fire-forget methods that do not expect a return value. For the modeled server, you can create and configure the name and method type.
- c At the top level of the AUTOSAR Dictionary, expand **AdaptiveApplications** and view **ProvidedPorts**. The ProvidedPorts define the properties for the ports used in Simulink Function blocks to respond to clients.

Click a port to view the adaptive service instance identification attributes **Instance Specifier** and **Instance Identifier**. For a deployed server to communicate with deployed clients, the service instance identification of the provided port of the modeled server must

match the instance identification of the required port of the modeled clients. For more information, see “Configure AUTOSAR Adaptive Service Instance Identification” on page 6-64.

- 6 View how the code properties map to the modeled server. Open the Code Mappings editor, then open the **Functions** tab to view server functions.
 - The **Source** column shows the modeled Function Element blocks.
 - The **Port** column shows the ProvidedPort name.
 - The **Method** column shows which method defined in the AUTOSAR Dictionary associates with each port and block.
- 7 Validate and simulate the server model.
- 8 Optionally build the component model to generate C++ code and export ARXML definitions.

Configure Synchronous AUTOSAR Adaptive Clients (Blocking)

Configure an AUTOSAR adaptive client that requires a service from a server. The client in this example models synchronous communication, resulting in blocked execution after sending the request to the server.

To model a synchronous AUTOSAR adaptive client in the Simulink environment, use a Simulink Function Caller block and configure a service interface to call the server. Optionally, you can generate C++ code and export ARXML definitions.

- 1 Create or open an export-function model configured for the AUTOSAR Adaptive Platform.
- 2 Model the synchronous AUTOSAR adaptive client. To model the client in the Simulink, use Function Caller blocks and Function Element Call blocks to call the service provided by the server.
- 3 Continue to “Validate, Simulate, and Optionally Generate Code for Adaptive Clients” on page 6-58.

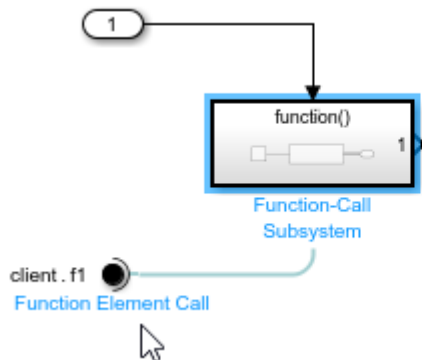
Configure Asynchronous AUTOSAR Adaptive Clients (Non-blocking)

Configure an AUTOSAR adaptive client that requires a service from a server. The client in this example models asynchronous communication, resulting in non-blocking operations with continued execution after sending the request to the server.

To model an asynchronous AUTOSAR adaptive client in the Simulink environment, use a Simulink Function Caller block and a Message Triggered subsystem, and configure the service interface to call the server. Optionally, you can generate C++ code and export ARXML definitions.

To configure an asynchronous adaptive client:

- 1 Create or open an export-function model configured for the AUTOSAR Adaptive Platform.
- 2 Add a Function Element Call port to model the asynchronous AUTOSAR adaptive client.



- 3 In the function-call subsystem, add a Function Caller block, right-click the Function Caller block, and select **Block Parameters (FunctionCaller)**. In the Block Parameters dialog box, select **Execute function call asynchronously**.

The asynchronous function-call block must have one output. When asynchronous behavior is selected, the Function Caller block emits a message instead of signal lines. The message signifies the asynchronous behavior of method execution.

The diagram shows a 'Function Caller' block with inputs 'data' and 'client.f1()' and output 'y'. A message signal '1' is connected to the output. The Block Parameters dialog box for Function Caller is shown, with the 'Execute function call asynchronously' checkbox checked.

Block Parameters: Function Caller

FunctionCaller

Call the function described in 'Function prototype' to compute output signals from input signals. These signals correspond to the arguments of the function called, and you can optionally provide examples in 'Input/Output argument specifications' if the function is placed outside of the current model hierarchy.

Parameters

Function prototype:

y = client.f1(data)

Input argument specifications (e.g., int8(1)):

double(1) 1

Output argument specifications (e.g., int8(1)):

double(1) 1

Sample time (-1 for inherited):

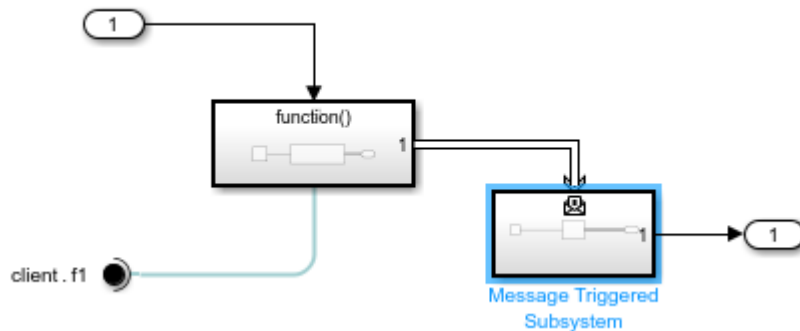
-1

Execute function call asynchronously

OK Cancel Help Apply

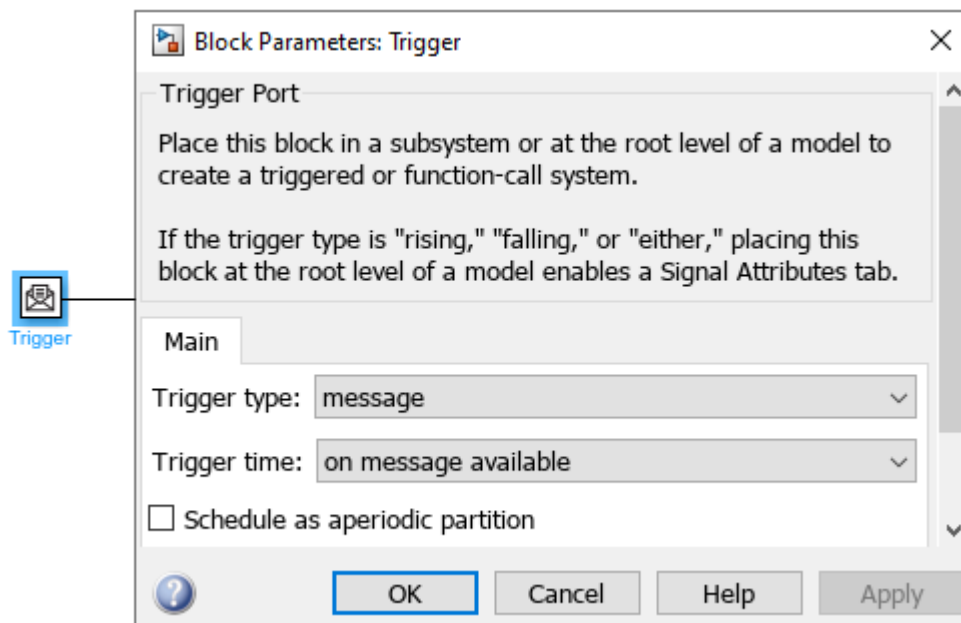
- 4 At the root level of the model, add a "Message Triggered Subsystem" and connect the message signal from the Function-Call Subsystem.

When the method called from the Function Caller completes, the output message triggers the message triggered subsystem, which acts as the callback that the client application registers for an asynchronous method. The message triggered subsystem executes whenever a message is available at the control port, independent of sample time.



- 5 Open the message triggered subsystem, right-click the Trigger Port block, and select **Block Parameters (TriggerPort)**.

In the Block Parameters dialog box, verify that **Trigger type** is set to message, and **Schedule as aperiodic partition** is not selected.




- 6 Add behavior to the message triggered subsystem.
- 7 Complete the model with any additional logic.
- 8 Continue to section “Validate, Simulate, and Optionally Generate Code for Adaptive Clients” on page 6-58.

Validate, Simulate, and Optionally Generate Code for Adaptive Clients

To validate and simulate an AUTOSAR adaptive client in the Simulink environment, review the service interface by using the AUTOSAR Dictionary and code property mappings by using the Code Mapping editor. Optionally, you can generate C++ code and export ARXML definitions.

- 1 On the **Apps** tab, click AUTOSAR Component Designer.
- 2 View the AUTOSAR properties of the service interface. Open the AUTOSAR Dictionary and view the properties derived from the component model. Optionally, you can create additional properties:

- a Select **Service Interfaces** to view or create an interface. This interface defines the properties for the modeled client component.

To create an AUTOSAR service interface, click the add button .

- b Under **Service Interfaces**, view or create the methods. For the modeled client, you can create and configure the name and method type.
- c At the top level of the AUTOSAR Dictionary, expand **AdaptiveApplications**, and view the **RequiredPorts**. **RequiredPorts** define the code properties for the ports used in Simulink Function Caller blocks that request services from servers.

Click a port to view the adaptive service instance identification manifest attributes **Instance Specifier** and **Instance Identifier**. For a deployed client to communicate with deployed servers, the service instance identification of the required port of the modeled client must match the instance identification of the provided port of the modeled server. For more information, see “Configure AUTOSAR Adaptive Service Instance Identification” on page 6-64.

- 3 View how the code properties map to the modeled client. Open the Code Mappings editor, then open the **Function Callers** tab to view client functions.
 - The **Source** column shows the modeled Function Caller blocks.
 - The **Port** column shows the RequiredPort name.
 - The **Method** column shows which method defined in the AUTOSAR Dictionary associates with each port and block.
- 4 Validate and simulate the client model.
- 5 (Optional) Build the component model to generate C++ code and export ARXML definitions.

Tips and Limitations

- Global Simulink functions are not supported for Adaptive AUTOSAR.
- Private scoped Simulink functions are not mapped to methods. They can be used to model behavior internal to the adaptive application component.
- Function Caller blocks configured for asynchronous behavior must have one output. Function Caller blocks with a void output or multiple outputs are not supported.

See Also

Simulink Function | Function Caller | Function-Call Subsystem | Message Triggered Subsystem | Function Element | Function Element Call

Related Examples

- “Configure AUTOSAR Adaptive Elements and Properties” on page 6-21
- “Map AUTOSAR Adaptive Elements for Code Generation” on page 6-37

More About

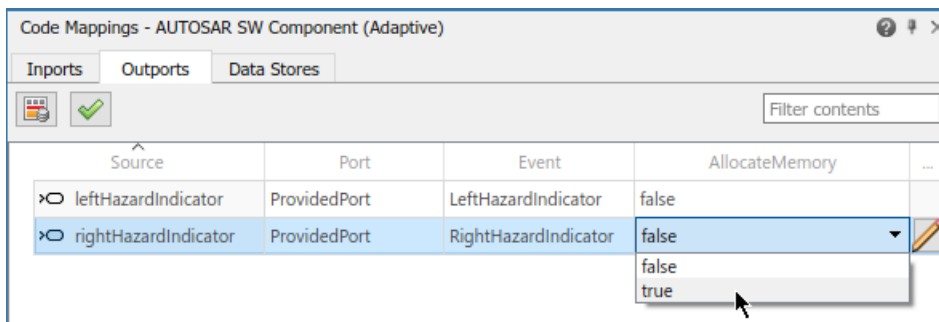
- “Model AUTOSAR Communication” on page 2-21
- “AUTOSAR Component Configuration” on page 4-3

Configure Memory Allocation for AUTOSAR Adaptive Service Data

To send service event data, the AUTOSAR Adaptive Platform supports these methods:

- By reference — The send function uses memory in the application address space. After the send returns, the application can modify the event data.
- By `ara::com` allocated memory — The application requests `ara::com` middleware to allocate memory for the data. This method avoids data copies by `ara::com` middleware and can be more efficient for frequent sends or large amounts of data. But the application loses access to the memory after the send returns.

To configure memory allocation for event sends, open the Code Mappings editor. Select the **Outports** tab and examine each outport. When you select an outport, the editor displays the code attribute `AllocateMemory`. To send event data by `ara::com` allocated memory, select the value `true`. To send event data by reference, select `false`.



If you set `AllocateMemory` to `true`, in the generated C++ model code, the corresponding event send uses an `ara::com` allocated buffer.

```
void autosar_LaneGuidanceModelClass::step()
{
    ...
    ara::com::SampleAllocateePtr<company::chassis::provided::skeleton::events::
        rightHazardIndicator::SampleType> *rightHazardIndicatorAllocateePtrRawPtr;
    std::shared_ptr< ara::com::SampleAllocateePtr<company::chassis::provided::
        skeleton::events::rightHazardIndicator::SampleType> >
        rightHazardIndicatorAllocateePtrSharedPtr;
    ...
    rightHazardIndicatorAllocateePtrSharedPtr = std::make_shared< ara::com::
        SampleAllocateePtr<company::chassis::provided::skeleton::events::
        rightHazardIndicator::SampleType> >
        (ProvidedPort->rightHazardIndicator.Allocate());
    rightHazardIndicatorAllocateePtrRawPtr =
        rightHazardIndicatorAllocateePtrSharedPtr.get();

    // Send: '<S8>/Message Send'
    *rightHazardIndicatorAllocateePtrRawPtr->get() = rtb_Merge1;

    // Send event
    ProvidedPort->rightHazardIndicator.Send(std::move
        (*rightHazardIndicatorAllocateePtrRawPtr));
}

```

See Also

Related Examples

- “Model AUTOSAR Adaptive Service Communication” on page 6-50
- “Map AUTOSAR Adaptive Elements for Code Generation” on page 6-37

More About

- “AUTOSAR Component Configuration” on page 4-3

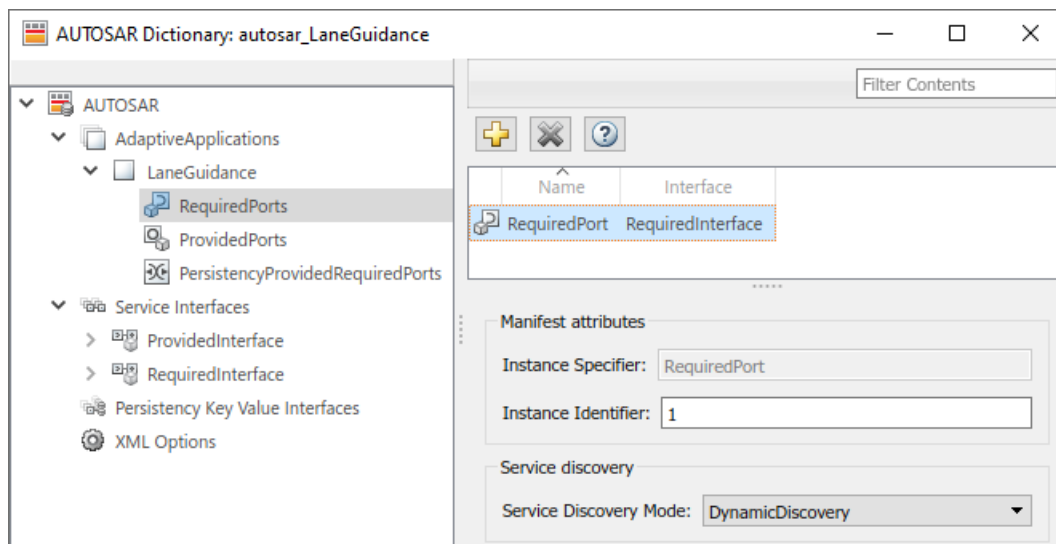
Configure AUTOSAR Adaptive Service Discovery Modes

AUTOSAR adaptive service communication provides the option to configure applications to use one-time or dynamic discovery to subscribe to services. The default discovery mode, `OneTime`, allows an AUTOSAR application to find and subscribe to services at initialization. This mode of discovery may require that services are started before the application. You can change the service discovery mode to `DynamicDiscovery` to enable an AUTOSAR application to find and subscribe to services as they become available.

You can configure, in your model or programmatically, the service discovery mode of each required service port as `OneTime` or `DynamicDiscovery`:

- From your model, you can use the AUTOSAR Dictionary to open the **Service discovery** attributes of required ports and select their service discovery modes.

This example shows how to set a required port to `DynamicDiscovery`.



- Programmatically, you can use the `set` function from the `getAUTOSARProperties` API to configure the service discovery mode.

This example shows how to set a required port to `DynamicDiscovery`.

```
hModel = 'autosar_LaneGuidance';
openExample(hModel);
apiObj = autosar.api.getAUTOSARProperties(hModel);
set(apiObj, "/LaneGuidance_pkg/LaneGuidance_swc/LaneGuidance/RequiredPort/", ...
    "ServiceDiscoveryMode", "DynamicDiscovery")
```

The service discovery mode value impacts the generated C++ code (in the model source code file) in the following two locations:

- The call site of the `StartFindService` API and registration of the callback (for both the `InstanceIdentifier` and `InstanceSpecifier` variants).

```
// Model initialize function
void autosar_LaneGuidanceModelClass::initialize()
{
    ProvidedPort = std::make_shared< company::chassis::provided::skeleton::
        ProvidedInterfaceSkeleton >(ara::com::InstanceIdentifier("2"), ara::com::
```

```

        MethodCallProcessingMode::kPoll);
    ProvidedPort->OfferService();
    company::chassis::required::proxy::RequiredInterfaceProxy::StartFindService
        (std::move(std::bind(&autosar_LaneGuidanceModelClass::RequiredPortSvcHandler,
            this, std::placeholders::_1, std::placeholders::_2)), ara::com::
            InstanceIdentifier("1"));
}

```

- The definition of the callback function.

```

void autosar_LaneGuidanceModelClass::RequiredPortSvcHandler(ara::com::
    ServiceHandleContainer< company::chassis::required::proxy::
    RequiredInterfaceProxy::HandleType > svcHandles, const ara::com::
    FindServiceHandle fsHandle)
{
    if ((!RequiredPort) && (svcHandles.size() > 0U)) {
        RequiredPort = std::make_shared< company::chassis::required::proxy::
            RequiredInterfaceProxy >(*svcHandles.begin());
        RequiredPort->leftCarInBlindSpot.Subscribe(1U);
        RequiredPort->leftLaneDistance.Subscribe(1U);
        RequiredPort->leftTurnIndicator.Subscribe(1U);
        RequiredPort->rightCarInBlindSpot.Subscribe(1U);
        RequiredPort->rightLaneDistance.Subscribe(1U);
        RequiredPort->rightTurnIndicator.Subscribe(1U);
        company::chassis::required::proxy::RequiredInterfaceProxy::StopFindService
            (fsHandle);
    }
}

```

See Also

Related Examples

- “Model AUTOSAR Adaptive Service Communication” on page 6-50
- “Configure AUTOSAR Adaptive Elements and Properties” on page 6-21

More About

- “AUTOSAR Component Configuration” on page 4-3

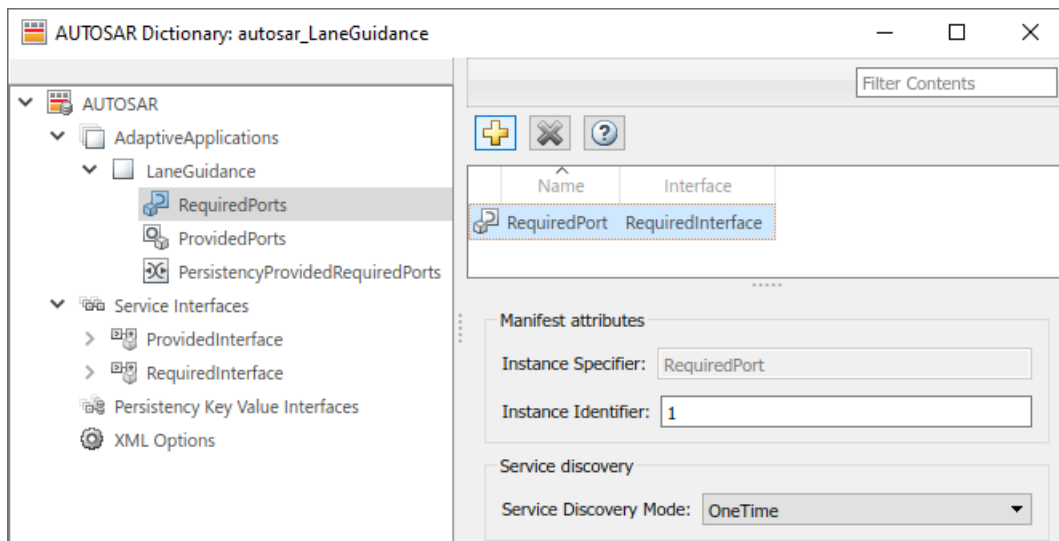
Configure AUTOSAR Adaptive Service Instance Identification

You can configure service instance identification for required and provided ports in an AUTOSAR adaptive component. When you build an adaptive software component model:

- Exported ARXML files include a service instance manifest file, which describes port-to-service instance mapping.
- Generated C++ code uses the configured service instance information in `ara::com` function calls.

To configure service instance identification:

- 1 Open the AUTOSAR Dictionary and select **XML Options**. Set XML option **Identify Service Instance Using** to indicate the form in which to generate service instance information. Select `InstanceIdentifier` or `InstanceSpecifier`. The form that you select is used to identify service instances in generated Proxy and Skeleton functions.
- 2 Go to the required ports and provided ports views in the dictionary. Select each listed port to display its **Manifest attributes**. For each port, based on the service instance form you selected in XML options, examine the value for **Instance Specifier** or **Instance Identifier**. You can enter a value or accept an existing value.



Building the model generates the service instance manifest file `model_ServiceInstanceManifest.arxml`. The manifest file describes service interface deployments, service instance to port mapping, and service interfaces for the adaptive component.

In the generated C++ code, `ara::com` function calls use the configured service instance information. For example, if you selected the `InstanceIdentifier` form, and set **Instance Identifier** to 1 for a required port, the generated function calls use that configuration.

```
// Model initialize function
void autosar_LaneGuidanceModelClass::initialize()
{
    {
        ara::com::ServiceHandleContainer< company::chassis::required::proxy::
            RequiredInterfaceProxy::HandleType > handles;
        handles = company::chassis::required::proxy::RequiredInterfaceProxy::
            FindService(ara::com::InstanceIdentifier("1"));
        ...
    }
}
```

See Also

Related Examples

- “Model AUTOSAR Adaptive Service Communication” on page 6-50
- “Configure AUTOSAR Adaptive Elements and Properties” on page 6-21

More About

- “AUTOSAR Component Configuration” on page 4-3


Model AUTOSAR Adaptive Persistent Memory

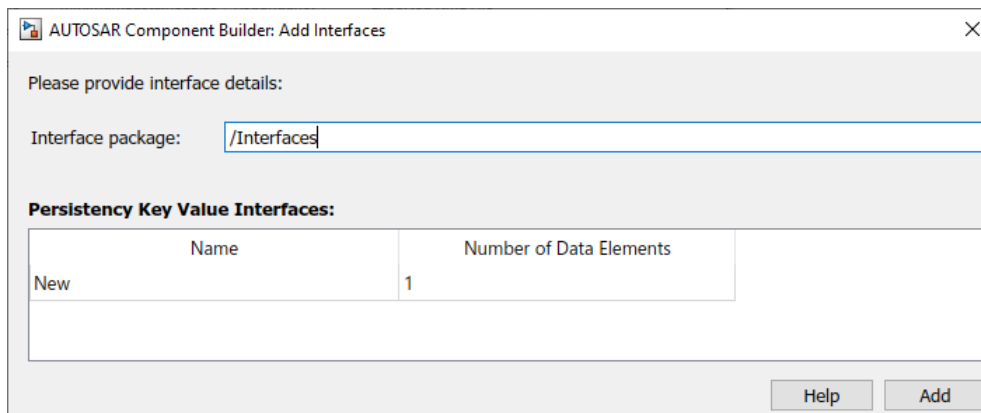
To model adaptive persistent memory in Simulink, you can:

- Create AUTOSAR persistency provided-required ports, persistency key value interfaces, and key value interface data elements.
- Create data stores and map them to AUTOSAR persistency ports and data elements.

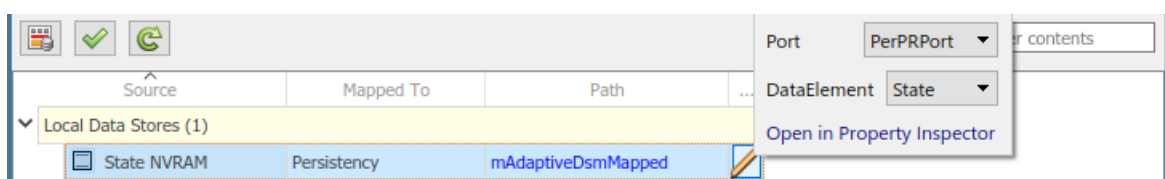
If you have Simulink Coder and Embedded Coder software, you can generate C++ code and ARXML descriptions for AUTOSAR persistent memory artifacts.

To implement adaptive persistent memory in Simulink:

- 1 Open a model configured for the AUTOSAR Adaptive Platform.
- 2 Open the AUTOSAR Dictionary and select **Persistency Key Value Interfaces**. To create an AUTOSAR persistency key value interface, click the **Add** button . In the Add Interfaces dialog box, specify the interface name and the number of associated data elements.



- 3 Expand the **Persistency Key Value Interfaces** node. Expand the new key value interface and select **DataElements**. In the data elements view, select each data element and configure its name.
- 4 At the top level of the AUTOSAR Dictionary, expand **AdaptiveApplications** and expand the adaptive software component. Use the **PersistencyProvidedRequiredPorts** view to add AUTOSAR persistency provided-required ports that you want to associate with the new persistency key value interface.
- 5 In the model window, to model AUTOSAR adaptive persistency, add data store memory blocks.
- 6 Open the Code Mappings editor. Use the **Data Stores** tab to map Simulink data store memory blocks to Persistency.
- 7 For each data store memory, select an AUTOSAR persistency provided-required port and an AUTOSAR persistency key value interface data element.



- 8 To programmatically map a data store and select port and data element for the data store, you can use the `mapDataStore` function.
- 9 After validating the adaptive component model configuration, you can simulate or generate code for AUTOSAR adaptive persistent memory.

See Also

`mapDataStore` | `getDataStore`

Related Examples

- “Map AUTOSAR Adaptive Elements for Code Generation” on page 6-37
- “Configure AUTOSAR Adaptive Elements and Properties” on page 6-21

Generate AUTOSAR Adaptive C++ Code and XML Descriptions

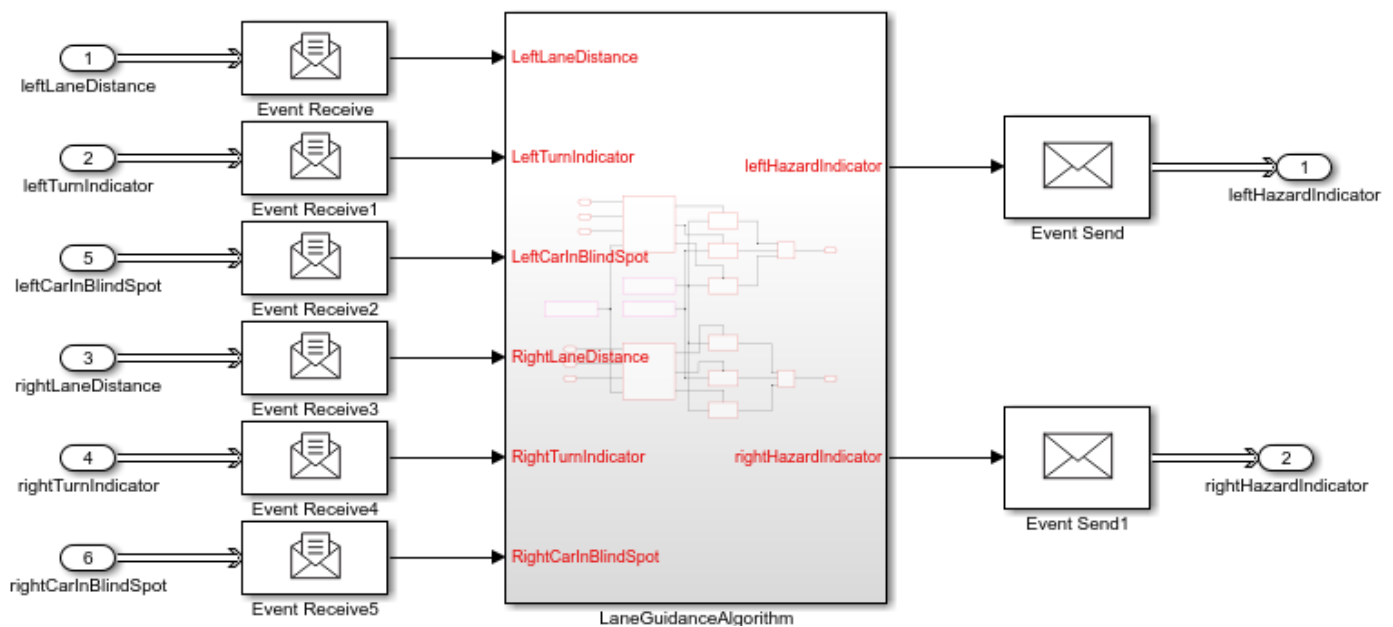
Generate AUTOSAR-compliant C++ code and export AUTOSAR XML (ARXML) descriptions from AUTOSAR adaptive component model.

If you have Simulink Coder and Embedded Coder software, you can build AUTOSAR component models. Building an adaptive component model generates algorithmic C++ code and exports ARXML descriptions that comply with AUTOSAR Adaptive Platform specifications. Use the generated C++ code and ARXML descriptions for testing in Simulink or integration into an AUTOSAR adaptive run-time environment.

Prepare AUTOSAR Adaptive Component Model for Code Generation

Open an adaptive component model from which you want to generate AUTOSAR C++ code and ARXML descriptions. This example uses AUTOSAR example model `autosar_LaneGuidance`.

```
open_system('autosar_LaneGuidance');
```



Optionally, to refine model configuration settings for code generation, you can use the Embedded Coder Quick Start (recommended). This example uses the Embedded Coder Quick Start. From the **Apps** tab, open the AUTOSAR Component Designer app. On the **AUTOSAR** tab, click **Quick Start**.

Work through the quick-start procedure. In the Output window, select output option **C++ code compliant with AUTOSAR Adaptive Platform**.

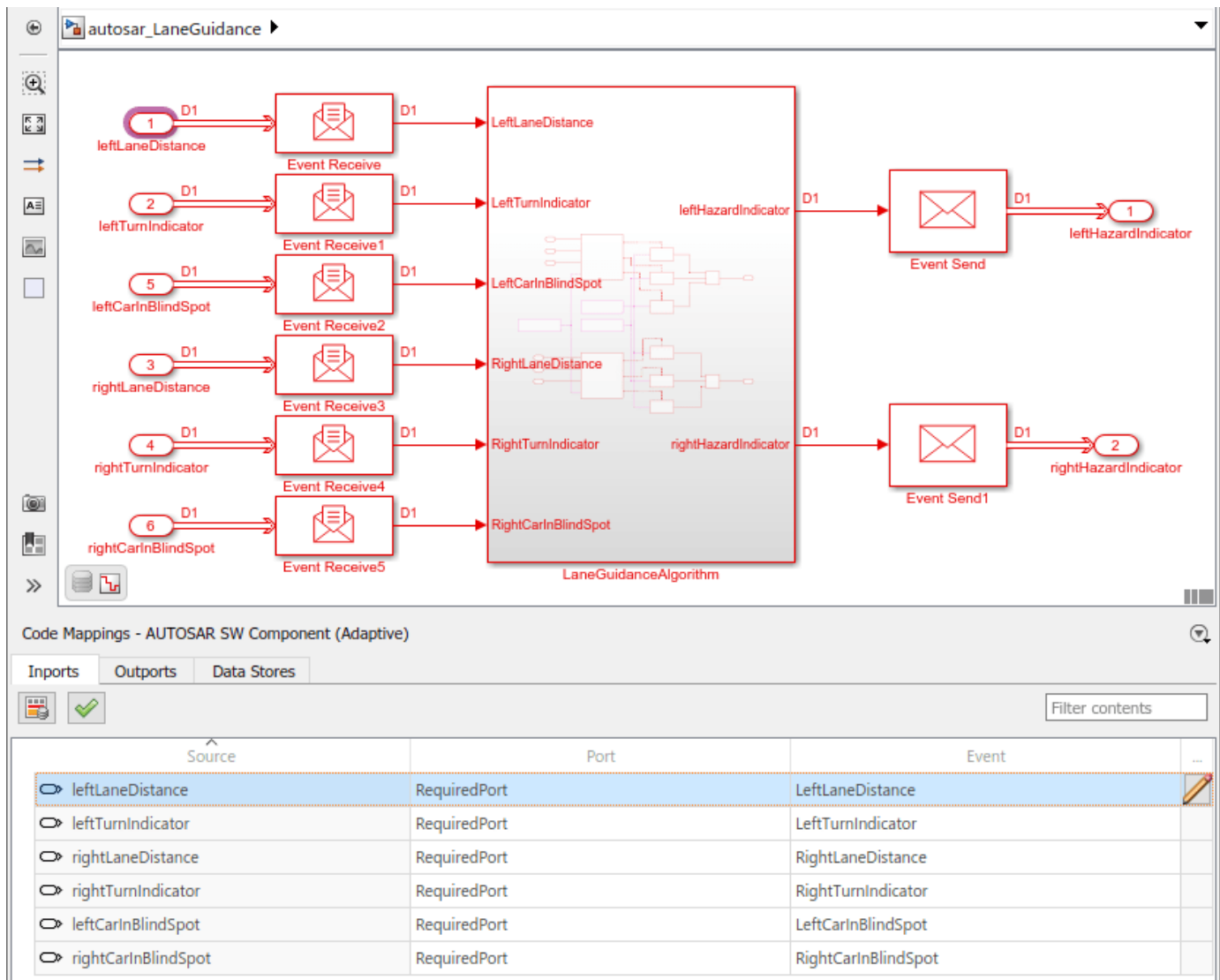
Select the output for your generated code.

- C code
- C code compliant with AUTOSAR
- C++ code
- C++ code compliant with AUTOSAR Adaptive Platform

The quick-start software takes the following steps to configure an AUTOSAR adaptive software component model:

- 1** Configures code generation settings for the model. If the AUTOSAR target is not selected, the software sets model configuration parameter **System target file** to `autosar_adaptive.tlc`.
- 2** If no AUTOSAR mapping exists, the software creates a mapped AUTOSAR adaptive software component for the model.
- 3** Performs a model build.

In the last window, when you click **Finish**, your model opens in the AUTOSAR code perspective.



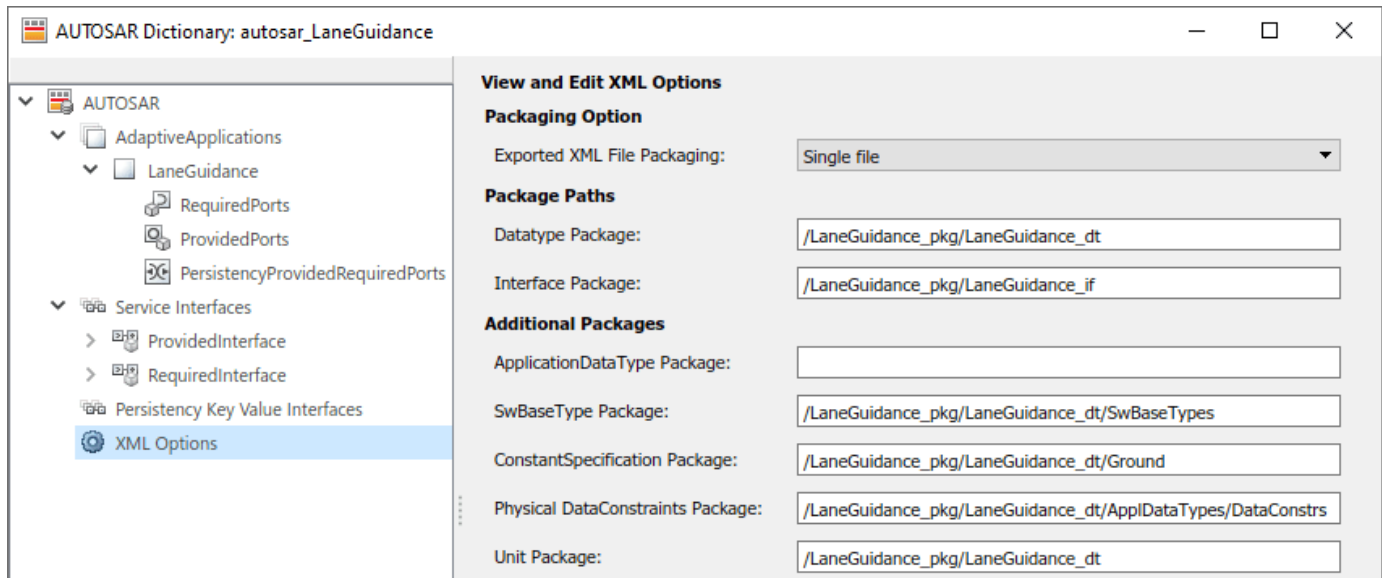
Inspect XML Options in AUTOSAR Dictionary

Before generating code, open the AUTOSAR Dictionary and examine the settings of AUTOSAR XML export parameters. On the **AUTOSAR** tab, select **Code Interface > AUTOSAR Dictionary**. In the AUTOSAR Dictionary, select **XML Options**.

The XML options view in the AUTOSAR Dictionary displays XML export parameters and their values. You can configure:

- XML file packaging for AUTOSAR elements created in Simulink
- AUTOSAR package paths
- Aspects of exported AUTOSAR XML content

This example sets **Exported XML file packaging** to **Single file**, so that ARXML for adaptive components, data types, and interfaces is exported into a single file, *modelName.arxml*. Export also generates ARXML manifest files.



Generate AUTOSAR C++ Code and XML Descriptions

To generate AUTOSAR C++ code and XML software descriptions that comply with Adaptive Platform specifications, build the model. In the model window, press **Ctrl+B**. The build process generates C++ code and ARXML descriptions to the model build folder, `autosar_LaneGuidance_autosar_adaptive`. Data types and related elements that are not used in the model are removed from the exported ARXML files. When the build completes, a code generation report opens.

The screenshot shows a window titled "Code Generation Report" with a search bar and "Match Case" option. The main content is divided into two panes. The left pane, titled "Content", lists various reports and code files. The right pane shows the source code for "autosar_LaneGuidance.cpp".

Content Pane:

- Summary
- Subsystem Report
- Code Interface Report
- Traceability Report
- Static Code Metrics Report
- Code Replacements Report
- Coder Assumptions

Code Pane:

```

206
207 // Model initialize function
208 void autosar_LaneGuidance::initialize()
209 {
210 {
211     ara::com::ServiceHandleContainer< company::chassis::required::proxy::
212         RequiredInterfaceProxy::HandleType > handles;
213     ProvidedPort = std::make_shared< company::chassis::provided::skeleton::
214         ProvidedInterfaceSkeleton >(ara::com::InstanceIdentifier("2"), ara::com::
215         MethodCallProcessingMode::kEvent);
216     ProvidedPort->OfferService();
217     handles = company::chassis::required::proxy::RequiredInterfaceProxy::
218         FindService(ara::com::InstanceIdentifier("1"));
219     if (handles.size() > 0U) {
220         RequiredPort = std::make_shared< company::chassis::required::proxy::
221             RequiredInterfaceProxy >(*handles.begin());
222
223         // Subscribe event
224         RequiredPort->LeftCarInBlindSpot.Subscribe(1U);
225         RequiredPort->LeftLaneDistance.Subscribe(1U);
226         RequiredPort->LeftTurnIndicator.Subscribe(1U);
227         RequiredPort->RightCarInBlindSpot.Subscribe(1U);
228         RequiredPort->RightLaneDistance.Subscribe(1U);
229         RequiredPort->RightTurnIndicator.Subscribe(1U);
230     }
231 }
232 }
233
234 // Model terminate function
235 void autosar_LaneGuidance::terminate()
236 {

```

Code Pane File List:

- Model files
 - autosar_LaneGuidance.cpp
 - autosar_LaneGuidance.h
- Shared files
 - rtwtypes.h
- Interface files
 - autosar_LaneGuidance.arxml
 - autosar_LaneGuidance_Execu
 - autosar_LaneGuidance_Servic
- Other files
 - MainUtils.hpp
- Other files
 - main.cpp
- ARA files
 - impl_type_double.h
 - providedinterface_common.h

Related Links

- “Code Generation”
- “AUTOSAR Component Configuration” on page 4-3
- “AUTOSAR Blockset”

Configure AUTOSAR Adaptive Code Generation

To generate AUTOSAR-compliant C++ code and ARXML component descriptions from a model configured for the AUTOSAR Adaptive platform:

- 1 In the Configuration Parameters dialog box, on the **Code Generation > AUTOSAR Code Generation Options** pane, configure AUTOSAR code generation parameters.
- 2 Configure AUTOSAR XML export options by using the AUTOSAR Dictionary or AUTOSAR property functions.
- 3 Optionally, customize the generated C++ class name and namespace for the adaptive model.
- 4 Optionally, modify the run-time logging behavior for the adaptive application.
- 5 Build the model.

Note CMake version 3.12 or above is required for AUTOSAR adaptive code generation.

In this section...

- “Select AUTOSAR Adaptive Schema” on page 6-73
- “Specify Maximum SHORT-NAME Length” on page 6-74
- “Specify XCP Slave Transport Layer” on page 6-74
- “Specify XCP Slave IP Address” on page 6-74
- “Specify XCP Slave Port” on page 6-75
- “Enable XCP Slave Message Verbosity” on page 6-75
- “Use Custom XCP Slave” on page 6-75
- “Inspect AUTOSAR Adaptive XML Options” on page 6-76
- “Customize Class Name and Namespace in Generated Code” on page 6-76
- “Configure Run-Time Logging Behavior” on page 6-76
- “Generate AUTOSAR Adaptive C++ and XML Files” on page 6-77

Select AUTOSAR Adaptive Schema

For import and export of ARXML files and generation of AUTOSAR-compliant C++ code, AUTOSAR Blockset supports the following AUTOSAR Adaptive Platform schema versions:

- R21-11 (00050)
- R20-11 (00049)
- R19-11 (00048)
- R19-03 (00047)
- R18-10 (00046)

Selecting the AUTOSAR adaptive system target file for your model for the first time sets the schema version parameter to the default value R21-11 (00050).

If you import ARXML files into Simulink, the ARXML importer detects and uses the schema version. It sets the schema version parameter in the model. For example, if you import ARXML files based on schema R21-11 (00050), the importer sets the matching schema version in the model.

When you build an AUTOSAR adaptive model, the code generator exports ARXML descriptions and generates C++ code that are compliant with the current AUTOSAR schema version value.

Before exporting your AUTOSAR software component, check the selected schema version. If you need to change the selected schema version, use the model configuration parameter **Generate XML file for schema version**.

Note Set the AUTOSAR model configuration parameters to the same values for top and referenced models. This guideline applies to **Generate XML file for schema version**, **Maximum SHORT-NAME length**, **Transport layer**, **IP address**, **Port**, **Verbose**, and **Use custom XCP Slave**.

Specify Maximum SHORT-NAME Length

The AUTOSAR standard specifies that the maximum length of SHORT - NAME XML elements is 128 characters.

To specify a maximum length for SHORT - NAME elements exported by the code generator, set the model configuration parameter **Maximum SHORT-NAME length** to an integer value between 32 and 128, inclusive. The default is 128 characters.

Specify XCP Slave Transport Layer

XCP is a network protocol originating from ASAM for connecting calibration systems to electronic control units. It enables read and write access to variables and memory contents of micro controller systems at runtime. As a two-layer protocol, XCP separates the protocol and transport layers and adheres to a Single-Master/Multi-Slave concept. Transport layer selection does not affect the XCP protocol layer.

Currently, the following transport layers are defined as standard by ASAM:

- XCP on CAN
- XCP on Sxl
- XCP on Ethernet (TCP/IP or UDP/IP)
- XCP on USB
- XCP on Flex Ray

To select the transport layer used by the AUTOSAR adaptive application (XCP Slave), use the model configuration parameter **Transport layer**. Selecting an XCP transport layer enables other XCP parameters.

For more information, see “Configure AUTOSAR Adaptive Data for Run-Time Calibration and Measurement” on page 6-80.

Specify XCP Slave IP Address

Internet Protocol (IP) is the principal communications protocol for relaying datagrams across network boundaries. The Internet Protocol is responsible for addressing host interfaces, encapsulating data into datagrams, and routing datagrams from a source host interface to a destination host interfaces across one or more IP networks.

Each datagram has two components: a header and a payload. The IP header includes source IP address, destination IP address, and other metadata needed to route and deliver the datagram. The payload is the data that it transported.

To specify the IP address of the machine on which the AUTOSAR adaptive application (XCP Slave) executes, use the model configuration parameter **IP address**. The **IP address** parameter is enabled by selecting a value for **Transport layer**.

For more information, see “Configure AUTOSAR Adaptive Data for Run-Time Calibration and Measurement” on page 6-80.

Specify XCP Slave Port

A port number is the logical address of each application or process that uses a network or the Internet to communicate. Port number primarily aids in the transmission of data between a network and an application. Port numbers work in collaboration with networking protocols to achieve this.

A port number uniquely identifies a network-based application on a computer. Each application is allocated a 16-bit integer port number. This number is assigned by the operating system, set manually by the user, or set as a default.

To specify the network port on which the AUTOSAR adaptive application (XCP Slave) serves XCP Master commands, use the model configuration parameter **Port**. The **Port** parameter is enabled by selecting a value for **Transport layer**.

For more information, see “Configure AUTOSAR Adaptive Data for Run-Time Calibration and Measurement” on page 6-80.

Enable XCP Slave Message Verbosity

Verbosity is the level of technical detail included in software messages. Verbose messages can help in debugging and understanding XCP communication.

To enable verbose messages for the AUTOSAR adaptive application (XCP Slave), select the model configuration parameter **Verbose**. The **Verbose** parameter is enabled by selecting a value for **Transport layer**.

For more information, see “Configure AUTOSAR Adaptive Data for Run-Time Calibration and Measurement” on page 6-80.

Use Custom XCP Slave

By default, the MathWorks XCP Slave is used for communication. You can use a custom XCP Slave for the Ethernet (TCP/IP) transport layer. A custom XCP Slave implementation is required to establish the interface. Define the implementation in header file `xcp_slave.h` in folder `matlabroot/toolbox/coder/autosar/adaptive`.

To enable use of a custom XCP Slave, select the model configuration parameter **Use custom XCP Slave**. The **Use custom XCP Slave** parameter is enabled by selecting a value for **Transport layer**.

For more information, see “Configure AUTOSAR Adaptive Data for Run-Time Calibration and Measurement” on page 6-80.

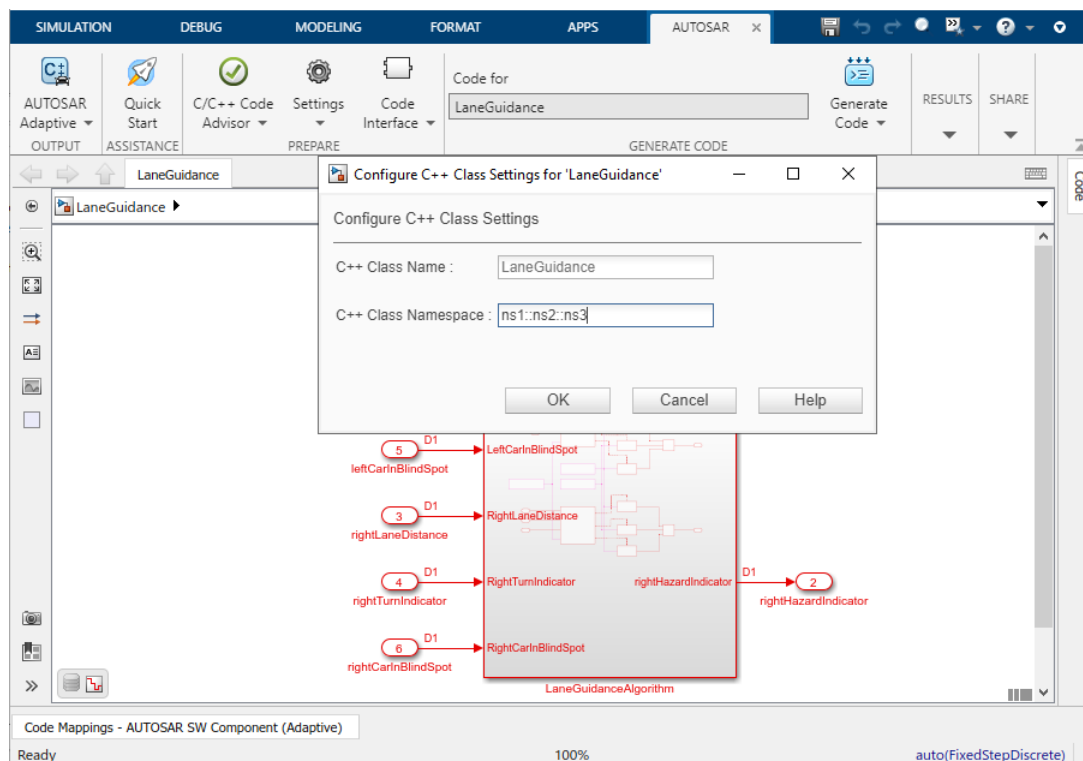
Inspect AUTOSAR Adaptive XML Options

Examine the XML options that you configured by using the AUTOSAR Dictionary. If you have not yet configured the options, see “Configure AUTOSAR Adaptive XML Options” on page 6-33.

Customize Class Name and Namespace in Generated Code

If you would like to customize the generated code, you can control the generated C++ class name and namespace for your AUTOSAR applications either interactively or programmatically.

To interactively configure these aspects of the generated code, from an open model, on the **AUTOSAR** tab, click **Code Interface**, select **Class Name & Namespace**, and customize the names in the opened configuration dialog box.



To programmatically configure the name and namespace, use the AUTOSAR functions `getClassName`, `setClassName`, `getClassNamespace`, and `setClassNamespace`.

Configure Run-Time Logging Behavior

Optionally, modify the `ara::log` based run-time logging behavior for the AUTOSAR adaptive application.

As defined in the AUTOSAR Specification of Diagnostic Log and Trace, adaptive applications can forward event logging information to a console, file, or network. This allows you to collate and analyze log data from multiple applications. By default, the application logs event messages to the local console.

To modify the default run-time logging behavior for an adaptive model, you use AUTOSAR property functions, including `set`. Code generation exports the specified logging properties to an ARXML execution manifest file. If you build a Linux® executable from the adaptive model, you can generate a JSON execution manifest file that modifies the default logging behavior for the executable. For more information, see “Configure Run-Time Logging for AUTOSAR Adaptive Executables” on page 6-89.

Generate AUTOSAR Adaptive C++ and XML Files

After configuring AUTOSAR code generation and XML options, generate code. To generate C++ code and export XML descriptions, build the adaptive component model.

The build process generates AUTOSAR-compliant C++ code and AUTOSAR XML descriptions to the model build folder. The exported XML files include:

- One or more *modelName*.arxml* files, based on whether you set **Exported XML file packaging** to `Single file` or `Modular`.
- Manifests for AUTOSAR executables and service instances.
- If you imported ARXML files into Simulink, updated versions of those files.

This table lists *modelName*.arxml* files that are generated based on the value of the **Exported XML file packaging** option configured in the AUTOSAR Dictionary.

Exported XML File Packaging Value	Exported File Name	Default Contents
Single file	<i>modelName.arxml</i>	AUTOSAR elements for adaptive software components, data types, and interfaces.
	<i>modelName_ExecutionManifest.arxml</i>	Deployment-related information for adaptive applications, including executables, process-to-machine mapping sets, and processes.
	<i>modelName_ServiceInstanceManifest.arxml</i>	Configuration of service-oriented communication, including service interface deployments, service instances, and service instance to port mappings.
Modular	<i>modelName_component.arxml</i>	Adaptive software components, including required and provided ports. This is the main ARXML file exported for the Simulink model. In addition to software components, the component file contains packageable elements that the exporter does not move to data type or interface files based on AUTOSAR element category.

Exported XML File Packaging Value	Exported File Name	Default Contents
	<i>modelname_datatype.arxml</i>	Data types and related elements, including: <ul style="list-style-type: none"> • Application data types • Standard Cpp implementation data types • Constant specifications • Physical data constraints • Units and unit groups • Software record layouts
	<i>modelname_interface.arxml</i>	Adaptive interfaces, including required and provided service interfaces with namespaces and events.
	<i>modelname_ExecutionManifest.arxml</i>	Deployment-related information for adaptive applications, including executables, process-to-machine mapping sets, and processes.
	<i>modelname_ServiceInstanceManifest.arxml</i>	Configuration of service-oriented communication, including service interface deployments, service instances, and service instance to port mappings.

You can merge the AUTOSAR adaptive XML component descriptions into an AUTOSAR authoring tool. The AUTOSAR component information is partitioned into separate files to facilitate merging. The partitioning attempts to minimize the number of merges that you must do. You do not need to merge the data type file into the authoring tool because data types are defined early in the design process. You must merge the internal behavior file because this information is part of the model implementation.

To help support the round trip of AUTOSAR elements between an AUTOSAR authoring tool (AAT) and the Simulink Model-Based Design environment, the code generator preserves AUTOSAR elements and their universal unique identifiers (UUIDs) across ARXML import and export. For more information, see “Round-Trip Preservation of AUTOSAR XML File Structure and Element Information” on page 3-37.

For an example of how to generate AUTOSAR-compliant C++ code and export AUTOSAR XML component descriptions from a Simulink model, see “Generate AUTOSAR Adaptive C++ Code and XML Descriptions” on page 6-68.

See Also

`autosar.api.getSimulinkMapping` | `getClassName` | `setClassName` | `getClassNamespace` | `setClassNamespace` | **Generate XML file for schema version** | **Maximum SHORT-NAME length** | **Transport layer** | **IP address** | **Port** | **Verbose** | **Use custom XCP Slave**

Related Examples

- “Configure AUTOSAR Adaptive XML Options” on page 6-33
- “Configure Run-Time Logging for AUTOSAR Adaptive Executables” on page 6-89
- “Generate AUTOSAR Adaptive C++ Code and XML Descriptions” on page 6-68

More About

- “Round-Trip Preservation of AUTOSAR XML File Structure and Element Information” on page 3-37

Configure AUTOSAR Adaptive Data for Run-Time Calibration and Measurement

AUTOSAR Blockset enables you to configure run-time calibration of adaptive application data based on XCP slave communication and ASAP2 (A2L) file generation. The XCP and ASAP2 capabilities are defined outside the Adaptive Platform (AP) specification, which as of Release 19-11 does not address data calibration.

As part of generating and deploying adaptive code, you can configure interfaces for XCP slave communication in the generated C++ code and export A2L files containing model data for calibration and measurement.

Before deploying adaptive code:

- In the Configuration Parameters dialog box, configure the model to generate XCP slave function calls in adaptive C++ code and to generate an XCP section in an ASAP2 (A2L) file.
- In the **Share** gallery under the **AUTOSAR** tab of the model, use **Generate Calibration Files** to generate ASAP2 (A2L) files that contain model data for calibration and measurement.

Configure XCP Communication Interface in Generated Code

To enable the communication capability, use the AUTOSAR adaptive model configuration parameter **Transport layer** to select an XCP transport layer. When **Transport layer** is set to a value other than **None**, Simulink adds XCP slave function calls to the generated C++ code. By default, the tool uses the MathWorks XCP Slave stack.

Selecting an XCP transport layer enables other XCP parameters. This image shows the XCP slave model configuration parameters.

The image shows a dialog box titled "XCP Slave Configuration". It contains the following fields and options:

- Transport layer:** A dropdown menu with "XCP On TCP/IP" selected.
- IP address:** A text input field containing "127.0.0.1".
- Port:** A text input field containing "17725".
- Verbose**
- Use custom XCP Slave**

Using the model configuration parameters, you can:

- Specify the transport layer that you want to use for communication.
- Specify the target machine IP address and port number. You can use the port for only one application.
- Optionally, enable verbose messages for XCP slave.
- Optionally, instead of the MathWorks XCP slave, you can use a custom XCP slave implementation based on the Ethernet transport layer. To use a custom XCP slave, provide implementations for the functions declared in the XCP slave header file by using the custom XCP slave API commands. The XCP slave header file is located in the MATLAB installation folder `matlabroot/toolbox/coder/autosar/adaptive_deployment/include`.

- Add custom XCP slave details in the configuration parameter **Toolchain details** or add the details manually to the `CMakeLists.txt` file.

The screenshot shows the 'Build process' configuration window. The left sidebar has 'Code Generation' selected. The main window is titled 'Build process' and contains the following sections:

- Generate code only:** A checked checkbox and a 'Zip file name' field set to '<empty>'. There is also an unchecked checkbox for 'Package code and artifacts'.
- Toolchain settings:** A 'Toolchain' dropdown menu set to 'AUTOSAR Adaptive | CMake' and a 'Build configuration' dropdown menu set to 'Faster Builds'.
- Toolchain details:** A collapsed section that is expanded to show a table with the following data:

Tool	Options
Build Type	Release
Required Packages	
Include Directories	
Link Libraries	
Library Paths	
Defines	

To generate ASAP2 (A2L) files, use **Generate Calibration Files** from the **Share** gallery under the **AUTOSAR** tab of the model. For more information, see “Generate ASAP2 and CDF Calibration Files” (Simulink Coder).

See Also

`coder.asap2.export` | **Transport layer**

Configure AUTOSAR Adaptive Model for External Mode Simulation

Embedded Coder Support Package for Linux Applications enables you to configure an AUTOSAR adaptive model for external mode simulation using XCP communication channel. Install the support package from **Add-Ons** button from MATLAB home. For more information about the support package, see “Support Package Installation” (Embedded Coder).

Steps to enable external mode simulation:

- 1 Open your AUTOSAR adaptive model.
- 2 Set the **Hardware board** configuration parameter in the **Hardware Implementation** to Embedded Coder Linux Docker Container.

Note Embedded Coder Linux Docker Container is only visible if the Embedded Coder Support Package for Linux Applications is installed.

- 3 Set the **Transport layer** parameter in the **XCP Slave Configuration** pane of the **Code Generation** > **AUTOSAR Code Generation** to XCP On TCP/IP.
- 4 Specify the IP address of the target machine and port number for XCP communication in the **Code Generation** > **AUTOSAR Code Generation** > **XCP Slave Configuration** section.
- 5 Enable the **External mode** option in the **Interface** pane of **Code Generation** section.
- 6 To log signals, select the signals you want to log and click the **Log Signals** in the **Prepare** pane of the **Simulation** section.
- 7 Open the **Linux Runtime Manager** from the Simulink apps gallery. Alternatively, enter this command at the MATLAB Command Window.


```
linux.RuntimeManager.open
```
- 8 Deploy the model on the Linux target machine. See “Build Simulink Model and Deploy Application” (Embedded Coder) for more information.
- 9 Select the model on the **Linux Runtime Manager** and click **Monitor & Tune** button. This will start the application and external mode simulation.

See Also

Linux Runtime Manager

Related Examples

- “Build Simulink Model and Deploy Application” (Embedded Coder)
- “Get Started with Embedded Coder Support Package for Linux Applications” on page 6-92
- “Event Communication Between AUTOSAR Adaptive Applications Using Message Polling” on page 6-96
- “Event Communication Between AUTOSAR Adaptive Applications Using Message Triggering” on page 6-100

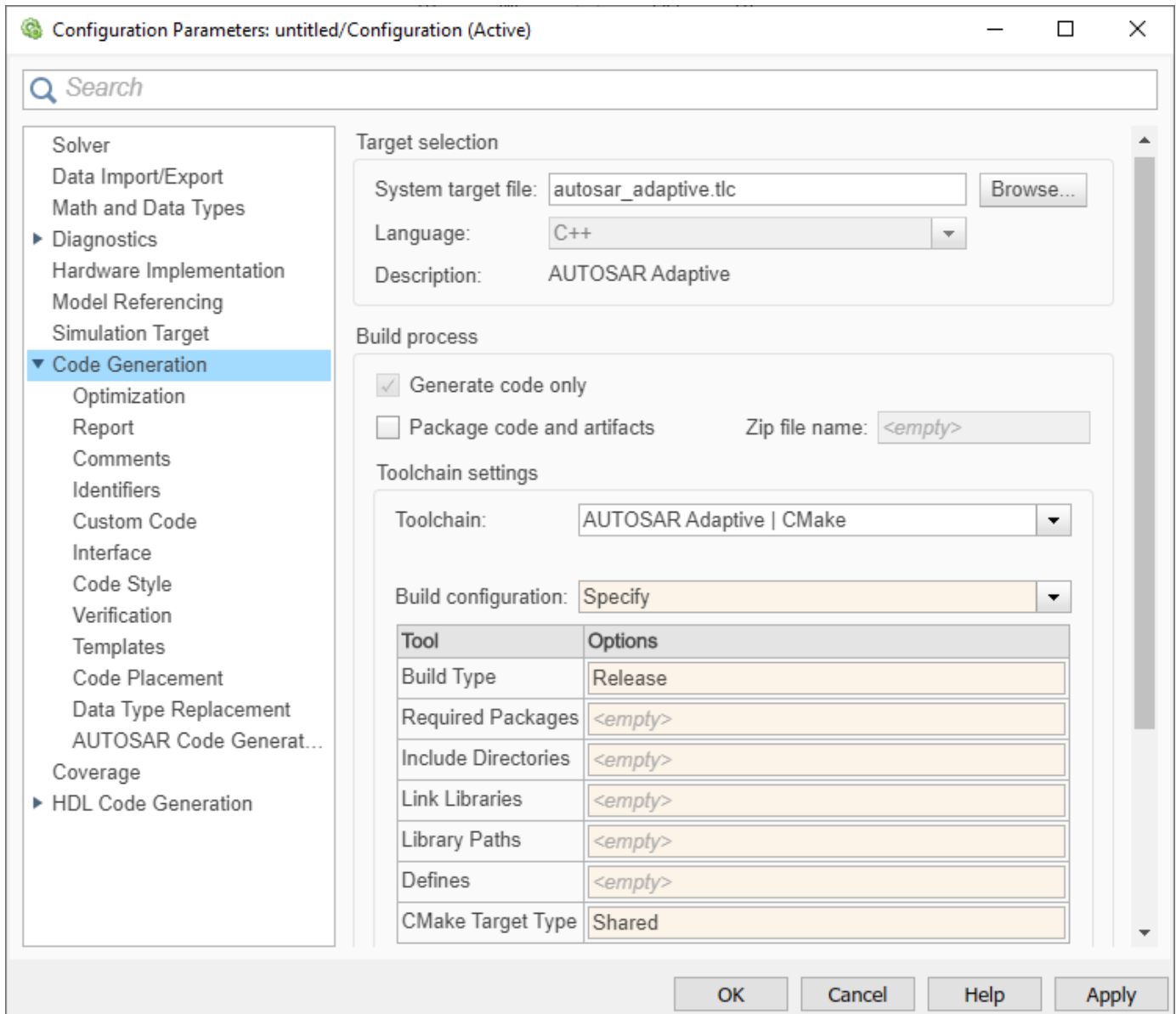
Build Library or Executable from AUTOSAR Adaptive Model

As part of generating code for an AUTOSAR adaptive model, you can generate a `CMakeLists.txt` file for building a static or shared library or an executable. The AUTOSAR Adaptive | CMake toolchain generates the `CMakeLists.txt` file following modular CMake patterns. You can link the resulting library against a `main.cpp` file or combine it with other model files in an integration environment.

Building the library file from `CMakeLists.txt` requires running CMake software.

To build a static or shared library:

- 1 Open a component model that is configured for the AUTOSAR adaptive target (`autosar_adaptive.tlc`).
- 2 Open the **Configuration Parameters** dialog box and select **Code Generation**. Under **Toolchain settings**:
 - a Set **Toolchain** to AUTOSAR Adaptive | CMake.
 - b Set **Build configuration** to Specify.
 - c Set **CMake Target Type** to Static (for a static library) or Shared (for a shared library).
 - d In the **Include Directories**, **Link Libraries**, and **Library Paths** fields, specify libraries and header files that must be generated in `CMakeLists.txt` to support compilation. For example, set **Include Directories** to the string `${START_DIR}/modelName_autosar_adaptive/stub/aragen`, where `modelName` is the name of the adaptive model.
 - e Click **OK**.



- 3 Build the model. The build generates C++ code, ARXML files, and a CMakeLists.txt file.
- 4 In the model build folder, open CMakeLists.txt and verify that it is configured for static or shared library generation. For example, check that:
 - a The CMakeLists.txt file contains


```
add_library(modelName SHARED...) % for shared library
```

 or


```
add_library(modelName STATIC...) % for static library
```
 - b The specifications for `target_include_directories`, `target_link_libraries`, and `link_directories` include the values specified in **Toolchain settings**.
- 5 Go to the model build folder outside MATLAB. To build the static or shared library file, enter these commands:

```
cmake CMakeLists.txt;  
make all;
```

The make generates a library file for the adaptive model (for example, `modelName.a` or `modelName.so`) in the model build folder. You can link the library against a `main.cpp` file or combine it with other model files in an integration environment.

To build an executable, do one of the following:

- Use **AUTOSAR Adaptive | CMake** toolchain. Follow the same procedure as for libraries, but set **CMake Target Type** to Executable.
- To generate a standalone executable, use the **AUTOSAR Adaptive Linux Executable** toolchain. For more information, see “Build Out of the Box Linux Executable from AUTOSAR Adaptive Model” on page 6-86.

See Also

More About

- “Build Out of the Box Linux Executable from AUTOSAR Adaptive Model” on page 6-86
- “Configure AUTOSAR Adaptive Data for Run-Time Calibration and Measurement” on page 6-80

Build Out of the Box Linux Executable from AUTOSAR Adaptive Model

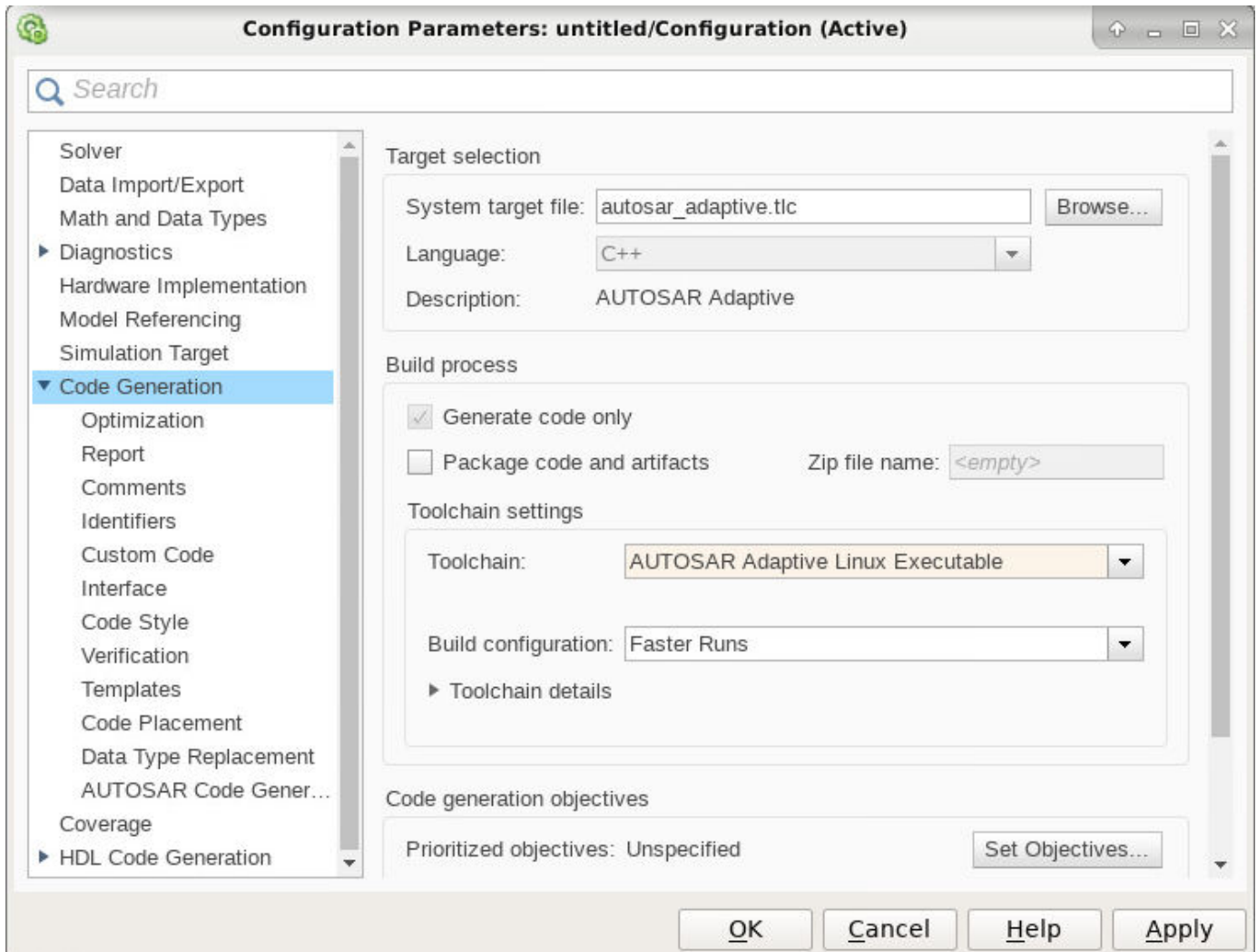
As part of generating code for an AUTOSAR adaptive model, you can generate a `CMakeLists.txt` file for building a Linux standalone executable. Then, on a Linux system, you can build the executable and run the resulting executable on Linux as a standalone application.

If the applications have matching DDS deployment artifacts, they can communicate with each other. Building the executable files from `CMakeLists.txt` requires running CMake software on a Linux system.

Note Executable generation from an AUTOSAR adaptive model is supported only on the Linux platform.

To build a Linux standalone executable:

- 1 Open a component model that you have configured for the AUTOSAR adaptive target (`autosar_adaptive.tlc`).
- 2 In the Configuration Parameters dialog box, select **Code Generation > Build process > Toolchain settings**. Set **Toolchain** to **AUTOSAR Adaptive Linux Executable**. The toolchain selection adds ARA functional cluster libraries provided by MathWorks.



Note The AUTOSAR Adaptive Linux Executable toolchain is supported only if the Embedded Coder Support Package for Linux Applications is installed. For more information, see “Support Package Installation” (Embedded Coder).

- 3 Build the model. The build generates C++ code, ARXML files, and a CMakeLists.txt file.
- 4 In the model build folder, open CMakeLists.txt and verify that it is configured for executable generation. For example, make sure that:
 - a The CMakeLists.txt file contains `add_executable(modelName ...)`.
 - b The specifications for `target_include_directories`, `target_link_libraries`, and `link_directories` include the values specified in **Toolchain settings**.
- 5 Verify the DDS deployment artifacts DDS Topic Name and DDS Domain ID from the generated ServiceInstanceManifest.arxml file.

Clear and re-create the mapping for the model with existing mappings (models created by using a MATLAB version prior to 22a), to have DDS binding as default. Otherwise, the model continues using user-defined bindings. To re-create the mapping, use this command:

```
autosar.api.create(<modelName>, 'default');
```

- 6 Get the support package root directory path using the below command in MATLAB:

```
path = matlabshared.supportpkg.getSupportPackageRoot
```

Copy the path to use it in the next step.

- 7 On a Linux system, outside MATLAB, go to the model build folder. To build the executable file, enter these commands:

```
cmake -DSPKG_ROOT=<path from step 6> CMakeLists.txt;  
make all;
```

The make generates an executable file for the adaptive model one level above the model build folder. You can run the executable on Linux as a standalone application.

Adaptive applications having event deployment artifacts with the same TOPIC - NAME and DOMAIN - ID can communicate with each other.

See Also

More About

- “Configure AUTOSAR Adaptive Code Generation” on page 6-73
- “Configure AUTOSAR Adaptive Data for Run-Time Calibration and Measurement” on page 6-80

Configure Run-Time Logging for AUTOSAR Adaptive Executables

As defined in the AUTOSAR Specification of Diagnostic Log and Trace, adaptive applications can forward event logging information to a console, a file, or network. This allows you to collate and analyze log data from multiple applications. By default, the application logs event messages to the local console. To view the log data from a file or network, you can use third-party tools.

To modify the default run-time logging behavior for an adaptive model, use AUTOSAR property functions, including `set`. Code generation exports the specified logging properties to an ARXML execution manifest file. The manifest file is used to configure aspects of the run-time behavior of the adaptive application Linux executable, such as the logging mode and verbosity level.

If you build a Linux executable from the adaptive model, you can use the AUTOSAR property function `createManifest` to generate a JSON execution manifest file. The JSON file modifies the default logging behavior for the executable. You can generate the JSON execution manifest file after you build the Linux executable. Before you run the Linux executable, verify that the JSON execution manifest file and executable file are in the same folder.

In this section...

“Logging to Console” on page 6-89

“Logging to File” on page 6-90

“Logging to Network” on page 6-90

Logging to Console

- 1 Open the AUTOSAR adaptive model.
- 2 Use AUTOSAR property functions to set the AUTOSAR property `LogMode` to Console:

```
apiObj = autosar.api.getAUTOSARProperties(modelName);
processPath = find(apiObj, '/', 'Process', 'PathType', 'FullyQualified');
set(apiObj, processPath{1}, 'LogTraceLogMode', 'Console');
```

- 3 Optionally, set the logging verbosity level to Verbose.

```
set(apiObj, processPath{1}, 'LogTraceDefaultLogLevel', 'Verbose');
```

- 4 Generate code and ARXML files for the model. The build generates the logging properties into the file `modelName_ExecutionManifest.arxml`.
- 5 If you intend to build and run a Linux standalone executable for the adaptive model, use the `createManifest` function to generate the manifest file `ExecutionManifest.json` in the current working folder.

```
createManifest(apiObj);
```

- 6 Before you run the Linux executable, verify that the JSON execution manifest file and executable file are in the same folder.
- 7 Execute the application and see the log messages appear on the console.

Logging to File

- 1 Open the AUTOSAR adaptive model.
- 2 Use AUTOSAR property functions to set the AUTOSAR property `LogMode` to `File`:

```
apiObj = autosar.api.getAUTOSARProperties(modelName);  
processPath = find(apiObj, '/', 'Process', 'PathType', 'FullyQualified');  
set(apiObj, processPath{1}, 'LogTraceLogMode', 'File');
```
- 3 Optionally, specify the path to the log file. By default the log file will be saved in the executable folder.

```
set(apiObj, processPath{1}, 'LogTraceFilePath', 'customFilePath');
```

- 4 Optionally, set the logging verbosity level to `Verbose`.

```
set(apiObj, processPath{1}, 'LogTraceDefaultLogLevel', 'Verbose');
```
- 5 Generate code and ARXML files for the model. The build generates the logging properties into the file `modelName_ExecutionManifest.arxml`.
- 6 If you intend to build and run a Linux standalone executable for the adaptive model, use the `createManifest` function to generate the manifest file `ExecutionManifest.json` in the current working folder.

```
createManifest(apiObj);
```
- 7 Before you run the Linux executable, verify that the JSON execution manifest file and executable file are in the same folder.
- 8 Execute the application and verify that the log file is created at the specified or default location.

Logging to Network

- 1 Open the AUTOSAR adaptive model.
- 2 Use AUTOSAR property functions to set the AUTOSAR property `LogMode` to `Network`:

```
apiObj = autosar.api.getAUTOSARProperties(modelName);  
processPath = find(apiObj, '/', 'Process', 'PathType', 'FullyQualified');  
set(apiObj, processPath{1}, 'LogTraceLogMode', 'Network');
```
- 3 Optionally, set the logging verbosity level to `Verbose`.

```
set(apiObj, processPath{1}, 'LogTraceDefaultLogLevel', 'Verbose');
```
- 4 Generate code and ARXML files for the model. The build generates the logging properties into the file `modelName_ExecutionManifest.arxml`.
- 5 Before you run the Linux executable, verify that the JSON execution manifest file and executable file are in the same folder.
- 6 Initialize the AUTOSAR run-time environment for adaptive applications with following command.

```
autosar.ara.initialize
```

Note The command `autosar.ara.initialize` will be removed in a future release. Use the Embedded Coder Support Package for Linux Applications instead. For more information, see “Support Package Installation” (Embedded Coder).

- 7 Execute the application and see the log messages appear on the network.

See Also

`createManifest`

Get Started with Embedded Coder Support Package for Linux Applications

This example shows how to deploy an application on a Linux target using Embedded Coder® Support Package For Linux® Applications.

You can use Embedded Coder Support Package for Linux Applications to generate code, create an executable program, run and stop the executable program in the target execution environment, and instrument the running application. You can also interact with multiple target computers at the same time.

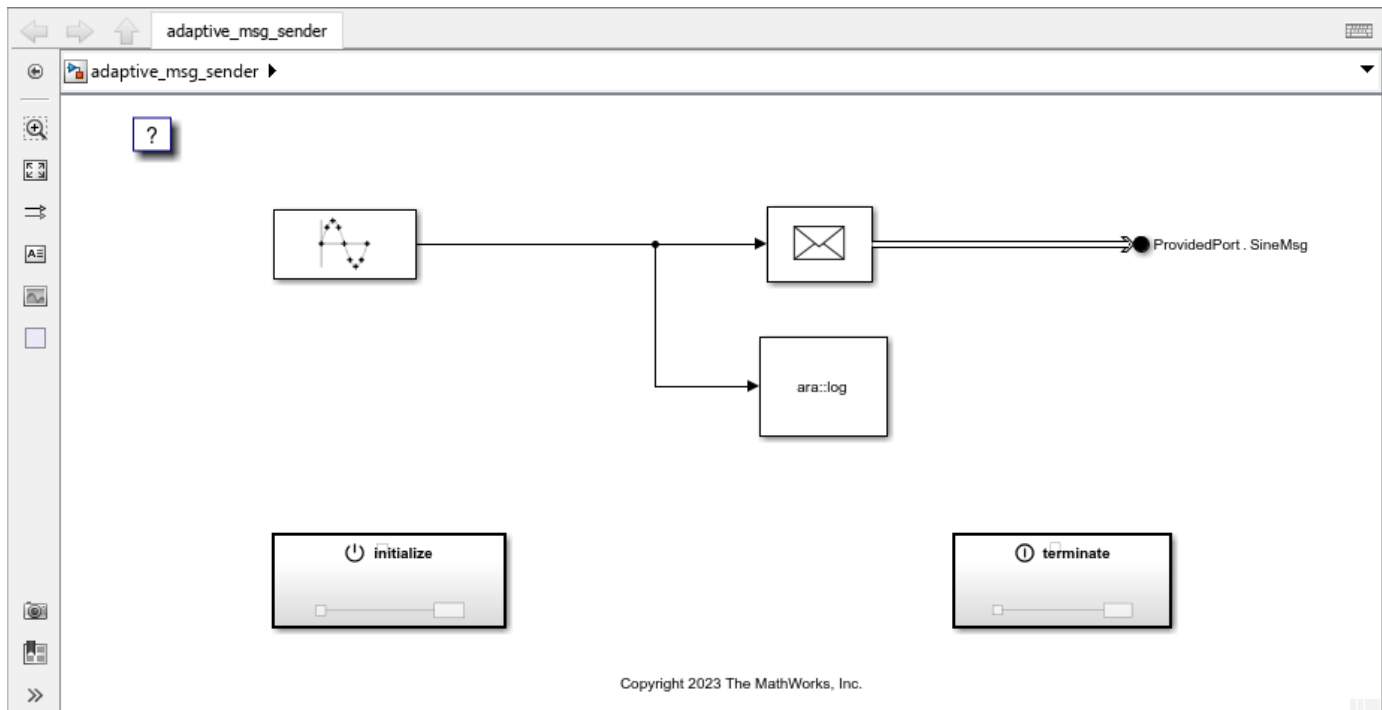
In this example, you deploy an AUTOSAR Adaptive application on a Linux target and control the life cycle of application.

Deploy Model

This example uses an AUTOSAR Adaptive model, which sends the data of a sine wave.

Open the model.

```
open_system('adaptive_msg_sender');
```



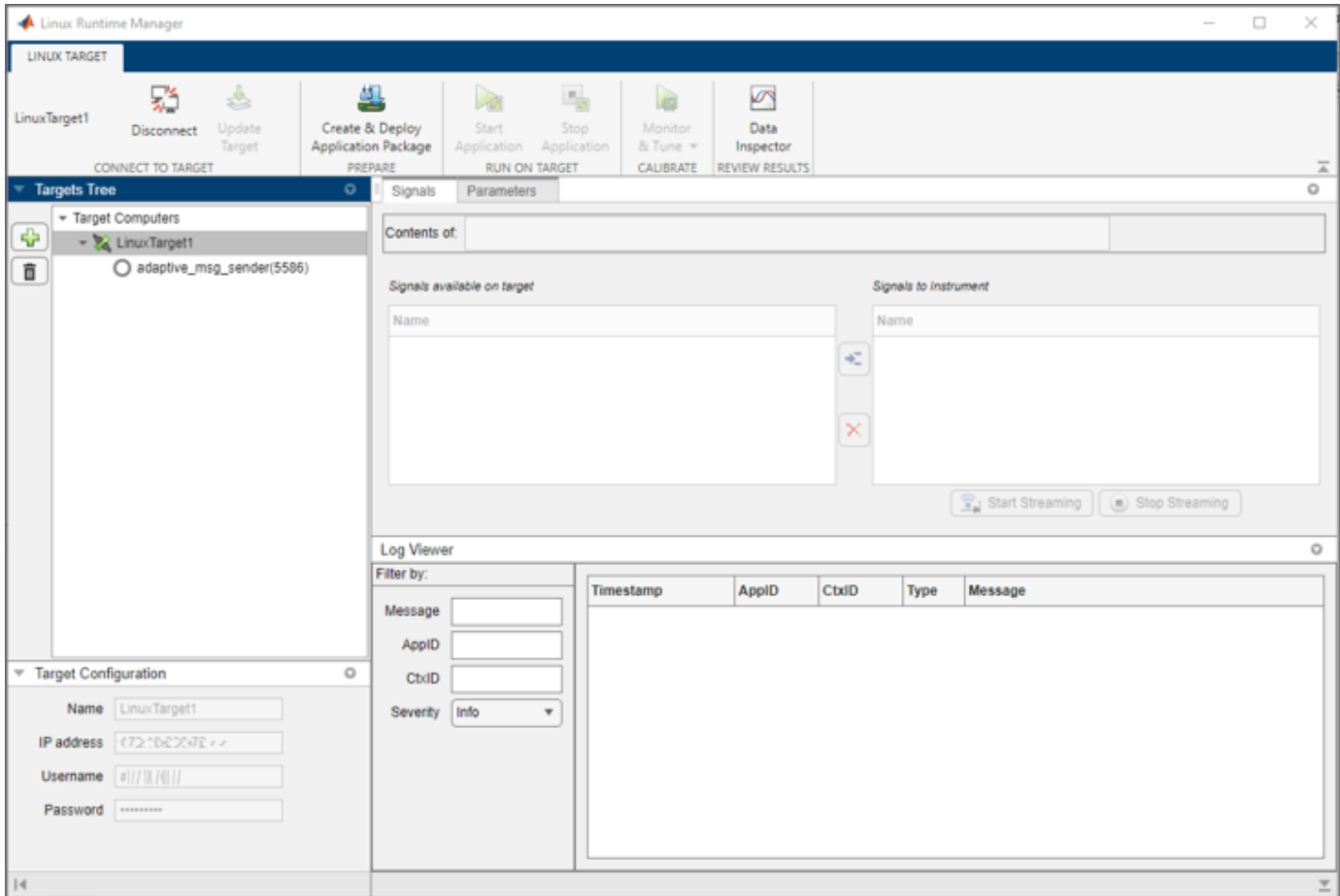
- Open the **Linux Runtime Manager** app

```
linux.RuntimeManager.open
```

- Connect to a Linux target computer by following the instructions in “Setup Linux Target Computer” (Embedded Coder).

- Deploy the model on the Linux target computer by following the instructions in “Build Simulink Model and Deploy Application” (Embedded Coder).

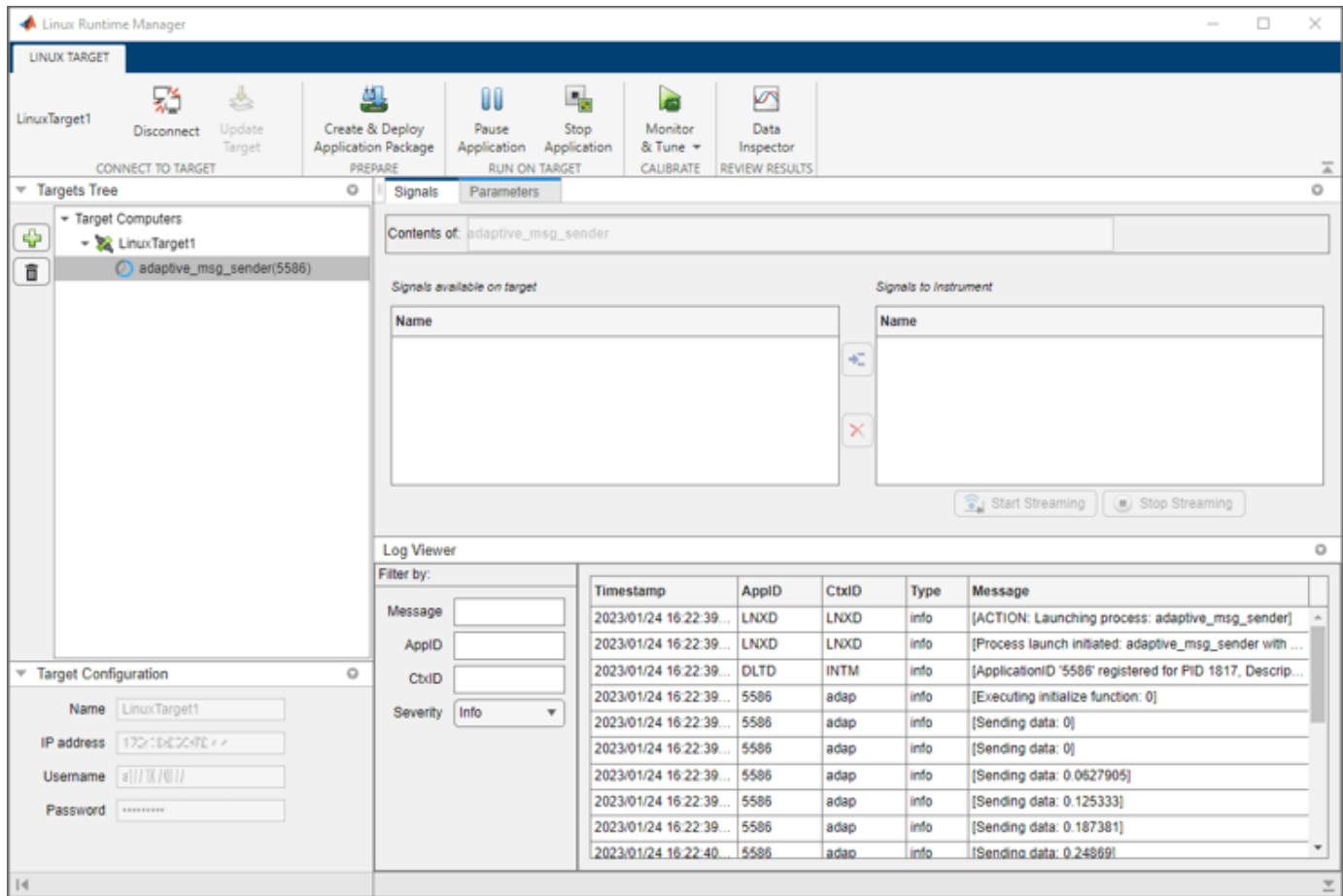
After you deploy the application, the **Linux Runtime Manager** app displays it in the **Targets Tree** pane.



Launch Application on Target

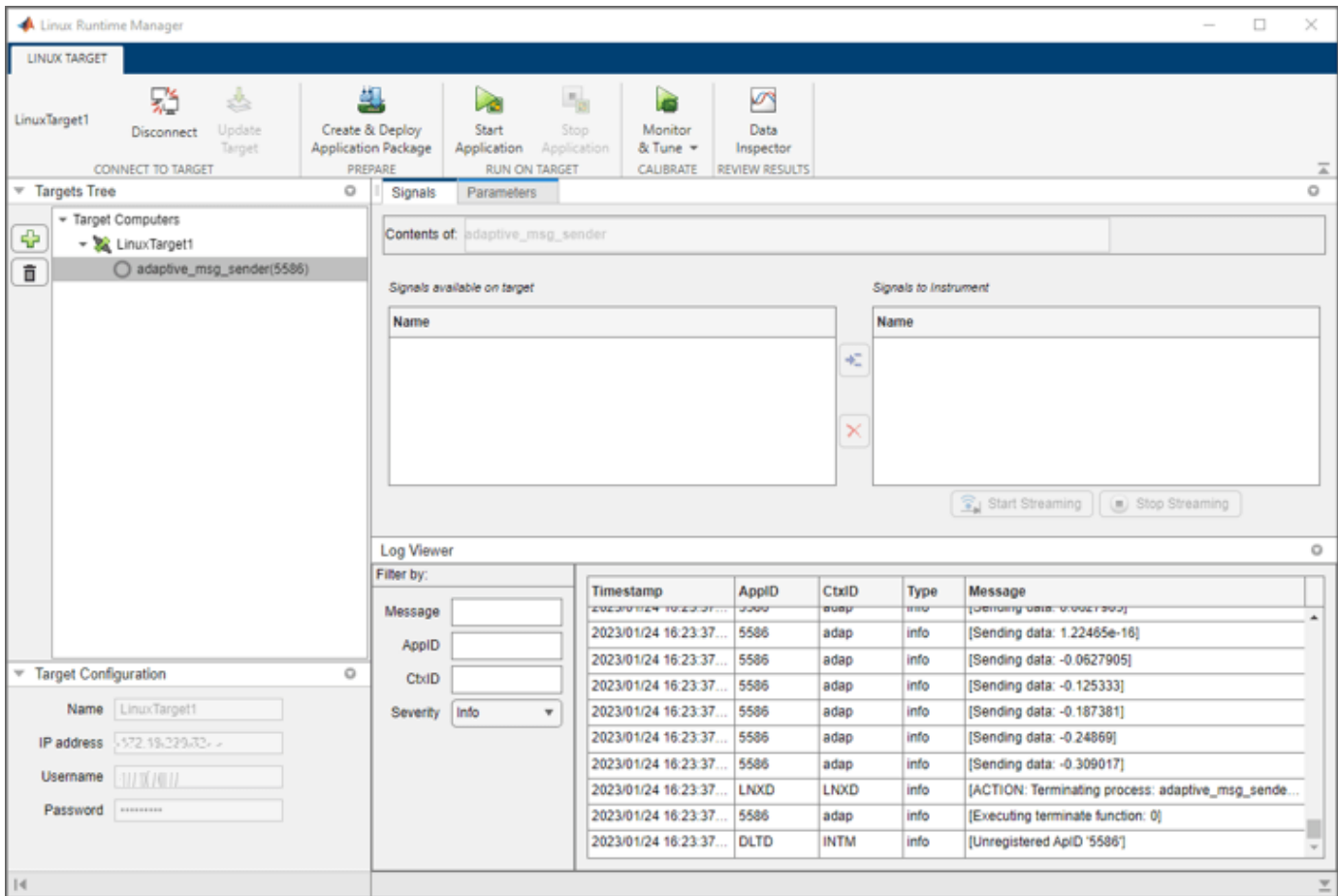
Select the application and click **Linux Target > Run On Target > Start Application** to launch the application on the target.

The app displays messages to indicate the execution of the initialization function and the data that **adaptive_msg_sender** sends in the **Log Viewer** pane.



Stop Application on Target

Stop the application by selecting it and clicking **Linux Target > Run On Target > Stop Application**. The app displays a message to indicate execution of the termination function in the **Log Viewer** pane.



See Also
[deployApplicationPackage](#)

Related Examples

- “Build Simulink Model and Deploy Application” (Embedded Coder)
- “Support Package Installation” (Embedded Coder)

Event Communication Between AUTOSAR Adaptive Applications Using Message Polling

This example shows how to deploy two AUTOSAR adaptive applications that use events to communicate with each other in message polling mode.

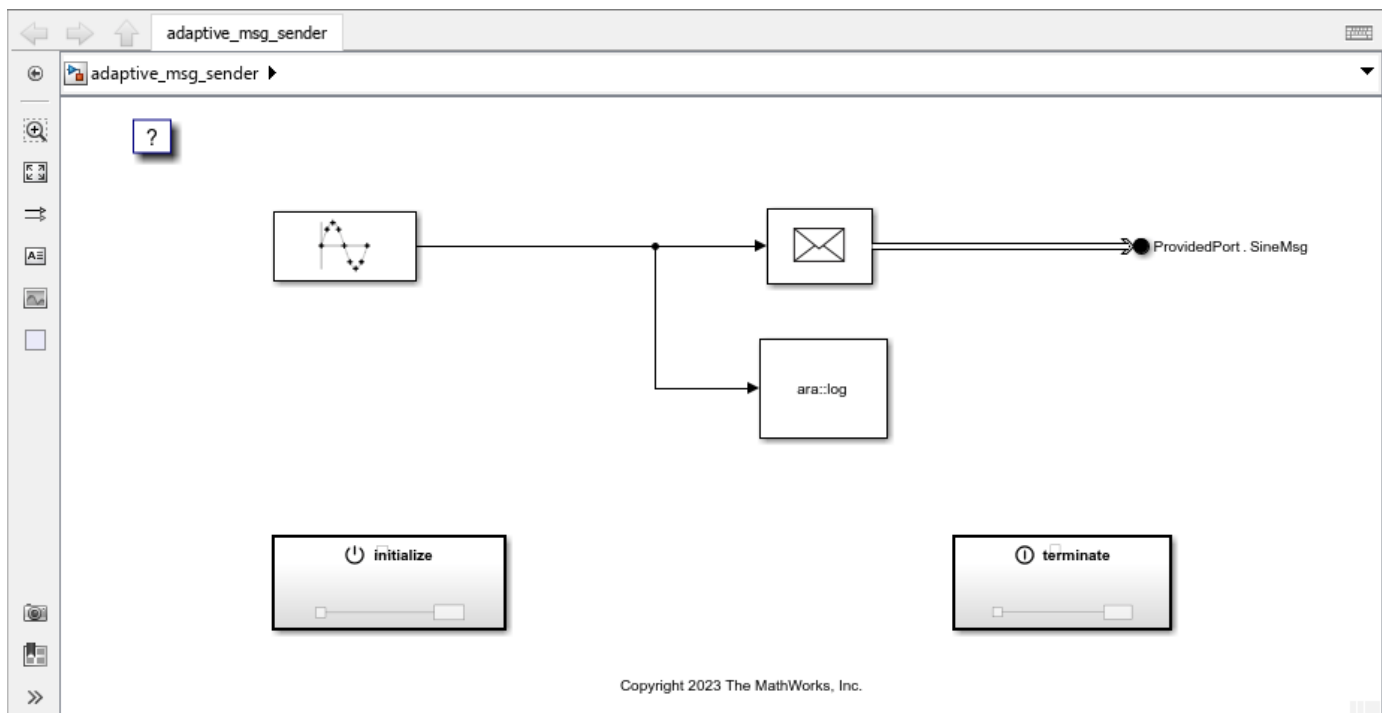
The sender application transmits the data of a sine wave periodically based on the sample time. The receiver application periodically checks the message availability from the other application in polling mode and receives messages when they are available.

Open the Models

This example the `adaptive_msg_sender` and `adaptive_msg_polling_receiver` AUTOSAR Adaptive models.

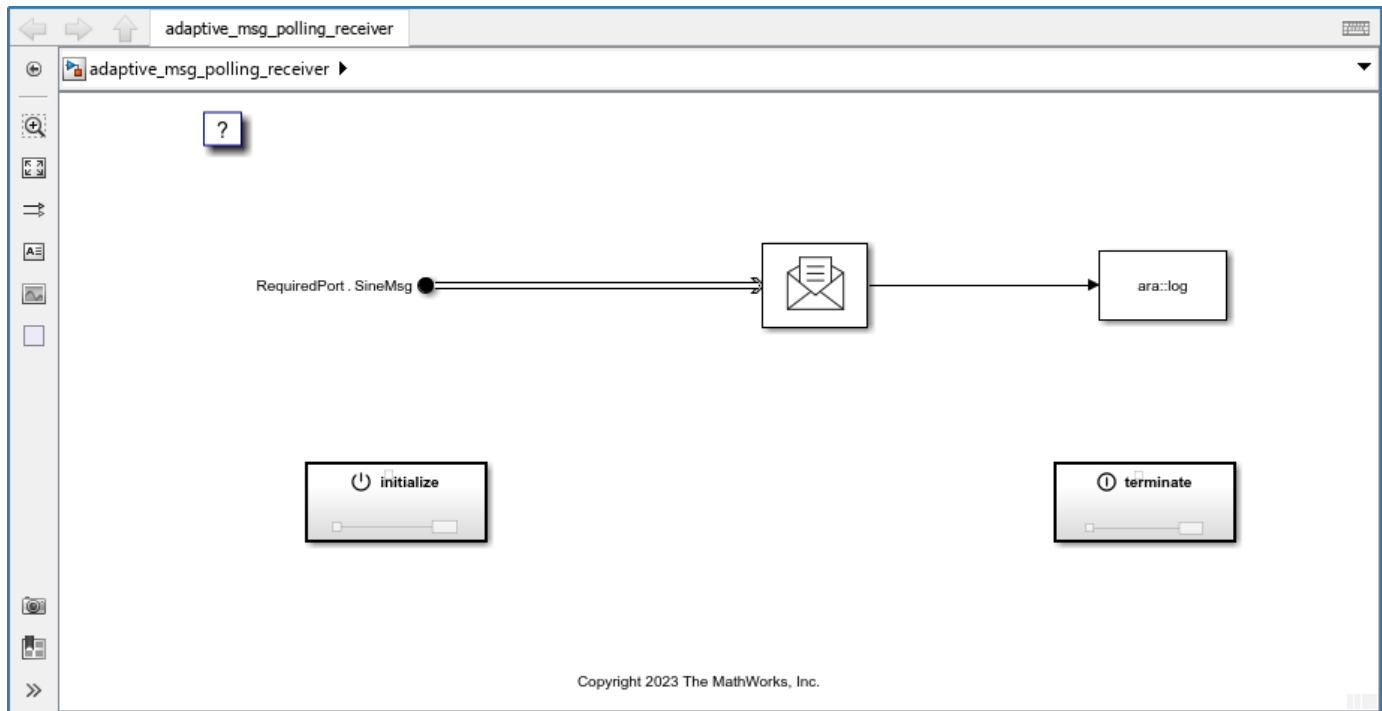
The `adaptive_msg_sender` model sends the data of a sine wave and uses an s-function block to log the data using `ara::log` messages.

```
open_system('adaptive_msg_sender');
```



The `adaptive_msg_polling_receiver` model receives the data of a sine wave and uses an s-function block to log the received data using `ara::log` messages.

```
open_system('adaptive_msg_polling_receiver');
```



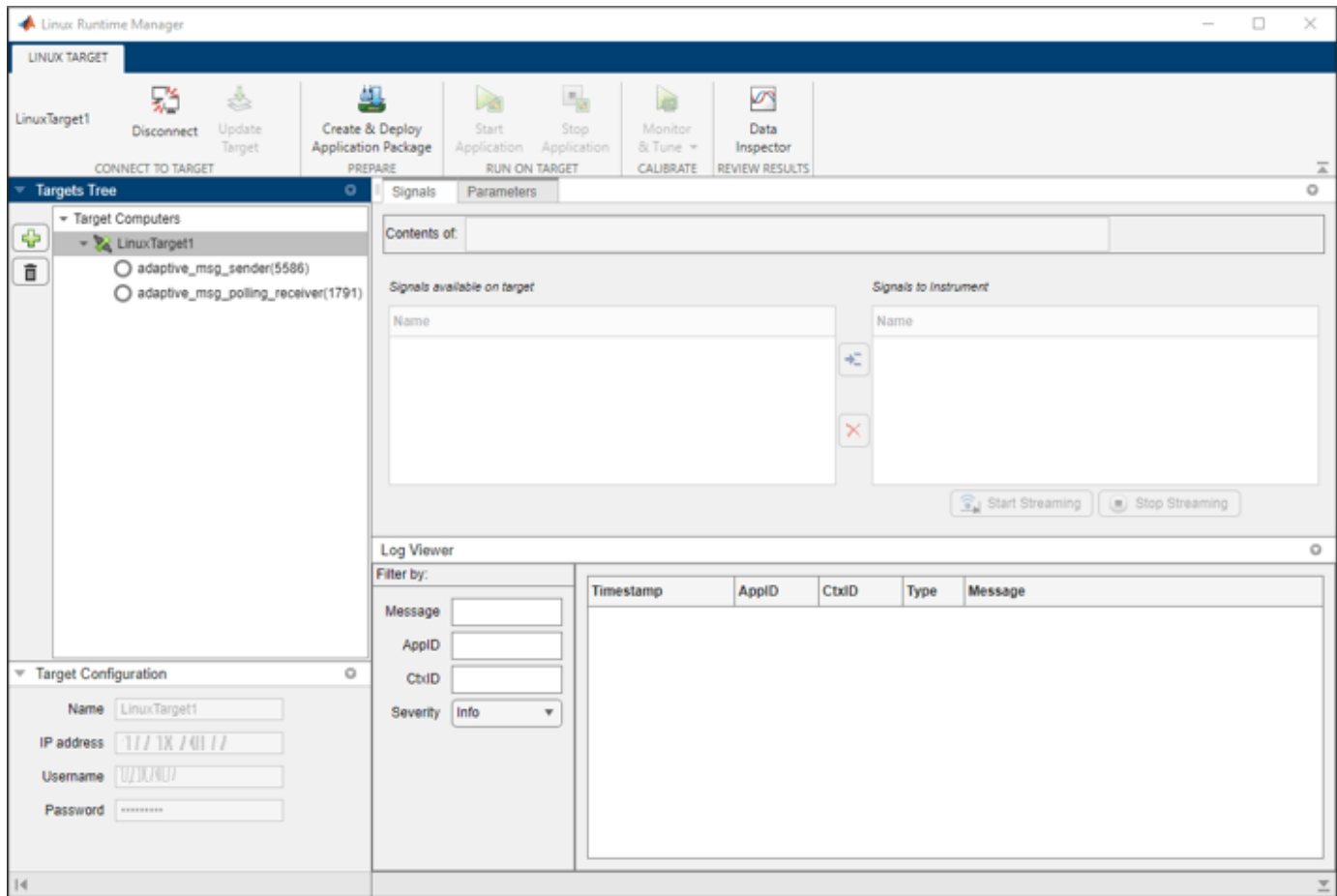
Deploy the Models

- Open **Linux Runtime Manager** app.

```
linux.RuntimeManager.open
```

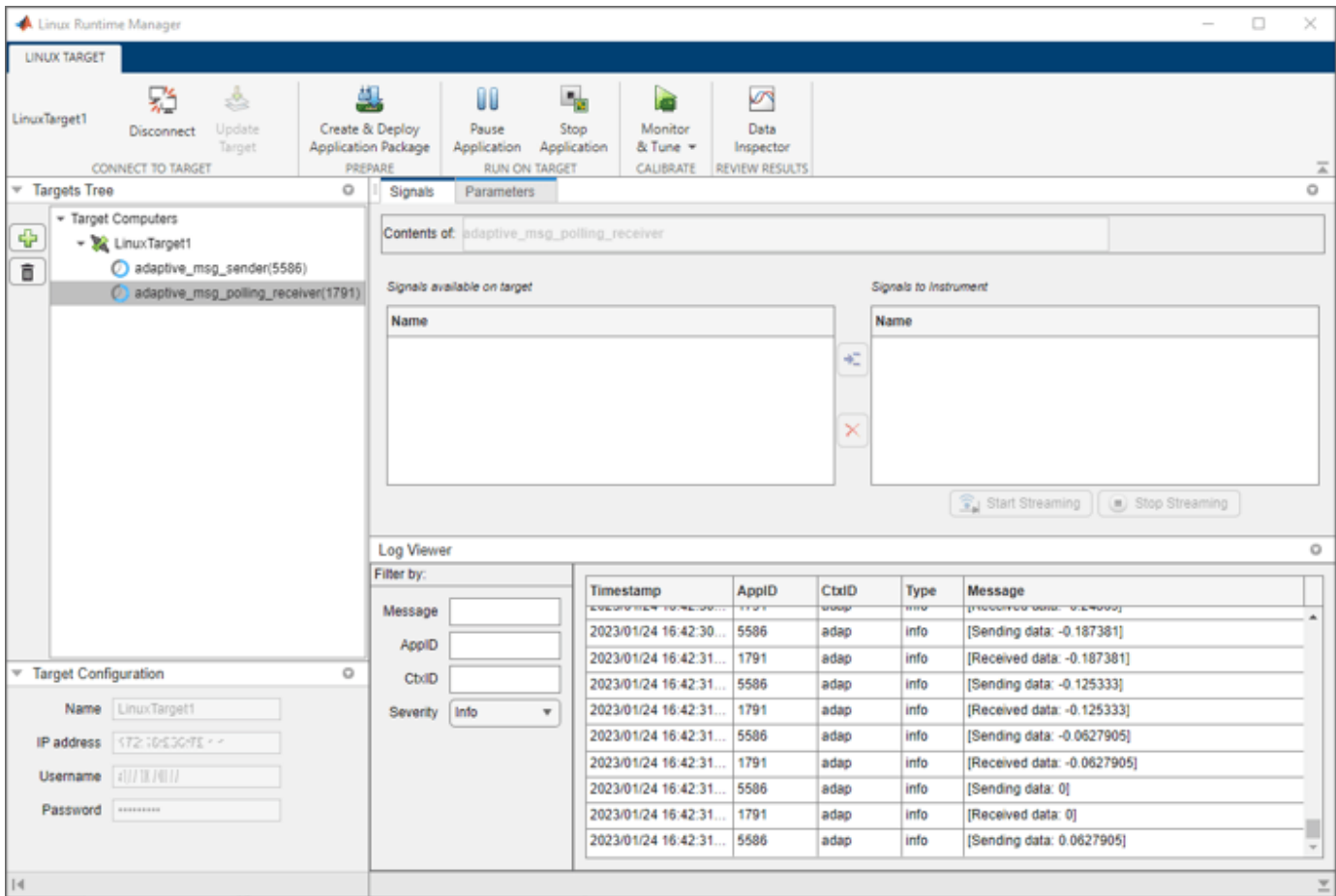
- Connect to a Linux target computer by following instructions in “Setup Linux Target Computer” (Embedded Coder).
- Deploy both the models on Linux target computer by following instructions in “Build Simulink Model and Deploy Application” (Embedded Coder).

After you deploy the applications, the **Linux Runtime Manager** app displays them in the **Targets Tree** pane.



- Select the `adaptive_msg_sender` application and click **Linux Target > Run On Target > Start Application** to launch the application on the target.
- Select the `adaptive_msg_polling_receiver` application and click **Linux Target > Run On Target > Start Application** to launch the application on the target.

The app displays `ara::log` messages that the deployed applications generate in the **Log Viewer** pane, which indicates that the applications are communicating.



- To stop the `adaptive_msg_sender` application, select it and click **Linux Target > Run On Target > Stop Application**.
- To stop the `adaptive_msg_polling_receiver` application, select it and click **Linux Target > Run On Target > Stop Application**.

See Also

`deployApplicationPackage`

Related Examples

- “Build Simulink Model and Deploy Application” (Embedded Coder)
- “Support Package Installation” (Embedded Coder)

Event Communication Between AUTOSAR Adaptive Applications Using Message Triggering

This example shows how to deploy two AUTOSAR adaptive applications that use events to communicate with each other in message triggering mode.

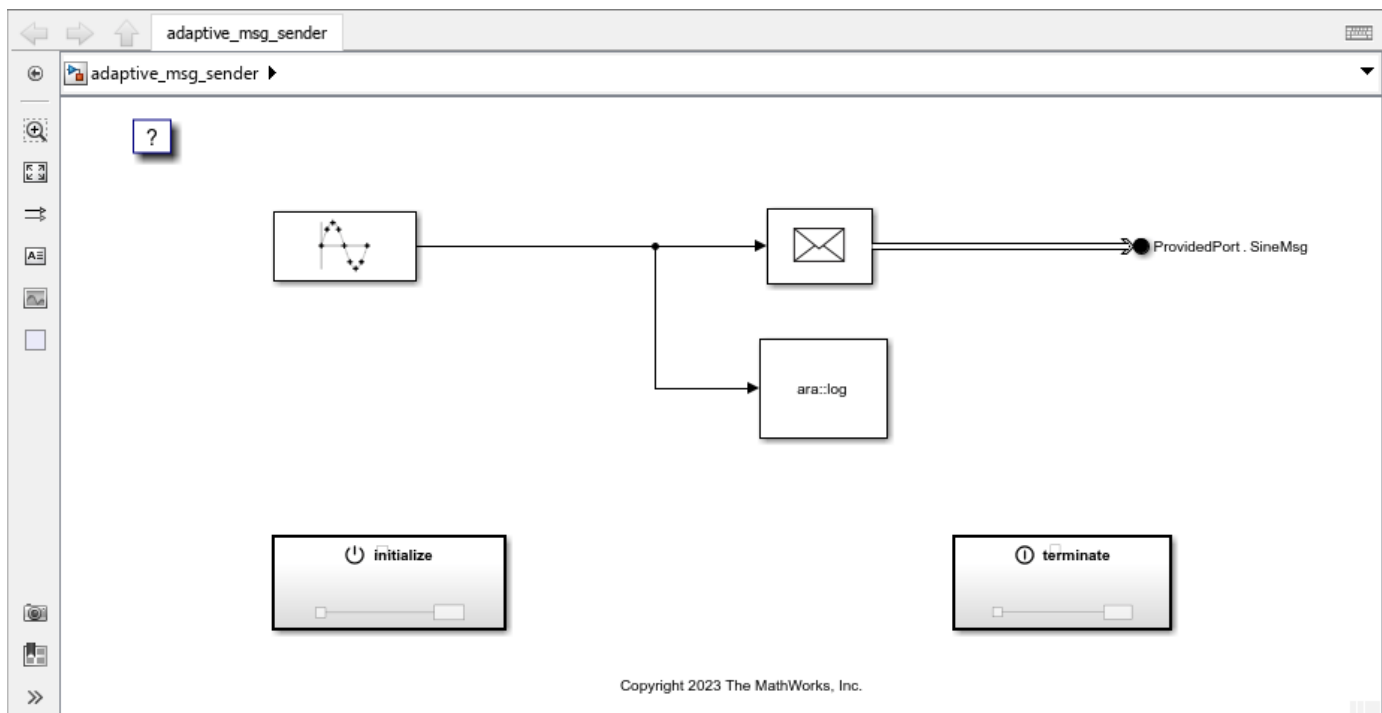
The sender application transmits the data of a sine wave periodically based on the sample time. The receiver application checks the message availability from the other application when ever the message triggering is invoked and receives messages when they are available.

Open the Models

This example uses the `adaptive_msg_sender` and `adaptive_msg_event_driven_receiver` AUTOSAR Adaptive models.

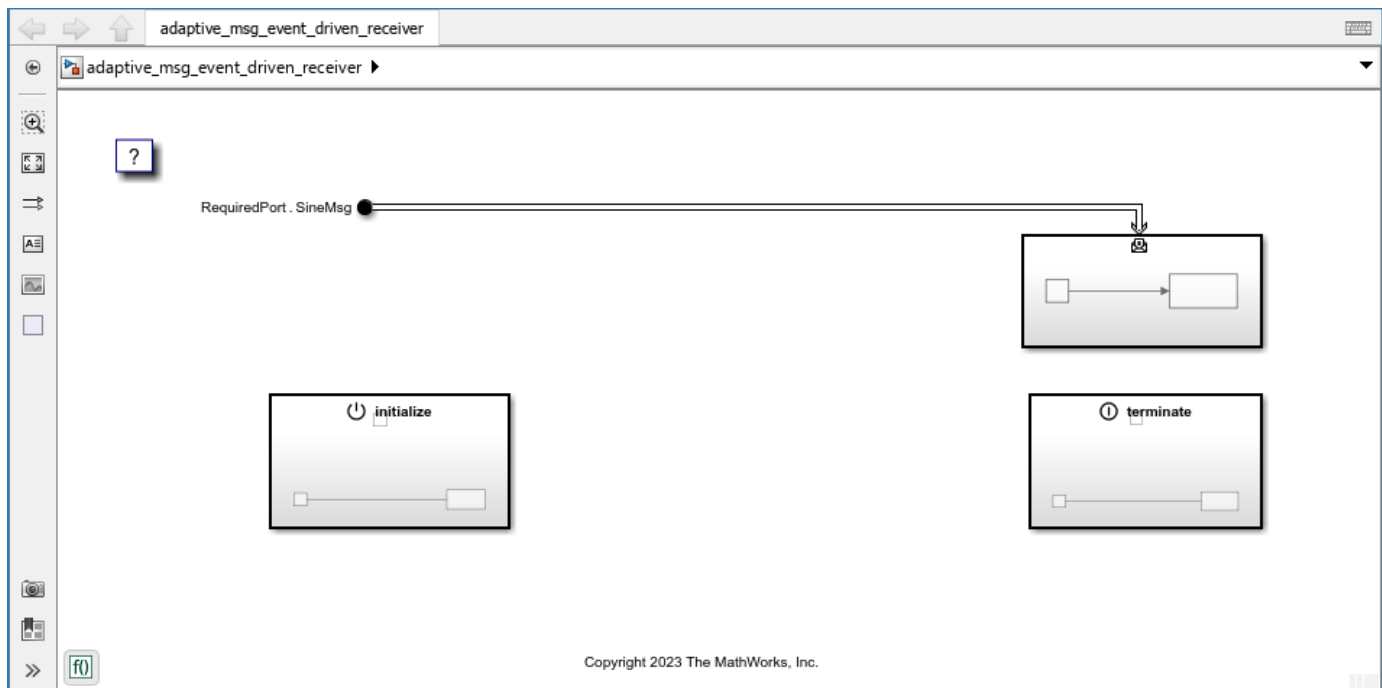
The `adaptive_msg_sender` model sends the data of a sine wave and uses an s-function block to log the data using `ara::log` messages.

```
open_system('adaptive_msg_sender');
```



The `adaptive_msg_event_driven_receiver` model receives the data of a sine wave and uses an s-function block to log the received data using `ara::log` messages.

```
open_system('adaptive_msg_event_driven_receiver');
```



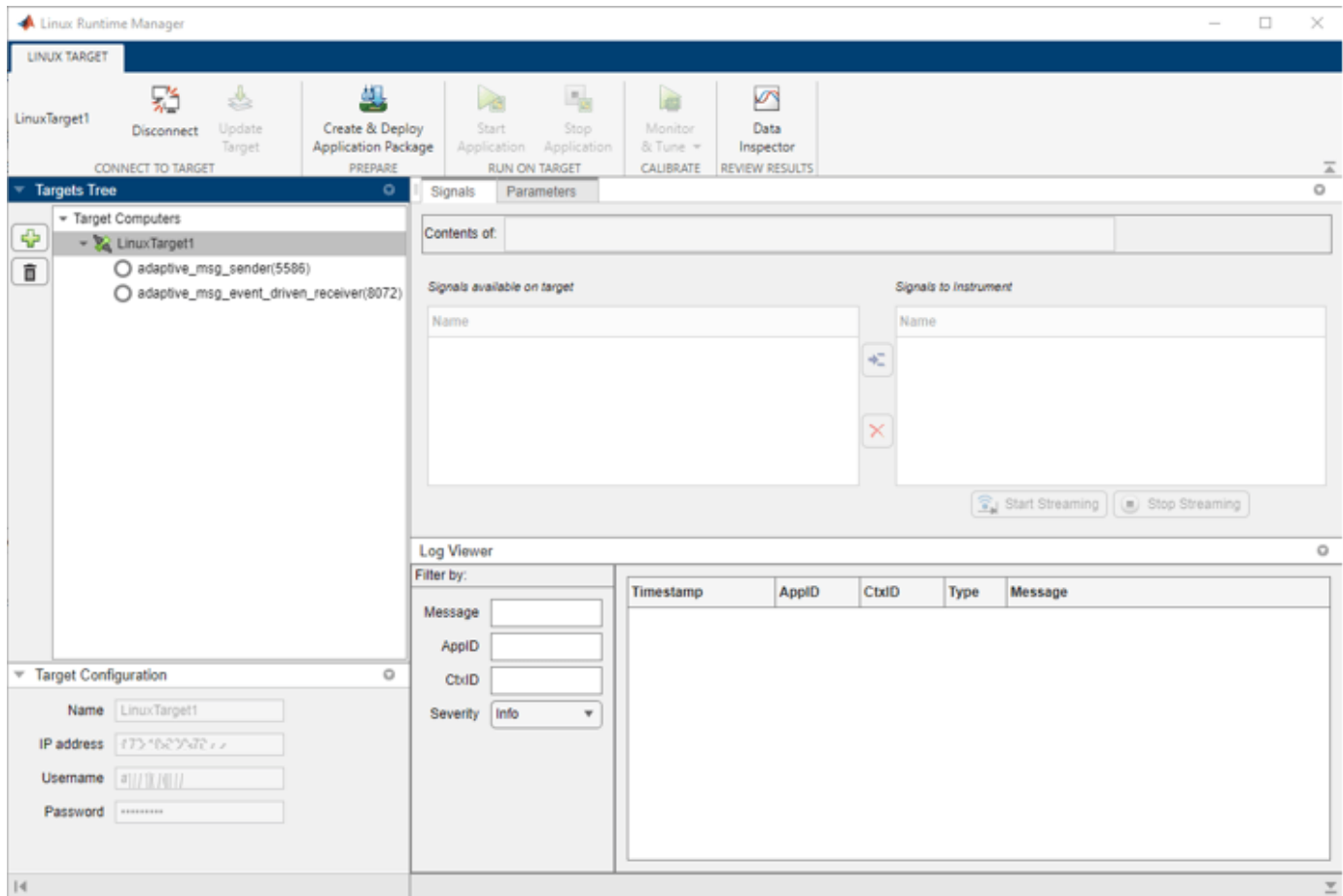
Deploy the Models

- Open **Linux Runtime Manager** app.

```
linux.RuntimeManager.open
```

- Connect to a Linux target computer by following instructions in “Setup Linux Target Computer” (Embedded Coder).
- Deploy both the models on Linux target computer by following instructions in “Build Simulink Model and Deploy Application” (Embedded Coder).

After you deploy the applications, the **Linux Runtime Manager** app displays them in the **Targets Tree** pane.



- Select the `adaptive_msg_sender` application and click **Linux Target > Run On Target > Start Application** to launch the application on the target.
- Select the `adaptive_msg_event_driven_receiver` application and click **Linux Target > Run On Target > Start Application** to launch the application on the target.

The app displays `ara::log` messages that the deployed applications generate in the **Log Viewer** pane, which indicates that the applications are communicating.

The screenshot displays the Linux Runtime Manager interface. The top toolbar contains several action buttons: Disconnect, Update Target, Create & Deploy Application Package, Pause Application, Stop Application, Monitor & Tune, and Data Inspector. Below the toolbar, the Targets Tree shows the configuration for LinuxTarget1, including the adaptive_msg_sender(5586) and adaptive_msg_event_driven_receiver(8072) applications. The Signals section shows the contents of the adaptive_msg_event_driven_receiver and the signals available on the target. The Log Viewer shows a list of messages with columns for Timestamp, AppID, CtxID, Type, and Message.

Timestamp	AppID	CtxID	Type	Message
2023/01/24 16:54:45...	8072	adap	info	[Received data: -0.951057]
2023/01/24 16:54:46...	5586	adap	info	[Sending data: -0.951057]
2023/01/24 16:54:46...	8072	adap	info	[Received data: -0.929776]
2023/01/24 16:54:46...	5586	adap	info	[Sending data: -0.929776]
2023/01/24 16:54:46...	8072	adap	info	[Received data: -0.904827]
2023/01/24 16:54:46...	5586	adap	info	[Sending data: -0.904827]
2023/01/24 16:54:46...	8072	adap	info	[Received data: -0.876307]
2023/01/24 16:54:46...	5586	adap	info	[Sending data: -0.876307]
2023/01/24 16:54:46...	8072	adap	info	[Received data: -0.844328]

- To stop the adaptive_msg_sender application, select it and click **Linux Target > Run On Target > Stop Application**.
- To stop the adaptive_msg_event_driven_receiver application, select it and click **Linux Target > Run On Target > Stop Application**.

See Also

deployApplicationPackage

Related Examples

- “Build Simulink Model and Deploy Application” (Embedded Coder)
- “Support Package Installation” (Embedded Coder)

AUTOSAR Composition and ECU Software Simulation

- “Import AUTOSAR Composition to Simulink” on page 7-2
- “Combine and Simulate AUTOSAR Software Components” on page 7-7
- “Model AUTOSAR Basic Software Service Calls” on page 7-12
- “Configure Calls to AUTOSAR Diagnostic Event Manager Service” on page 7-14
- “Configure Calls to AUTOSAR Function Inhibition Manager Service” on page 7-18
- “Configure Calls to AUTOSAR NVRAM Manager Service” on page 7-28
- “Configure AUTOSAR Basic Software Service Implementations for Simulation” on page 7-33
- “Simulate AUTOSAR Basic Software Services and Run-Time Environment” on page 7-36
- “Configure and Simulate AUTOSAR Function Inhibition Service Calls” on page 7-49
- “Simulate and Verify AUTOSAR Component Behavior by Using Diagnostic Fault Injection” on page 7-53

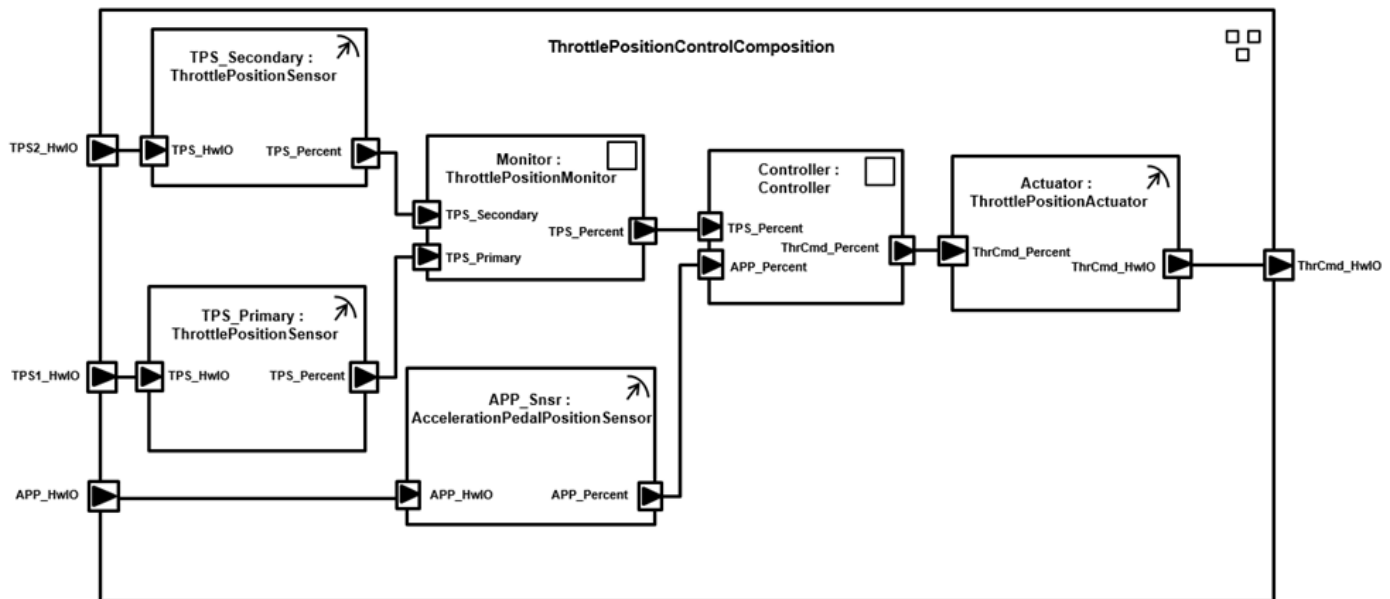
Import AUTOSAR Composition to Simulink

Create Simulink® model from XML description of AUTOSAR software composition.

Import AUTOSAR Composition from ARXML File to Simulink

Here is an AUTOSAR software composition that implements a throttle position control system. The composition contains six interconnected AUTOSAR software component prototypes -- four sensor/actuator components and two application components.

The composition was created in an AUTOSAR authoring tool and exported to the file `ThrottlePositionControlComposition.arxml`.



Use the MATLAB function `createCompositionAsModel` to import the AUTOSAR XML (ARXML) description and create an initial Simulink representation of the AUTOSAR composition. First, parse the ARXML description file and list the compositions it contains.

```
ar = arxml.importer('ThrottlePositionControlComposition.arxml');
names = getComponentNames(ar, 'Composition')

names = 1x1 cell array
    {'/Company/Components/ThrottlePositionControlComposition'}
```

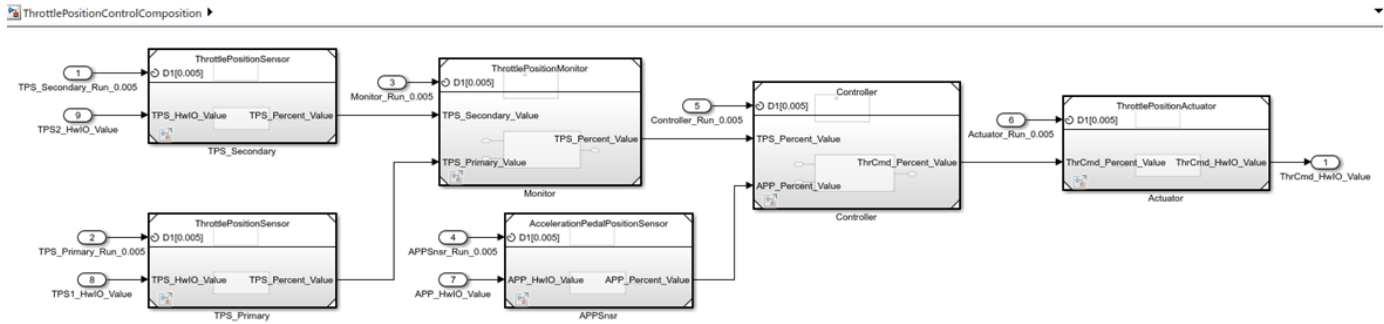
For the listed software composition, use `createCompositionAsModel` to create a Simulink representation.

```
createCompositionAsModel(ar, '/Company/Components/ThrottlePositionControlComposition');
```

```
Created model 'ThrottlePositionSensor' for component 1 of 5: /Company/Components/ThrottlePosition
Created model 'ThrottlePositionMonitor' for component 2 of 5: /Company/Components/ThrottlePositi
Created model 'Controller' for component 3 of 5: /Company/Components/Controller
Created model 'AccelerationPedalPositionSensor' for component 4 of 5: /Company/Components/Acceler
Created model 'ThrottlePositionActuator' for component 5 of 5: /Company/Components/ThrottlePosit.
```


Created model 'ThrottlePositionControlComposition' for composition 1 of 1:
/Company/Components/ThrottlePositionControlComposition

The function call creates a composition model that contains six component models, one for each atomic software component in the composition. Simulink inports and outports represent AUTOSAR ports and signal lines represent AUTOSAR component connectors.



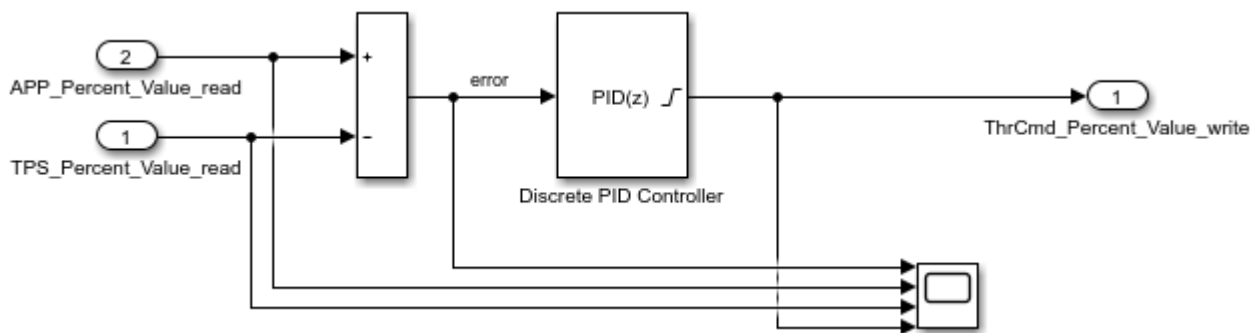
Develop AUTOSAR Component Algorithms, Simulate, and Generate Code

After creating an initial Simulink representation of the AUTOSAR composition, you develop each component in the composition. For each component, you refine the AUTOSAR configuration and create algorithmic model content.

For example, the Controller component model in the ThrottlePositionControlComposition composition model contains an atomic subsystem Runnable_Step_sys, which represents an AUTOSAR periodic runnable. The Runnable_Step_sys subsystem contains the initial stub implementation of the controller behavior.



Here is a possible implementation of the throttle position controller behavior. (To explore this implementation, see the model `autosar_sw_controller`, which is provided with the example “Design and Simulate AUTOSAR Components and Generate Code” on page 4-77.) The component takes as inputs an APP sensor percent value from a pedal position sensor and a TPS percent value from a throttle position sensor. Based on these values, the controller calculates the *error*. The error is the difference between where the operator wants the throttle, based on the pedal sensor, and the current throttle position. In this implementation, a Discrete PID Controller block uses the error value to calculate a throttle command percent value to provide to a throttle actuator. A scope displays the error value and the Discrete PID Controller block output value over time.



As you develop AUTOSAR components, you can:

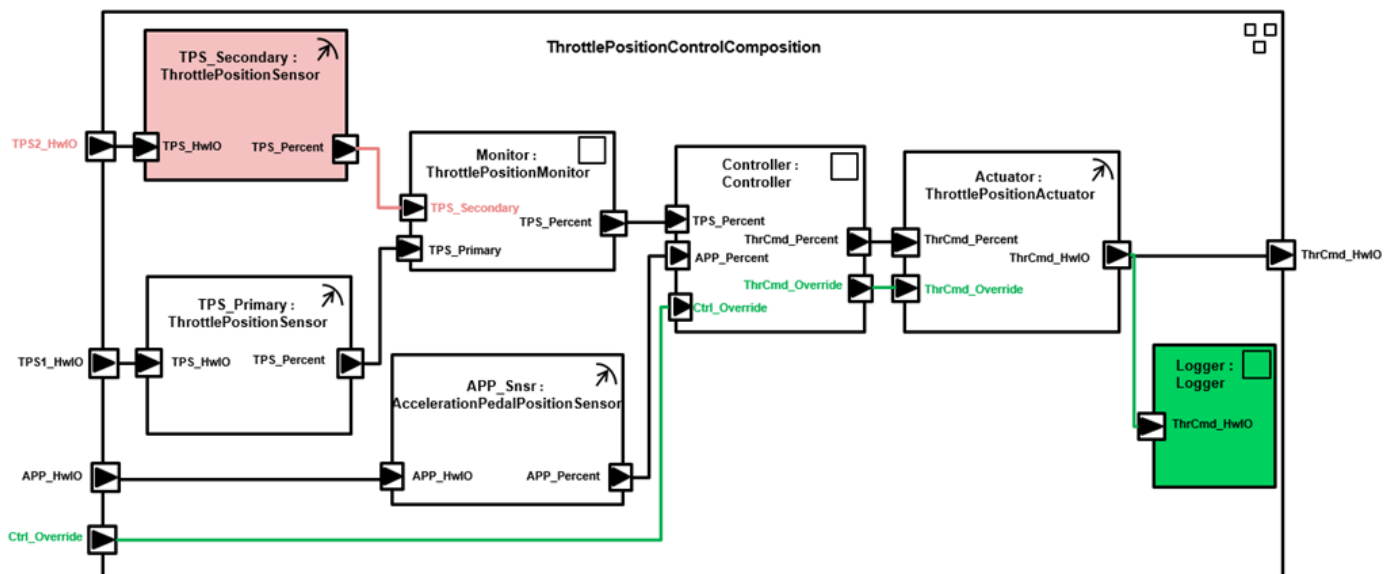
- Simulate component models individually or together in a containing composition.
- Generate ARXML component description files and algorithmic C code for testing in Simulink or integration into an AUTOSAR run-time environment. (AUTOSAR code generation requires Simulink Coder and Embedded Coder.)

For more information on developing, simulating, and building AUTOSAR components, see example “Design and Simulate AUTOSAR Components and Generate Code” on page 4-77.

Update AUTOSAR Composition Model with Architectural Changes from Authoring Tool

Suppose that, after you imported the AUTOSAR software composition into Simulink and began developing algorithms, architectural changes were made to the composition in the AUTOSAR authoring tool.

Here is the revised composition. The changes delete a sensor component, add a logger component, and add ports and connections at the composition and component levels. In the AUTOSAR authoring tool, the revised composition is exported to the file `ThrottlePositionControlComposition_updated.arxml`.



Use the MATLAB function `updateModel` to import the architectural revisions from the ARXML file. The function updates the AUTOSAR composition model with the changes and reports the results.

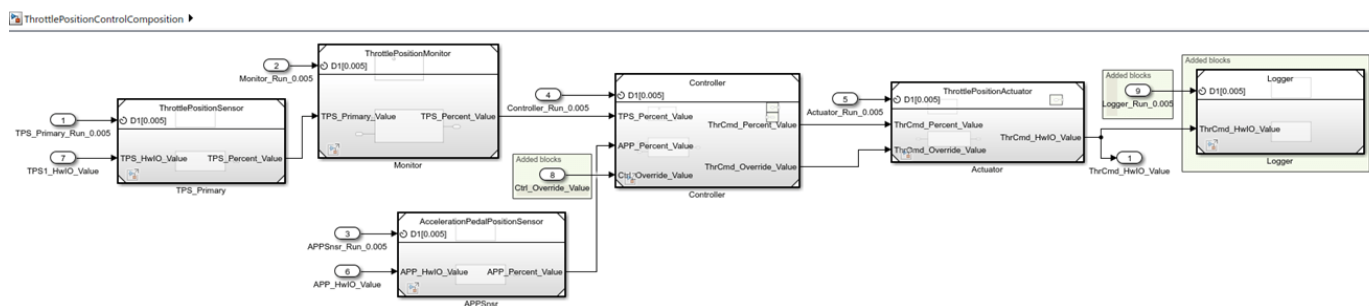
```

ar2 = arxml.importer('ThrottlePositionControlComposition_updated.arxml');
updateModel(ar2, 'ThrottlePositionControlComposition');

### Updating model ThrottlePositionSensor
### Saving original model as ThrottlePositionSensor_backup.slx
### Creating HTML report ThrottlePositionSensor_update_report.html
Updated model 'ThrottlePositionSensor' for component 1 of 6:
/Company/Components/ThrottlePositionSensor
### Updating model ThrottlePositionMonitor
### Saving original model as ThrottlePositionMonitor_backup.slx
### Creating HTML report ThrottlePositionMonitor_update_report.html
Updated model 'ThrottlePositionMonitor' for component 2 of 6:
/Company/Components/ThrottlePositionMonitor
Updated model 'Logger' for component 3 of 6: /Company/Components/Logger
### Updating model Controller
### Saving original model as Controller_backup.slx
### Creating HTML report Controller_update_report.html
Updated model 'Controller' for component 4 of 6: /Company/Components/Controller
### Updating model AccelerationPedalPositionSensor
### Saving original model as AccelerationPedalPositionSensor_backup.slx
### Creating HTML report AccelerationPedalPositionSensor_update_report.html
Updated model 'AccelerationPedalPositionSensor' for component 5 of 6:
/Company/Components/AccelerationPedalPositionSensor
### Updating model ThrottlePositionActuator
### Saving original model as ThrottlePositionActuator_backup.slx
### Creating HTML report ThrottlePositionActuator_update_report.html
Updated model 'ThrottlePositionActuator' for component 6 of 6:
/Company/Components/ThrottlePositionActuator
Updated model 'ThrottlePositionControlComposition' for composition 1 of 1:
/Company/Components/ThrottlePositionControlComposition
### Updating model ThrottlePositionControlComposition
### Saving original model as ThrottlePositionControlComposition_backup.slx
### Creating HTML report ThrottlePositionControlComposition_update_report.html

```

After the update, in the composition model, highlighting indicates where changes occurred.



The function also generates and displays an HTML AUTOSAR update report. The report lists changes that the update made to Simulink and AUTOSAR elements in the composition model. In the report, you can click hyperlinks to navigate from change descriptions to model changes, and to navigate from the main report to individual component reports.

Related Links

- [createCompositionAsModel](#)
- [updateModel](#)

- “Component Creation”
- “Import AUTOSAR Software Component Updates” on page 3-25
- “Design and Simulate AUTOSAR Components and Generate Code” on page 4-77

Combine and Simulate AUTOSAR Software Components

When you develop multiple AUTOSAR software component models that are interconnected and work together, you can combine them in an AUTOSAR composition model for simulation. A composition is an AUTOSAR software component that aggregates related groups of software components.

To create a Simulink representation of an AUTOSAR composition, take one of these actions:

- Import an AUTOSAR XML (ARXML) description of a composition (Classic Platform).
- Create a model and use Model blocks to reference and connect AUTOSAR component models.

Alternatively, if you have System Composer software, you can create an AUTOSAR architecture model and use Software Composition blocks to model AUTOSAR compositions. For more information, see “Software Architecture Modeling”.

When you simulate a composition model, you simulate the combined behavior of the aggregated AUTOSAR components.

After you develop AUTOSAR components and compositions, you can test groups of components that belong together in a system-level simulation. For example, you can create a system-level model containing compositions, components, a scheduler, a plant model, and potentially Basic Software service components and callers. You can configure system-level models to perform closed-loop or open-loop system simulations.

In this section...

“Import AUTOSAR Composition as Model (Classic Platform)” on page 7-7

“Create Composition Model for Simulating AUTOSAR Components” on page 7-8

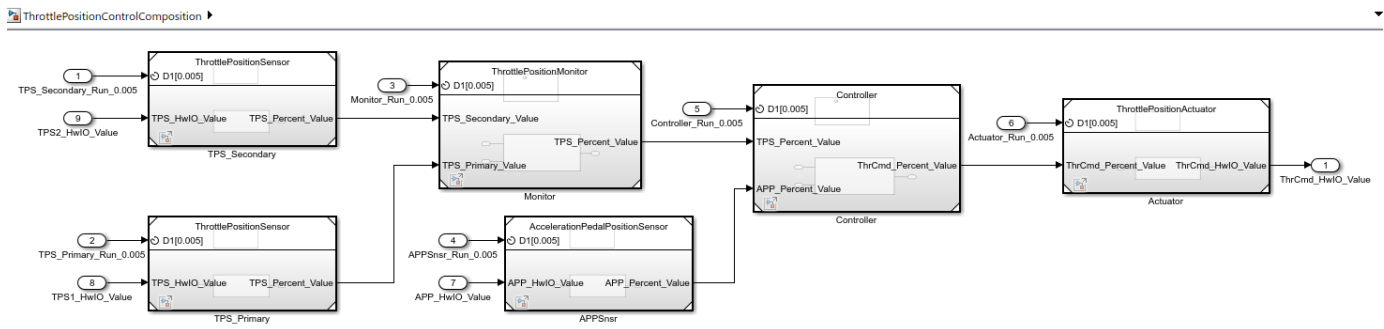
“Alternatives for AUTOSAR System-Level Simulation” on page 7-9

Import AUTOSAR Composition as Model (Classic Platform)

A composition is an AUTOSAR software component that aggregates related groups of software components. Compositions support component scaling and help to manage complexity in a design.

If you are developing software components for the AUTOSAR Classic Platform, you can create an AUTOSAR composition model by importing a composition description from ARXML files. Use the AUTOSAR importer function `createCompositionAsModel`. This function call creates composition model `ThrottlePositionControlComposition` from the example ARXML file `ThrottlePositionControlComposition.arxml`.

```
openExample('ThrottlePositionControlComposition.arxml');
ar = arxml.importer('ThrottlePositionControlComposition.arxml');
createCompositionAsModel(ar, '/Company/Components/ThrottlePositionControlComposition');
```



To simulate the combined behavior of the aggregated AUTOSAR components, simulate the composition model. Click the **Run** button in the model window or enter this MATLAB command.

```
simOutComposition = sim('ThrottlePositionControlComposition');
```

For more information, see “Import AUTOSAR Composition to Simulink” on page 7-2.

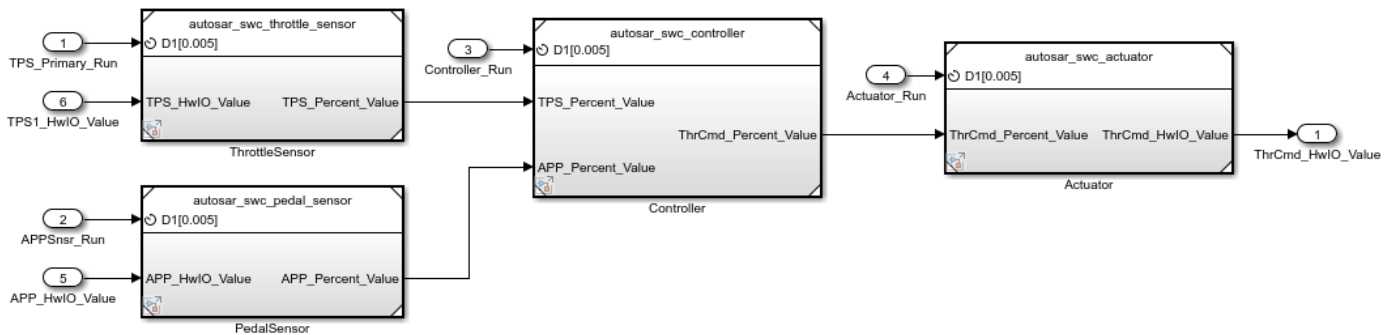
Create Composition Model for Simulating AUTOSAR Components

To combine related AUTOSAR software components in a composition model for simulation, create a Simulink model and use Model blocks to reference and connect AUTOSAR component models.

This example creates an AUTOSAR composition model. The created model is a simplified version of AUTOSAR example model `autosar_composition`. To expedite configuration and resolve issues, you can compare the new model against example model `autosar_composition`. If needed, you can copy elements such as inports and outports between the models. For a diagram of the finished composition model, see step 4.

- 1 Move AUTOSAR software component models that you want to simulate together into a working folder and `cd` to that folder. This example uses component models copied from `matlabroot/examples/autosarblockset/main` (`cd` to folder).
 - `autosar_swc_actuator`
 - `autosar_swc_controller`
 - `autosar_swc_pedal_sensor`
 - `autosar_swc_throttle_sensor`
- 2 Create a Simulink model. Save the model to the working folder with the name `composition`.
- 3 For each AUTOSAR component model:
 - a Open the component model separately and verify that it simulates.
 - b In the `composition` model, add a Model block and configure the block to reference the component. In the Model block parameters, select option **Schedule rates**. This option allows rate-based runnable tasks to be scheduled on the same basis as exported functions.
 - c Add ports that the component requires.
 - d Component model `autosar_swc_throttle_sensor` requires a special adjustment, because parent model `composition` (unlike example model `autosar_composition`) references the component only once. Open Model Explorer, select the model workspace for `autosar_swc_throttle_sensor`, select data object `TPSPercent_LkupTbl`, and clear the **Argument** option.

- 4 When you have created Model blocks for each AUTOSAR component, connect the components as shown here.



To simulate the combined behavior of the aggregated AUTOSAR components, simulate the composition model. Click the **Run** button in the model window or enter this MATLAB command.

```
simOutComposition = sim('composition');
```

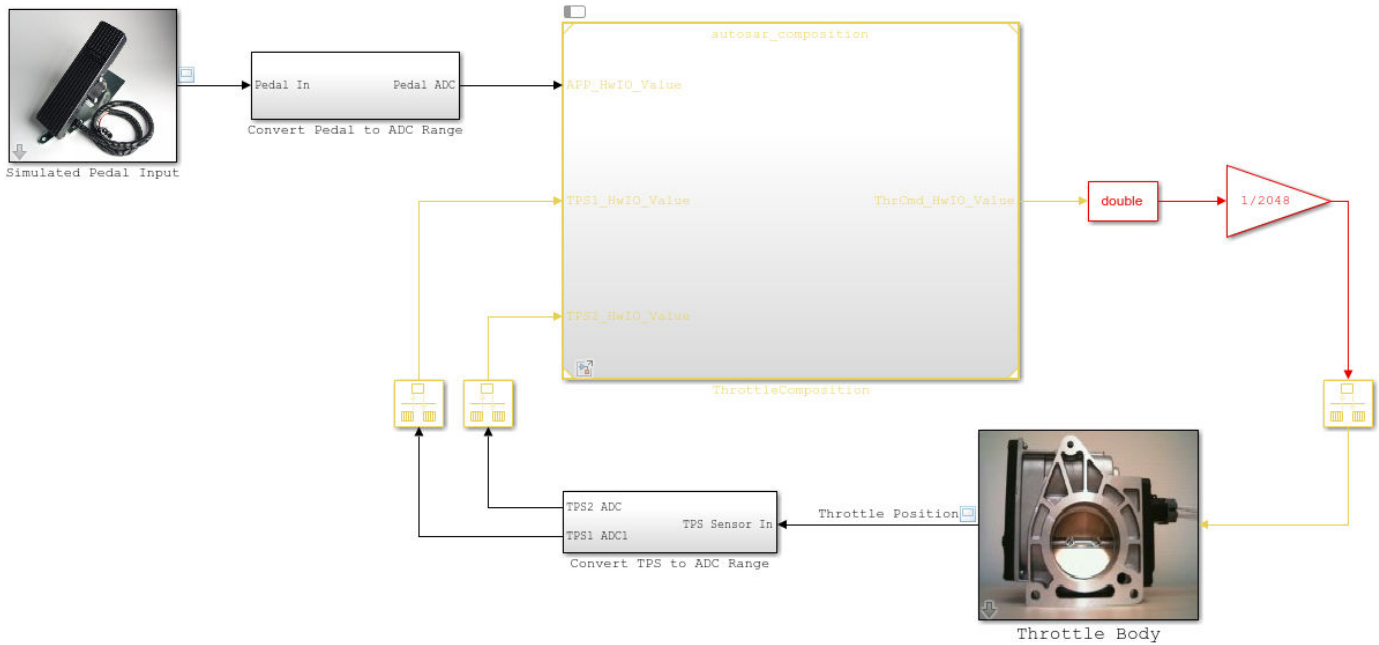
For more information, see “Design and Simulate AUTOSAR Components and Generate Code” on page 4-77.

Alternatives for AUTOSAR System-Level Simulation

After you develop AUTOSAR components and compositions, you can test groups of components that belong together in a system-level simulation. For example, you can create a system-level model containing compositions, components, a plant model, and potentially Basic Software service components and callers. You can configure system-level models to perform closed-loop or open-loop system simulations. For a system-level model, use a Simulink model or a Simulink Test test harness model.

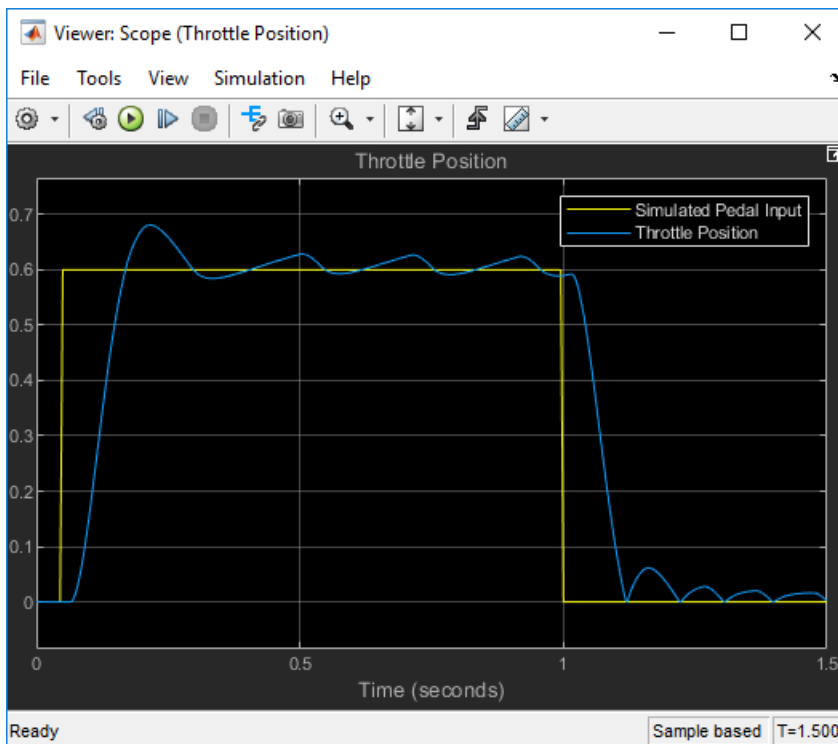
For an example of a closed-loop simulation, open example model `autosar_system`. This model provides a system-level test harness for the AUTOSAR composition model `autosar_composition`.

```
openExample('autosar_system');
```



A throttle position scope opens with the model. If you simulate system-level model `autosar_system`, the scope indicates how well the throttle-position control algorithms in composition model `autosar_composition` are tracking the pedal input. To improve the behavior, you can modify component algorithms in the composition or change a sensor source.

```
simOutSystem = sim('autosar_system');
```



For more information, see “Design and Simulate AUTOSAR Components and Generate Code” on page 4-77.

For an example of open-loop simulation using Simulink Test, see “Testing AUTOSAR Compositions” (Simulink Test). This example performs back-to-back testing for an AUTOSAR composition model.

For an example of simulating AUTOSAR Basic Software services, see “Simulate AUTOSAR Basic Software Services and Run-Time Environment” on page 7-36.

See Also

`createCompositionAsModel`

Related Examples

- “Import AUTOSAR Composition to Simulink” on page 7-2
- “Design and Simulate AUTOSAR Components and Generate Code” on page 4-77
- “Testing AUTOSAR Compositions” (Simulink Test)
- “Simulate AUTOSAR Basic Software Services and Run-Time Environment” on page 7-36

More About

- “AUTOSAR Software Components and Compositions” on page 1-11


Model AUTOSAR Basic Software Service Calls

For the AUTOSAR Classic Platform, the AUTOSAR standard defines important services as part of Basic Software (BSW) that runs in the AUTOSAR Runtime Environment (RTE). Examples include services provided by the Diagnostic Event Manager (Dem), the Function Inhibition Manager (FiM), and the NVRAM Manager (NvM). In the AUTOSAR RTE, AUTOSAR software components typically access BSW services using client-server or sender-receiver communication.

To support system-level modeling of AUTOSAR components and services, AUTOSAR Blockset provides an AUTOSAR Basic Software block library. The library contains preconfigured Function Caller blocks for modeling component calls to AUTOSAR BSW services.

- Diagnostic Event Manager (Dem) blocks — Calls to Dem service interfaces, including DiagnosticInfoCaller, DiagnosticMonitorCaller, DiagnosticOperationCycleCaller, and DiagnosticEventAvailableCaller.
- Function Inhibition Manager (FiM) blocks — Calls to FiM service interfaces, including Function Inhibition Caller and Control Function Available Caller.
- NVRAM Manager (NvM) blocks — Calls to NvM service interfaces, including NvMAdminCaller and NvMServiceCaller.

To implement client calls to AUTOSAR BSW service interfaces in your AUTOSAR software component, you drag and drop Basic Software blocks into an AUTOSAR model. Each block has prepopulated parameters, such as **Client port name** and **Operation**. If you modify the operation selection, the software updates the block inputs and outputs to correspond. The **Operation** parameter must be set to an operation supported by the schema currently specified by the model. The list of operations on the block parameters dialog reflects the operations supported by the current schema. To configure the schema, in the Configuration Parameters **Code Generation > AUTOSAR Code Generation Options** pane, specify the **Generate XML file for schema version** parameter.

To configure the added blocks in the AUTOSAR software component, click the **Update** button  in the Code Mappings editor view of the model. The software creates AUTOSAR client-service interfaces, operations, and ports, and maps each Simulink function caller to an AUTOSAR client port and operation.

For more information, see “Configure Calls to AUTOSAR Diagnostic Event Manager Service” on page 7-14, “Configure Calls to AUTOSAR Function Inhibition Manager Service” on page 7-18, and “Configure Calls to AUTOSAR NVRAM Manager Service” on page 7-28.

To simulate an AUTOSAR component model that calls BSW services, create a containing composition, system, or harness model. In that containing model, provide reference implementations of the Dem and NvM service operations called by the component.

The AUTOSAR Basic Software block library includes a Diagnostic Service Component block and an NVRAM Service Component block. The blocks provide reference implementations of Dem/FiM and NvM service operations. To support simulation of component calls to the Dem, FiM, and NvM services, include the blocks in the containing model. You can insert the blocks in either of two ways:

- Automatically insert the blocks by creating a Simulink Test harness model.
- Manually insert the blocks into a containing composition, system, or harness model.

For more information, see “Configure AUTOSAR Basic Software Service Implementations for Simulation” on page 7-33 and “Simulate AUTOSAR Basic Software Services and Run-Time Environment” on page 7-36.

See Also

Related Examples

- “Configure Calls to AUTOSAR Diagnostic Event Manager Service” on page 7-14
- “Configure Calls to AUTOSAR Function Inhibition Manager Service” on page 7-18
- “Configure Calls to AUTOSAR NVRAM Manager Service” on page 7-28
- “Configure AUTOSAR Basic Software Service Implementations for Simulation” on page 7-33
- “Simulate AUTOSAR Basic Software Services and Run-Time Environment” on page 7-36

More About

- “Model AUTOSAR Nonvolatile Memory” on page 2-40
- “Model AUTOSAR Communication” on page 2-21

Configure Calls to AUTOSAR Diagnostic Event Manager Service

For the AUTOSAR Classic Platform, the AUTOSAR standard defines important services as part of Basic Software (BSW) that runs in the AUTOSAR Runtime Environment (RTE). Examples include services provided by the Diagnostic Event Manager (Dem), the Function Inhibition Manager (FiM), and the NVRAM Manager (NvM). In the AUTOSAR RTE, AUTOSAR software components typically access BSW services using client-server or sender-receiver communication.

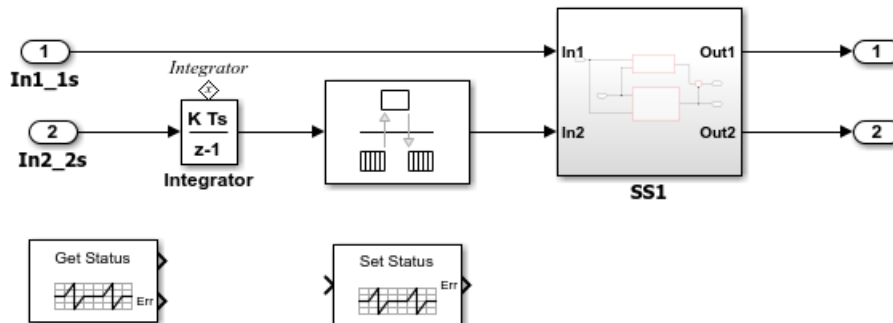
To support system-level modeling of AUTOSAR components and services, AUTOSAR Blockset provides an AUTOSAR Basic Software block library. The library contains preconfigured blocks for modeling component calls to AUTOSAR BSW services and reference implementations of the BSW services. For information about using the blocks to model client calls to AUTOSAR BSW service interfaces, see “Model AUTOSAR Basic Software Service Calls” on page 7-12.

For a live-script example of simulating AUTOSAR BSW services, see example “Simulate AUTOSAR Basic Software Services and Run-Time Environment” on page 7-36.

Here is an example of configuring client calls to Dem service interfaces in your AUTOSAR software component.

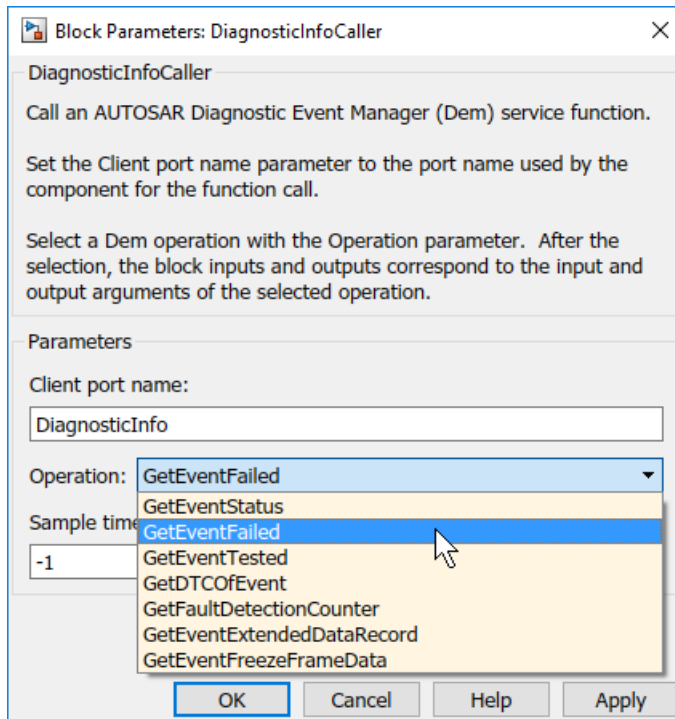
- 1 Open a model that is configured for AUTOSAR code generation. Using the Library Browser or by typing block names in the model window, add Dem blocks to the model. This example adds the blocks DiagnosticInfoCaller and DiagnosticMonitorCaller to the example model `autosar_sw_c`.

```
openExample('autosar_sw_c');
```

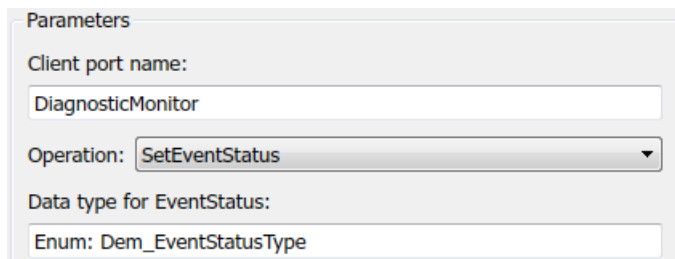



- 2 Open each block and examine the parameters, especially **Operation**. If you select a different operation and click **Apply**, the software updates the block inputs and outputs to match the arguments of the selected operation.

This example changes the **Operation** for the DiagnosticInfoCaller block from `GetEventStatus` to `GetEventFailed`. (For an example of using `GetEventFailed` in a throttle position monitor implementation, see example “Simulate AUTOSAR Basic Software Services and Run-Time Environment” on page 7-36.) The **Operation** parameter must be set to an operation supported by the schema currently specified by the model. The list of operations reflects the operations supported by the current schema.



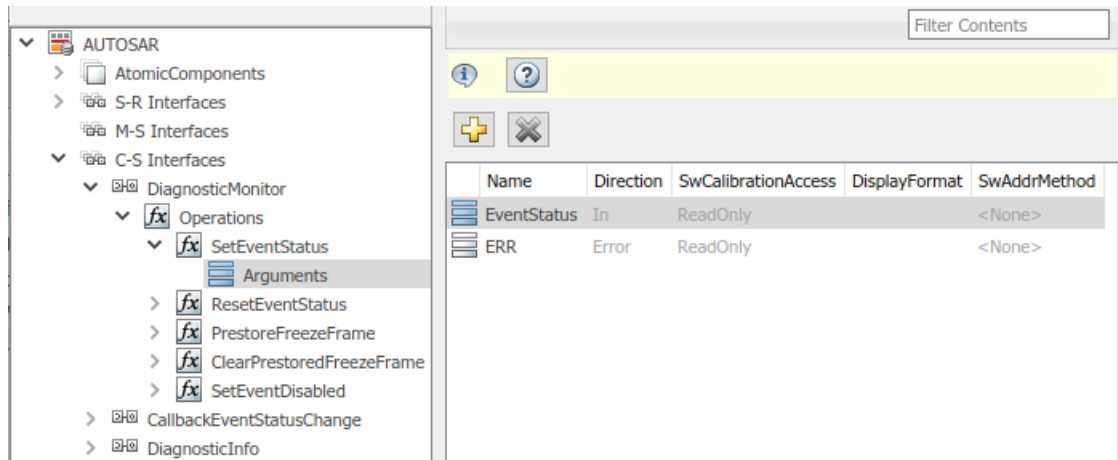
For some Dem operations, such as `GetDTCOfEvent` and `SetEventStatus`, the block parameters dialog box displays a data type parameter. The parameter specifies an enumerated data type for a function input that represents a Dem format type or event status. Default data types are provided, such as `Dem_DTCFormatType` or `Dem_EventStatusType`. For more information about format type or event status values, see the AUTOSAR standard *Specification of Diagnostic Event Manager*.



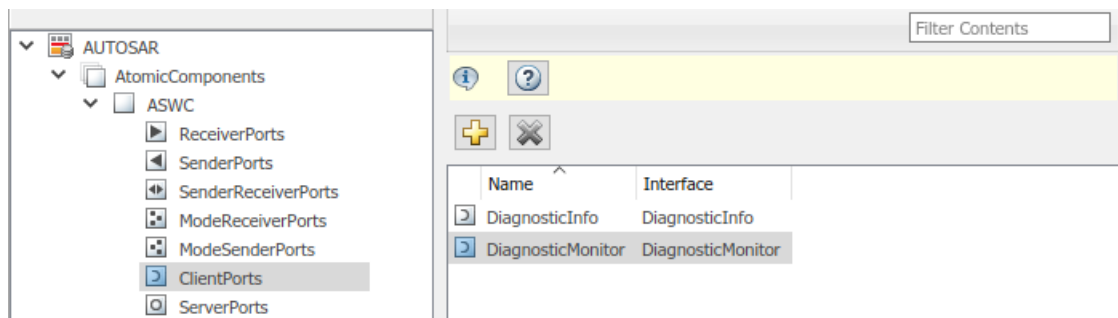
- 3 Open the Code Mappings editor. To update the Simulink to AUTOSAR mapping of the model with changes to Simulink function callers, click the **Update** button . The software creates AUTOSAR client-service interfaces, operations, and ports, and maps each Simulink function caller to an AUTOSAR client port and operation.

For example, for the `DiagnosticMonitorCaller` block in this example, for which the `SetEventStatus` operation is selected:

- The software creates C-S interface `DiagnosticMonitor`, and under `DiagnosticMonitor`, its supported operations. For each operation, arguments are provided with read-only properties. Here are the arguments for the `DiagnosticMonitor` operation `SetEventStatus` displayed in the AUTOSAR Dictionary.



- The software creates a client port with the default name `DiagnosticMonitor`. Unlike the C-S-interface, operation, and argument names, the client port name can be customized. The client port is mapped to the `DiagnosticMonitor` interface.



- The Code Mappings editor, **Function Callers** tab, maps the `DiagnosticMonitor` function caller block to AUTOSAR client port `DiagnosticMonitor` and AUTOSAR operation `SetEventStatus`.

The screenshot shows the Code Mappings editor with the 'Function Callers' tab selected. The table below shows the mappings:

Source	ClientPort	Operation
<code>DiagnosticInfo_GetEventFailed</code>	DiagnosticInfo	GetEventFailed
<code>DiagnosticMonitor_SetEventStatus</code>	DiagnosticMonitor	SetEventStatus

- 4 Optionally, build your component model and examine the generated C and ARXML code. The C code includes the client calls to the BSW services, for example:

```

/* FunctionCaller: '<Root>/DiagnosticInfoCaller' */
Rte_Call_DiagnosticInfo_GetEventFailed(&rtb_DiagnosticInfoCaller_o1);

/* FunctionCaller: '<Root>/DiagnosticMonitorCaller' */
Rte_Call_DiagnosticMonitor_SetEventStatus(DEM_EVENT_STATUS_PASSED);

```

Generated RTE include files define the server operation call points, such as `Rte_Call_DiagnosticMonitor_SetEventStatus`, and argument data types, such as enumeration type `Dem_EventStatusType`.

The ARXML code defines the BSW service operations called by the component as server call points, for example:

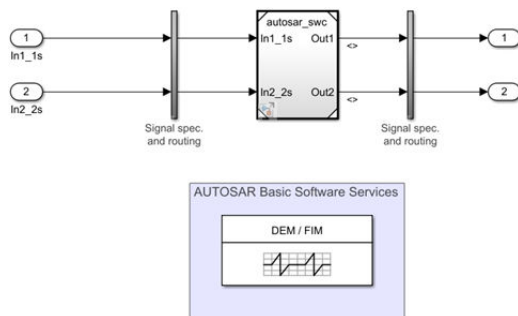
```

<SERVER-CALL-POINTS>
...
  <SYNCHRONOUS-SERVER-CALL-POINT UUID="...">
    <SHORT-NAME>SC_DiagnosticMo_334e61e63627b44b</SHORT-NAME>
    <OPERATION-IREF>
      <CONTEXT-R-PORT-REF DEST="R-PORT-PROTOTYPE">
        /Company/Powertrain/Components/ASWC/DiagnosticMonitor
      </CONTEXT-R-PORT-REF>
      <TARGET-REQUIRED-OPERATION-REF DEST="CLIENT-SERVER-OPERATION">
        /AUTOSAR/Services/Dem/DiagnosticMonitor/SetEventStatus
      </TARGET-REQUIRED-OPERATION-REF>
    </OPERATION-IREF>
    <TIMEOUT>1.0E-06</TIMEOUT>
  </SYNCHRONOUS-SERVER-CALL-POINT>
</SERVER-CALL-POINTS>

```

- 5 To simulate the component model, create a containing composition, system, or test harness model. In that containing model, insert reference implementations of the Dem GetEventFailed and GetEventStatus service operations.

The AUTOSAR Basic Software block library provides a Diagnostic Service Component block, which provides reference implementations of Dem service operations. You can manually insert the block into a containing composition, system, or harness model, or automatically insert the block by creating a Simulink Test harness model.



For more information, see “Configure AUTOSAR Basic Software Service Implementations for Simulation” on page 7-33 and “Simulate AUTOSAR Basic Software Services and Run-Time Environment” on page 7-36.

See Also

DiagnosticInfoCaller | DiagnosticMonitorCaller | DiagnosticOperationCycleCaller | DiagnosticEventAvailableCaller | Diagnostic Service Component

Related Examples

- “Configure AUTOSAR Basic Software Service Implementations for Simulation” on page 7-33
- “Simulate AUTOSAR Basic Software Services and Run-Time Environment” on page 7-36
- “Configure AUTOSAR Client-Server Communication” on page 4-142

More About

- “Model AUTOSAR Basic Software Service Calls” on page 7-12
- “Model AUTOSAR Communication” on page 2-21

Configure Calls to AUTOSAR Function Inhibition Manager Service

For the AUTOSAR Classic Platform, the AUTOSAR standard defines important services as part of Basic Software (BSW) that runs in the AUTOSAR Runtime Environment (RTE). Examples include services provided by the Diagnostic Event Manager (Dem), the Function Inhibition Manager (FiM), and the NVRAM Manager (NvM). In the AUTOSAR RTE, AUTOSAR software components typically access BSW services using client-server or sender-receiver communication.

To support system-level modeling of AUTOSAR components and services, AUTOSAR Blockset provides an AUTOSAR Basic Software block library. The library contains preconfigured blocks for modeling component calls to AUTOSAR BSW services and reference implementations of the BSW services. For information about using the blocks to model client calls to AUTOSAR BSW service interfaces, see “Model AUTOSAR Basic Software Service Calls” on page 7-12.

For live-script examples of simulating AUTOSAR BSW services, see examples “Simulate AUTOSAR Basic Software Services and Run-Time Environment” on page 7-36 and “Configure and Simulate AUTOSAR Function Inhibition Service Calls” on page 7-49.

As defined in the AUTOSAR specification, the Function Inhibition Manager provides a control mechanism for selectively inhibiting (that is, deactivating) function execution in software component runnables, based on function identifiers (FIDs) with inhibition conditions. For example, an FID can represent functionality that must be stopped if a specific failure occurs.

The Function Inhibition Manager is closely related to the Diagnostic Event Manager, because inhibition conditions can be based on the status of diagnostic events. For example, if a sensor failure event is reported to the Diagnostic Event Manager, the Function Inhibition Manager can inhibit the associated function identifier and stop execution of the corresponding functionality.

AUTOSAR Blockset provides FiM and Dem blocks that allow you to query the status of function inhibition conditions and configure function inhibition criteria based on diagnostic event status.

In this section...

“Model Function Inhibition” on page 7-18

“Scope Failures to Operation Cycles” on page 7-23

“Control Function Availability During Failure or For Testing” on page 7-23

“Configure Service Calls for Function Inhibition” on page 7-24

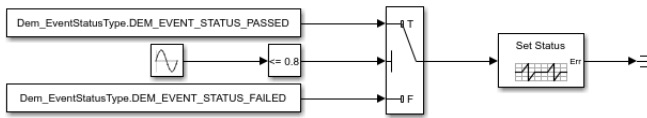
Model Function Inhibition

AUTOSAR software components use function inhibition to switch functions on or off depending on the state of Diagnostic Event Manager (Dem) events. Software components can react to an event such as a sensor success or failure by allowing or preventing execution of an associated function.

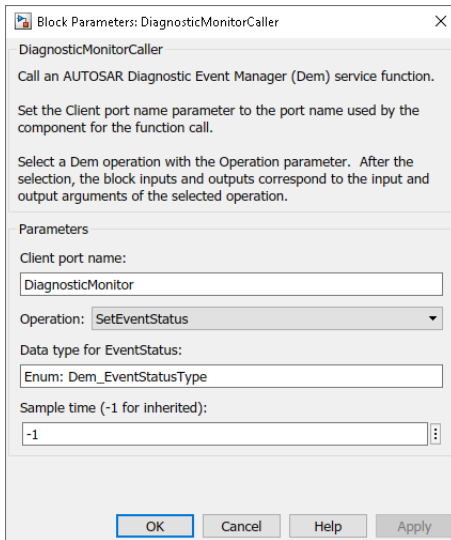
Consider an AUTOSAR software component model in which a Dem Set Status block serves as a monitor for new functionality within the component. Dem events passed by using the Set Status block indicate whether conditions such as sensor failure have occurred. Event status determines whether execution of associated downstream functions can proceed. The functions run only if no failure events are reported.

To implement function inhibition for new functionality in the component:

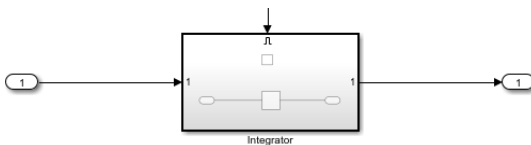
- 1 Open the AUTOSAR software component model.



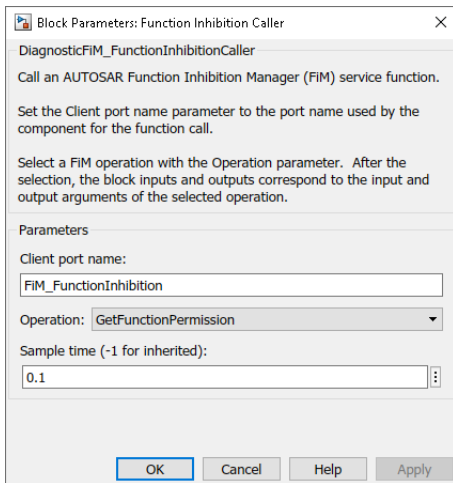
Open the Set Status block dialog box. Examine the client port name and operation values. Confirm that the client port and its client interface are defined in the AUTOSAR Dictionary.



- 2 Model the new functionality such that it will execute only if event status indicates no sensor failures. This example places the new functionality in an enabled subsystem.



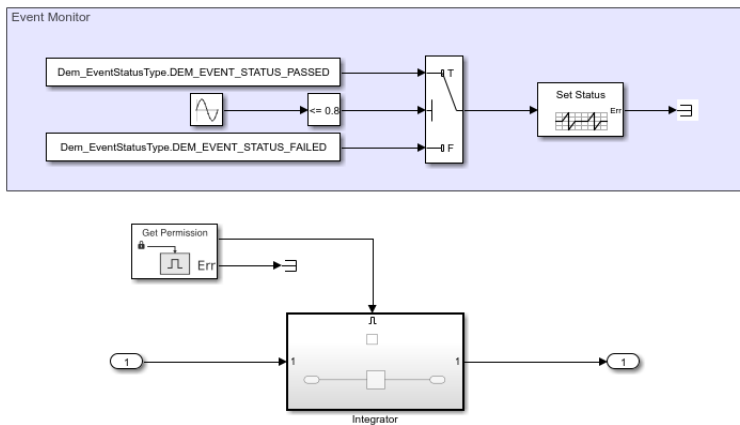
- 3 Add a Function Inhibition Caller block to the model. Open the block dialog box and configure the block to request a GetFunctionPermission operation from the FiM FunctionInhibition service interface. Specify a client port name and a sample time.



Open the Code Mappings editor and click the **Update** button. The software creates the specified client port and interface in the AUTOSAR Dictionary, and maps the Get Permission caller block to the specified AUTOSAR client port and operation.

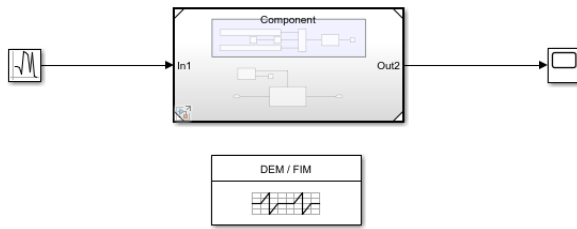
- 4 Connect the Get Permission block to the enable port of the subsystem that contains the new functionality. The block represents evaluation of the function inhibition conditions for the new functionality. If the functionality is not inhibited and therefore has permission to run, the Get Permission block returns true, enabling the subsystem.

Here is the revised software component model.



- 5 Place the software component model in a test harness.

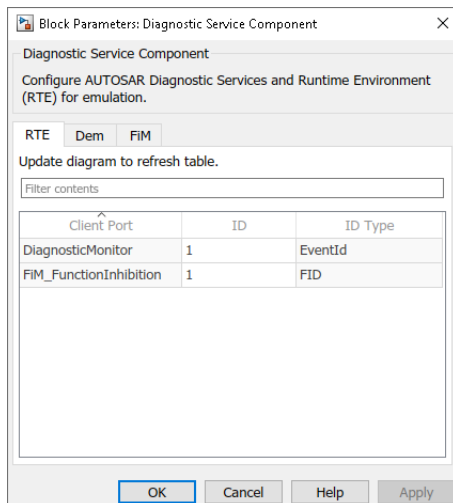
In the test harness model, to provide reference implementations of the Dem Get Status and FiM Get Permission services for simulation, add a Diagnostic Service Component block. Update the model.



- Open the Diagnostic Service Component block dialog box. To refresh RTE and FiM tables in the dialog box at any point, update the model.

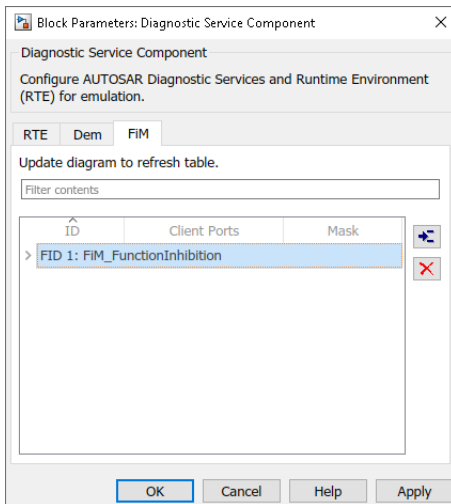
On the **RTE** tab, you configure event and function identifiers for events that can trigger function inhibition. In this example, the model hierarchy contains only one event port and one FID port, so the **RTE** tab requires no further configuration.

If the model hierarchy contains additional ports for multiple Get Permission blocks, with functionality distributed across multiple components, you can use the **RTE** tab to assign ports to the same FID to group them or separate FIDs to address them individually. For a function inhibition example with multiple ports and distributed functionality, see “Configure and Simulate AUTOSAR Function Inhibition Service Calls” on page 7-49.



- After event and function identifiers are configured, switch to the **FiM** tab. On the **FiM** tab, you add and configure the inhibition conditions that determine when Get Permission blocks allow functionality to operate.

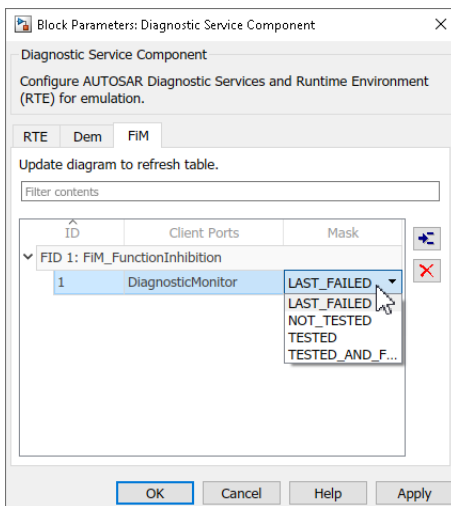
The **FiM** tab lists function identifiers representing functions in the model hierarchy for which function inhibition is implemented. In this example, the model hierarchy contains one FID.



- 8 To add an inhibition condition for the FID, select the FID and click the **Add inhibition condition** button. A row for the inhibition condition appears under the FID.

In the row, select an event ID value (matching an event ID listed in the **RTE** tab). Then, for the FID and event ID pair, select an inhibition mask value. The AUTOSAR specifications define mask values in a FiMinhibitionMask values table.

In this example, the function represented by FID 1 is inhibited if the event represented by event ID 1 is LAST_FAILED.



- 9 Update and simulate the harness model.

Next steps in developing the software component include:

- Addressing how to trigger the event.
- Adding the functionality to inhibit.

For a larger-scale example of modeling AUTOSAR function inhibition, see “Configure and Simulate AUTOSAR Function Inhibition Service Calls” on page 7-49.

Scope Failures to Operation Cycles

In an AUTOSAR software component, operation cycles represent automotive cycles, such as ignition cycles, power cycles, warm up cycles, or on-board diagnostic (OBD) cycles. A cycle can be started, stopped, or queried by using Diagnostic Event Manager services. You can use operation cycles to determine if a given event has failed within a given time.

Operation cycles split a simulation into periods of time, such as one minute cycles. In each cycle, the software can check if a diagnostic condition (event) has been TESTED (a FiM condition) during that cycle and inhibit functions accordingly.

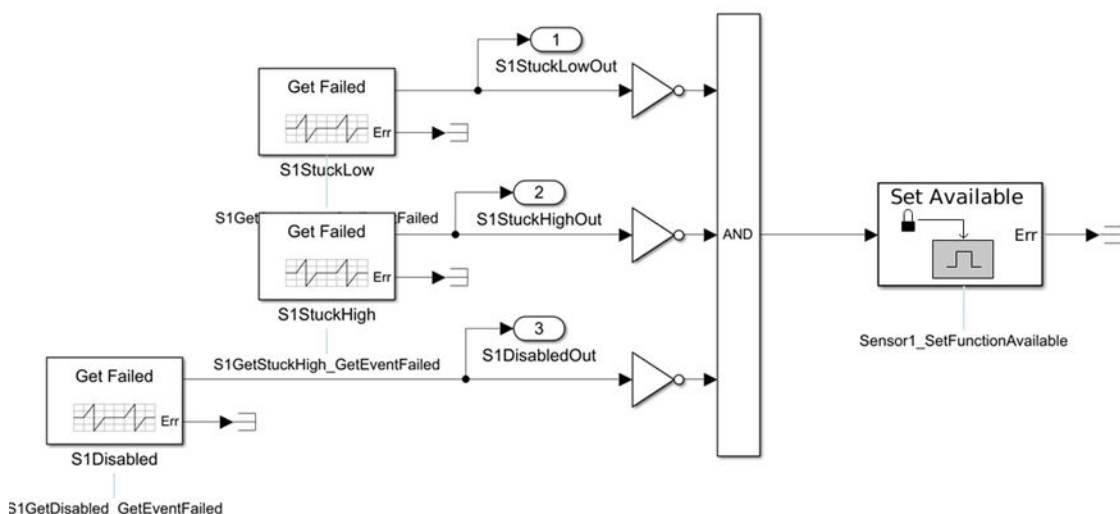
The BSW block DiagnosticOperationCycleCaller supports the `SetOperationCycleState` and `GetOperationCycleState` services. A component calls the services to control component operation cycles, which are used to scope failures to a time period. Calling `SetOperationCycleState` with the value `Dem_OperationCycleStateType.DEM_CYCLE_STATE_START` starts an operation cycle. Passing in the value `Dem_OperationCycleStateType.DEM_CYCLE_STATE_END` ends an operation cycle. Calling `GetOperationCycleState` queries the current state of an operation cycle.

For an example use of the DiagnosticOperationCycleCaller block and the `SetOperationCycleState` service, see “Configure and Simulate AUTOSAR Function Inhibition Service Calls” on page 7-49.

Control Function Availability During Failure or For Testing

The Function Inhibition Manager supports inhibition criteria for restricting functional blocks from executing until logical and functional predecessors have run, or for restricting execution of a safety system until a failure is verified. However, you can restrict use of functionality independently of inhibition criteria. For example, a sensor component can disable reading of its sensor data during a failure or during testing of other system functionality.

The BSW block Control Function Available Caller supports the `SetFunctionAvailable` service, which provides a granular mechanism to inhibit specific functionality. A component uses `SetFunctionAvailable` with an input signal value of false to inhibit associated functionality, so that the Get Permission block for the functionality returns 0. In this example, a sensor monitor uses `SetFunctionAvailable` to inform a central monitor component whether sensor measurements are available.



The central monitor uses Function Inhibition Caller blocks and the `GetFunctionPermission` service to decide whether to account for measurements coming from each sensor. The central monitor has as many Get Permission blocks as there are sensors.

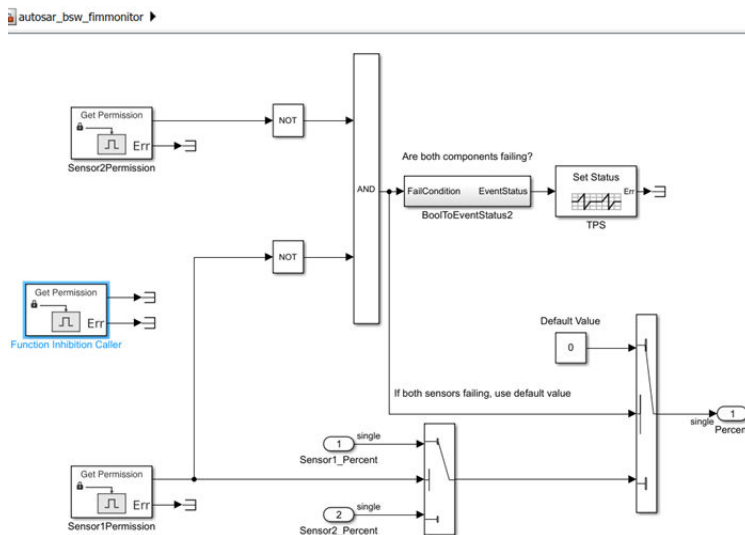
The **FiM** tab of the Diagnostic Service Component block configures the details of failure events. If a function is available, the **FiM** tab ID and mask settings control function inhibition. If a function is not available, `GetFunctionPermission` always returns false.

Configure Service Calls for Function Inhibition

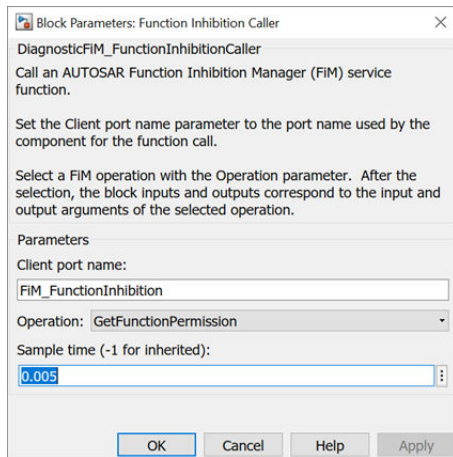
As part of implementing function inhibition, you configure client calls to FiM-related service interfaces in your AUTOSAR software component. Here is an example of configuring client calls to query the status of function inhibition conditions.


- 1 Open a model that is configured for AUTOSAR code generation. This example uses example model `autosar_bsw_fimmonitor`, which is associated with the example “Configure and Simulate AUTOSAR Function Inhibition Service Calls” on page 7-49. Using the Library Browser or by typing block names in the model window, add FiM block Function Inhibition Caller to the model.

For the purposes of this example, connect the block outputs to Terminator blocks.



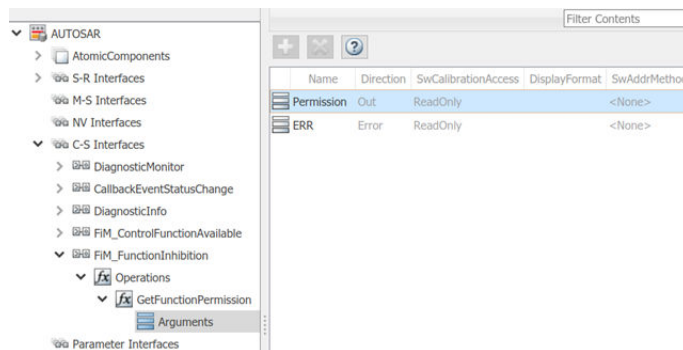
- 2 Open the new block and examine the parameters. For the FiM service call, the **Client port name** is `FiM_FunctionInhibition` and the **Operation** is `GetFunctionPermission`. The **Operation** parameter must be set to an operation supported by the schema currently specified by the model. The list of operations reflects the operations supported by the current schema. Set **Sample time** to `0.005`, which matches the other `GetFunctionPermission` caller blocks in the model.



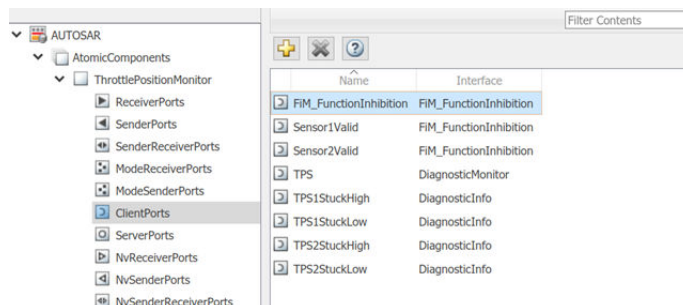
- 3 Open the Code Mappings editor. To update the Simulink to AUTOSAR mapping of the model with changes to Simulink function callers, click the **Update** button . The software creates AUTOSAR client-service interfaces, operations, and ports, and maps each Simulink function caller to an AUTOSAR client port and operation.

For example, for the Function Inhibition Caller block in this example, for which the `GetFunctionPermission` operation is selected:

- The software creates C-S interface `FiM_FunctionInhibition`, and under `FiM_FunctionInhibition`, its supported operation, `GetFunctionPermission`. Operation arguments are provided with read-only properties. In the AUTOSAR Dictionary, here are the arguments for the `FiM_FunctionInhibition` operation `GetFunctionPermission`.



- The software creates a client port with the default name `FiM_FunctionInhibition`. Unlike the C-S-interface, operation, and argument names, the client port name can be customized. The client port is mapped to the `FiM_FunctionInhibition` interface.



- The Code Mappings editor maps the Function Inhibition Caller function caller block to AUTOSAR client port FiM_FunctionInhibition and AUTOSAR operation GetFunctionPermission.

Functions	Inports	Outports	Parameters	Data Stores	Signals/States	Data Transfers	Function Callers
							Filter contents
	Source				ClientPort		Operation
⊗	FIM_FunctionInhibition_GetFunctionPermission				FIM_FunctionInhibition		GetFunctionPermission
⊗	Sensor1Valid_GetFunctionPermission				Sensor1Valid		GetFunctionPermission
⊗	Sensor2Valid_GetFunctionPermission				Sensor2Valid		GetFunctionPermission
⊗	TPS_SetEventStatus				TPS		SetEventStatus

- Optionally, build your component model and examine the generated C and ARXML code. The C code includes the client calls to the BSW services, for example:

```
/* FunctionCaller: '<Root>/Function Inhibition Caller' */
Rte_Call_FiM_FunctionInhibition_GetFunctionPermission
(&rtb_FunctionInhibitionCaller_o1);
```

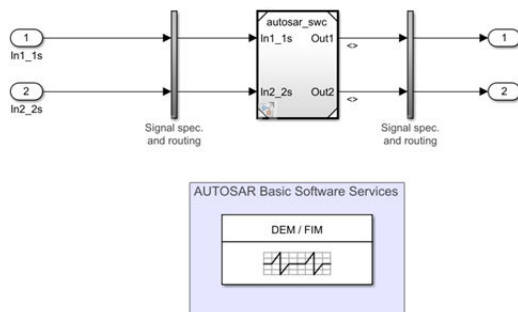
Generated RTE include files define the server operation call points, such as Rte_Call_FiM_FunctionInhibition_GetFunctionPermission.

The ARXML code defines the BSW service operations called by the component as server call points, for example:

```
<SERVER-CALL-POINTS>
<SYNCHRONOUS-SERVER-CALL-POINT UUID="...">
  <SHORT-NAME>SC_FiM_Function_60fb8d34c7807f7b</SHORT-NAME>
  <OPERATION-IREF>
    <CONTEXT-R-PORT-REF DEST="R-PORT-PROTOTYPE">
      /ThrottlePositionMonitorCompo_pkg/ThrottlePositionMonitorCompo_sw
      /ThrottlePositionMonitor/FiM_FunctionInhibition
    </CONTEXT-R-PORT-REF>
    <TARGET-REQUIRED-OPERATION-REF DEST="CLIENT-SERVER-OPERATION">
      /AUTOSAR/Services/FiM/FiM_FunctionInhibition/GetFunctionPermission
    </TARGET-REQUIRED-OPERATION-REF>
  </OPERATION-IREF>
  <TIMEOUT>1.0E-06</TIMEOUT>
</SYNCHRONOUS-SERVER-CALL-POINT>
...
</SERVER-CALL-POINTS>
```

- To simulate the component model, create a containing composition, system, or test harness model. In that containing model, insert a reference implementation of the FiM GetFunctionPermission service operation.

The AUTOSAR Basic Software block library provides a Diagnostic Service Component block, which provides reference implementations of Dem and FiM service operations. You can manually insert the block into a containing composition, system, or harness model, or automatically insert the block by creating a Simulink Test harness model.



For more information, see “Configure AUTOSAR Basic Software Service Implementations for Simulation” on page 7-33.

Example “Configure and Simulate AUTOSAR Function Inhibition Service Calls” on page 7-49 provides a set of example models, which together illustrate key aspects of implementing function inhibition, including:

- Query the status of inhibition conditions (`FunctionInhibition` operation `GetFunctionPermission`).
- Configure inhibition criteria based on event status (Diagnostic Service Component block dialog, **RTE** and **FiM** tabs).
- Define operation cycles to scope failures to a time period (Dem `OperationCycle` operation `SetOperationCycleState`).

See Also

Function Inhibition Caller | Control Function Available Caller | `DiagnosticOperationCycleCaller` | Diagnostic Service Component

Related Examples

- “Configure Calls to AUTOSAR Diagnostic Event Manager Service” on page 7-14
- “Configure AUTOSAR Basic Software Service Implementations for Simulation” on page 7-33
- “Simulate AUTOSAR Basic Software Services and Run-Time Environment” on page 7-36
- “Configure and Simulate AUTOSAR Function Inhibition Service Calls” on page 7-49
- “Configure AUTOSAR Client-Server Communication” on page 4-142

More About

- “Model AUTOSAR Basic Software Service Calls” on page 7-12
- “Model AUTOSAR Communication” on page 2-21

Configure Calls to AUTOSAR NVRAM Manager Service

For the AUTOSAR Classic Platform, the AUTOSAR standard defines important services as part of Basic Software (BSW) that runs in the AUTOSAR Runtime Environment (RTE). Examples include services provided by the Diagnostic Event Manager (Dem), the Function Inhibition Manager (FiM), and the NVRAM Manager (NvM). In the AUTOSAR RTE, AUTOSAR software components typically access BSW services using client-server or sender-receiver communication.

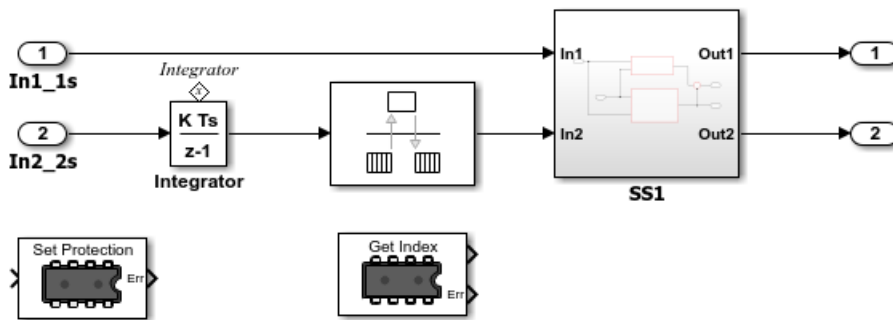
To support system-level modeling of AUTOSAR components and services, AUTOSAR Blockset provides an AUTOSAR Basic Software block library. The library contains preconfigured blocks for modeling component calls to AUTOSAR BSW services and reference implementations of the BSW services. For information about using the blocks to model client calls to AUTOSAR BSW service interfaces, see “Model AUTOSAR Basic Software Service Calls” on page 7-12.

For a live-script example of simulating AUTOSAR BSW services, see example “Simulate AUTOSAR Basic Software Services and Run-Time Environment” on page 7-36.

For more information about modeling software component access to AUTOSAR nonvolatile memory, see “Model AUTOSAR Nonvolatile Memory” on page 2-40.

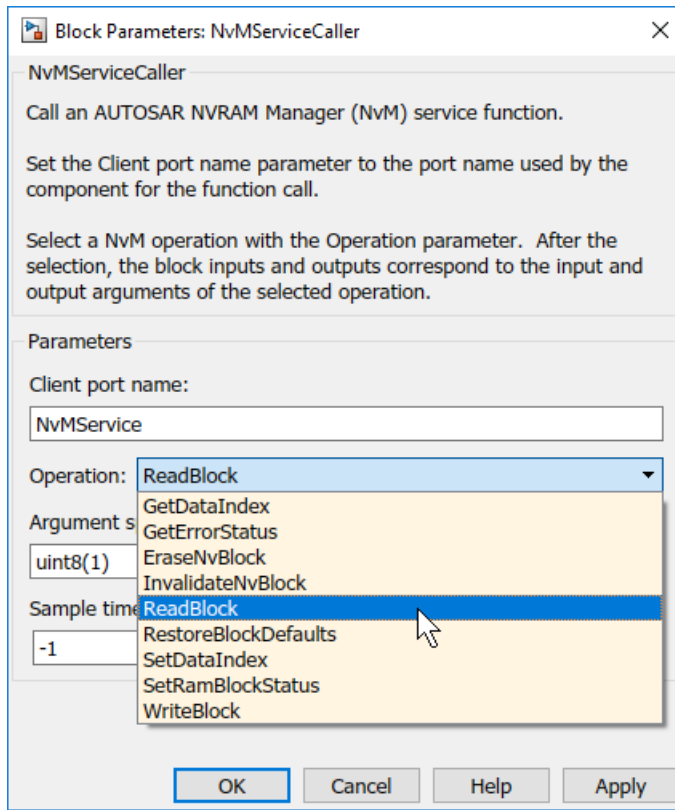
Here is an example of configuring client calls to NvM service interfaces in your AUTOSAR software component.

- 1 Open a model that is configured for AUTOSAR code generation. Using the Library Browser or by typing block names in the model window, add NvM blocks to the model. This example adds the blocks NvMAdminCaller and NvMServiceCaller to a writable copy of the example model `autosar_sw.c`.



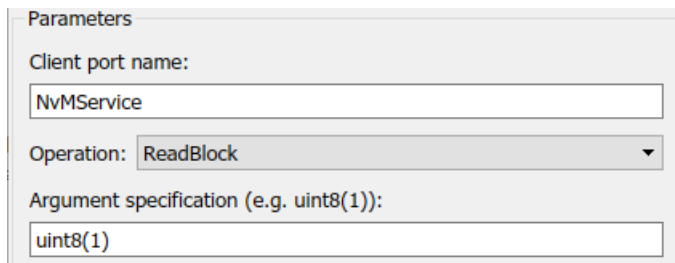
- 2 Open each block and examine the parameters, especially **Operation**. If you select a different operation and click **Apply**, the software updates the block inputs and outputs to match the arguments of the selected operation.


This example changes the **Operation** for the NvMServiceCaller block from `GetDataIndex` to `ReadBlock`. (For an example of using `readBlock` in a throttle position sensor implementation, see example “Simulate AUTOSAR Basic Software Services and Run-Time Environment” on page 7-36.) The **Operation** parameter must be set to an operation supported by the schema currently specified by the model. The list of operations reflects the operations supported by the current schema.



For some NvM operations, such as `ReadBlock` and `WriteBlock`, the block parameters dialog box displays an argument specification parameter. The parameter specifies data type and dimension information for data to be read or written by the operation, set to `uint8(1)` by default.

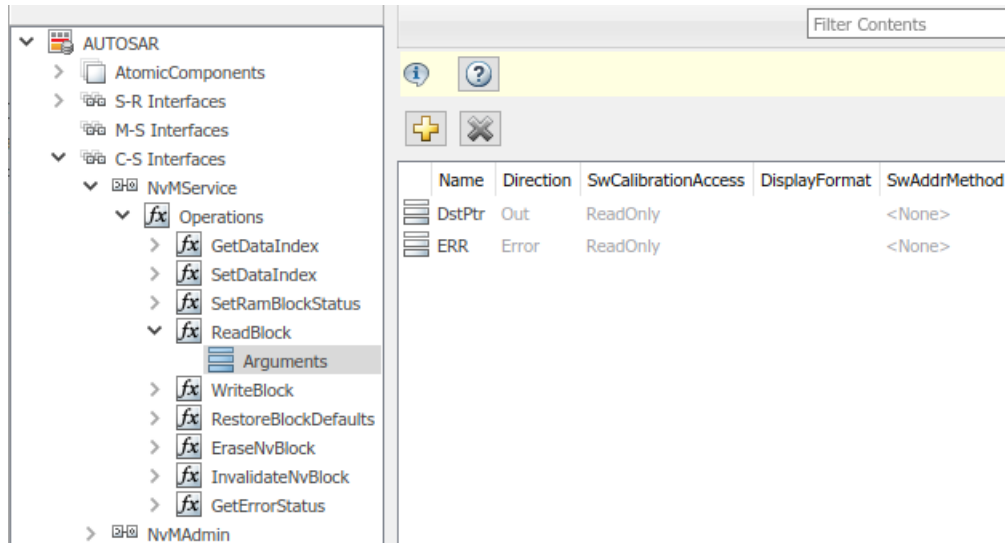
- To specify a multidimensional data type, you can use array syntax, such as `int8([1 1; 1 1])`.
- To specify a structured data type, you can create a Simulink.Parameter data object, type it with a Simulink.Bus object, and reference the parameter name.



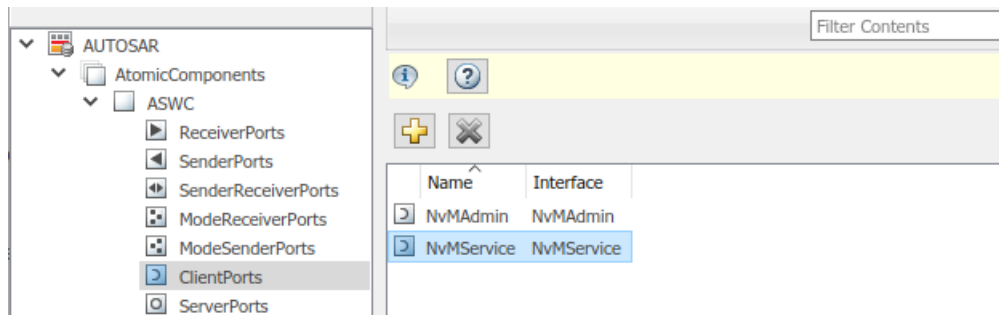
- 3 Open the Code Mappings editor. To update the Simulink to AUTOSAR mapping of the model with changes to Simulink function callers, click the **Update** button . The software creates AUTOSAR client-service interfaces, operations, and ports, and maps each Simulink function caller to an AUTOSAR client port and operation.

For example, for the `NvMServiceCaller` block in this example, for which the `ReadBlock` operation is selected:

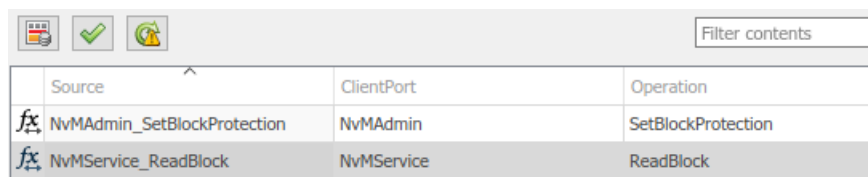
- The software creates C-S interface NvMService, and under NvMService, its supported operations. For each operation, arguments are provided with read-only properties. Here are the arguments for the NvMService operation ReadBlock displayed in the AUTOSAR Dictionary.



- The software creates a client port with the default name NvMService. Unlike the C-S-interface, operation, and argument names, the client port name can be customized. The client port is mapped to the NvMService interface.

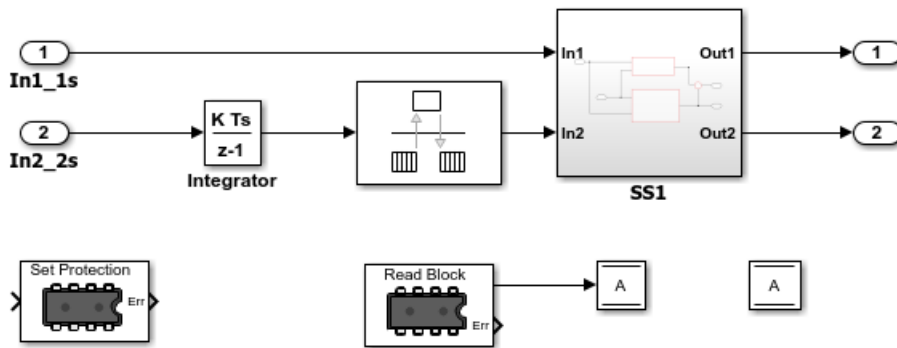


- The Code Mappings editor, **Function Callers** tab, maps the NvMService function caller block to AUTOSAR client port NvMService and AUTOSAR operation ReadBlock.



4 Optionally, build your model and examine the generated C and ARXML code.

In the block dialog step, if you selected operation ReadBlock for the NvMServiceCaller block, code generation requires adding data store blocks to the model. Connect the block first output to a Data Store Write block, and add a Data Store Memory block. For both blocks, specify data store name A. For example:



The C code includes the client calls to the BSW services, for example:

```
/* FunctionCaller: '<Root>/NmServiceCaller' */
Rte_Call_NvMService_ReadBlock(&rtDW.A);
...
/* FunctionCaller: '<Root>/NmAdminCaller' */
Rte_Call_NvMAdmin_SetBlockProtection(false);
```

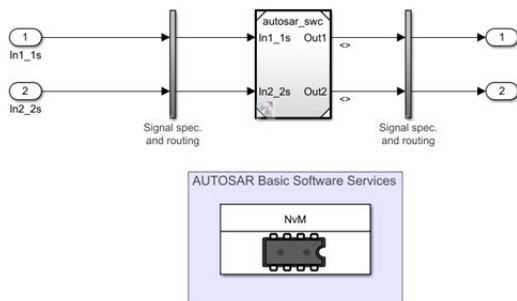
Generated RTE include files define the server operation call points, such as Rte_Call_NvMService_ReadBlock.

The ARXML code defines the BSW service operations called by the component as server call points, for example:

```
<SERVER-CALL-POINTS>
...
<ASYNCHRONOUS-SERVER-CALL-POINT UUID="...">
  <SHORT-NAME>SC_NvMService_ReadBlock</SHORT-NAME>
  <OPERATION-IREF>
    <CONTEXT-R-PORT-REF DEST="R-PORT-PROTOTYPE">
      /Company/Powertrain/Components/ASWC/NvMService
    </CONTEXT-R-PORT-REF>
    <TARGET-REQUIRED-OPERATION-REF DEST="CLIENT-SERVER-OPERATION">
      /AUTOSAR/Services/NvM/NvMService/ReadBlock
    </TARGET-REQUIRED-OPERATION-REF>
  </OPERATION-IREF>
  <TIMEOUT>1</TIMEOUT>
</ASYNCHRONOUS-SERVER-CALL-POINT>
</SERVER-CALL-POINTS>
```

- To simulate the component model, create a containing composition, system, or test harness model. In that containing model, insert reference implementations of the NvM ReadBlock and SetBlockProtection service operations.

The AUTOSAR Basic Software block library provides an NVRAM Service Component block, which provides reference implementations of NvM service operations. You can manually insert the block into a containing composition, system, or harness model, or automatically insert the block by creating a Simulink Test harness model.



For more information, see “Configure AUTOSAR Basic Software Service Implementations for Simulation” on page 7-33 and “Simulate AUTOSAR Basic Software Services and Run-Time Environment” on page 7-36.

See Also

NvMAdminCaller | NvMServiceCaller | NVRAM Service Component

Related Examples

- “Configure AUTOSAR Basic Software Service Implementations for Simulation” on page 7-33
- “Simulate AUTOSAR Basic Software Services and Run-Time Environment” on page 7-36
- “Configure AUTOSAR Client-Server Communication” on page 4-142

More About


- “Model AUTOSAR Nonvolatile Memory” on page 2-40
- “Model AUTOSAR Basic Software Service Calls” on page 7-12
- “Model AUTOSAR Communication” on page 2-21

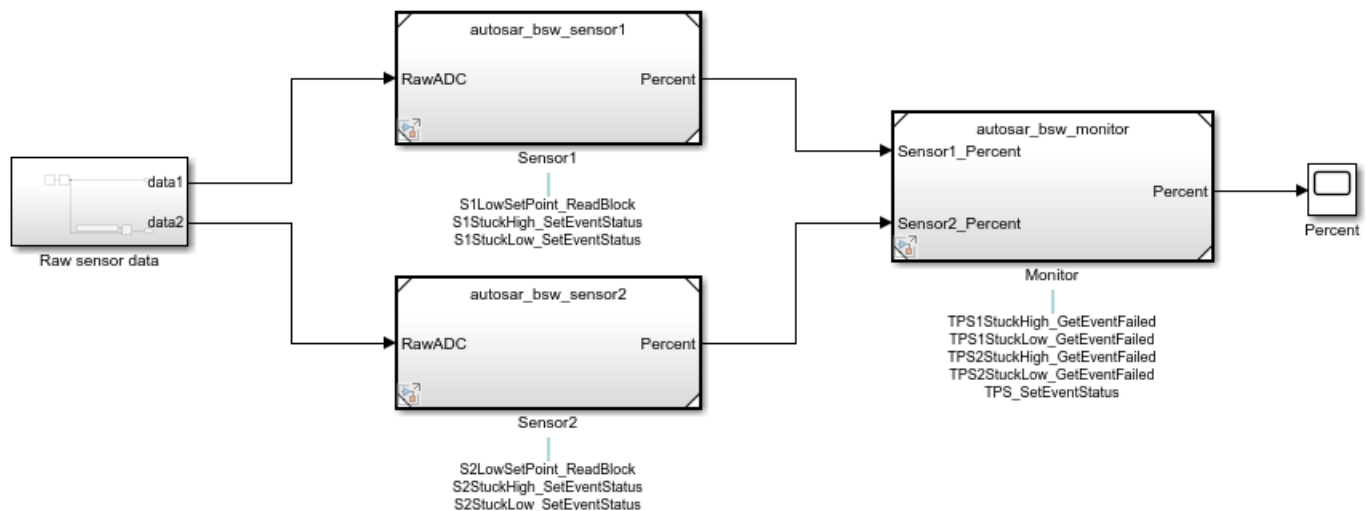
Configure AUTOSAR Basic Software Service Implementations for Simulation

AUTOSAR Blockset provides reference implementations of Diagnostic Event Manager (Dem), Function Inhibition Manager (FiM), and NVRAM Manager (NvM) services supported by AUTOSAR Basic Software (BSW) caller blocks. When coupled with the BSW caller blocks, the reference implementations allow you to configure and run system- or composition-level simulations of AUTOSAR BSW service calls. The ability to simulate calls into BSW services can help identify modeling problems before the AUTOSAR generated code reaches the AUTOSAR Runtime Environment (RTE).

To configure BSW caller blocks and BSW service reference implementations for simulation:

- 1 In one or more AUTOSAR component models, configure calls to AUTOSAR BSW services. Follow the procedures described in “Configure Calls to AUTOSAR Diagnostic Event Manager Service” on page 7-14, “Configure Calls to AUTOSAR Function Inhibition Manager Service” on page 7-18, or “Configure Calls to AUTOSAR NVRAM Manager Service” on page 7-28.
- 2 For simulation purposes, create a composition, system, or harness model that contains instances of the AUTOSAR component models. This procedure uses AUTOSAR example model `autosar_bsw_presim`, which is used in example “Simulate AUTOSAR Basic Software Services and Run-Time Environment” on page 7-36. The referenced component models call NvM service operation `ReadBlock` and Dem service operations `SetEventStatus` and `GetEventFailed`.

 autosar_bsw_presim ▶



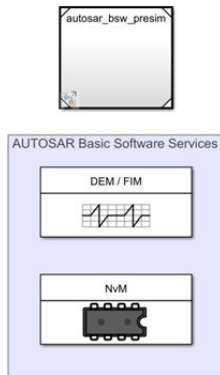
Alternatively, as shown in the next step, you can use Simulink Test to create a harness model.

- 3 In the containing model, provide reference implementations of the Dem or NvM service operations that your AUTOSAR component models call. For Dem and NvM service operations, the AUTOSAR Basic Software block library provides Diagnostic Service Component and NVRAM Service Component blocks.

You can insert a Service Component block in either of two ways:

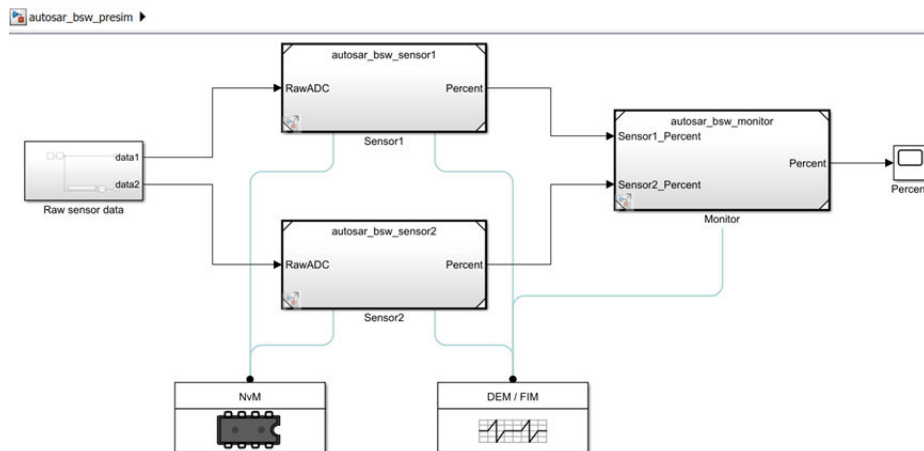
- Automatically insert the block by creating a Simulink Test harness model. In an AUTOSAR component model or a containing model, on the **Apps** tab, click **Simulink Test**. Then, on the

Tests tab, click **Add Test Harness**. In the Create Test Harness dialog box, click **OK**. The software compiles the model, adds a Diagnostic or NVRAM Service Component block, and creates ports and other elements required for simulation. For example, here is a test harness created for the presimulation integration model in example “Simulate AUTOSAR Basic Software Services and Run-Time Environment” on page 7-36.



- Manually insert the block into a containing composition, system, or harness model. Using the Library Browser or `add_block` command, or by typing block names in the model window, add a service component block to the containing model. Example “Simulate AUTOSAR Basic Software Services and Run-Time Environment” on page 7-36 uses these commands to add Diagnostic Service Component and NVRAM Service Component blocks to a containing model and then update the model diagram.

```
add_block('autosarlibdem/Diagnostic Service Component',...
  'autosar_bsw_presim/Diagnostic Service Component');
add_block('autosarlibnvm/NVRAM Service Component',...
  'autosar_bsw_presim/NVRAM Service Component');
set_param('autosar_bsw_presim','SimulationCommand','update');
```



- Each service component block has prepopulated parameters. Examine the parameter settings and consider if modifications are required, based on how you are using the Dem, FiM, and NvM service operations. For more information, see Diagnostic Service Component and NVRAM Service Component.
- Simulate the containing model. The simulation exercises the AUTOSAR Dem and NvM service calls in the component models. For a sample simulation, see example “Simulate AUTOSAR Basic Software Services and Run-Time Environment” on page 7-36.

See Also

Diagnostic Service Component | NVRAM Service Component

Related Examples

- “Simulate AUTOSAR Basic Software Services and Run-Time Environment” on page 7-36
- “Configure Calls to AUTOSAR Diagnostic Event Manager Service” on page 7-14
- “Configure Calls to AUTOSAR Function Inhibition Manager Service” on page 7-18
- “Configure Calls to AUTOSAR NVRAM Manager Service” on page 7-28

More About

- “Model AUTOSAR Basic Software Service Calls” on page 7-12
- “Model AUTOSAR Communication” on page 2-21

Simulate AUTOSAR Basic Software Services and Run-Time Environment

Simulate AUTOSAR component calls to Basic Software memory and diagnostic services by using reference implementations.

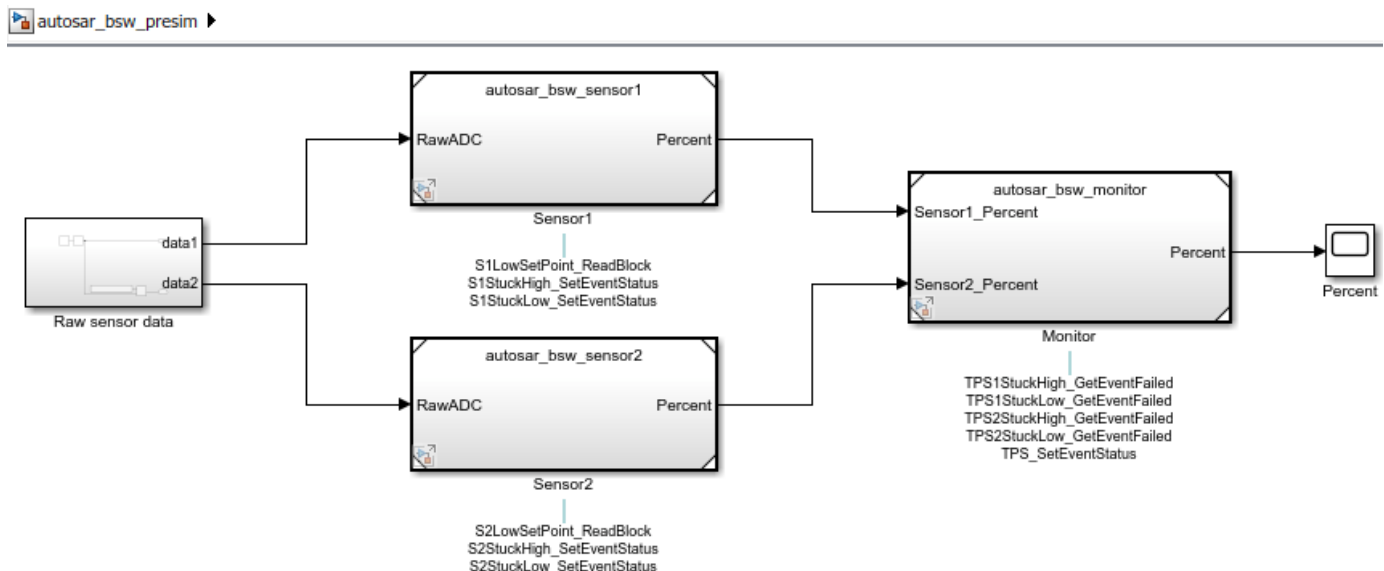
Configure Calls to AUTOSAR Basic Software Services

The AUTOSAR standard defines Basic Software (BSW) services that run in the AUTOSAR run-time environment. The services include NVRAM Manager (NvM) Diagnostic Event Manager (Dem), and Function Inhibition Manager (FiM) services. In the AUTOSAR run-time environment, AUTOSAR software components typically access BSW services using client-server or sender-receiver communication.

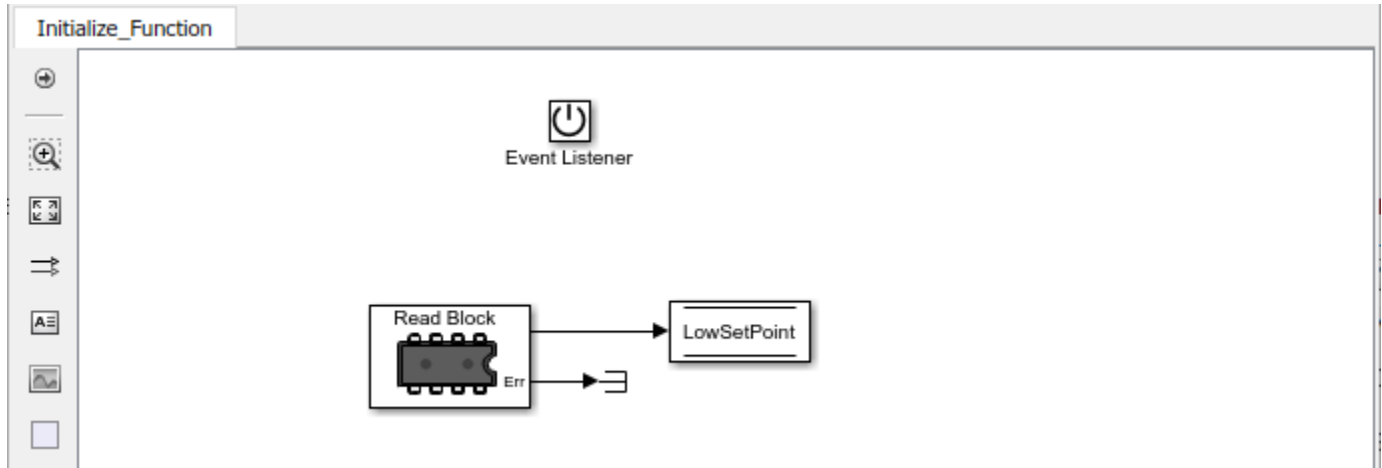
In your AUTOSAR software component model, to implement client calls to NvM, Dem, and FiM service interfaces, you drag and drop preconfigured NvM, Dem, and FiM caller blocks. Each block has prepopulated parameters, such as **Client port name** and **Operation**. You configure the block parameters, for example, to select a service operation to call. To configure the added caller blocks in the AUTOSAR software component, you synchronize the model. The software creates AUTOSAR client-service interfaces, operations, and ports, and maps each Simulink function call to an AUTOSAR client port and operation. For more information, see “Configure Calls to AUTOSAR NVRAM Manager Service” on page 7-28, “Configure Calls to AUTOSAR Diagnostic Event Manager Service” on page 7-14, and “Configure Calls to AUTOSAR Function Inhibition Manager Service” on page 7-18.

Here is a throttle position integration model, which integrates two throttle position sensor components and a throttle position monitor component. The sensor components take a raw throttle position sensor (TPS) value and convert it to a TPS percent value. The monitor component takes the TPS percent values provided by the primary and secondary sensor components and decides which TPS signal to pass through. The sensor components call BSW NvM and Dem services, and the monitor component calls BSW Dem services.

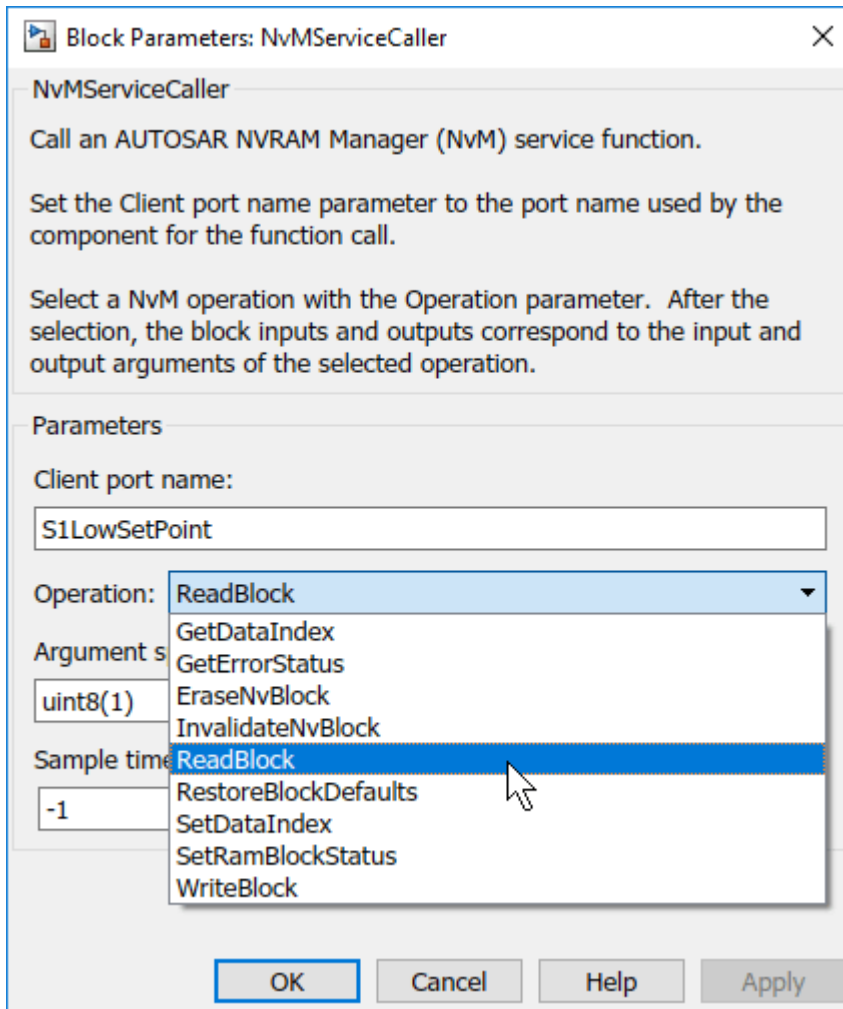
```
open_system('autosar_bsw_presim');
```



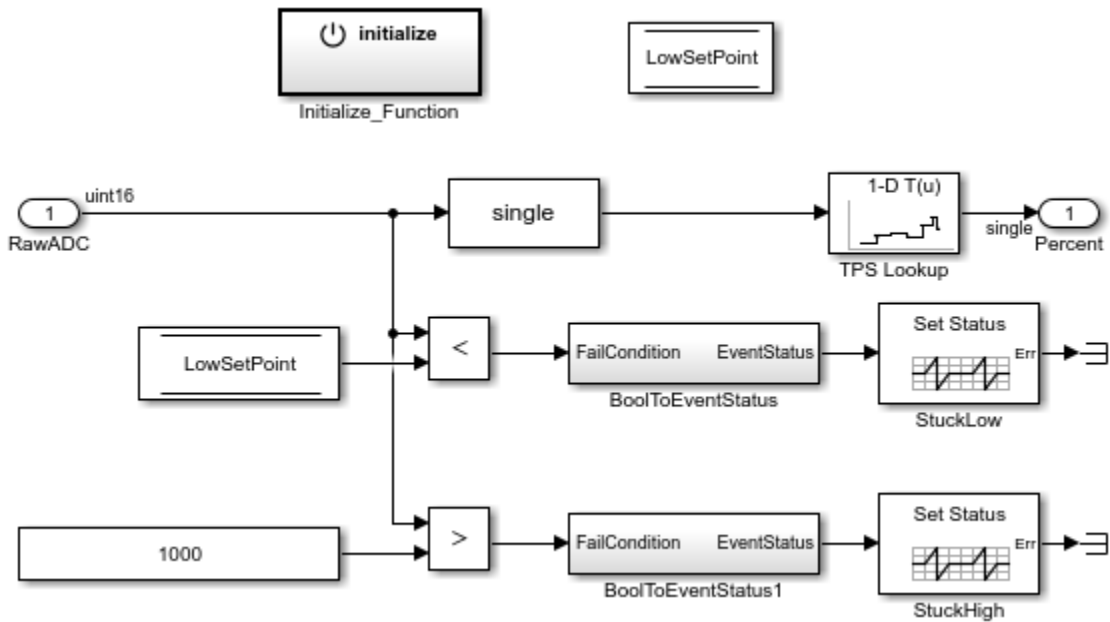
Sensor components `autosar_bsw_sensor1` and `autosar_bsw_sensor2` each contain an Initialize Function block, which calls the NvM service interface `NvMService`. The calls are implemented using the Basic Software library block `NvMServiceCaller`. Each block is configured to call the `NvMService` operation `ReadBlock`. The `ReadBlock` calls use client ports `S1LowSetPoint` and `S2LowSetPoint`. Here is the Initialize Function block for `autosar_bsw_sensor1`.



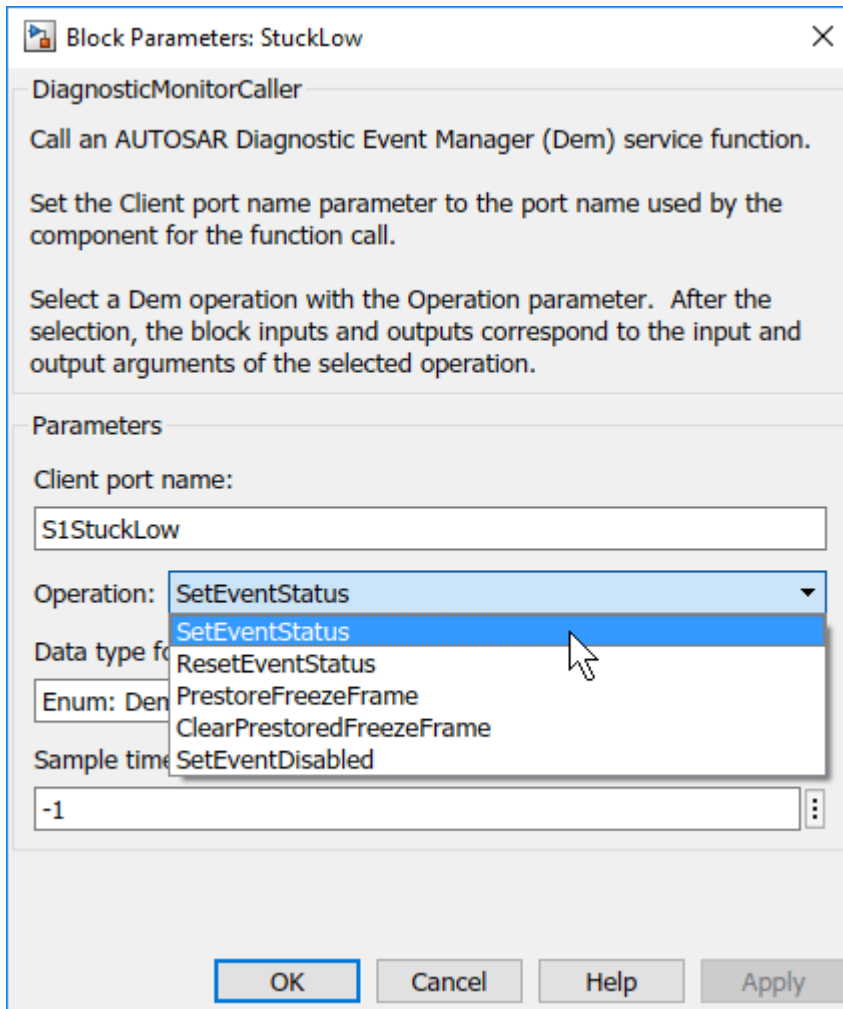
Here is the `NvMServiceCaller` block dialog box for the `ReadBlock` call in the Initialize Function block. For more information, see `NvMServiceCaller`.



Sensor components `autosar_bsw_sensor1` and `autosar_bsw_sensor2` each contain two calls to the Dem service interface `DiagnosticMonitor`. Both calls are implemented using the Basic Software library block `DiagnosticMonitorCaller`. Each block is configured to call the `DiagnosticMonitor` operation `SetEventStatus`. The `SetEventStatus` calls use client ports `S1StuckLow`, `S1StuckHigh`, `S2StuckLow`, and `S2StuckHigh`.

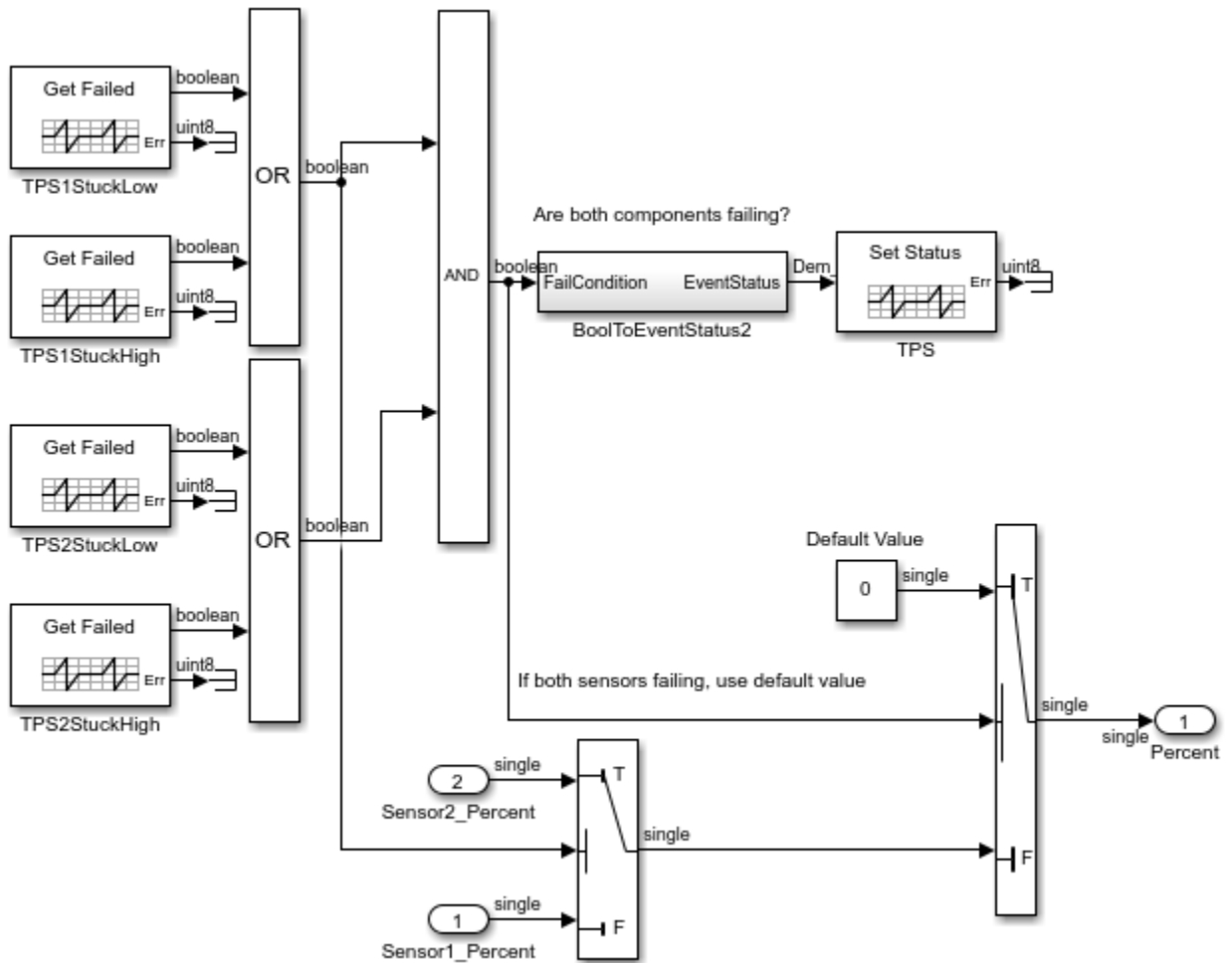


Here is the DiagnosticMonitorCaller block dialog box for the StuckLow call in the first sensor component. For more information, see DiagnosticMonitorCaller.

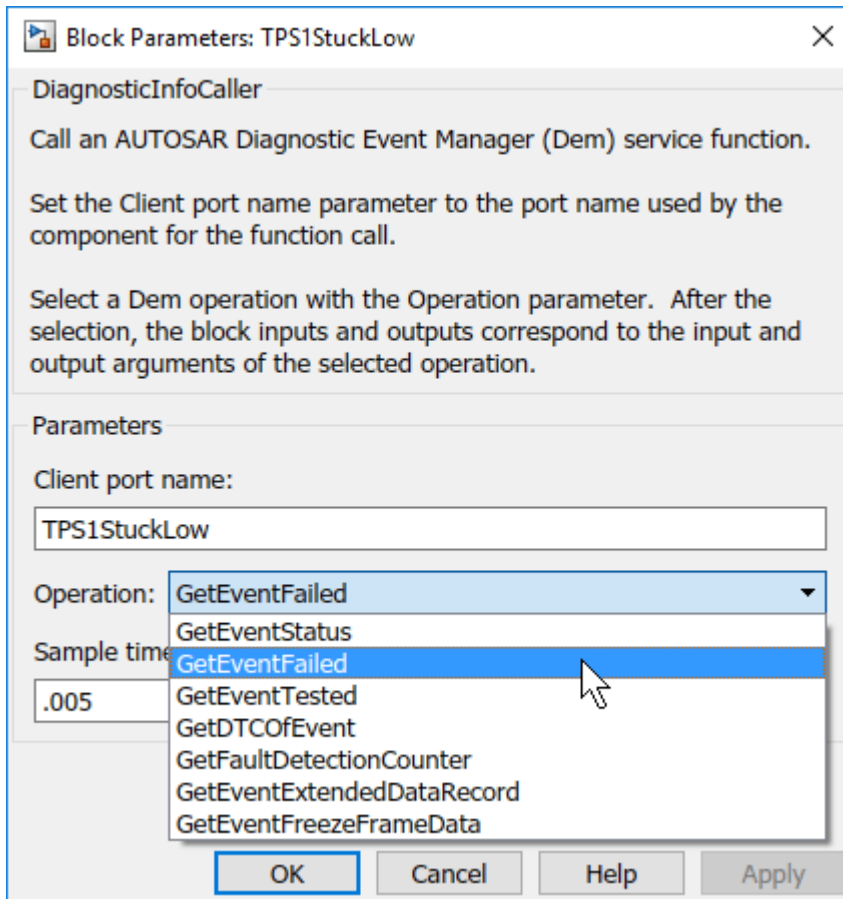


Monitor component `autosar_bsw_monitor` contains a call to the Dem service interface `DiagnosticMonitor` and four calls to the Dem service interface `DiagnosticInfo`.

- As in the sensor component, a `DiagnosticMonitorCaller` block implements the `DiagnosticMonitor` call, and it is configured to call the `SetEventStatus` operation. The client port name is `TPS`.
- The four `DiagnosticInfo` calls are implemented using the Basic Software library block `DiagnosticInfoCaller`. Each block is configured to call the `DiagnosticInfo` operation `GetEventFailed`. The `GetEventFailed` calls use client ports `TPS1StuckLow`, `TPS1StuckHigh`, `TPS2StuckLow`, and `TPS2StuckHigh`.



Here is the DiagnosticInfoCaller block dialog box for the TPS1StuckLow call. For more information, see DiagnosticInfoCaller.



If you have Simulink Coder and Embedded Coder software, you can generate C code and export ARXML descriptions for the NvM and Dem service calls. Open and build each component model. For example, to build model `autosar_bsw_monitor`, open the model. Press **Ctrl+B** or enter the MATLAB command `slbuild('autosar_bsw_monitor')`.

To see the results of the model build, examine the code generation report.

Configure Reference Implementations of AUTOSAR Basic Software Services for Simulation

To simulate an AUTOSAR component model that calls BSW services, create a containing composition, system, or harness model. In that containing model, provide reference implementations of the NvM, Dem, and FiM service operations called by the component.

The AUTOSAR Basic Software block library includes an NVRAM Service Component block and a Diagnostic Service Component block. The blocks provide reference implementations of NvM, Dem, and FiM service operations. To support simulation of component calls to the NvM, Dem, and FiM services, include the blocks in the containing model. You can insert the blocks in either of two ways:

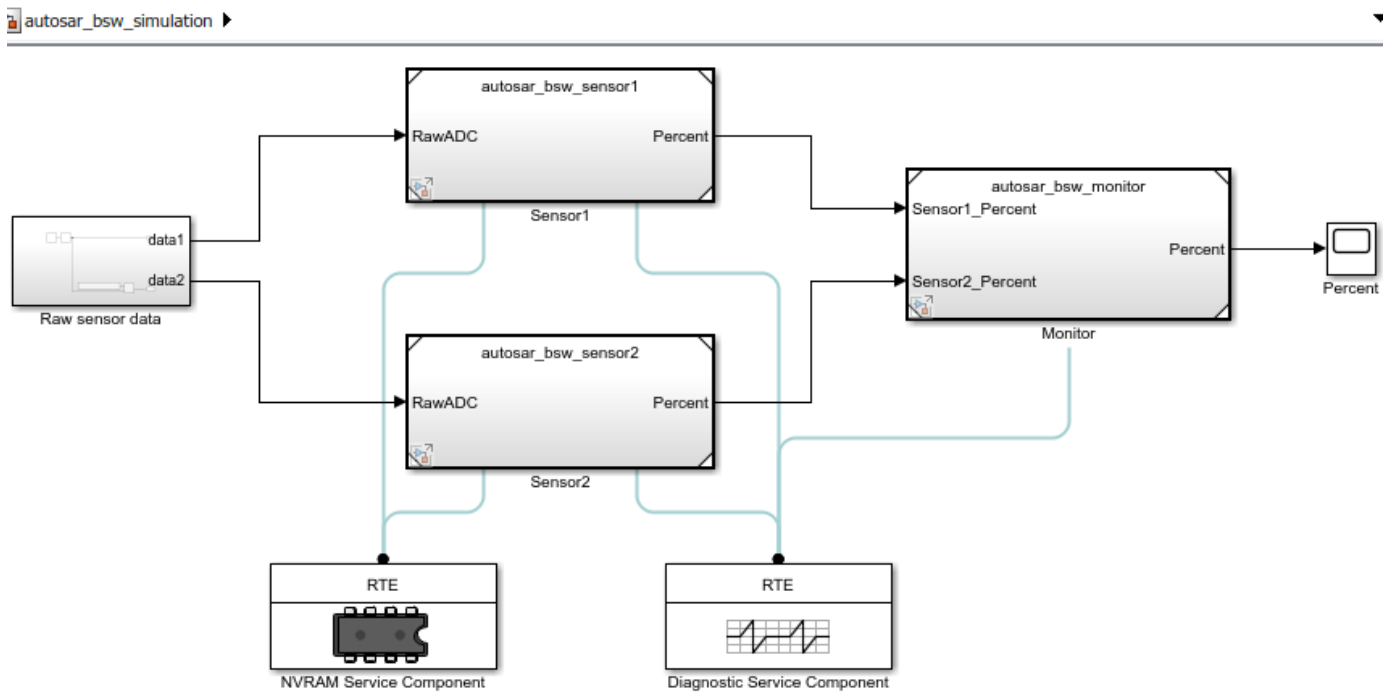
- Automatically insert the blocks by creating a Simulink Test harness model
- Manually insert the blocks into a containing composition, system, or harness model

To automatically insert Service Component blocks for a model that calls BSW NvM, Dem, and FiM services, open the model (or a containing model) and create a Simulink Test test harness (requires Simulink Test). For more information, see “Create a Test Harness” (Simulink Test). Creating a test

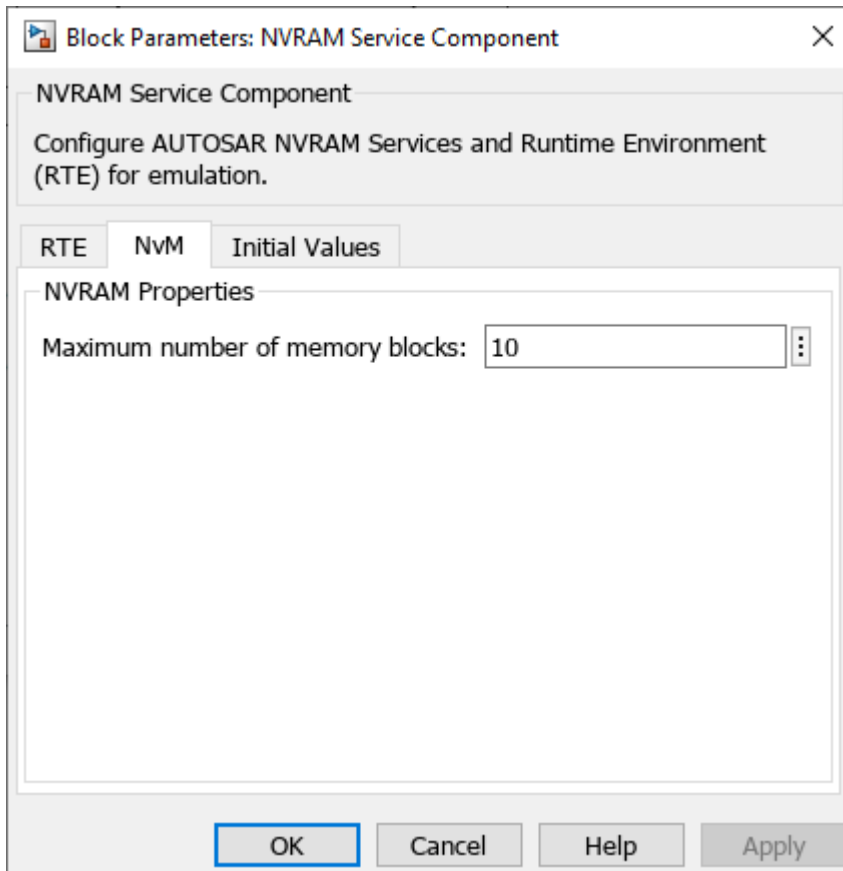
harness compiles the model, adds the Service Component blocks, and creates ports and other elements required for simulation.

This example manually inserts Service Component blocks for NvM and Dem service calls. Open the integration model `autosar_bsw_presim`. Using the Library Browser or `add_block` commands, or by typing block names in the model window, add the NVRAM and Diagnostic Service Component blocks to the model.

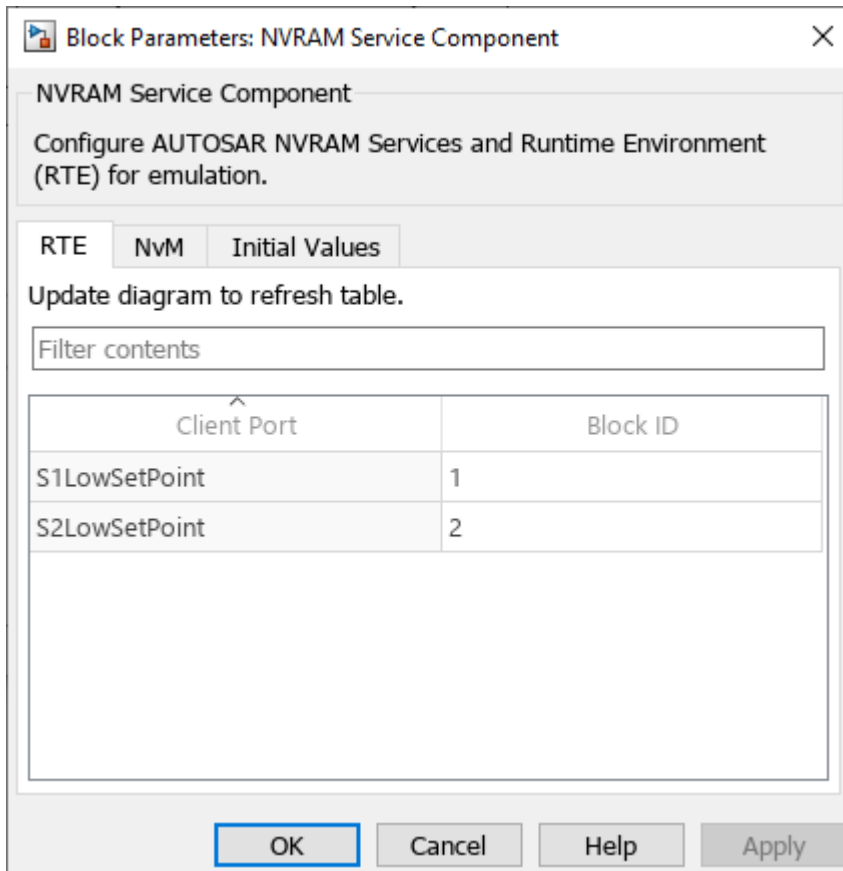
```
open_system('autosar_bsw_presim');
add_block('autosarlibnvm/NVRAM Service Component','autosar_bsw_presim/NVRAM Service Component');
add_block('autosarlibdem/Diagnostic Service Component','autosar_bsw_presim/Diagnostic Service Component');
set_param('autosar_bsw_presim','SimulationCommand','update');
```



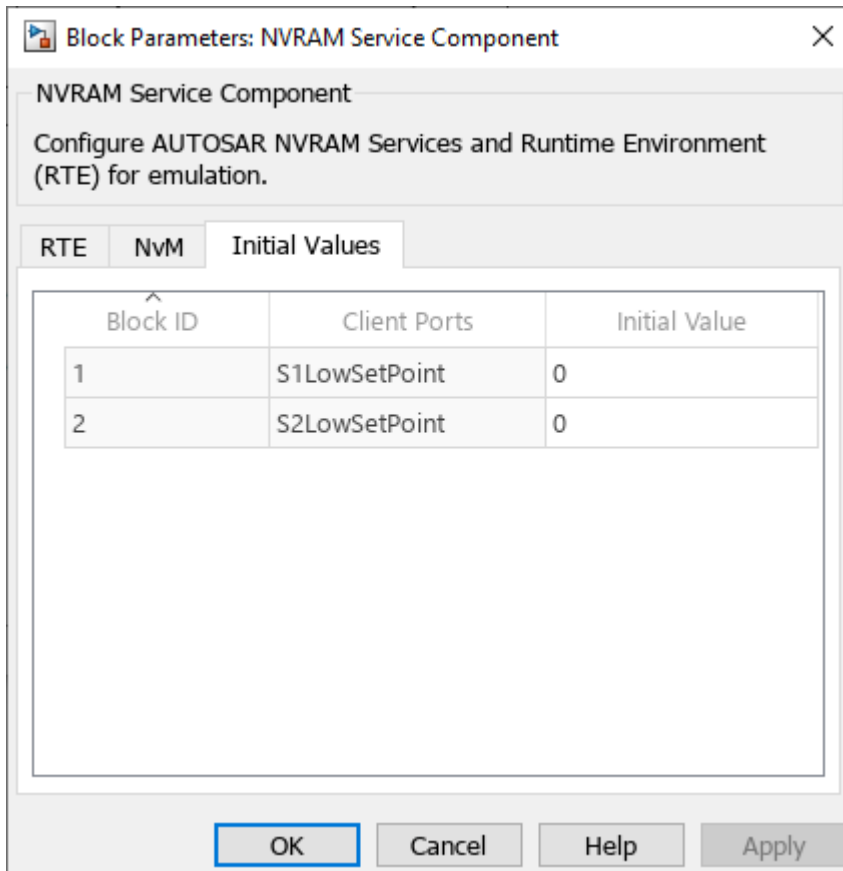
The NVRAM Service Component block has prepopulated parameters, including run-time environment (RTE) parameters and **NVRAM Properties** parameters. Examine the parameter settings and consider if any require modifying, based on how you are using the NvM service operations. For more information, see NVRAM Service Component.



The RTE tab table lists component client ports and their mapping to NvM service block IDs. Each row in the table represents a call into NvM services from a Basic Software caller block. Calls that act on the same NvM block typically use the same block ID. This example maps the NvM ReadBlock client ports to different block IDs.

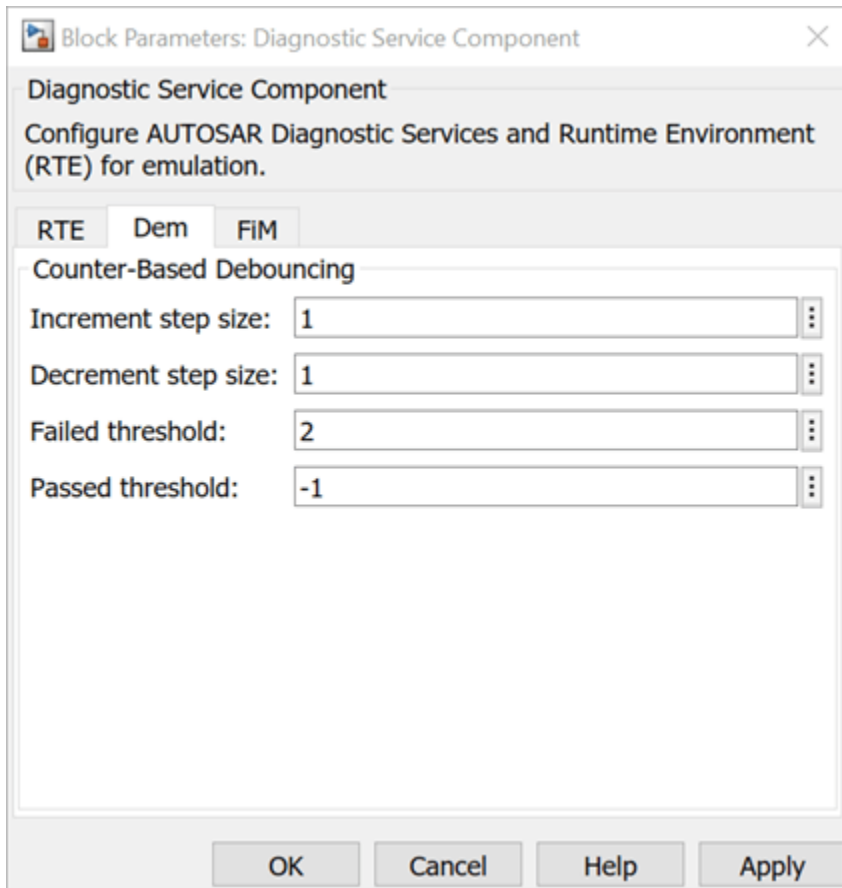


The Initial Values tab table lists component client ports and their initial values for simulation. The default initial value is 0.

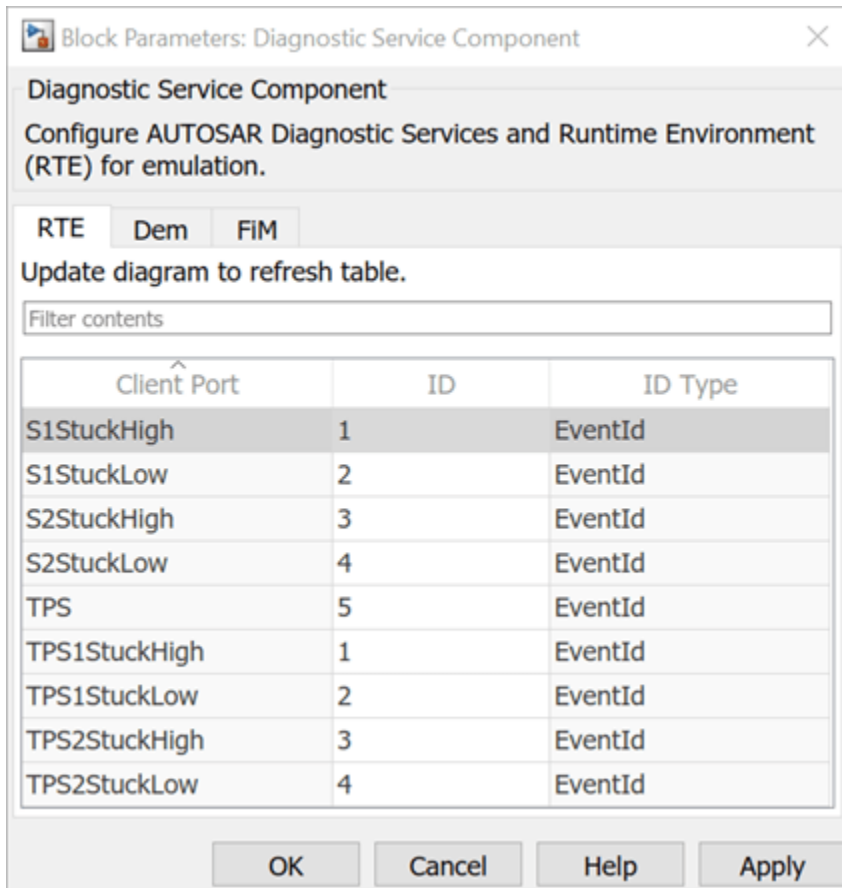


The Diagnostic Service Component block has prepopulated parameters, including RTE parameters and Dem **Counter-Based Debouncing** parameters. Examine the parameter settings and consider if any require modifying, based on how you are using the Dem service operations.

The **Counter-Based Debouncing** parameters control the counter-based debounce algorithm provided by the Dem service reference implementations. During multiple simulation runs, you can tune event step size and threshold parameters and observe the effects. For more information, see Diagnostic Service Component.



The RTE tab table lists component client ports and their mapping to Dem or FiM service IDs (in this example, event IDs). Each row in the table represents a call into Dem services from a Basic Software caller block. Calls that act on the same event typically use the same event ID. This example maps the Dem `SetEventStatus` client ports to different event IDs, and then maps the Dem `GetEventFailed` client ports to event IDs that are shared with `SetEventStatus` ports. For example, `SetEventStatus` port `S1StuckHigh` and `GetFailedEvent` port `TPS1StuckHigh` share event ID 1; `S1StuckLow` and `TPS1StuckLow` share event ID 2; and so on.



Simulate Calls to AUTOSAR NvM and Dem Services

After configuring NVRAM and Diagnostic Service Component blocks in the integration model, simulate the model. The simulation exercises the AUTOSAR NvM and Dem service calls in the throttle position sensor and monitor component models.

```
open_system('autosar_bsw_simulation');
simOutIntegration = sim('autosar_bsw_simulation');
```

Related Links

- “Model AUTOSAR Basic Software Service Calls” on page 7-12
- “Configure Calls to AUTOSAR NVRAM Manager Service” on page 7-28
- “Configure Calls to AUTOSAR Diagnostic Event Manager Service” on page 7-14
- “Configure AUTOSAR Basic Software Service Implementations for Simulation” on page 7-33

Configure and Simulate AUTOSAR Function Inhibition Service Calls

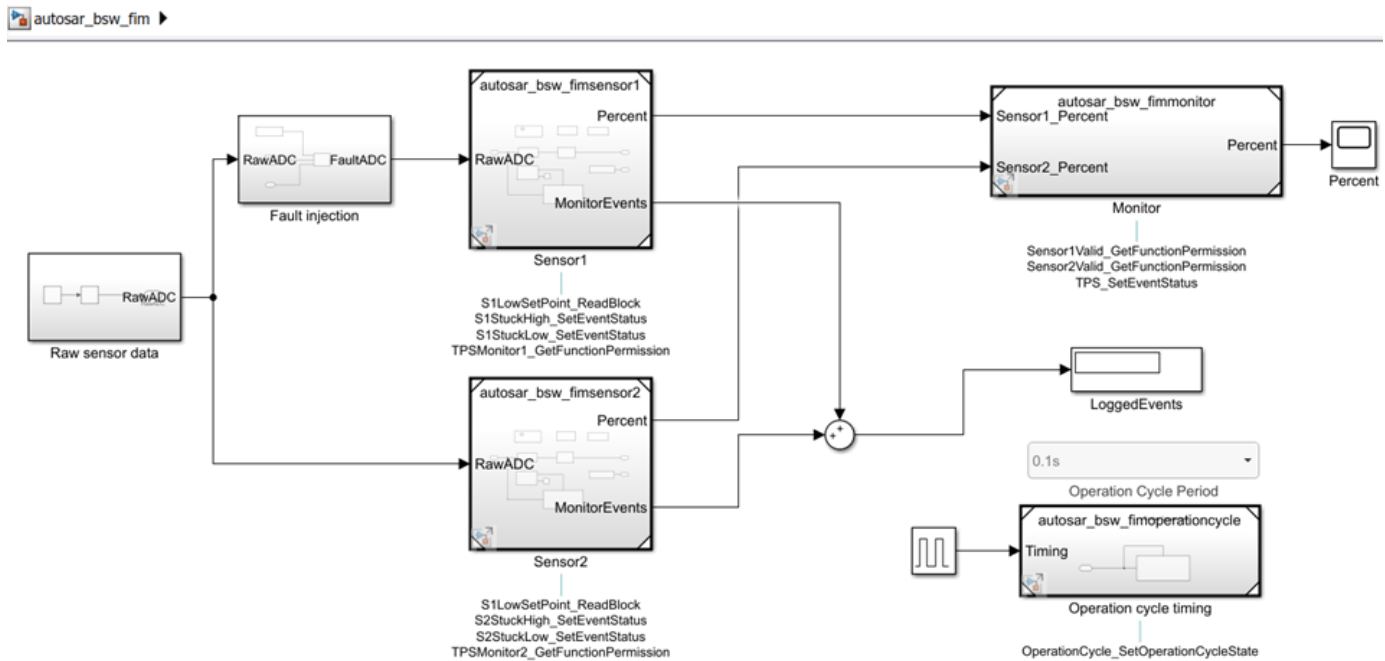
Simulate AUTOSAR component calls to Basic Software function inhibition and related services by using reference implementations.

Configure Calls to AUTOSAR Basic Software Services

The AUTOSAR standard defines Basic Software (BSW) services that run in the AUTOSAR run-time environment. The services include Diagnostic Event Manager (Dem), Function Inhibition Manager (FiM), and NVRAM Manager (NvM) services. In the AUTOSAR run-time environment, AUTOSAR software components typically access BSW services using client-server or sender-receiver communication.

In your AUTOSAR software component model, to implement client calls to FiM and related Dem service interfaces, you drag and drop preconfigured FiM and Dem caller blocks. Each block has prepopulated parameters, such as **Client port name** and **Operation**. You configure the block parameters, for example, to select a service operation to call. To configure the added caller blocks in the AUTOSAR software component, you synchronize the model. The software creates AUTOSAR client-service interfaces, operations, and ports, and maps each Simulink function call to an AUTOSAR client port and operation. For more information, see “Configure Calls to AUTOSAR Function Inhibition Manager Service” on page 7-18.

Here is a function inhibition integration model, which integrates two sensor components, a monitor component, and an operation cycle component. The sensor components call BSW FiM and Dem (and NvM) services, the monitor component calls BSW FiM and Dem services, and the operation cycle component calls a BSW Dem service.



The sensor and monitor components each call the FiM service interface `FunctionInhibition`. The calls are implemented using the BSW library block `Function Inhibition Caller`. Each block instance is configured to call the `FunctionInhibition` operation `GetFunctionPermission`.

The operation cycle component calls the Dem service interface `OperationCycle`. The call is implemented using the BSW library block `DiagnosticOperationCycleCaller`. The block is configured to call the `OperationCycle` operation `SetOperationCycleState`.

Configure Reference Implementations of AUTOSAR Basic Software Services for Simulation

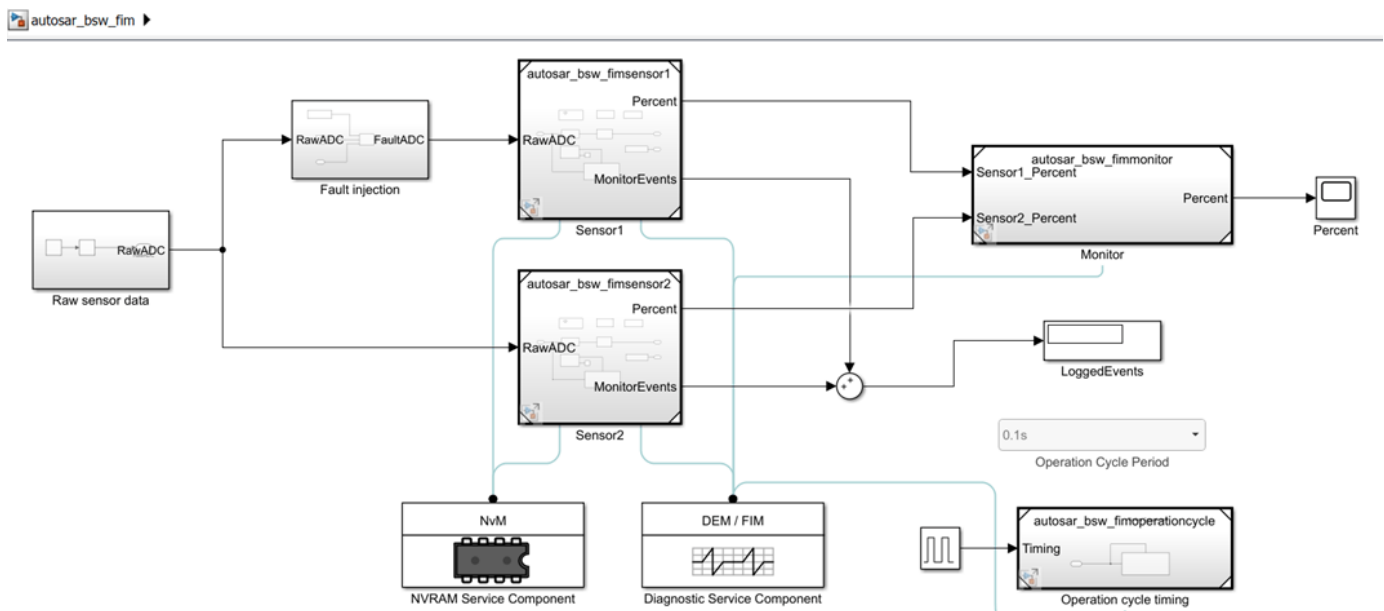
To simulate an AUTOSAR component model that calls BSW services, create a containing composition, system, or harness model. In that containing model, provide reference implementations of the Dem, FiM, and NvM service operations called by the component.

The AUTOSAR Basic Software block library includes a `Diagnostic Service Component` block and an `NVRAM Service Component` block. The blocks provide reference implementations of Dem, FiM, and NvM service operations. To support simulation of component calls to the Dem, FiM, and NvM services, include the blocks in the containing model. You can insert the blocks in either of two ways:

- Automatically insert the blocks by creating a Simulink Test harness model
- Manually insert the blocks into a containing composition, system, or harness model, and then update the model

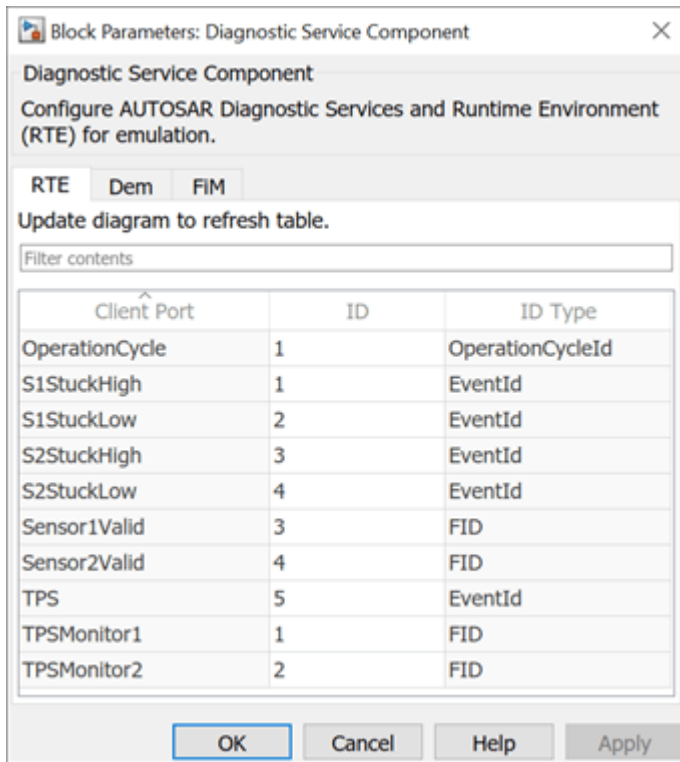
Here is the function inhibition integration model after manually inserting `Diagnostic` and `NVRAM Service Component` blocks. To display function connections, on the **Debug** tab, select **Information Overlays > Function Connectors**.

```
open_system('autosar_bsw_fim');
```

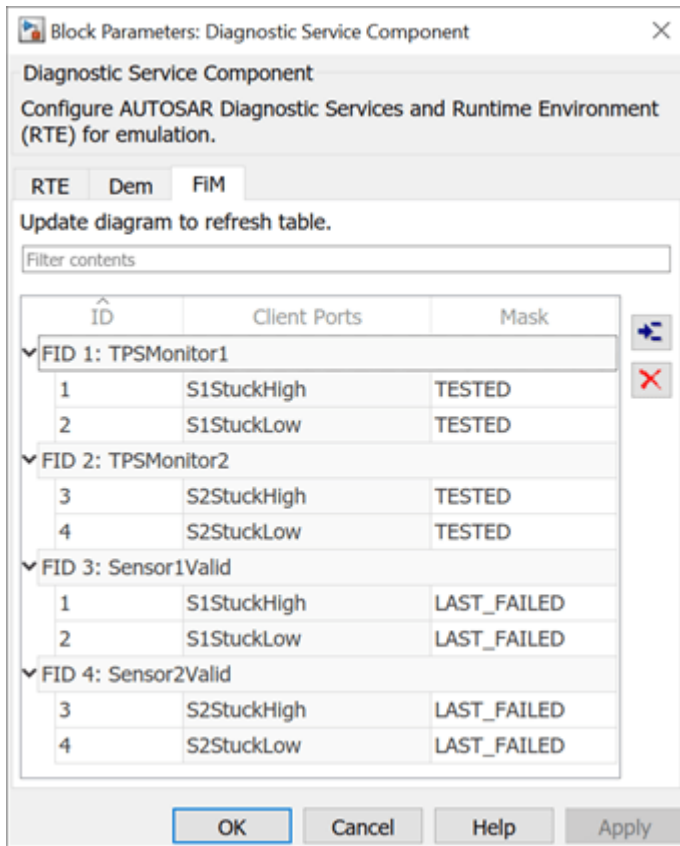


The `Diagnostic Service Component` block has prepopulated parameters, including RTE service ID parameters, Dem **Counter-Based Debouncing** parameters, and FiM inhibition condition parameters. The RTE tab lists component client ports and their mapping to Dem or FiM service IDs for events, operation cycles, or functions with inhibition conditions. Each row in the table represents

a call into Dem or FiM services from a Basic Software caller block, for which you can modify an ID value.



The FIM tab lists function identifiers (FIDs) and their associated inhibition conditions and client ports. The tab provides graphical controls for adding or removing inhibition conditions for a selected FID. For each inhibition condition, select ID and mask values.



For more information, see “Configure Calls to AUTOSAR Function Inhibition Manager Service” on page 7-18.

Simulate Calls to AUTOSAR FiM and Dem Services

After configuring Diagnostic and NVRAM Service Component blocks in the integration model, simulate the model. The simulation exercises the AUTOSAR FiM and Dem service calls in the sensor, monitor, and operation cycle component models.

```
open_system('autosar_bsw_fim');
simOutIntegration = sim('autosar_bsw_fim');
```

Related Links

- “Model AUTOSAR Basic Software Service Calls” on page 7-12
- “Configure Calls to AUTOSAR Function Inhibition Manager Service” on page 7-18
- “Configure AUTOSAR Basic Software Service Implementations for Simulation” on page 7-33

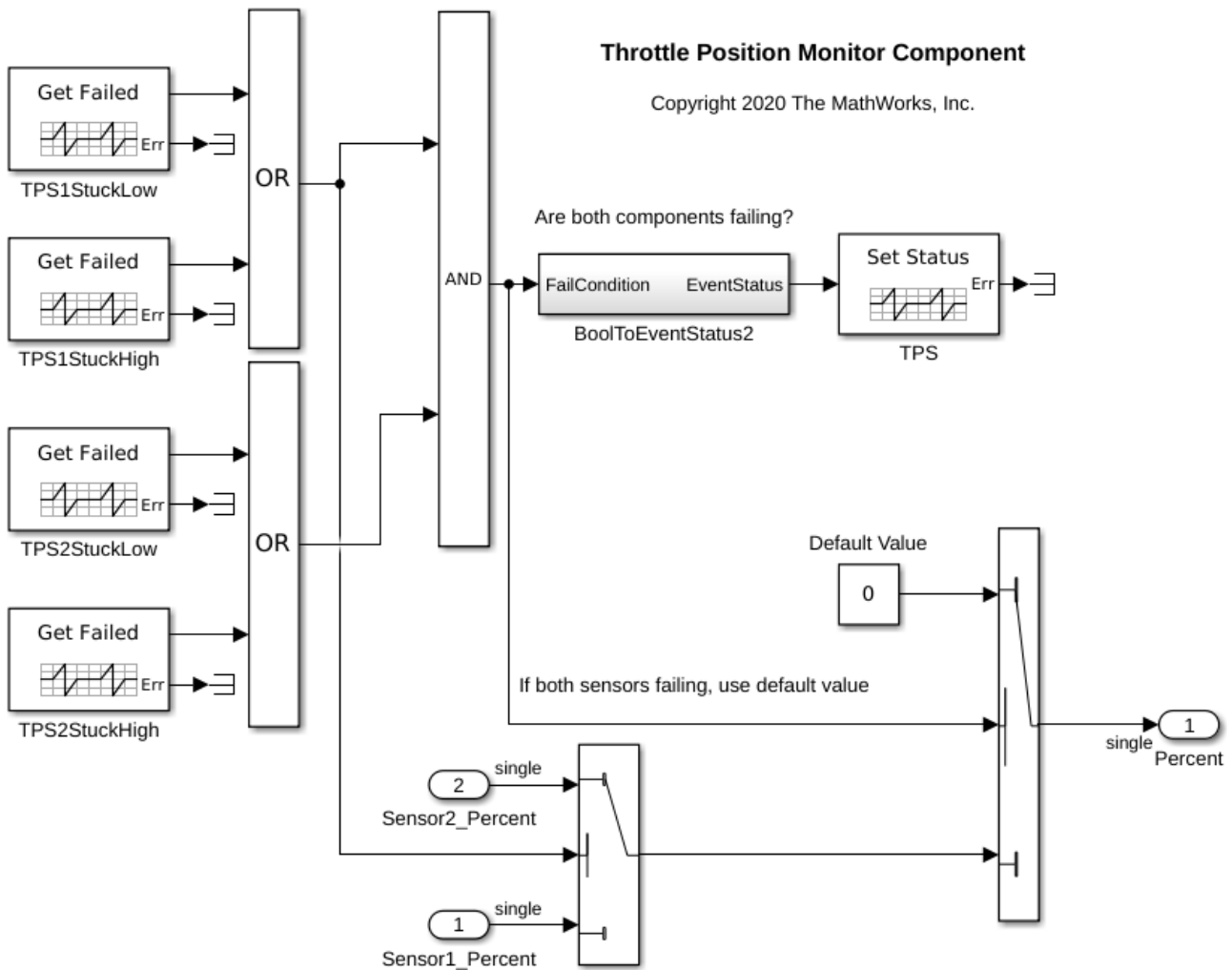
Simulate and Verify AUTOSAR Component Behavior by Using Diagnostic Fault Injection

This example shows how to simulate and verify the behavior of AUTOSAR components modeled in Simulink® that contain calls into the AUTOSAR Diagnostic Event Manager (Dem). You can gain quick testing coverage by overriding the diagnostic status of specific events or verify component recovery by injecting transient event failures.

Override AUTOSAR Diagnostic Statuses to Gain Coverage

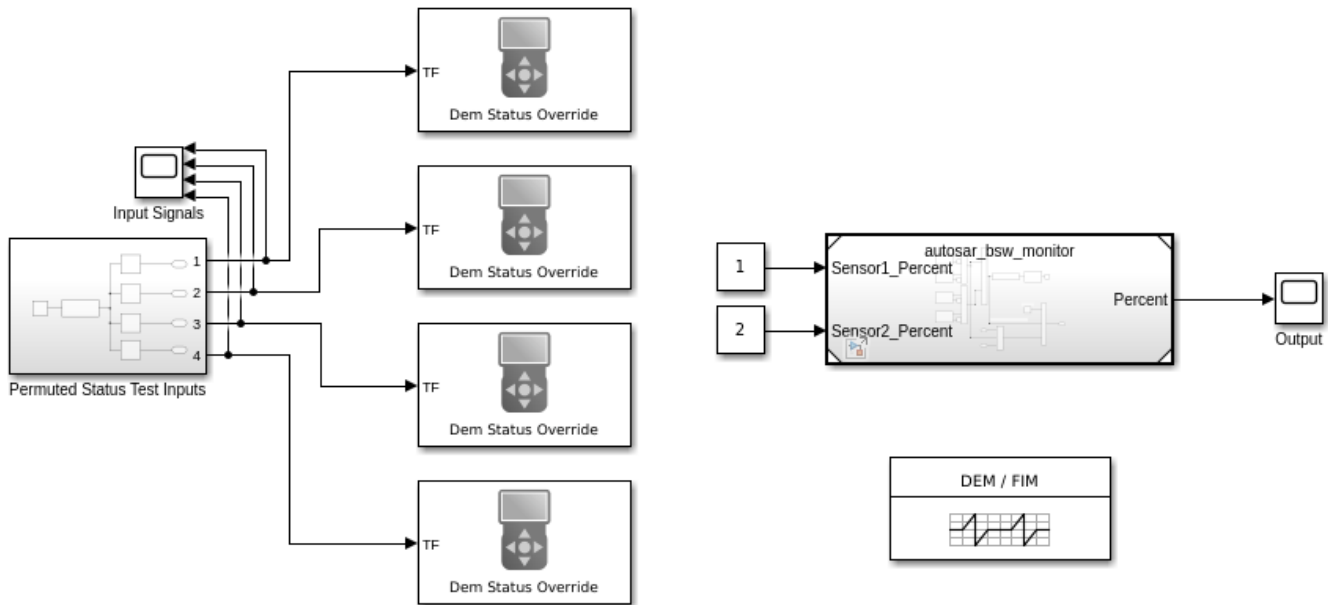
Component models often call into the AUTOSAR Diagnostic Event Manager (Dem) to communicate localized errors to the rest of the system. They may also query the diagnostic status of other systems by making calls to `getEventStatus`.

To show how to simulate and verify behavior by overriding the status of events, an example throttle position monitor component is shown. This component contains four calls to the Diagnostic Event Manager to query if four specific events have failed on their last evaluation. The status of these events determines if the primary and secondary sensor inputs can be passed on to the rest of the system. If both sensors report event failures, then a default value is passed on as the sensor input and a `setEventStatus` call reports a general failure to the system.

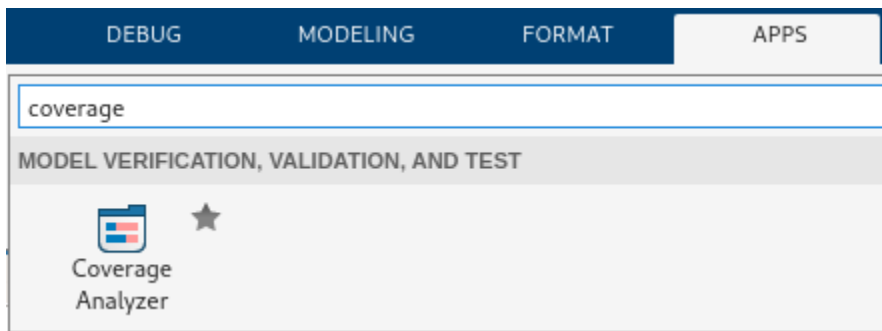


To verify that this component works as designed, a test harness model capable of exercising the branches of the model is shown below. In the test harness, the values 1 and 2 distinguish the two sensor inputs and a Diagnostic Service Component block provides simulation for calls to GetEventFailed and SetEventStatus. The Dem Status Override blocks are then added to override the Test Failed bit of the UDS status byte for each event.

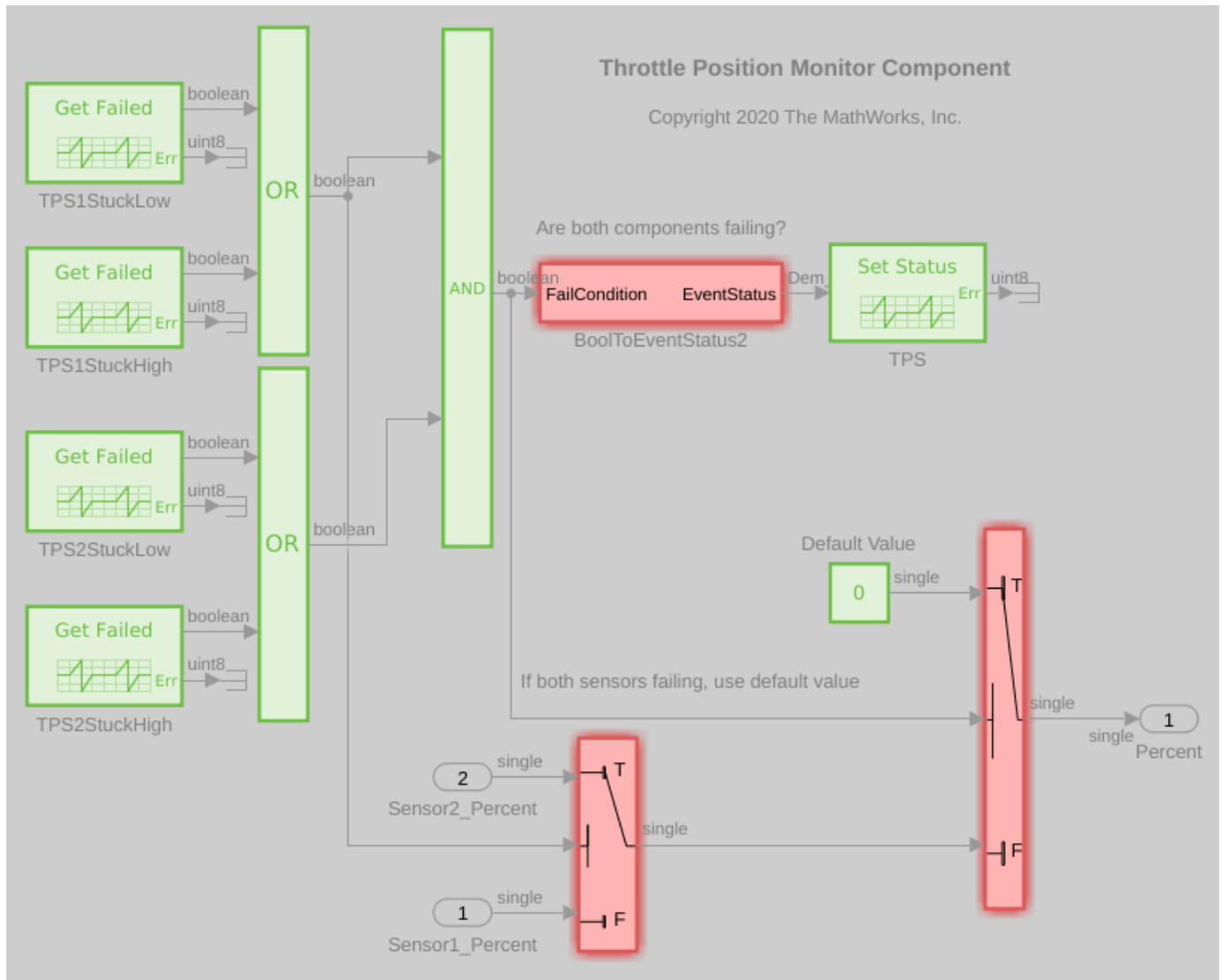
```
open_system('autosar_bsw_override_test');
```



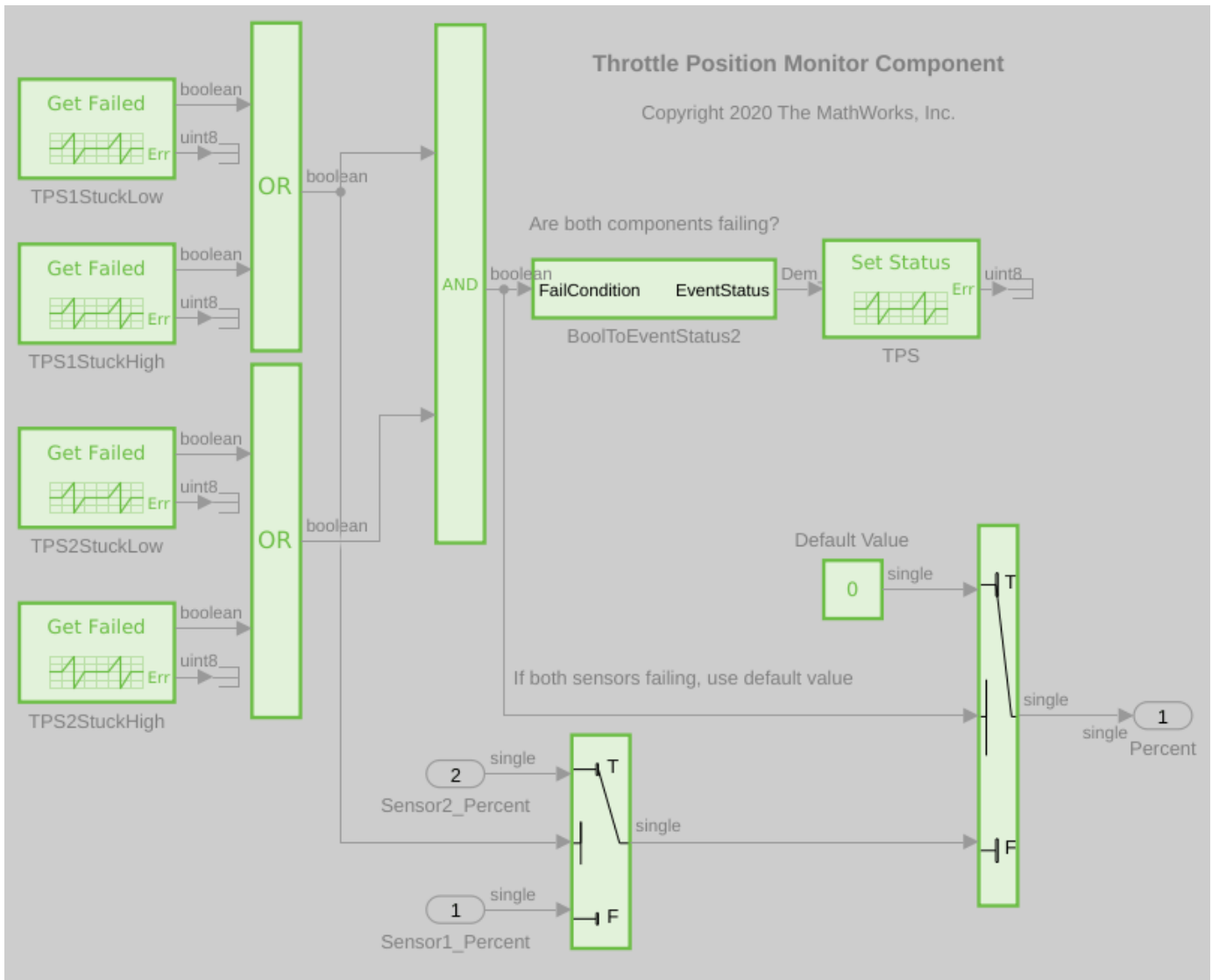
To get a more visual understanding of the test conditions and coverage for the example component in the test harness, from the Apps gallery, you can use the **Coverage Analyzer** app.



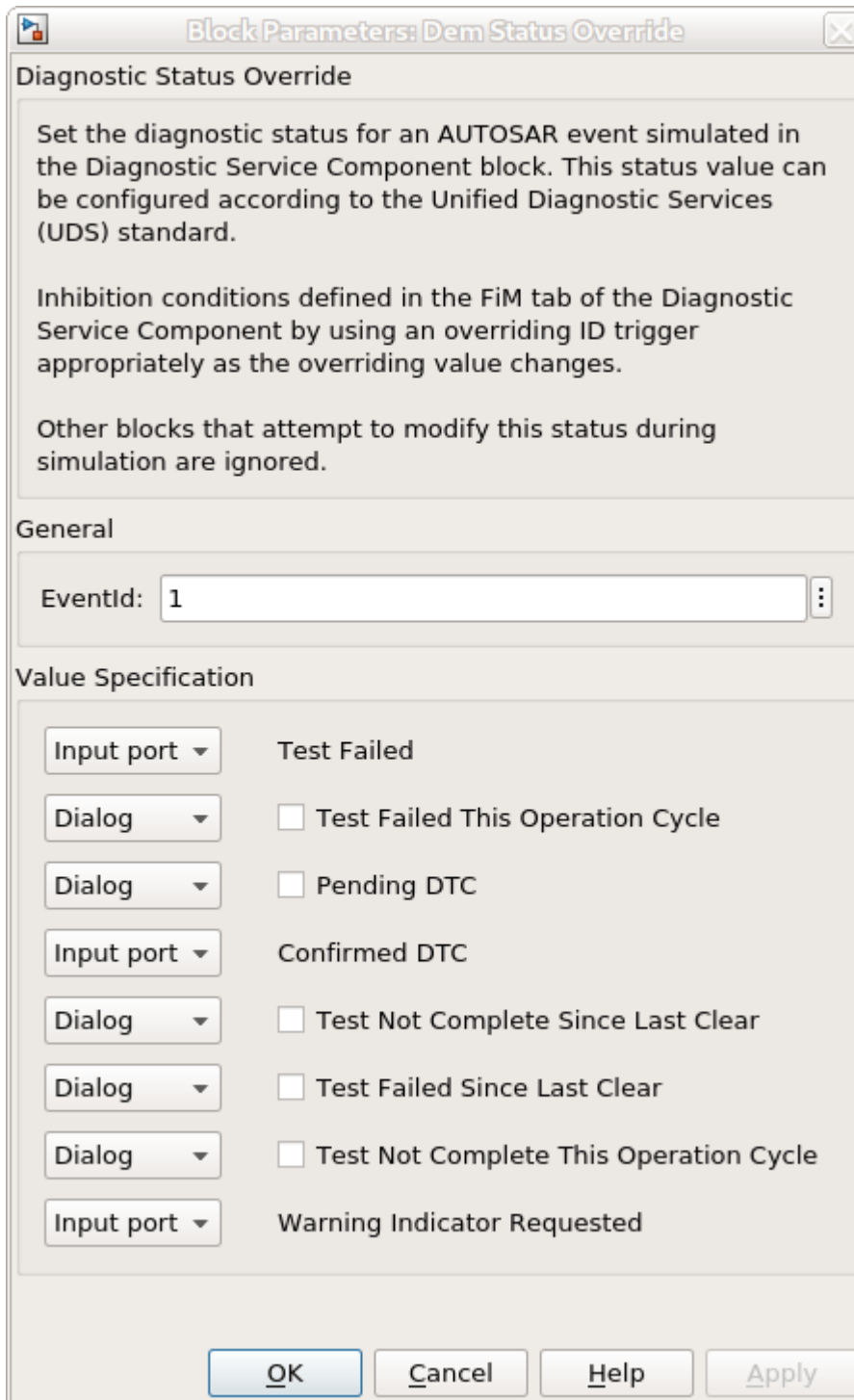
The view below shows the test coverage of the component model when the Dem Status Override blocks are removed. Given that the GetEventFailed call returns false without external input, you are not able to achieve full decision coverage on the Switch blocks in the component.



When Dem Status Override blocks are used in the harness, you are able to obtain full coverage of the component.



This example has shown how to obtain testing coverage by using the Dem Status Override block. While this specific example was narrow in scope, and it focused on the Test Failed bit, you can use the Dem Status Override block to configure each bit of the Unified Diagnostic Service (UDS) status byte to create any combination.



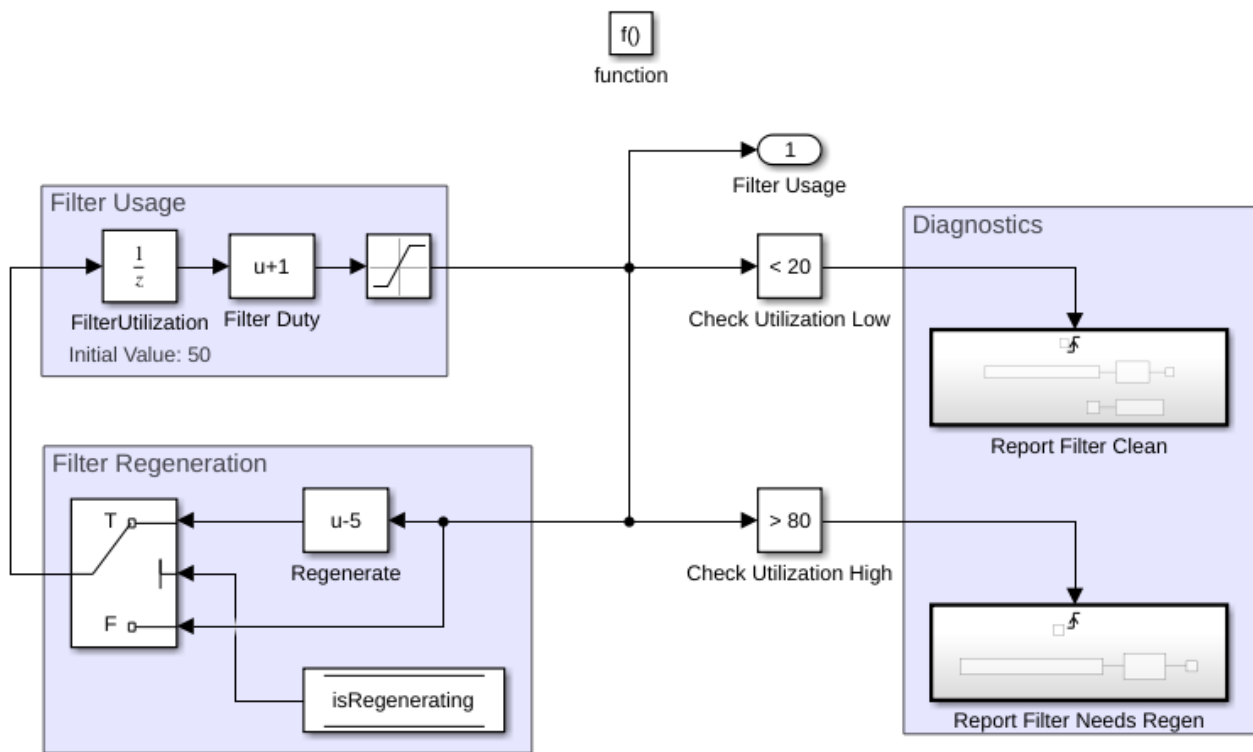
In general, you can use the Dem Status Override block to manually override the bits of the UDS status byte or have bits simulate independently. If you select **Dialog** and configure a bit, you override the status of that bit. Status bits are robust against changes with `setEventStatus` calls and the effects of event availability and operation cycle changes. Additionally, downstream services such as Function Inhibition (FiM) respond accordingly to the overridden status. If you select **Input port**,

you allow certain status bits to update based on a connected input signal, where values greater than zero turn the bit on. This functionality enables a component to override certain bits of a status while allowing others to continue to simulate independently.

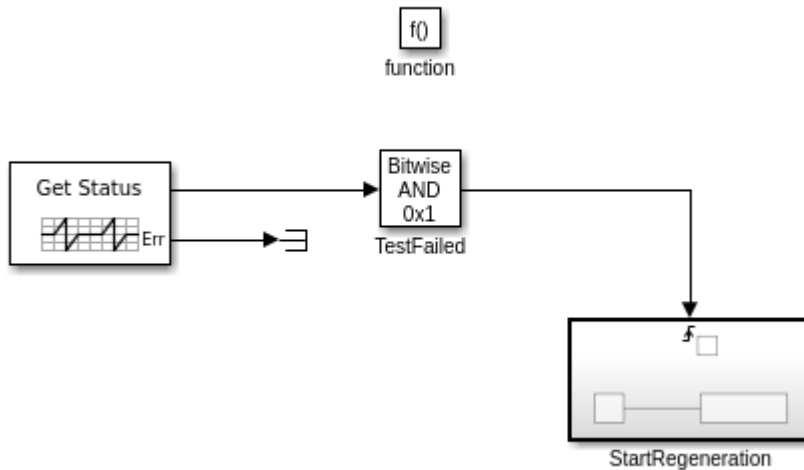
Inject Transient Events to Verify Component Recovery

In AUTOSAR there are often component managing systems that are also responsible for monitoring their own status. In such systems, it is useful to be able to test if such a component can recover from faults to the point that it can again report success. For this type of testing, the Dem Status Override block used in the preceding example is not suitable because it prevents the tested component from reporting a successful status again. Instead, a Dem Status Inject block can be used to create this type of test condition.

To show how to inject a transient event to verify component recovery, an example component for a regenerative particulate filter is shown. In this component, the state of the filter is stored in the state of a Delay block. During simulation, the filter incrementally increases within a nominal range of 20-80% utilization. This usage is monitored and reported to the diagnostic services. A representative regeneration mode is also shown. When regeneration is enabled the filter utilization quickly drops.



Within this component, the diagnostic services Test Failed bit is also queried. When a rising edge is detected, indicating that the filter has reached its threshold utilization, the component enters a regeneration cycle by setting the *isRegenerating* Data Store to true.

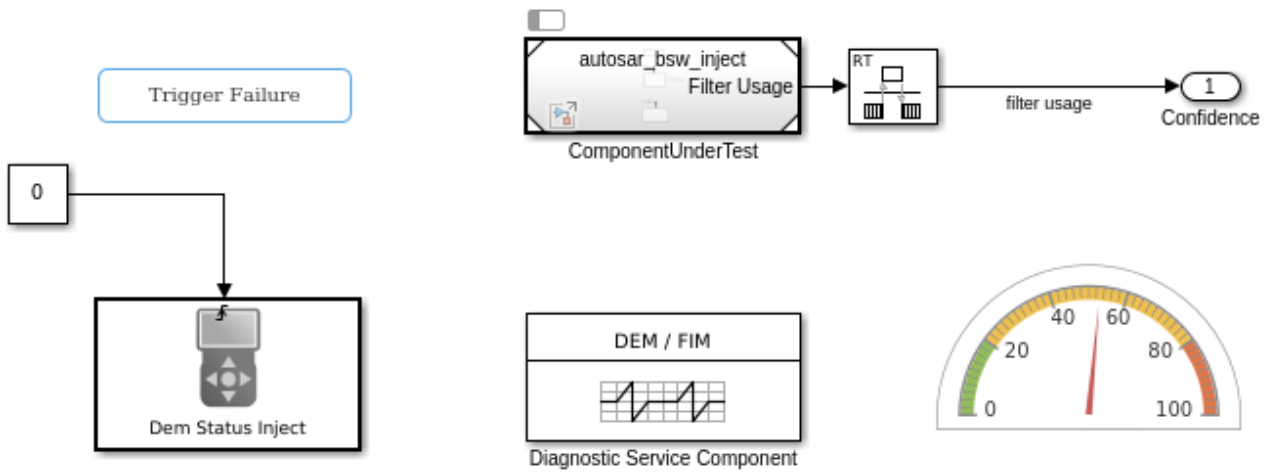


To verify the self recovery of this component, the test harness model shown was created. Specifically, you want to verify that the filter stays within the nominal range of utilization and that it enters and exits regeneration cycles correctly when diagnostic events occur. When the component simulates, it is expected to perpetually alternate between regeneration and nominal usage cycles by firing its own diagnostic events as required. You also want to test that the component can trigger a regeneration cycle if it receives a diagnostic event from the run-time environment (RTE).

The test harness shows the component filter utilization with the gauge shown in its dashboard. The Diagnostic Service Component block provides simulation for the calls to the Diagnostic Event Manager. When the harness simulates the component model, the gauge shows the filter gradually increase to 80% and then quickly lower to 20% as the filter is regenerated before repeating the cycle.

To verify that regeneration can be triggered from a diagnostic event provided by the RTE, you can use a Dem Status Inject block. The block has been specified to use `EventId 1`, which matches the client ports in the component defined in the Diagnostic Service Component. The fault type has been set to `Event Fail` to set the `Test Failed` bit to true when the event is injected. In the test harness model, the failure injection can be triggered by using the dashboard button. Alternatively, you could have provided this failure through an input signal source for an automated test configuration. The test harness can simulate the component to show its response to the fault injection. You can see when a fault is injected by using the dashboard button because you can see the gauge drop even if it has not reached the upper threshold to show that the component model has entered a regeneration cycle. This example model is configured to run indefinitely, so you must click **Stop** to end the simulation.

```
open_system('autosar_bsw_inject_test');
```



This example has shown how to test component recovery by using the Dem Status Inject block. While this specific example was narrow in scope, and it focused on the Test Failed bit, you can use the Dem Status Inject block to configure each bit of the Unified Diagnostic Service (UDS) status byte to create any combination.

Block Parameters: Dem Status Inject

Diagnostic Status Inject

Set the diagnostic status for an AUTOSAR event simulated in the Diagnostic Service Component block. This status value can be configured according to the Unified Diagnostic Services (UDS) standard.

Inhibition conditions defined in the FIM tab of the Diagnostic Service Component by using an overriding ID trigger appropriately as the overriding value changes.

This block simulates along with other blocks affecting this status.

General

EventId:

Fault type:

Trigger type:

Value Specification

Test Failed	Set
Test Failed This Operation Cycle	Set
Pending DTC	Set
Confirmed DTC	
Test Not Complete Since Last Clear	Clear
Test Failed Since Last Clear	Set
Test Not Complete This Operation Cycle	Clear
Warning Indicator Requested	

In general, you can use the Dem Status Inject block to manually override the bits of the UDS status byte. This block then simulates and updates its status with other blocks in the model to show

recovery. Downstream services such as Function Inhibition (FiM) respond accordingly to this status throughout simulation.

See Also

Dem Status Inject | Dem Status Override

Related Examples

- “Simulate AUTOSAR Basic Software Services and Run-Time Environment” on page 7-36
- “Configure Calls to AUTOSAR Diagnostic Event Manager Service” on page 7-14
- “Configure Calls to AUTOSAR Function Inhibition Manager Service” on page 7-18

More About

- “Model AUTOSAR Basic Software Service Calls” on page 7-12

AUTOSAR Software Architecture Modeling

- “Create AUTOSAR Architecture Models” on page 8-2
- “Add and Connect AUTOSAR Classic Components and Compositions” on page 8-4
- “Add and Connect AUTOSAR Adaptive Components and Compositions” on page 8-10
- “Import AUTOSAR Composition from ARXML” on page 8-16
- “Create Profiles Stereotypes and Views for AUTOSAR Architecture Analysis” on page 8-20
- “Link AUTOSAR Components to Requirements” on page 8-25
- “Define AUTOSAR Component Behavior by Creating or Linking Models” on page 8-27
- “Configure AUTOSAR Scheduling and Simulation” on page 8-38
- “Generate and Package AUTOSAR Composition XML Descriptions and Component Code” on page 8-43
- “Author AUTOSAR Classic Compositions and Components in Architecture Model” on page 8-50
- “Import AUTOSAR Composition into Architecture Model” on page 8-63
- “Configure AUTOSAR Architecture Model Programmatically” on page 8-67
- “Manage Shared Interfaces and Data Types for AUTOSAR Architecture Models” on page 8-71
- “Create AUTOSAR Architecture from a Component in System Composer Model” on page 8-81

Create AUTOSAR Architecture Models

An AUTOSAR architecture model provides resources and a canvas for developing AUTOSAR composition and component models (requires System Composer). From the architecture model, you can:

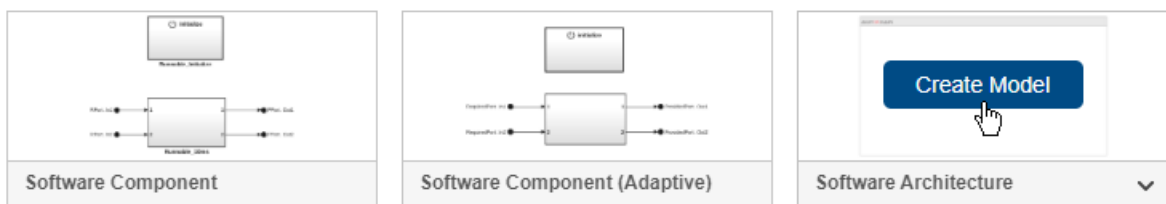
- Add and connect AUTOSAR compositions and components.
- View component or composition dependencies.
- Link components to requirements (requires Requirements Toolbox™).
- Add Simulink behavior to components by creating or linking models.
- Configure scheduling and simulation.
- Export composition and component ARXML descriptions and generate component code (requires Embedded Coder).

Architecture models provide an end-to-end AUTOSAR software design workflow. In Simulink, you can author a high-level application design for AUTOSAR Classic or Adaptive Platform architectures, implement behavior for application components, and simulate the application. For classic architecture compositions, you can add Basic Software (BSW) service calls and service implementations.

To create an architecture model, open an AUTOSAR Blockset model template from the Simulink Start Page. For example:

- 1 Open the Simulink Start Page. Enter the MATLAB `simulink` command or select Simulink menu sequences that create a new model.
- 2 On the **New** tab, scroll down to AUTOSAR Blockset and expand the list of model templates. Place your cursor over the **Software Architecture** template and click **Create Model**.

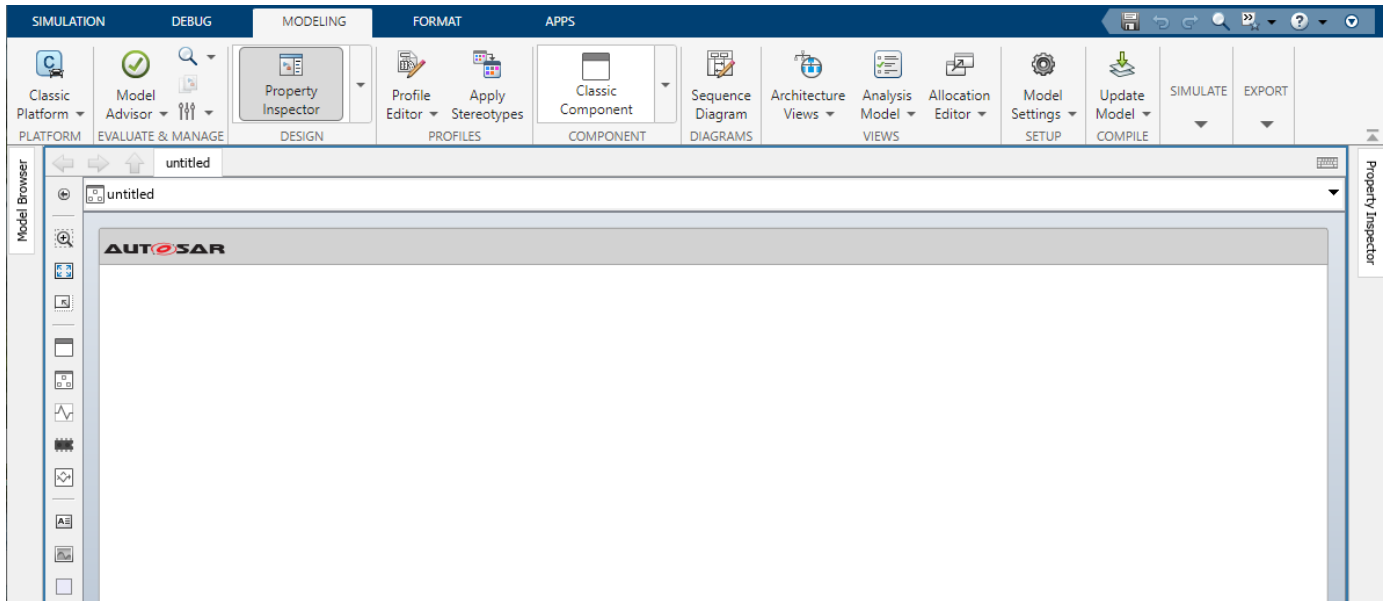
▼ AUTOSAR Blockset



A new AUTOSAR architecture model opens.

- 3 On the **Modeling** tab, click **Platform** and select the AUTOSAR platform kind, **Classic Platform** or **Adaptive Platform**. Mixing classic and adaptive components in the same architecture model is not supported.
- 4 Explore the controls and content in the software architecture canvas.
 - In the Simulink Toolstrip, the **Modeling** tab supports common tasks for architecture modeling.
 - To the left of the model window, the palette includes icons for adding different types of AUTOSAR components to the model:

- For classic architectures, supported blocks include the Classic Component, Software Composition, and for Basic Software (BSW) modeling, Diagnostic Service Component and NVRAM Service Component.
- For adaptive architectures, supported blocks include the Adaptive Component and Software Composition.
- The composition editor provides a view of AUTOSAR software architecture based on the AUTOSAR Virtual Function Bus (VFB). The model canvas initially is empty.



After you create an AUTOSAR architecture model, develop the top level of the design. See “Add and Connect AUTOSAR Classic Components and Compositions” on page 8-4 or “Add and Connect AUTOSAR Adaptive Components and Compositions” on page 8-10.

See Also

Software Component | Adaptive Component | Software Composition | Diagnostic Service Component | NVRAM Service Component

Related Examples

- “Add and Connect AUTOSAR Classic Components and Compositions” on page 8-4
- “Author AUTOSAR Classic Compositions and Components in Architecture Model” on page 8-50
- “Configure AUTOSAR Architecture Model Programmatically” on page 8-67

Add and Connect AUTOSAR Classic Components and Compositions

After you create an AUTOSAR architecture model, develop the top-level AUTOSAR classic or adaptive software design. The composition editor provides a view of AUTOSAR software architecture based on the AUTOSAR Virtual Function Bus (VFB).

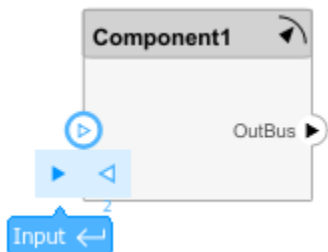
To begin, on the Simulink Toolstrip **Modeling** tab, in the **Platform** section, confirm the architecture model is configured for **Classic Platform**. After that, starting at the top level of the architecture model, use the composition editor and the **Modeling** tab to add and connect AUTOSAR software compositions and classic components.

Alternatively, you can import a software composition from ARXML files. See “Import AUTOSAR Composition from ARXML” on page 8-16.

Add and Connect Classic Component Blocks

To add and connect AUTOSAR classic software components in an architecture model:

- For each component required by the design, from the **Modeling** tab or the palette to the left of the canvas, add a Classic Component block. You can use the Property Inspector to set the component **Kind** — **Application**, **ComplexDeviceDriver**, **EcuAbstraction**, **SensorActuator**, or **ServiceProxy**.
- Add component require ports and provide ports. To add each component port, click an edge of a Classic Component block. When port controls appear, select **Input** for a require port or **Output** for a provide port.



- To connect the Classic Component blocks to other blocks, connect the block ports with signal lines.
- To connect the Classic Component blocks to architecture or composition model root ports, drag lines from the component ports to the containing model boundary.



Releasing the connection creates a root port at the boundary.



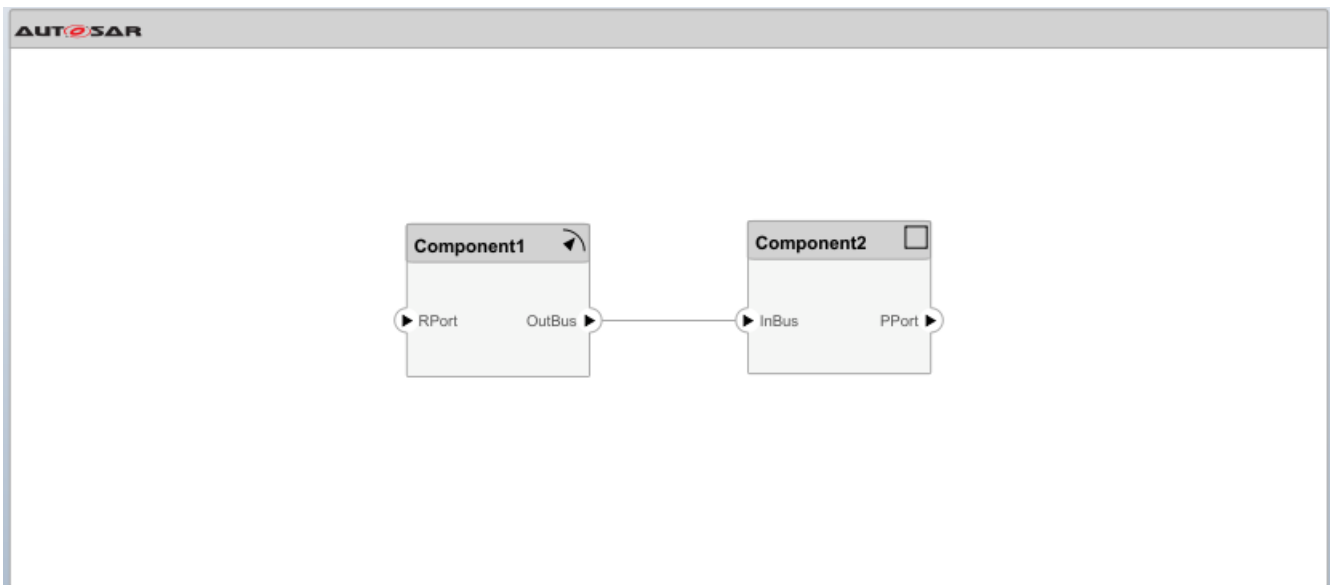
- Configure additional AUTOSAR properties by using the Property Inspector.

For example, to author a simple design:

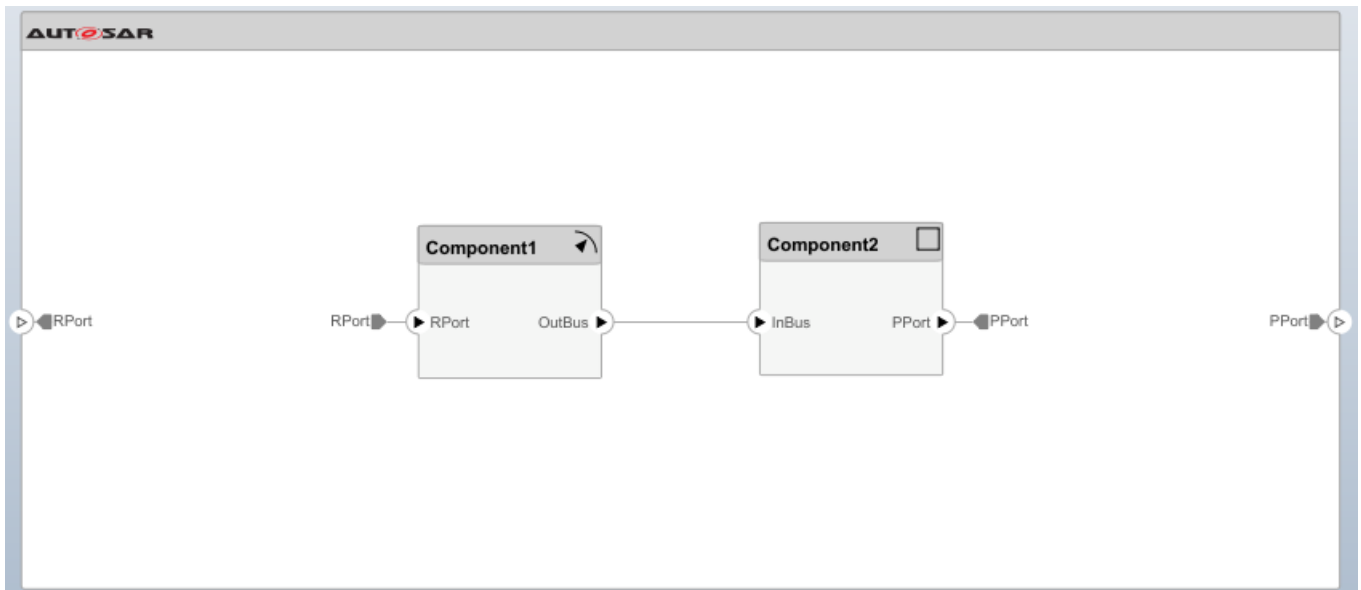
- 1 Using the Simulink Start Page, create an AUTOSAR architecture model. (For more information, see “Create AUTOSAR Architecture Models” on page 8-2.) The model canvas initially is empty.
- 2 Select the architecture platform, Classic or Adaptive.

From the **Modeling** tab, in the **Platform** section, select **Classic Platform**. Mixing classic and adaptive components in the same architecture model is not supported.

- 3 From the **Modeling** tab or the palette, add two Classic Component blocks. Place them next to each other, left and right.
 - a For each block, use the Property Inspector to set the component **Kind** — **SensorActuator** for the left block and **Application** for the right block.
 - b Add a provide (output) port to the left component block and a require (input) port to the right component block. Connect the two ports.
 - c Add a require (input) port to the left component block and a provide (output) port to the right component block.



- 4 Connect the new require and provide ports to architecture model root ports. Drag a line from each port to the model boundary.



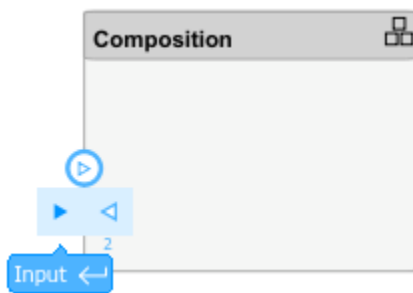
The simple design is complete, but behavior is not yet defined for the AUTOSAR components. The next step is to add Simulink behavior to the AUTOSAR components by creating, importing, or linking models. See “Define AUTOSAR Component Behavior by Creating or Linking Models” on page 8-27. For a more detailed design example, see “Author AUTOSAR Classic Compositions and Components in Architecture Model” on page 8-50.

If you have Requirements Toolbox software, you can link components in an AUTOSAR architecture model to requirements. See “Link AUTOSAR Components to Requirements” on page 8-25.

Add and Connect Composition Blocks to a Classic Model

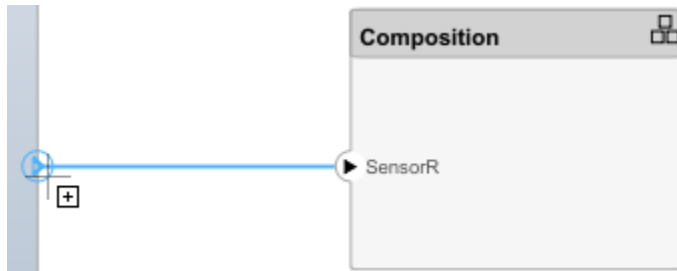
To add and connect an AUTOSAR software composition nested in an architecture model:

- From the **Modeling** tab or the palette to the left of the canvas, add a Software Composition block.
- Add composition require ports and provide ports. To add each composition port, click an edge of the Software Composition block. When port controls appear, select **Input** for a require port or **Output** for a provide port.

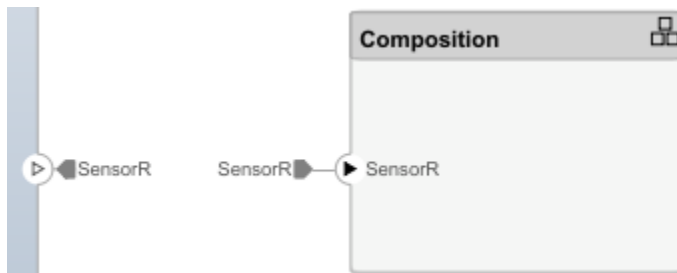


Alternatively, open the Software Composition block. To add each composition port, click the boundary of the composition diagram. When port controls appear, select **Input** for a require port or **Output** for a provide port.

- To connect the Software Composition block to other blocks, connect the block ports with signal lines.
- To connect the Software Composition block to architecture or composition model root ports, drag a line from the composition ports to the containing model boundary.



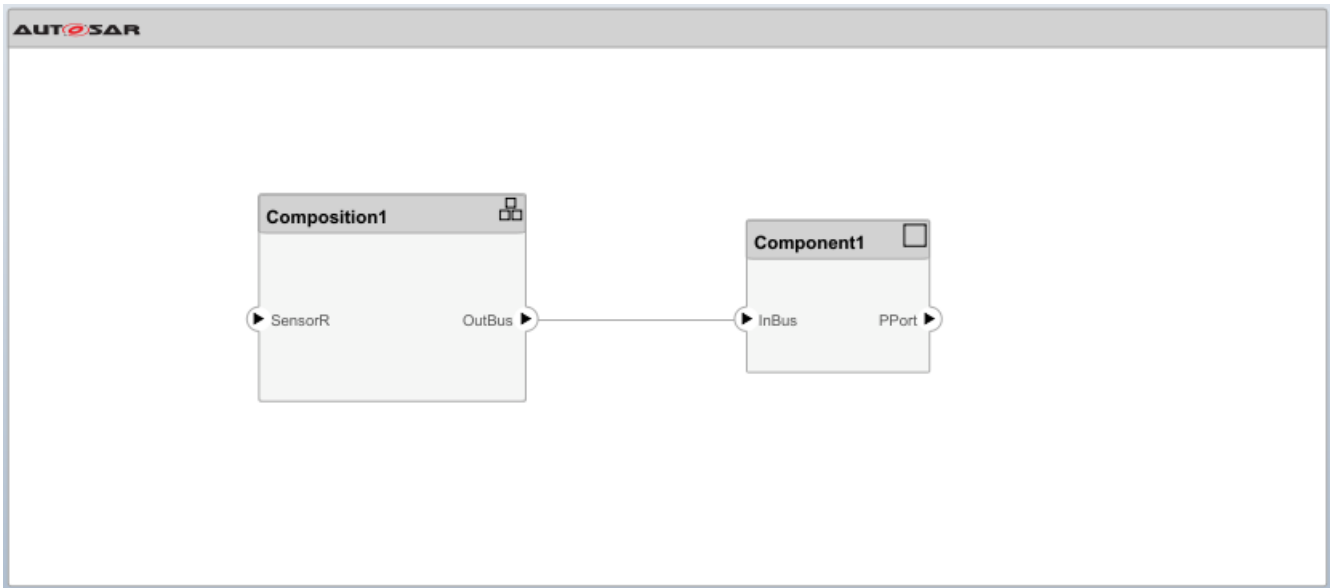
Releasing the connection creates a root port at the boundary.



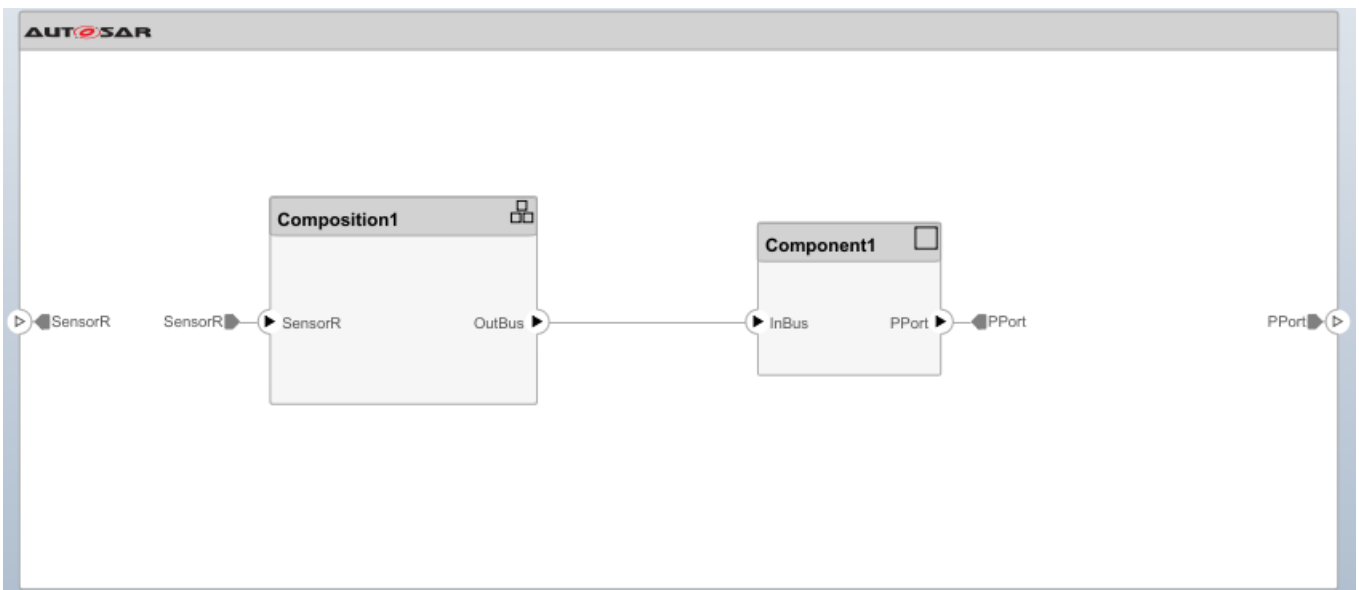
- Configure additional AUTOSAR properties by using the Property Inspector.

For example, to author a simple nested composition:

- 1 Using the Simulink Start Page, create an AUTOSAR architecture model. (For more information, see “Create AUTOSAR Architecture Models” on page 8-2.) The model canvas initially is empty.
- 2 From the **Modeling** tab or the palette, add a Software Composition block and a Classic Component block. Place them next to each other, left and right.
 - a Add a provide (output) port to the left composition block and a require (input) port to the right component block. Connect the two ports.
 - b Add a require (input) port to the left composition block and a provide (output) port to the right component block.



- 3 Connect the unconnected require and provide ports to architecture model root ports. Drag from each port to the model boundary.



Typically, an AUTOSAR composition contains a set of AUTOSAR components and compositions with a shared purpose. To populate a composition, open the Software Composition block and begin adding more Classic Component and Software Composition blocks. For a more detailed design example, see “Author AUTOSAR Classic Compositions and Components in Architecture Model” on page 8-50.

See Also

Software Component | Software Composition

Related Examples

- “Import AUTOSAR Composition from ARXML” on page 8-16
- “Add and Connect AUTOSAR Adaptive Components and Compositions” on page 8-10
- “Define AUTOSAR Component Behavior by Creating or Linking Models” on page 8-27
- “Create Profiles Stereotypes and Views for AUTOSAR Architecture Analysis” on page 8-20
- “Link AUTOSAR Components to Requirements” on page 8-25
- “Author AUTOSAR Classic Compositions and Components in Architecture Model” on page 8-50
- “Configure AUTOSAR Architecture Model Programmatically” on page 8-67

Add and Connect AUTOSAR Adaptive Components and Compositions

After you create an AUTOSAR architecture model, develop the top-level AUTOSAR classic or adaptive software design. The composition editor provides a view of AUTOSAR software architecture based on the AUTOSAR Virtual Function Bus (VFB).

To begin, on the Simulink Toolstrip **Modeling** tab, click **Platform** and select **Adaptive Platform** to configure adaptive architecture modeling. After that, you can use the composition editor and the **Modeling** tab to add and connect AUTOSAR software compositions and adaptive components.

In this section...

“Add and Connect Adaptive Component Blocks” on page 8-10

“Add and Connect Composition Blocks to an Adaptive Model” on page 8-13

Alternatively, you can import a software composition from ARXML files. See “Import AUTOSAR Composition from ARXML” on page 8-16.

Add and Connect Adaptive Component Blocks

To add and connect AUTOSAR adaptive software components in an architecture model:

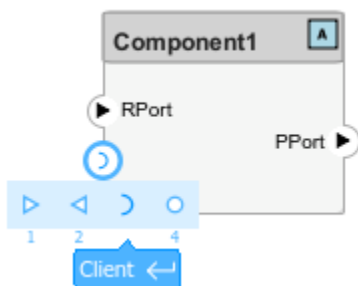
- Add components.

From the **Modeling** tab or the palette to the left of the canvas, add an Adaptive Component block. By using the Property Inspector, you can inspect the block. The component **Kind** for adaptive architecture modeling is **AdaptiveApplication**.

- Add component require ports and provide ports.

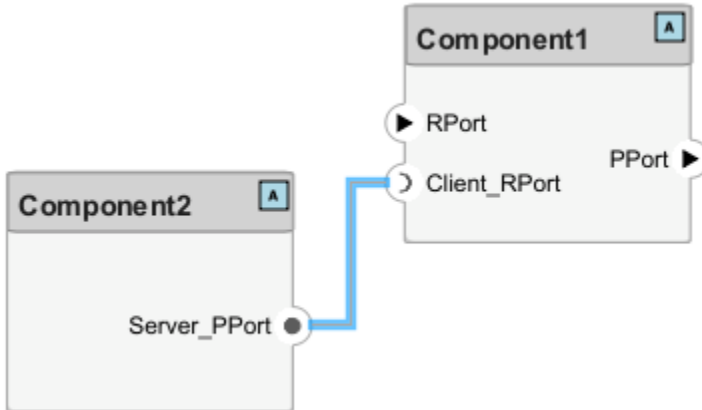
To add each component port, click an edge of an Adaptive Component block. Port controls appear for modeling communication over the service interfaces.

- For event communication, select **Input** for a require port or **Output** for a provide port.
- For method communication, select **Client** for a require port or **Server** for a provide port.



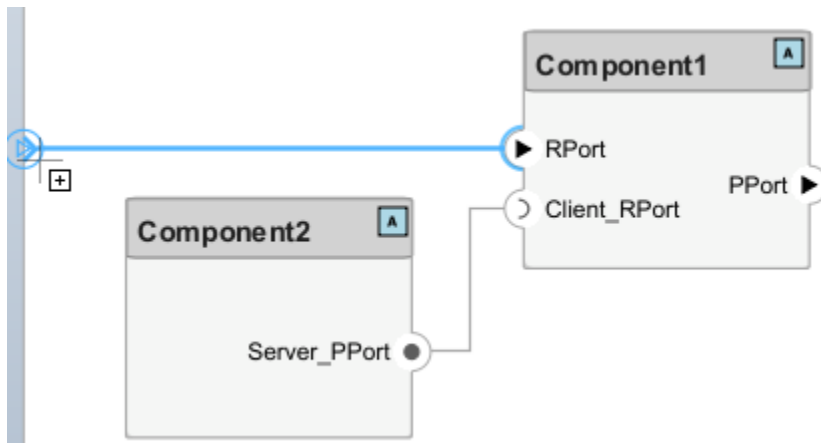
- Connect block ports.

To connect the Adaptive Component blocks to other blocks, connect the block ports with signal lines. Connecting adaptive component blocks together represents the service oriented communication between the two communication endpoints.

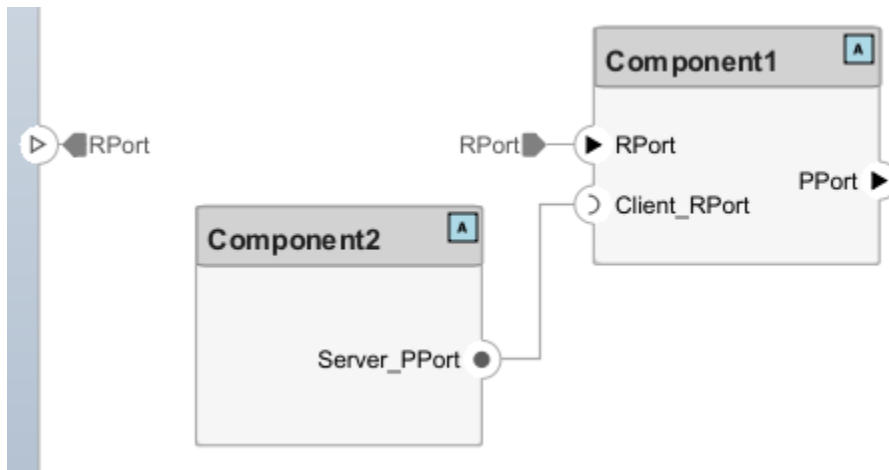


- Connect root ports.

To connect the Adaptive Component blocks to architecture or composition model root ports, drag lines from the component ports to the containing model boundary.



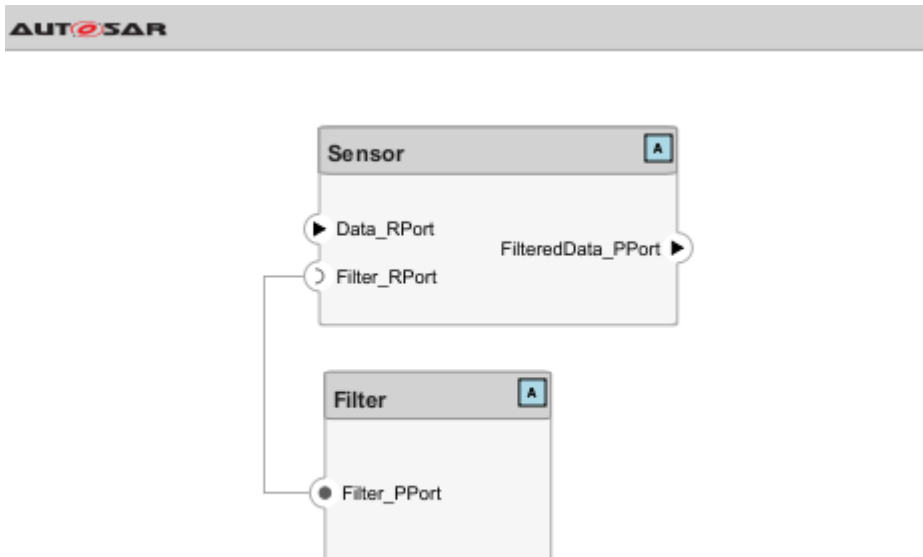
Releasing the connection creates a root port at the boundary.



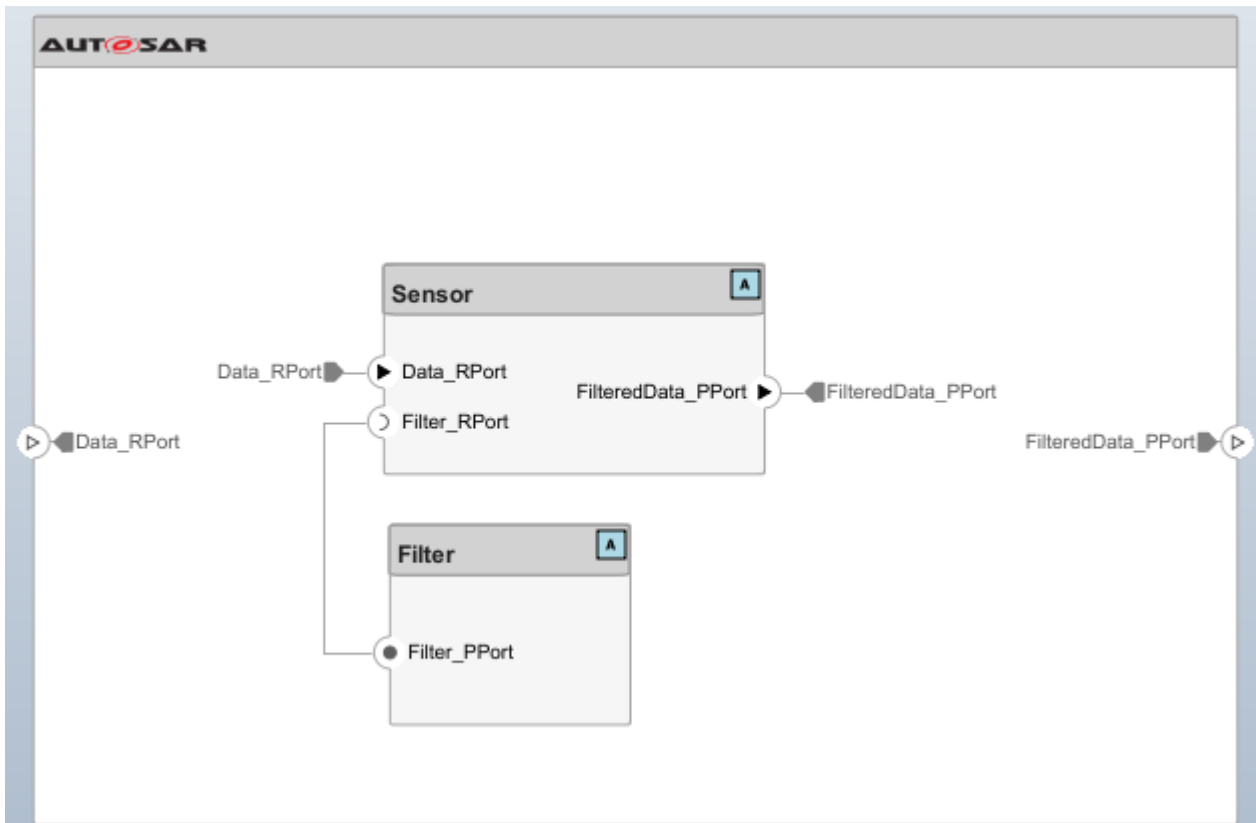
- Configure additional AUTOSAR properties by using the Property Inspector.

As an example, add and connect components for a client-server architecture model. The model consists of a sensor client component receiving raw data over a require port, filtering that data by using a method call to a filter service component, and sending the filtered data over a provider port.

- 1 Using the Simulink Start Page, create an AUTOSAR architecture model. (For more information, see “Create AUTOSAR Architecture Models” on page 8-2.) The model canvas initially is empty.
- 2 From the **Modeling** tab, click **Platform** and select Adaptive Platform.
- 3 From the **Modeling** tab or the palette, add two Adaptive Component blocks. Label the first block **Sensor** and the second block **Filter**. Place the **Filter** component below the **Sensor** component.
- 4 Add ports to the **Sensor** and **Filter** components.
 - a Add a Client port to the **Sensor** component and label it **Filter_RPort**.
 - b Add a Server port to the **Filter** component and label it **Filter_PPort**.
 - c Connect the two ports.
 - d Add an Input port to the **Sensor** component and label it **Data_RPort**.
 - e Add an Output port to the **Sensor** component and label it **FilteredData_PPort**. Resize the **Sensor** component as needed to fit the labels.



- 5 Connect the **Data_RPort** and **FilteredData_PPort** ports to architecture model root ports. Drag a line from each port to the model boundary.



The design is complete, but behavior is not yet defined for the AUTOSAR components.

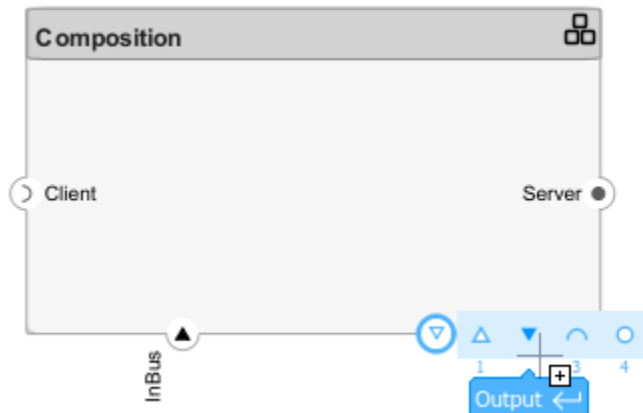
The next step is to add Simulink behavior to the AUTOSAR components by creating, importing, or linking models. See “Define AUTOSAR Component Behavior by Creating or Linking Models” on page 8-27.

If you have Requirements Toolbox software, you can link components in an AUTOSAR architecture model to requirements. See “Link AUTOSAR Components to Requirements” on page 8-25.

Add and Connect Composition Blocks to an Adaptive Model

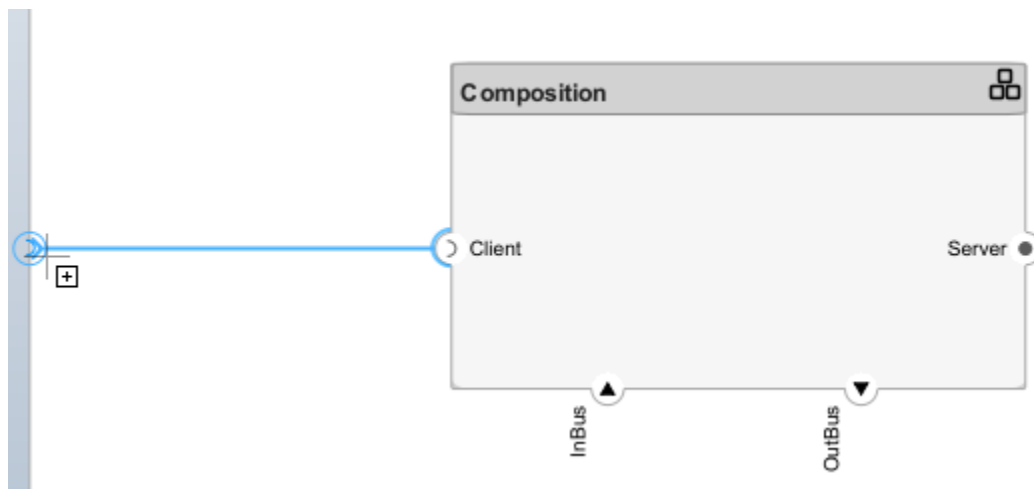
To add and connect an AUTOSAR software composition nested in an architecture model:

- From the **Modeling** tab or the palette to the left of the canvas, add a Software Composition block.
- Add composition require ports and provide ports. To add each composition port, click an edge of the Software Composition block. Port controls appear for modeling communication over the service interfaces.
 - For event communication, select **Input** for a require port or **Output** for a provide port.
 - For method communication, select **Client** for a require port or **Server** for a provide port.



Alternatively, open the Software Composition block. To add each composition port, click the boundary of the composition diagram. When port controls appear, select **Input** or **Client** for require ports, or **Output** or **Server** for provide ports as required for your interface.

- To connect the Software Composition block to other blocks, connect the block ports with signal lines.
- To connect the Software Composition block to architecture or composition model root ports, drag lines from the composition ports to the containing model boundary.



Releasing the connection creates a root port at the boundary.

- Configure additional AUTOSAR properties by using the Property Inspector.

An AUTOSAR composition contains a set of AUTOSAR components and compositions with a shared purpose. To populate a composition, open the Software Composition block and begin adding more Adaptive Component and Software Composition blocks.

See Also

Adaptive Component | Software Composition

Related Examples

- “Import AUTOSAR Composition from ARXML” on page 8-16
- “Add and Connect AUTOSAR Classic Components and Compositions” on page 8-4
- “Define AUTOSAR Component Behavior by Creating or Linking Models” on page 8-27
- “Create Profiles Stereotypes and Views for AUTOSAR Architecture Analysis” on page 8-20
- “Link AUTOSAR Components to Requirements” on page 8-25
- “Author AUTOSAR Classic Compositions and Components in Architecture Model” on page 8-50
- “Configure AUTOSAR Architecture Model Programmatically” on page 8-67

Import AUTOSAR Composition from ARXML

After you create an AUTOSAR architecture model, develop the top-level AUTOSAR software design. The composition editor provides a view of AUTOSAR software architecture based on the AUTOSAR Virtual Function Bus (VFB).

If you have an ARXML description of an AUTOSAR software composition, you can import the composition into an AUTOSAR architecture model. The import creates a Simulink representation of the composition at the top level of the architecture model.

Importing a composition requires an open AUTOSAR architecture model with no functional content. To import a composition, open the AUTOSAR Importer app or call the architecture function `importFromARXML`.

If you do not have an ARXML description to import, you can use the composition editor to add and connect AUTOSAR software compositions and components. See “Add and Connect AUTOSAR Classic Components and Compositions” on page 8-4 or “Add and Connect AUTOSAR Adaptive Components and Compositions” on page 8-10.

Import AUTOSAR Composition By Using AUTOSAR Importer App

The example in this section describes importing a classic architecture composition. The workflow for importing an adaptive composition is the same.

To import an AUTOSAR software composition from ARXML files into an architecture model:

- 1 Create or open an AUTOSAR architecture model that has no functional content.

```
archModel = autosar.arch.createModel("myArchModel");
```

By default, `autosar.arch.createModel` creates an AUTOSAR architecture model for the Classic Platform. To specify the Adaptive Platform or explicitly specify the Classic Platform, use the `platform` name-value argument. Mixing classic and adaptive components in the same architecture model is not supported.

```
archModel = autosar.arch.createModel("myArchModel", "platform", "Classic");
```

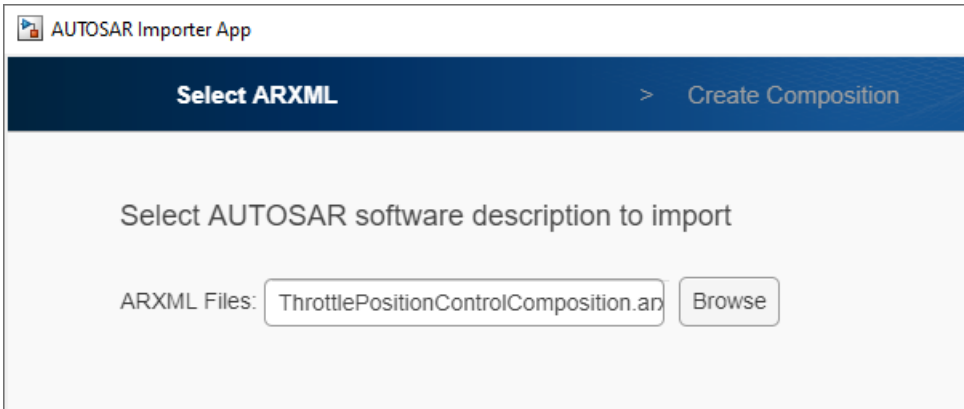
- 2 On the **Modeling** tab, in the **Platform** section, note the platform kind for the architecture model.

If needed, change the platform kind before importing ARXML files.

- 3 On the **Modeling** tab, select **Import from ARXML**.
- 4 In the AUTOSAR Importer app, in the **Select ARXML** pane, in the **ARXML Files** field, enter the names of one or more ARXML files (comma separated) that describe an AUTOSAR software composition.

For this example, use `ThrottlePositionControlComposition.arxml` to import a classic architecture composition.

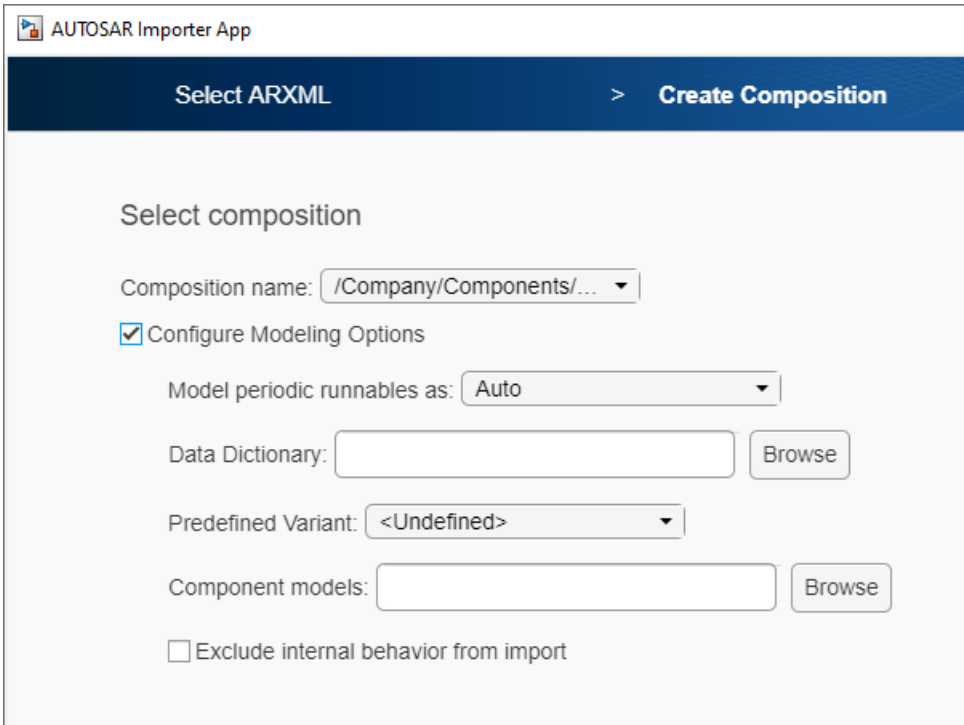
```
openExample('autosarblockset/ImportAUTOSARComponentToSimulinkExample', ...
'supportingfile', 'ThrottlePositionControlComposition.arxml');
```



Click **Next**. The app parses the specified ARXML file.

- 5 In the **Create Composition** pane, the **Composition name** menu lists the compositions found in the parsed ARXML file. Select the composition /Company/Components/ThrottlePositionControlComposition.

Optionally, to view additional modeling options for composition creation, select **Configure Modeling Options**.



You can specify:

- Simulink data dictionary in which to place data objects for imported AUTOSAR data types.
- Names of existing Simulink behavior models to link to imported AUTOSAR components.

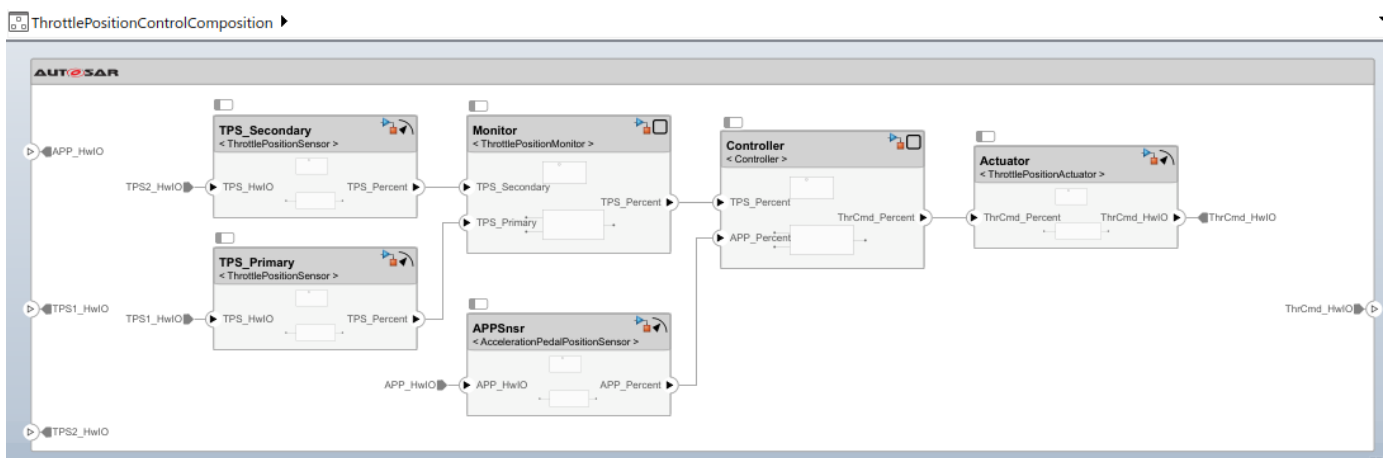
When importing classic compositions, you can additionally specify:

- Whether to include or exclude AUTOSAR software components, which define composition behavior. By default, the import includes components within the composition.
- Component options to apply when creating Simulink behavior models for imported AUTOSAR software components. For example, how to model periodic runnables, or a `PredefinedVariant` or `SwSystemconstantValueSets` with which to resolve component variation points.

For more information about modeling options and behavior, see `importFromARXML`.

- 6 To finish importing the composition into the architecture model, click **Finish**. The Diagnostic Viewer displays the progress of the composition creation.

On completion, the imported composition appears in the software architecture canvas.



Because this composition import was configured to include AUTOSAR classic software components (modeling option **Exclude internal behavior from import** was cleared), the import created Simulink models for each component in the composition.

Next you develop each component in the composition. For each component model, you refine the AUTOSAR configuration and create algorithmic model content. For an example of developing component algorithms, see “Design and Simulate AUTOSAR Components and Generate Code” on page 4-77, section “Develop AUTOSAR Component Algorithms”.

Import AUTOSAR Composition By Calling `importFromARXML`

The example in this section describes importing a classic architecture composition. The workflow for importing an adaptive composition is the same.

You can access the AUTOSAR classic ARXML file `ThrottlePositionControlComposition.arxml` by using the `openExample` command.

```
openExample('autosarblockset/ImportAUTOSARComponentToSimulinkExample', ...
'supportingfile', 'ThrottlePositionControlComposition.arxml');
```

To programmatically import an AUTOSAR software composition from ARXML files into an architecture model, call the `importFromARXML` function. This example:

- 1 Creates AUTOSAR architecture model `myArchModel`.

- 2 Imports software composition `/Company/Components/ThrottlePositionControlComposition` from AUTOSAR example file `ThrottlePositionControlComposition.arxml` into the architecture model.

```
% Create AUTOSAR architecture model
modelName = "myArchModel";
archModel = autosar.arch.createModel(modelName); % Defaults to the Classic Platform

% Import composition from file ThrottlePositionControlComposition.arxml
importerObj = arxml.importer("ThrottlePositionControlComposition.arxml"); % Parse ARXML
importFromARXML(archModel, importerObj, ...
    "/Company/Components/ThrottlePositionControlComposition");

Creating model 'ThrottlePositionSensor' for component 1 of 5:
/Company/Components/ThrottlePositionSensor
Creating model 'ThrottlePositionMonitor' for component 2 of 5:
/Company/Components/ThrottlePositionMonitor
Creating model 'Controller' for component 3 of 5:
/Company/Components/Controller
Creating model 'AccelerationPedalPositionSensor' for component 4 of 5:
/Company/Components/AccelerationPedalPositionSensor
Creating model 'ThrottlePositionActuator' for component 5 of 5:
/Company/Components/ThrottlePositionActuator
Importing composition 1 of 1:
/Company/Components/ThrottlePositionControlComposition
```

For more information about import options and behavior, see the `importFromARXML` reference page.

See Also

`importFromARXML`

Related Examples

- “Import AUTOSAR Composition into Architecture Model” on page 8-63
- “Add and Connect AUTOSAR Classic Components and Compositions” on page 8-4
- “Add and Connect AUTOSAR Adaptive Components and Compositions” on page 8-10
- “Define AUTOSAR Component Behavior by Creating or Linking Models” on page 8-27
- “Create Profiles Stereotypes and Views for AUTOSAR Architecture Analysis” on page 8-20
- “Link AUTOSAR Components to Requirements” on page 8-25
- “Author AUTOSAR Classic Compositions and Components in Architecture Model” on page 8-50
- “Configure AUTOSAR Architecture Model Programmatically” on page 8-67

Create Profiles Stereotypes and Views for AUTOSAR Architecture Analysis

AUTOSAR architectures can be large and complex. Often development work is distributed, with different engineers working on different structural or functional parts of an architecture model. To help analyze structural or functional aspects of an architecture model, you can create filtered views of the model hierarchy.

- A spotlight view displays the upstream and downstream dependencies of a selected architecture component or composition.
- A custom view displays a subset of components from the architecture model, based on filtering conditions that you specify. You can filter model elements for operational, functional, or physical analysis.

Filtering an AUTOSAR architecture model for specific attributes and saving the filtered view with the model can help engineers focus and collaborate on their parts of the architecture.

Create Profiles and Stereotypes

You can use stereotypes to capture nonfunctional properties of elements in an AUTOSAR architecture model. To capture these properties, create a profile containing stereotype definitions and apply these stereotypes on the modeling elements. Define new profiles and stereotypes using the **Profile Editor**.

For example, in an AUTOSAR architecture model, you may want to define a custom stereotype for all sensor components in your model.

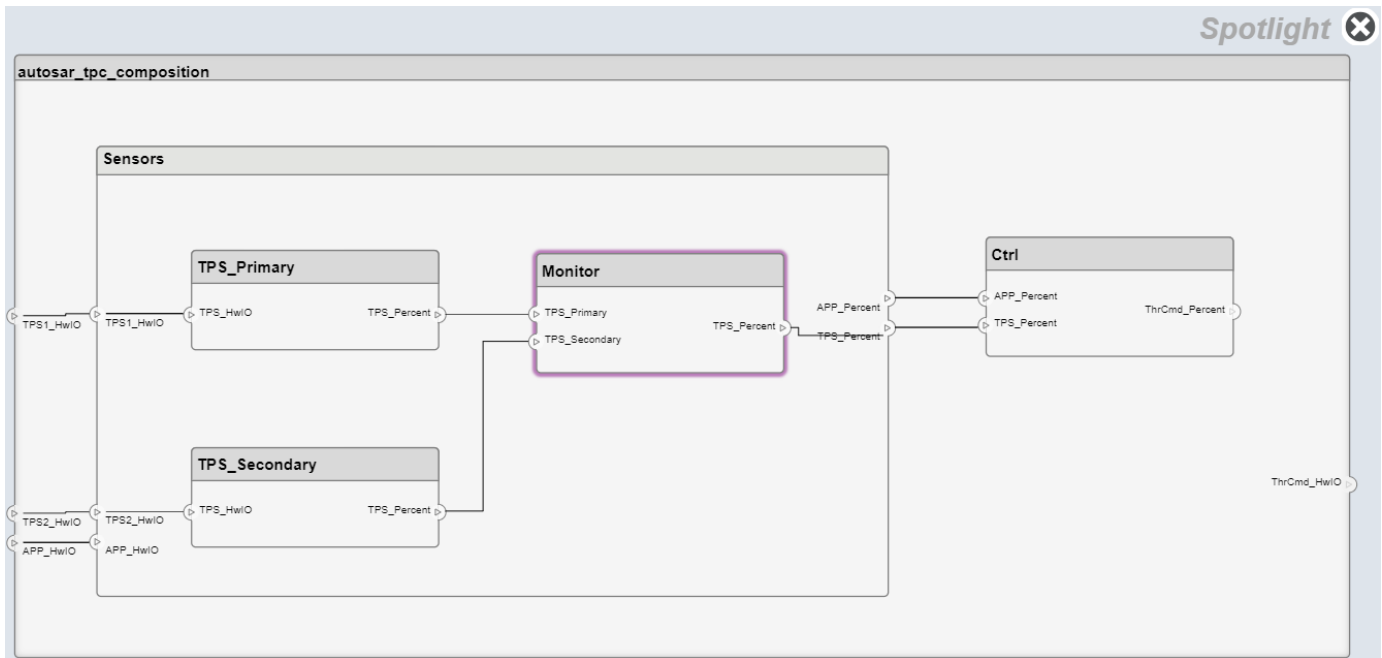
Once you have defined profiles and stereotypes for your architecture model, you can create custom views that display a subset of the stereotypes in the model and perform additional analysis by quantitatively evaluating the architecture for certain characteristics. For more information, see “Define Stereotypes and Perform Analysis” (System Composer) and “Use Stereotypes and Profiles” (System Composer).

View Component or Composition Dependencies

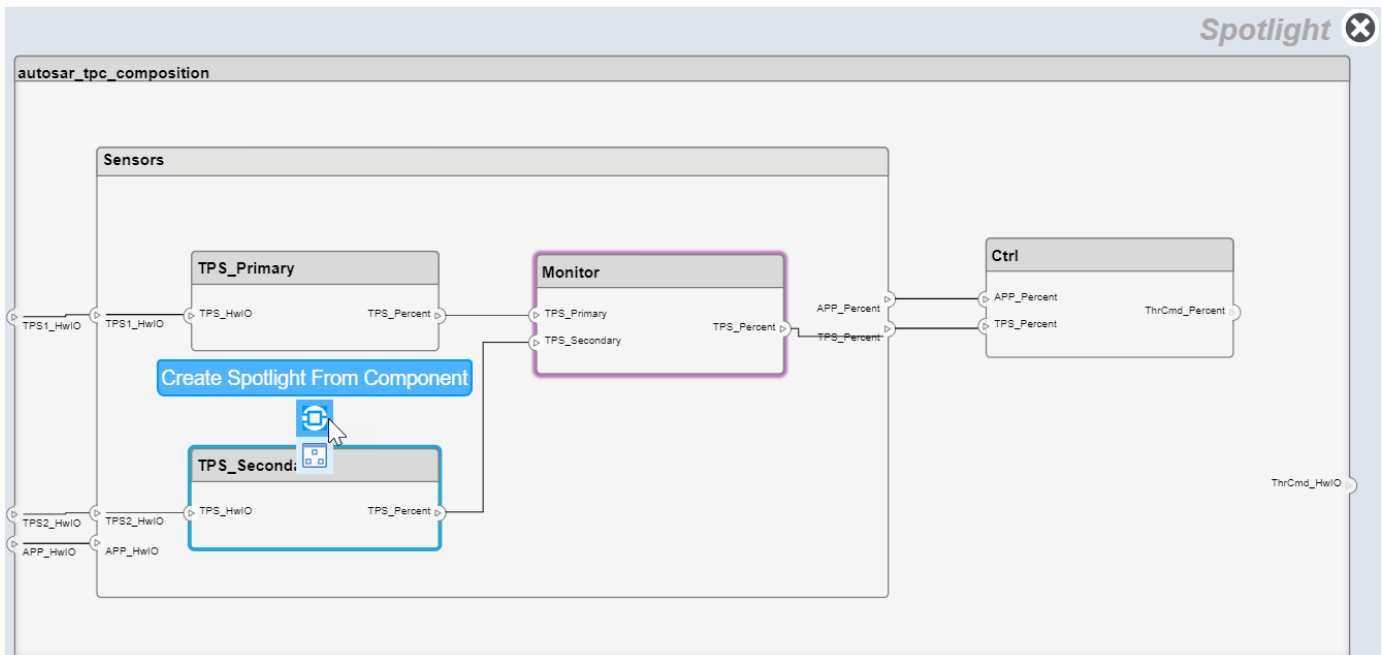
In an AUTOSAR architecture model, to help analyze component or composition dependencies, you can create a spotlight view. A spotlight view is a simplified view of an architecture component or composition that captures its upstream and downstream dependencies.

To create a spotlight view, open an architecture model and select a component or composition. On the **Modeling** tab, select **Architecture Views > Spotlight**.

The spotlight view displays the model elements to which the component or composition connects in a hierarchy. You cannot edit the spotlight diagram layout. This figure shows a spotlight view of component **Monitor** in AUTOSAR example model `autosar_tpc_composition`. (To open the example model in a local working folder, use the command `openExample('autosar_tpc_composition')`.)



While in the spotlight view, you can move the spotlight focus to another component or composition. Select another component or composition, place your cursor over the displayed ellipsis, and select model cue **Create Spotlight from Component**.



To keep a spotlight view visible during model development, you can create the view in a separate model window. To create a separate model window, select a component or composition, right-click the selected block to open its context menu, and select **Open in New Window**. In the new window, create a spotlight view.

Updating the architecture model diagram with changes refreshes open spotlight views.

To return from a spotlight view to the architecture model view, click the **Spotlight** close icon or select a component or composition and select model cue **Show in Composition**.

Simulink does not save spotlight views with the architecture model.

Create Custom Views for Analysis

To help analyze structural and functional aspects of an AUTOSAR architecture model, you can create a custom view. Based on filtering conditions that you specify, a custom view shows a subset of components from the architecture model. You can filter model elements for operational, functional, or physical analysis. To create a custom view, open the Architecture Views Gallery. In an open architecture model, on the **Modeling** tab, select **Architecture Views**.

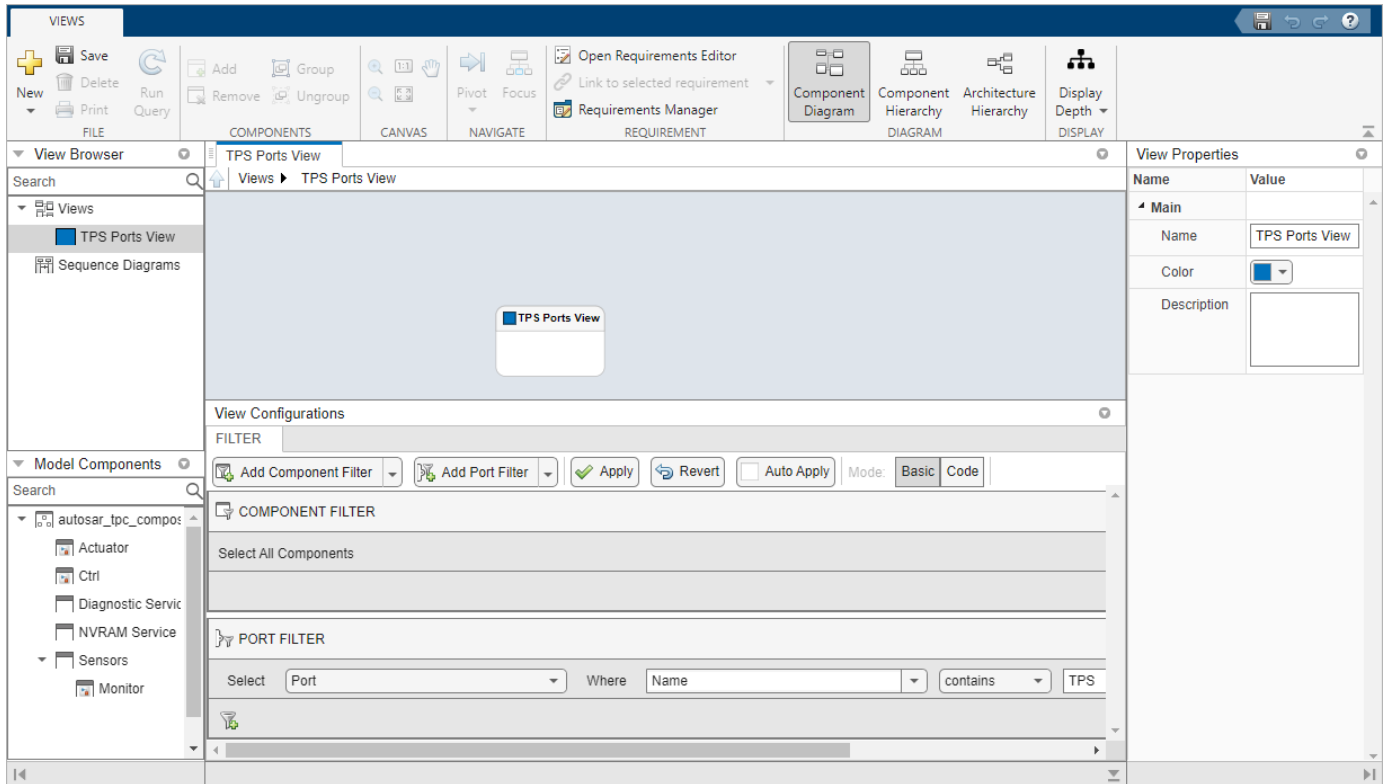
As an example, suppose you want to create a view for the architecture model `autosar_tpc_composition` to show components that handle throttle position sensor (TPS) signals. The following workflow shows how to create this custom view:

- 1 Open the example model in a local working folder by using the command `openExample('autosar_tpc_composition')`.
- 2 Open the Architecture Views gallery. On the **Modeling** tab, select **Architecture Views**.
- 3 Create a view. In the gallery view, click **New > View**. To name the view, in the **View Properties** pane, enter the name `TPS Ports View`.
- 4 Configure the view. Use the **Filter** tab to specify the constraints for the view.
 - a Specify the components. For the custom view, specify the components in the architecture model you would like to include in the custom view. There are several ways to specify the components, the options include `Add Component Filter`, `Select All Components`, `Add Custom Component Filter`, and `Clear All Component Filters`.

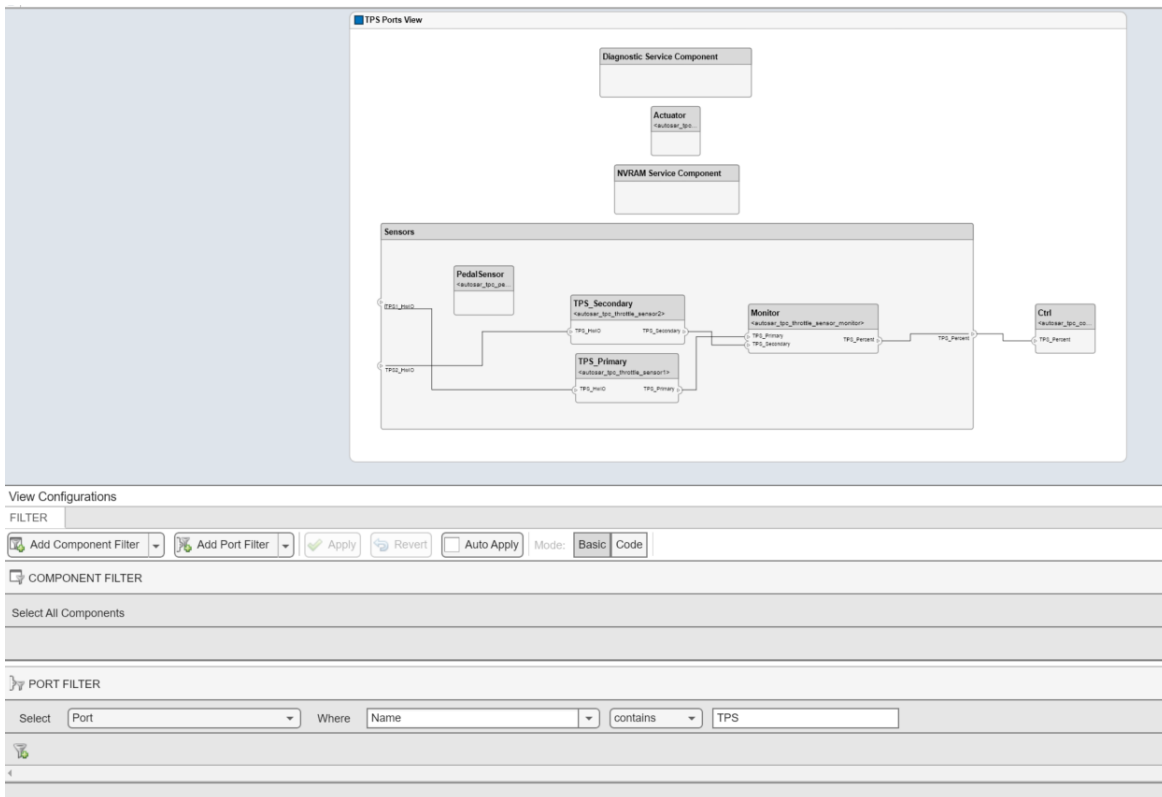
For this example, select `Select All Components` to bring all of the components from the example model into the view.

- b Specify the ports. For your custom view, specify the ports in the architecture model that you would like to include in the custom view. There are several ways to specify the ports, the options include `Add Port Filter`, `Exclude All Ports`, `Hide unconnected ports`, `Hide connectors`, `Add Custom Port Filter`, and `Clear All Port Filters`.

For this example, select `Add Port Filter` to filter the ports in the architecture model. Configure the filter so that it selects ports where the name contains TPS. When the filter is applied the custom view will show the components that contain the throttle position sensor (TPS).



5 Apply the view. To display the updated TPS Ports View, click **Apply**.



When you save the architecture model, the view is saved in the Architecture Views Gallery. Other users can then access and share the view. For more information, see “Create Architecture Views Interactively” (System Composer).

See Also

Related Examples

- “Create Spotlight Views” (System Composer)
- “Create Architecture Views Interactively” (System Composer)
- “Define AUTOSAR Component Behavior by Creating or Linking Models” on page 8-27
- “Link AUTOSAR Components to Requirements” on page 8-25
- “Author AUTOSAR Classic Compositions and Components in Architecture Model” on page 8-50

Link AUTOSAR Components to Requirements

If you have Requirements Toolbox™ software, you can link components in an AUTOSAR architecture model to requirements. By linking model elements that implement requirements to the associated requirements, you can track implementation of the requirements. If a requirement or an implementation changes, you can make adjustments to keep them in sync.

To link a component to a requirement:

Open an architecture model. For example, `autosar_tpc_composition`.

```
open_system('autosar_tpc_composition')
```

In the **Apps** tab, click **Requirements Manager**. In the model window, the **Requirements** tab opens, with the Requirements Browser docked at the bottom.

Create or open a requirements set. If you opened example model `autosar_tpc_composition`, you can use the example requirements file `TPC_Requirements.slsreqx`.

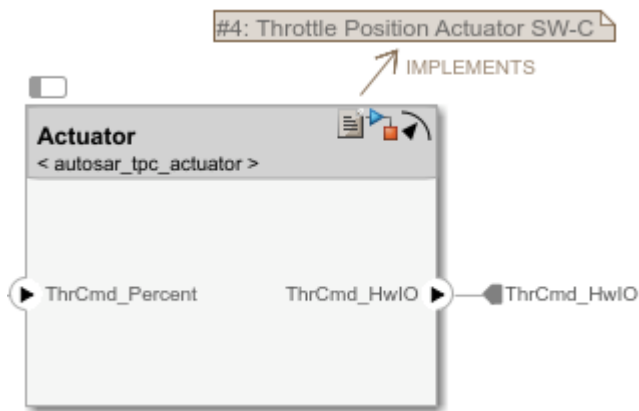
In the Requirements Browser, open the requirements file. The requirements set contains requirements for four components in the model.

The screenshot displays the AUTOSAR Requirements Manager interface. The main window shows a component diagram with three components: **Sensors**, **Ctrl** (autosar_tpc_controller), and **Actuator** (autosar_tpc_actuator). The **Actuator** component is highlighted with a blue border. The **Requirements** browser at the bottom shows a table of requirements for the `TPC_Requirements` set.

Index	ID	Summary
1	#1	Primary Throttle Position Sensor SW-C
2	#2	Secondary Throttle Position Sensor SW-C
3	#3	Accelerator Pedal Position Sensor SW-C
4	#4	Throttle Position Actuator SW-C

The **Property Inspector** on the right shows the properties for the selected **Actuator** component. The **Main** section is expanded, showing the **Name** as `Actuator` and the **Kind** as `SensorActuator`.

To link a requirement to an AUTOSAR component, drag the requirement from the Requirements Browser to the component block. For example, drag requirement 4 to the **Actuator** component block.



For more information, see “Link Blocks and Requirements” (Requirements Toolbox) and “Author Requirements in MATLAB or Simulink” (Requirements Toolbox).

See Also

Related Examples

- “Define AUTOSAR Component Behavior by Creating or Linking Models” on page 8-27
- “Create Profiles Stereotypes and Views for AUTOSAR Architecture Analysis” on page 8-20
- “Author AUTOSAR Classic Compositions and Components in Architecture Model” on page 8-50

More About

- “Author Requirements in MATLAB or Simulink” (Requirements Toolbox)
- “Link Blocks and Requirements” (Requirements Toolbox)

Define AUTOSAR Component Behavior by Creating or Linking Models

After you add and connect software composition and component blocks in your AUTOSAR architecture model, add Simulink behavior to the components. For each AUTOSAR software component block, you can:

- Create a model based on the block interface.
- Link to an implementation model.
- Create a model from an AUTOSAR XML (ARXML) component description.

To initiate these actions, select a Classic Component or Adaptive Component block, place your cursor over the displayed ellipsis, and select a component model cue — **Create Simulink Behavior**, **Link to Model**, or **Create Component Model from ARXML**.



The selections open dialog boxes that help you create or link a model that defines the Simulink behavior of the component.

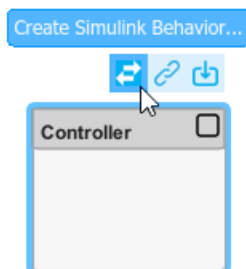
The creation and linking actions can be initiated in other ways, for example, from an architecture block context menu or from the toolstrip **Modeling** tab.

After you associate an implementation model with an AUTOSAR component, if you have Embedded Coder software, you can use component block cues or right-click options to generate code and export ARXML files. The ARXML export uses the XML options of the parent architecture model.

When the components in an architecture model have defined behavior, you can simulate the behavior of the aggregated components. See “Configure AUTOSAR Scheduling and Simulation” on page 8-38.

Create Simulink Behavior Based on Block Interface

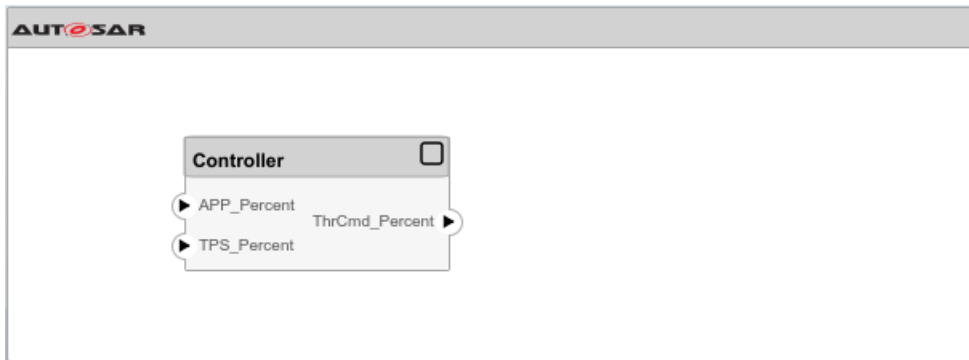
To create a stub implementation model and map it to an AUTOSAR software component, use the cue **Create Simulink Behavior** on the Classic Component or Adaptive Component.



Clicking the cue creates a model based on the interface of the authored component. Ports that you created on the software component block are present in the implementation model.

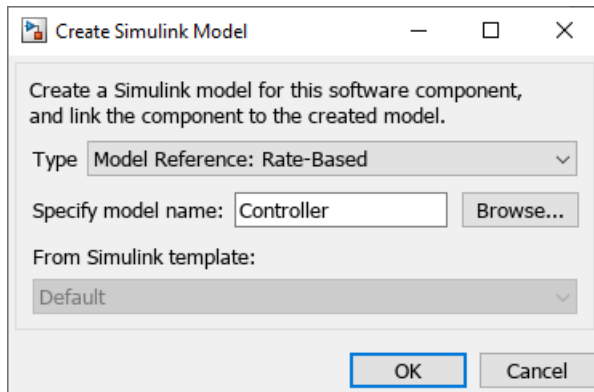
This example describes how to create Simulink behavior for a classic architecture component. The workflow for creating behavior for an adaptive component is the same.

- 1 Create or open an architecture model. To create a model, open the Simulink Start Page. Under AUTOSAR Blockset, open the **Software Architecture** template.
- 2 From the **Modeling** tab, in the **Platform** section, confirm that the AUTOSAR platform is correct. For this example, select **Classic Platform**.
- 3 From the **Modeling** tab or the palette to the left of the canvas, add a Classic Component block to the model and name it **Controller**. The Property Inspector displays the component **Kind** property as **Application**, which is correct for this component.
- 4 Click the block edges to add require (input) ports named **APP_Percent** and **TPS_Percent** and a provide (output) port named **ThrCmd_Percent**. (For a controller component with the same naming, see the example “Author AUTOSAR Classic Compositions and Components in Architecture Model” on page 8-50.)



- 5 Select the **Controller** block, place your cursor over the displayed ellipsis, and select cue **Create Simulink Behavior**. A model creation dialog box opens.
 - a Enter the type of model reference, periodic-rate runnable (**Rate-Based**) or function-call runnable (**Export-Function**). For more information about modeling patterns, see “Modeling Patterns for AUTOSAR Runnables” on page 2-10.

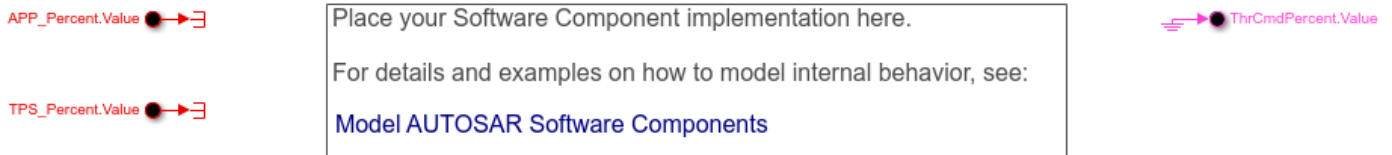
For this example, to create behavior for a classic architecture component, select **Rate-Based**.
 - b Enter a name for the new model or accept the block name default.
 - c Select a custom Simulink template for the new model or accept the default, a blank template. For more information about creating your own Simulink templates, see “Create Template from Model”.



To create a stub implementation model and map it to the AUTOSAR Controller component, click **OK**.


Model Controller.slx is created in the working folder.

- 6 To view the initial model content, open the Controller block. The ports are stubbed with Ground and Terminator blocks so that the model can immediately be updated and simulated.



- 7 In the open Controller model, to view the model mapping and dictionary, open the AUTOSAR Component Designer app. This view shows the mapping and properties of the model port APP_Percent.Value. The model port maps to AUTOSAR component port APP_Percent.

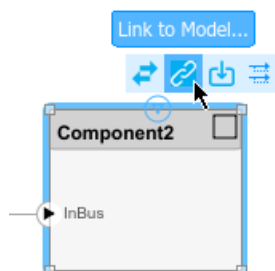


To view and modify additional AUTOSAR properties for the currently-selected element, click the  icon.

- After creating the stub model representation of the AUTOSAR component, use Simulink tools to develop the component implementation. You refine the AUTOSAR configuration and create algorithmic model content. For an example Controller block implementation, see the model `autosar_tpc_controller` provided with example “Author AUTOSAR Classic Compositions and Components in Architecture Model” on page 8-50.

Link to Implementation Model

To reference an existing Simulink implementation model from an AUTOSAR software component, use the block cue **Link to Model** for the Classic Component or Adaptive Component block. Clicking the cue initiates linking of the component block to an implementation model that you specify. By linking to existing models, you can deploy verified implementation models in your AUTOSAR design without requalification.



The implementation model must meet model linking requirements. The model must:

- Use the same AUTOSAR target as the architecture model.
- Have a complete mapping of Simulink model elements to AUTOSAR component elements.
- Implement root-level ports with In Bus Element and Out Bus Element blocks instead of Inport and Outport blocks.
- Use a fixed-step solver.
- Map to an AUTOSAR software component that is not already mapped to a different model in the composition hierarchy.

If the specified implementation model meets the linking requirements, the software links the component block to the model and updates the block and model interfaces to match.

If the implementation model does not meet one or more of the linking requirements, the software opens the AUTOSAR Model Linker app, which offers fixes for the unmet requirements. For example, if an implementation model uses root Inport and Outport blocks, the app offers to fix the issue by converting the signal ports to bus ports. When you click **Fix All**, the software fixes the unmet requirements and finishes linking the component block to the model.

This example describes how to link an AUTOSAR classic component to an existing implementation model. The workflow for linking an adaptive component to an implementation model is the same.

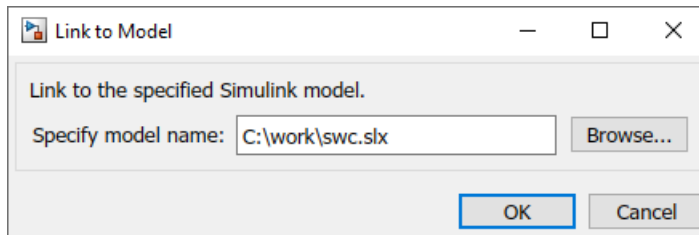
- 1 Create or open an architecture model. To create a model, open the Simulink Start Page. Under AUTOSAR Blockset, open the **Software Architecture** template.
- 2 From the **Modeling** tab, in the **Platform** section, confirm that the AUTOSAR platform is correct. For this example, select **Classic Platform**.
- 3 From the **Modeling** tab or the palette, add a Classic Component block to the model. The Property Inspector displays the component **Kind** property as **Application**, which is correct for this component.



- 4 Link the Component block to an implementation model that is not already configured for architecture model use. For example, select a model that is not configured for AUTOSAR or uses signal ports instead of bus ports at the root level. This example uses the swc model.

```
openExample('swc')
```

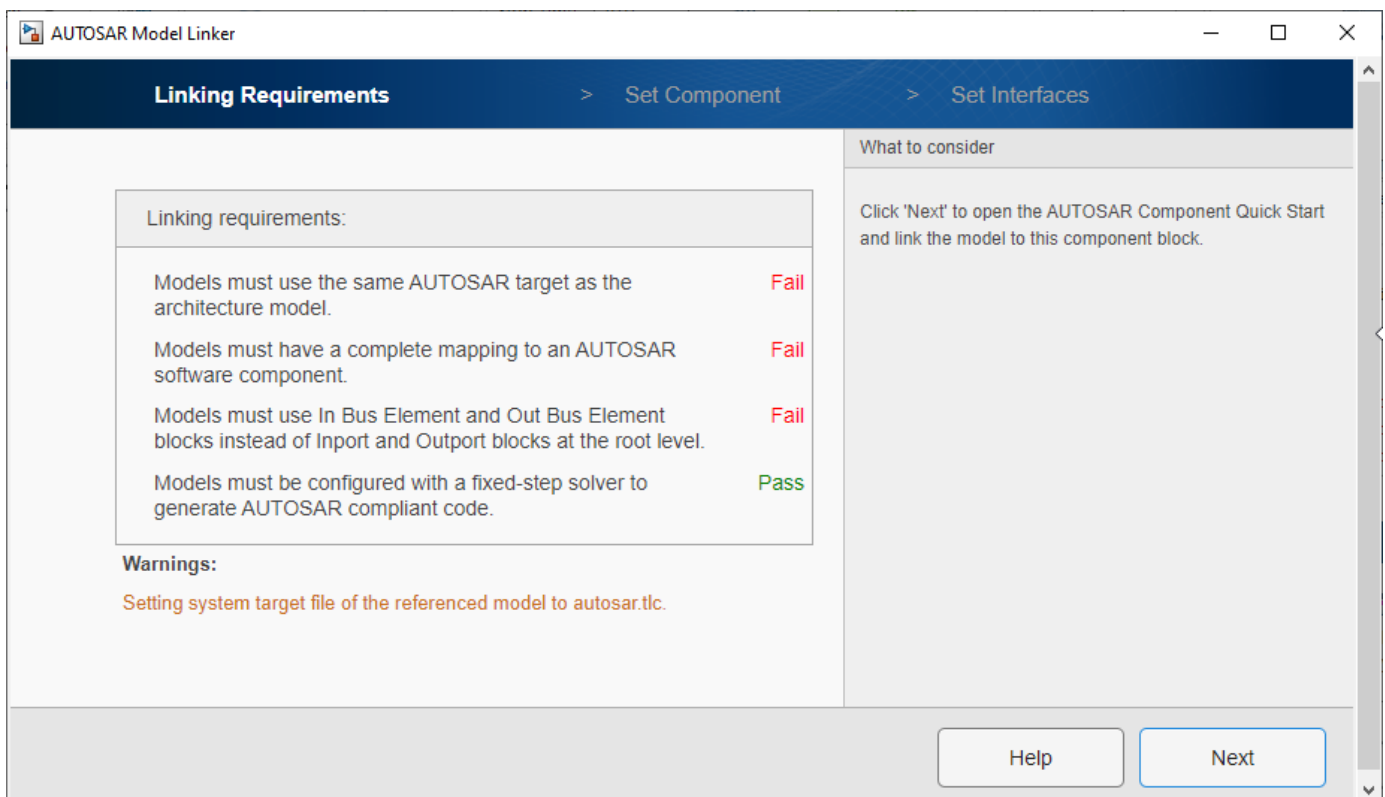
- 5 Select the Component block, place your cursor over the displayed ellipsis, and select cue **Link to Model**. In the Link to Model dialog box, browse to the implementation model swc.



To reference the implementation model from the AUTOSAR Component component, click **OK**.

If the specified implementation model does not meet one or more of the linking requirements, the software opens the AUTOSAR Model Linker app, which offers fixes for the unmet requirements.

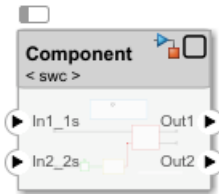
- 6 Observe the linking requirements for swc.



If the **Linking Requirements** pane displays a **Fix All** button, you are ready to fix the unmet linking requirements and link the component block to the implementation model. Click **Fix All**.

If the implementation model does not have a complete AUTOSAR component mapping, as in this example, you must map the model before linking. Click **Next** and work through mapping panes **Set Component** and **Set Interfaces**. For more information, see “Create AUTOSAR Software Component in Simulink” on page 3-2. When you complete the **Set Interfaces** pane, click **Fix All**.

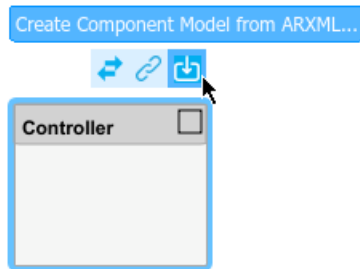
- 7 Simulink links the Component block to model swc and updates the block interface to match the model implementation.



- 8 To view the model content, open the Component block. In the open Component model, to view the model mapping and dictionary, open the AUTOSAR Component Designer app.
- 9 After linking the AUTOSAR component to the implementation model, you can connect the component block to other blocks or root ports in the design.

Create Model from ARXML Component Description

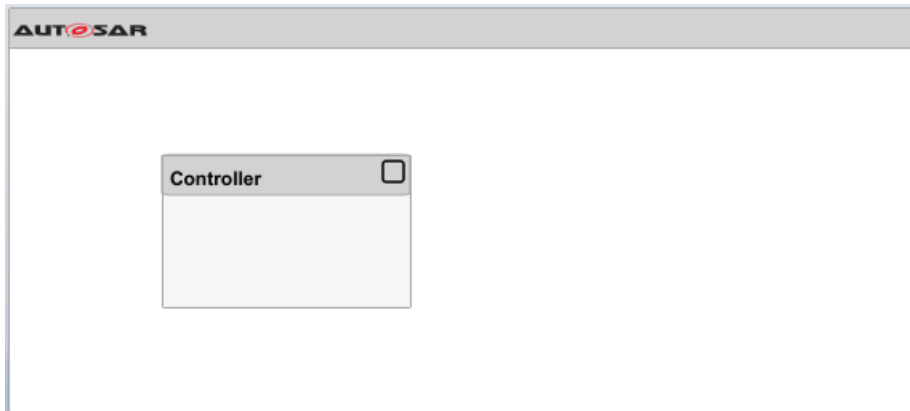
To create an AUTOSAR implementation model from an ARXML component description and map it to an AUTOSAR software component, use the block cue **Create Component Model from ARXML** for the Classic Component block or the Adaptive Component block.



Clicking the cue creates a model based on a specified ARXML description, links the component block to the model, and updates the block and model interfaces to match.

This example describes how to create an AUTOSAR classic model based on a specified ARXML description. The workflow is the same for creating an adaptive model from ARXML.

- 1 Create or open an architecture model. To create a model, open the Simulink Start Page. Under AUTOSAR Blockset, open the **Software Architecture** template.
- 2 From the **Modeling** tab, in the **Platform** section, confirm that the AUTOSAR platform is correct. For this example, select **Classic Platform**.
- 3 From the **Modeling** tab or from the palette, add a Classic Component block to the model and name it **Controller**. The Property Inspector displays the component **Kind** property as **Application**, which is correct for this component.



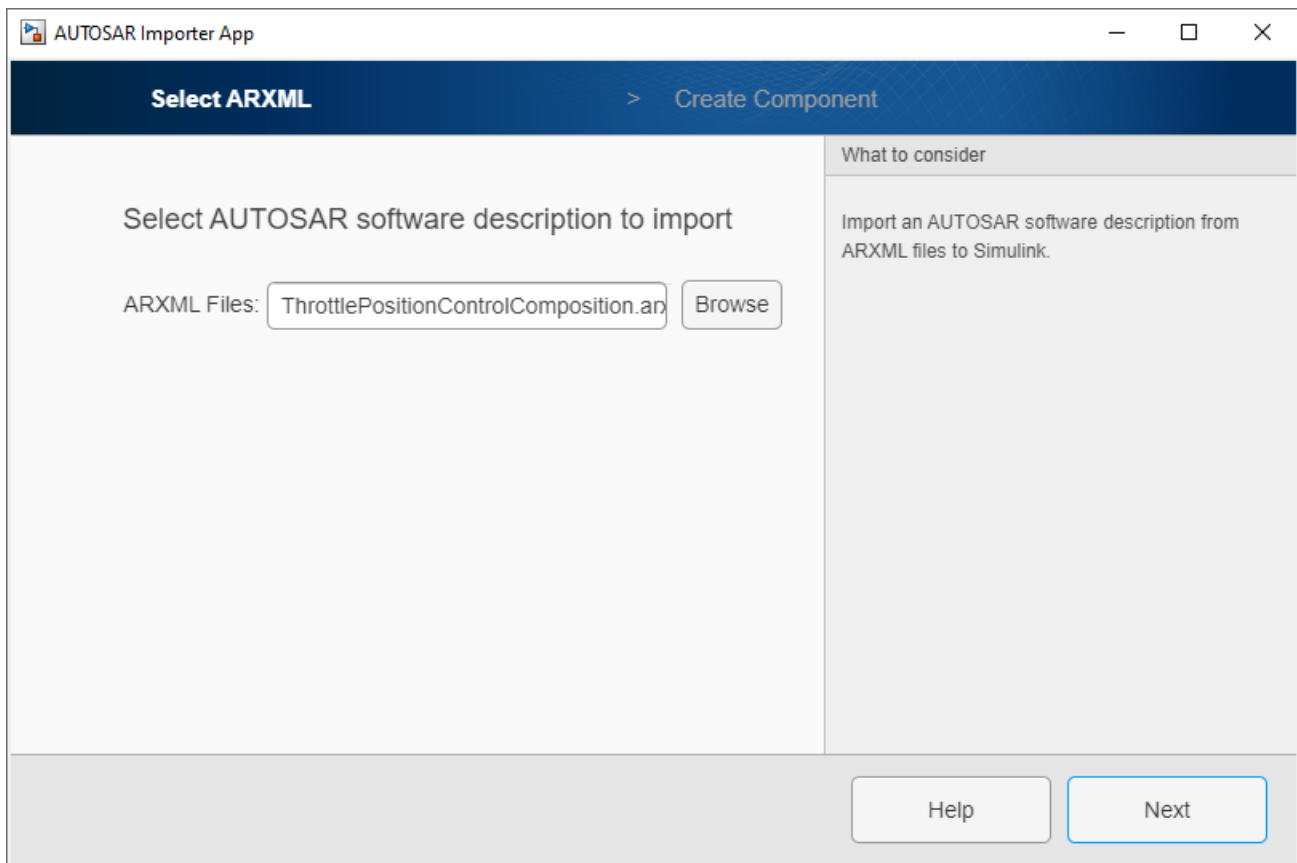
- 4 The example “Import AUTOSAR XML Descriptions Into Simulink” on page 3-13 provides an ARXML file that includes a controller component description. The ARXML file is on the default MATLAB search path. Open the ARXML file using this MATLAB command:

```
openExample('autosarblockset/ImportAUTOSARComponentToSimulinkExample',...  
'supportingfile','ThrottlePositionControlComposition.arxml');
```

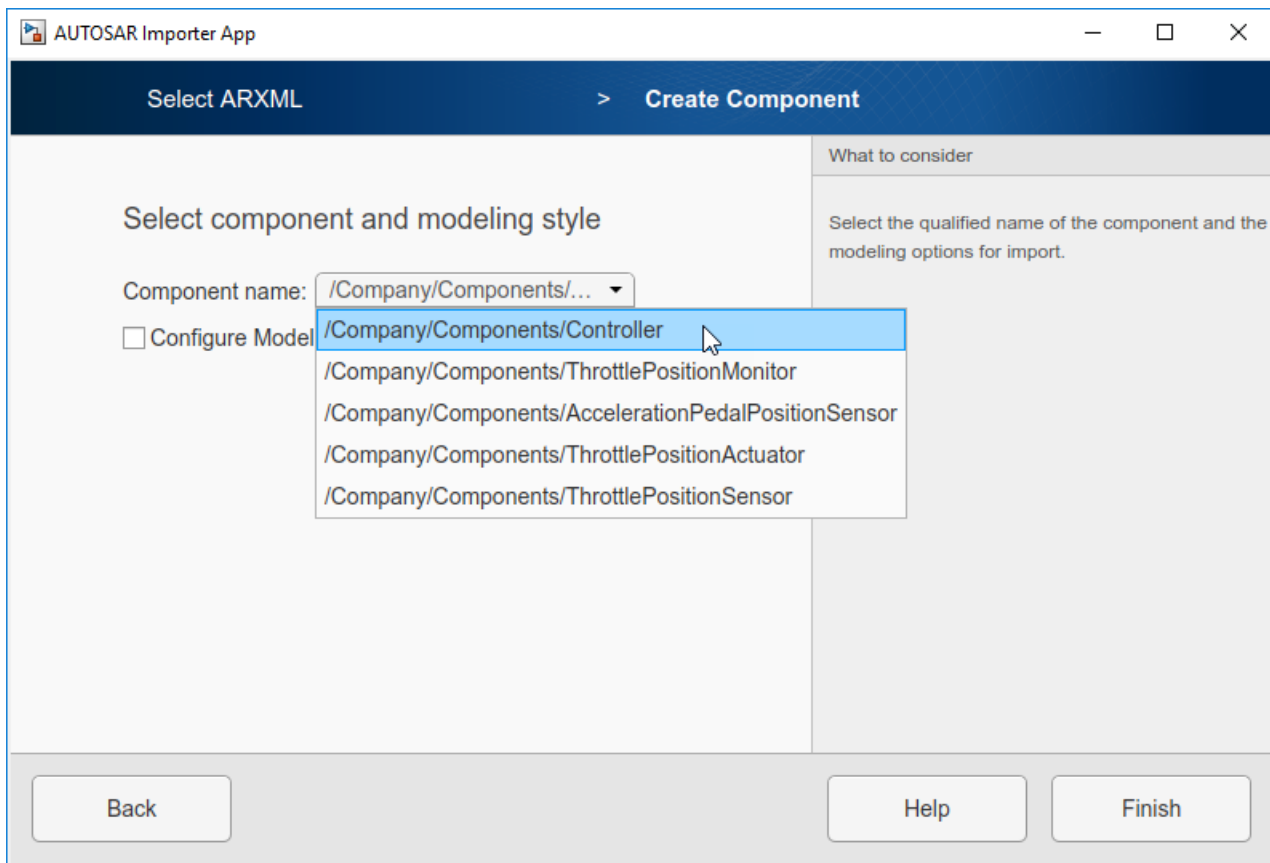
- 5 Select the **Controller** block, place your cursor over the displayed ellipsis, and select cue **Create Component Model from ARXML**. The AUTOSAR Importer App opens.

Work through the import and model creation procedure.

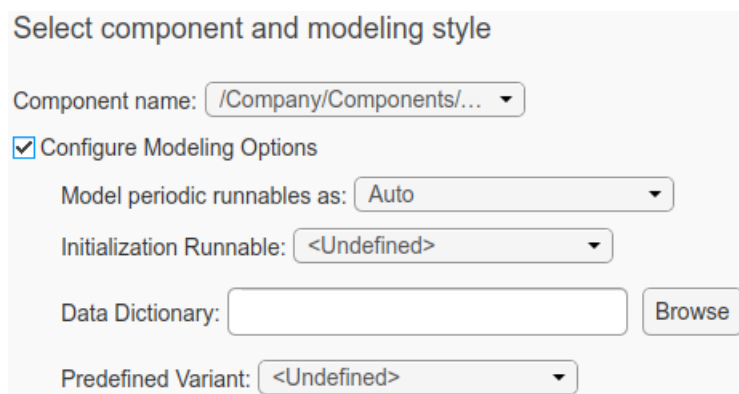
- 6 In the **Select ARXML** pane, browse to one or more AUTOSAR XML files that provide one or more software component descriptions. This example uses a file copied in an earlier step, `ThrottlePositionControlComposition.arxml`. To import the description, click **Next**.



- 7 In the **Create Component** pane, select the software component from which to create a model. From the list of components imported in the previous step, this example selects **Controller**.



To view optional settings for model creation, select **Configure Modeling Options**.



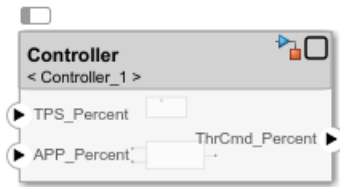
You can:

- Model periodic runnables as atomic subsystems or function-call subsystems, or accept a default modeling style selection (**Auto**).
- Select an existing AUTOSAR runnable as the initialization runnable for the component. In this example, **Controller_Init** is available for selection.
- Specify a Simulink data dictionary into which to import data objects corresponding to AUTOSAR data types in the XML file. If the specified dictionary does not already exist, the importer creates it. The model is then associated with the data dictionary.

- Select an AUTOSAR PredefinedVariant defined in the AUTOSAR XML file to initialize SwSystemconst data that serves as input to control variation points. For more information, see “Control AUTOSAR Variants with Predefined Value Combinations” on page 4-227. In this example, no PredefinedVariant is available for selection.

For more information about model creation options and behavior, see `createComponentAsModel`.

- 8 To create the model and map it to the AUTOSAR Controller component, click **Finish**. Simulink creates model `Controller.slx` in the working folder and updates the block interface to match the model implementation.



- 9 To view the model content, open the Controller block. In the open Controller model, to view the model mapping and dictionary, open the AUTOSAR Component Designer app.
- 10 After creating the AUTOSAR implementation model and linking the AUTOSAR component to it, connect the component block to other blocks or root ports in the design. For a fully connected controller component, see example “Author AUTOSAR Classic Compositions and Components in Architecture Model” on page 8-50.

See Also

Software Component | `createComponentAsModel`

Related Examples

- “Configure AUTOSAR Scheduling and Simulation” on page 8-38
- “Author AUTOSAR Classic Compositions and Components in Architecture Model” on page 8-50
- “Configure AUTOSAR Ports By Using Simulink Bus Ports” on page 4-138
- “Import AUTOSAR XML Descriptions Into Simulink” on page 3-13
- “Control AUTOSAR Variants with Predefined Value Combinations” on page 4-227
- “Configure AUTOSAR Architecture Model Programmatically” on page 8-67

Configure AUTOSAR Scheduling and Simulation

You can simulate the behavior of the aggregated components in an AUTOSAR classic architecture model or an adaptive architecture model. To test the model, you can connect a test harness model that provides input values and other modeling elements. To simulate, go to the top level of the architecture model and click **Run**.

As an example of configuring scheduling and simulation for an AUTOSAR classic architecture model, you can:

- Add Basic Software (BSW) blocks to simulate calls to BSW services.
- Create a test harness model to connect inputs and plant elements to the architecture model.
- Use the Schedule Editor to schedule and specify the execution order of component runnables.

Simulate Basic Software Service Calls

For the AUTOSAR Classic Platform, AUTOSAR Blockset provides Basic Software (BSW) blocks, which allow you to model software component calls to BSW services that run in the AUTOSAR run-time environment. BSW services include NVRAM Manager (NvM), Diagnostic Event Manager (Dem), and Function Inhibition Manager (FiM). In the run-time environment, AUTOSAR software components typically access BSW services using client-server or sender-receiver communication.

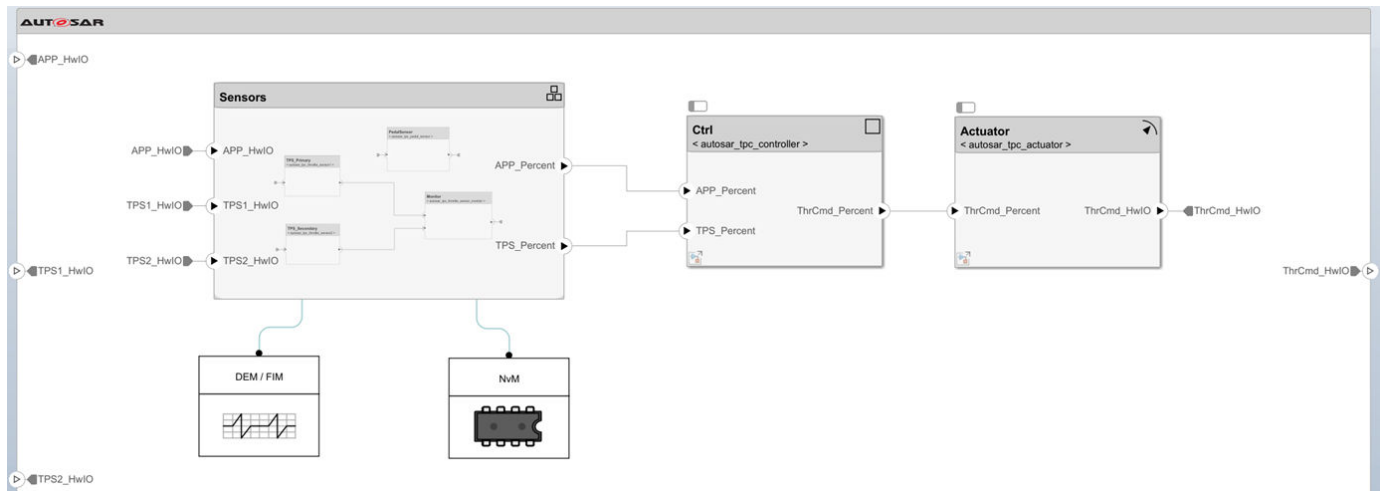
To simulate AUTOSAR components that call BSW services, you create a containing architecture, composition, or test harness model and add preconfigured BSW service component blocks. The blocks provide reference implementations of BSW service operations.

If the components in your architecture model use BSW caller blocks, make sure that the architecture model contains BSW service implementations. For more information, see “Model AUTOSAR Basic Software Service Calls” on page 7-12 and “Simulate AUTOSAR Basic Software Services and Run-Time Environment” on page 7-36.

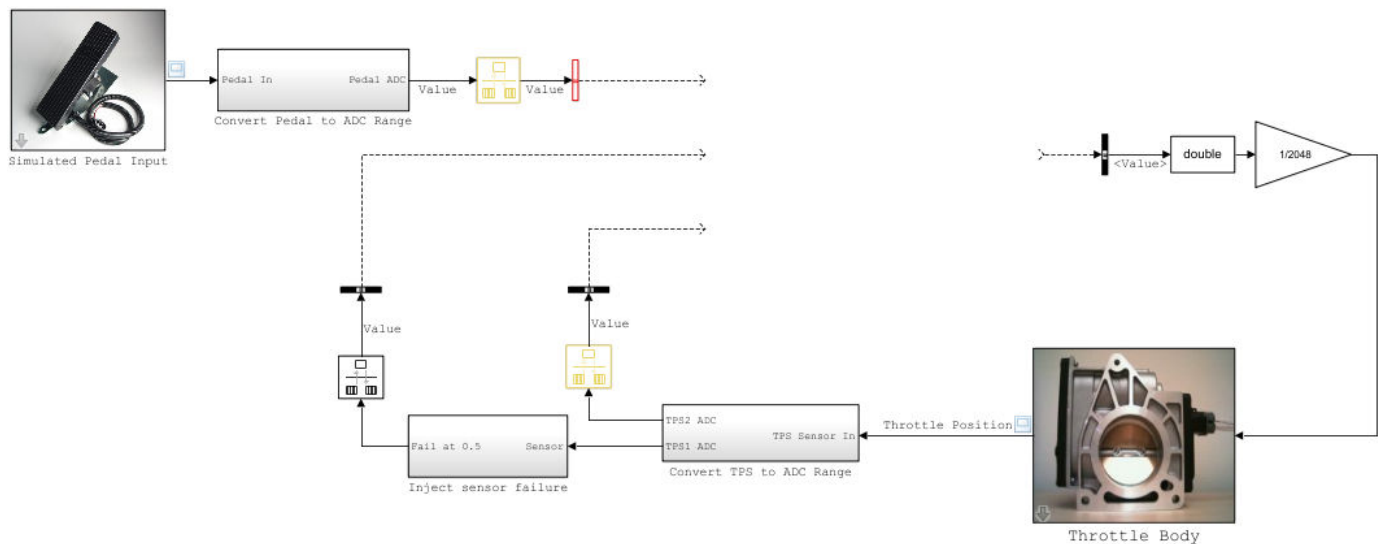
For an example of using BSW blocks in an AUTOSAR architecture model, see “Author AUTOSAR Classic Compositions and Components in Architecture Model” on page 8-50.

Connect a Test Harness

After you develop an architecture model, you can connect it to a test harness model that provides meaningful input values and plant model elements. For example, consider the architecture model `autosar_tpc_composition` from example “Author AUTOSAR Classic Compositions and Components in Architecture Model” on page 8-50. The model has three require (input) ports and one provide (output) port.

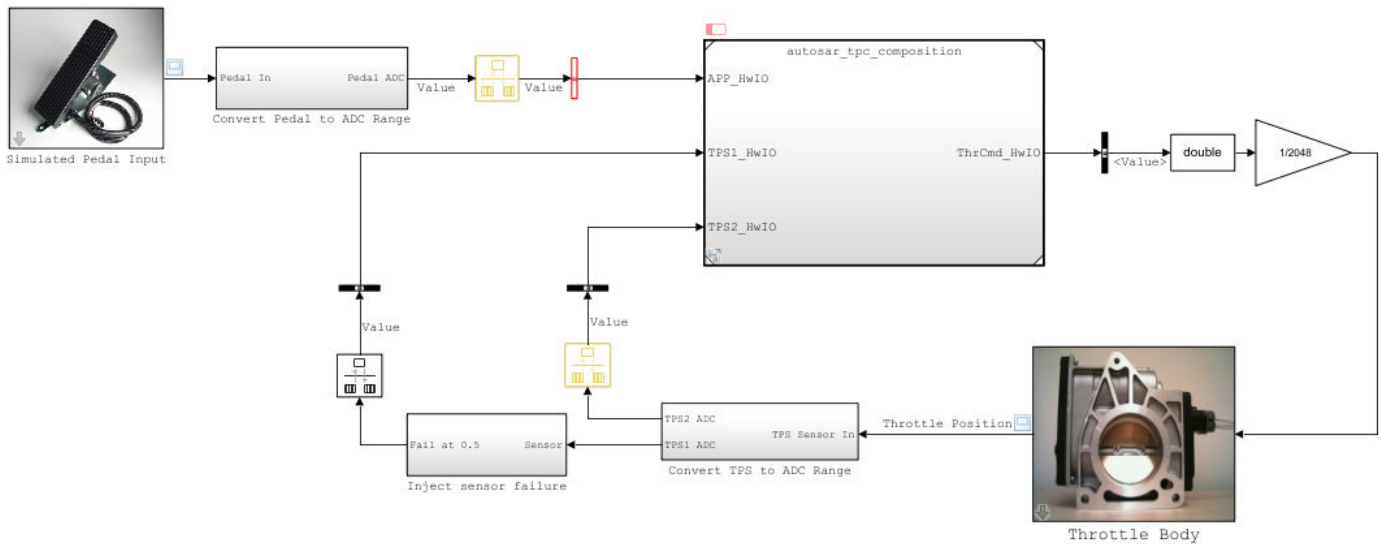


Here is a test harness model for simulating the architecture model `autosar_tpc_composition`. The test harness contains a plant model with a pedal input block and signals that correspond with the architecture model require and provide ports. This model was adapted from example model `autosar_tpc_system`.



To connect the architecture model to the test harness:

- 1 Insert a Model block.
- 2 Configure the Model block to reference the architecture model.
- 3 In the Model block dialog box, select the option **Schedule rates**. For the associated parameter **Schedule rates with**, select **Schedule Editor**. The architecture model components have explicit partitions that you can schedule with the **Schedule Editor**.
- 4 Connect the architecture model ports to the test harness signals.



To view and run the completed test harness model, open example model `autosar_tpc_system`. (To open the model in a local working folder, use `openExample('autosar_tpc_system')`.)

Schedule Component Runnables

For AUTOSAR Classic Platform software components that contain multiple runnables, the AUTOSAR Timing Extensions specification defines execution order constraints. These constraints specify the execution order of runnable entities within a component. You can view and manipulate the constraints at the component level or, in AUTOSAR architecture models, at the Virtual Function Bus (VFB) level.

In architecture models, you can:

- Import VFB-level execution order constraints from ARXML files.
- Use the Schedule Editor to modify the execution order of AUTOSAR component runnables. The editor displays every runnable in every component in the composition hierarchy.
- As part of composition export, export VFB-level execution order constraints to an ARXML timing module, `modelName_timing.arxml`.

To schedule and specify the execution order of AUTOSAR component runnables, use the Schedule Editor. From a standalone component model or an architecture model, you can:

- View a graphical representation of component runnables as partitions in an AUTOSAR component or architecture model.
- Create partitions and map them to AUTOSAR runnables.
- Directly specify the execution order of runnables.

The Schedule Editor supports multiple modeling styles, including rate-based and export-function modeling. For more information, see “Using the Schedule Editor” and “Create Partitions”. For AUTOSAR component model examples, see “Configure AUTOSAR Runnable Execution Order” on page 4-181.

In an AUTOSAR architecture model, to open the Schedule Editor, open the **Modeling** tab and select **Design Tools > Schedule Editor**. The editor displays every runnable in every component in the

composition hierarchy. Here is the execution order view when you open the Schedule Editor from the example architecture model `autosar_tpc_composition`. Use the editor controls to modify the execution order of the runnables.

The screenshot shows the Schedule Editor window for `autosar_tpc_composition`. The interface includes a toolbar with options like Manage Partitions, Order, Update Diagram, Save Model, Highlight, Arrange, Timing Legend, and Layout. The main workspace displays a dependency graph with nodes representing runnables and their execution order. The right-hand panel shows the execution order table and the property inspector for the selected `PedalSensor.D1` partition.

Order	Name	Trigger
1	■ D1 <i>implicit</i>	0.005
2	■ PedalSensor.D1	0.005
3	■ TPS_Primary.D1	0.005
4	■ TPS_Secondary.D1	0.005
5	■ Monitor.D1	0.005
6	■ Ctrl.D1	0.005
7	■ Actuator.D1	0.005

PROPERTY INSPECTOR

Partition

Name	PedalSensor.D1
Rate	0.005
Type	Explicit periodic partition

Exporting a composition from an AUTOSAR architecture model exports VFB-level execution order constraints into the file `modelName_timing.arxml`. The ARXML module aggregates timing information from the entire composition hierarchy. This ARXML code shows the execution order constraint exported for the runnables in `autosar_tpc_composition`, based on the Schedule Editor configuration.

```
<VFB-TIMING UUID="...">
  <SHORT-NAME>TPC_Composition</SHORT-NAME>
  <TIMING-REQUIREMENTS>
    <EXECUTION-ORDER-CONSTRAINT UUID="...">
      <SHORT-NAME>EOC</SHORT-NAME>
      <BASE-COMPOSITION-REF DEST="COMPOSITION-SW-COMPONENT-TYPE">
        /Components/TPC_Composition
      </BASE-COMPOSITION-REF>
      <ORDERED-ELEMENTS>
        <EOC-EXECUTABLE-ENTITY-REF UUID="...">
          <SHORT-NAME>PedalSensor_PedalSensor_Step</SHORT-NAME>
          <COMPONENT-IREF>
            <TARGET-COMPONENT-REF DEST="SW-COMPONENT-PROTOTYPE">
              /Components/Sensors/PedalSensor
            </TARGET-COMPONENT-REF>
          </COMPONENT-IREF>
          <EXECUTABLE-REF DEST="RUNNABLE-ENTITY">
            /Components/PedalSensor/PedalSensor_IB/PedalSensor_Step
          </EXECUTABLE-REF>
        </EOC-EXECUTABLE-ENTITY-REF>
      </ORDERED-ELEMENTS>
    </EXECUTION-ORDER-CONSTRAINT>
  </TIMING-REQUIREMENTS>
</VFB-TIMING>
```

```

    <SUCCESSOR-REFS>
      <SUCCESSOR-REF DEST="EOC-EXECUTABLE-ENTITY-REF">
        /Timing/TPC_Composition/EOC/TPS_Primary_ThrottleSensor1_Step
      </SUCCESSOR-REF>
    </SUCCESSOR-REFS>
  </EOC-EXECUTABLE-ENTITY-REF>
<EOC-EXECUTABLE-ENTITY-REF UUID="...">
  <SHORT-NAME>TPS_Primary_ThrottleSensor1_Step</SHORT-NAME>
  ...
</EOC-EXECUTABLE-ENTITY-REF>
<EOC-EXECUTABLE-ENTITY-REF UUID="...">
  <SHORT-NAME>TPS_Secondary_ThrottleSensor2_Step</SHORT-NAME>
  ...
</EOC-EXECUTABLE-ENTITY-REF>
<EOC-EXECUTABLE-ENTITY-REF UUID="...">
  <SHORT-NAME>Monitor_ThrottleSensorMonitor_Step</SHORT-NAME>
  ...
</EOC-EXECUTABLE-ENTITY-REF>
<EOC-EXECUTABLE-ENTITY-REF UUID="...">
  <SHORT-NAME>Ctrl_Controller_Step</SHORT-NAME>
  ...
</EOC-EXECUTABLE-ENTITY-REF>
<EOC-EXECUTABLE-ENTITY-REF UUID="...">
  <SHORT-NAME>Actuator_Actuator_Step</SHORT-NAME>
  <COMPONENT-IREF>
    <TARGET-COMPONENT-REF DEST="SW-COMPONENT-PROTOTYPE">
      /Components/TPC_Composition/Actuator
    </TARGET-COMPONENT-REF>
  </COMPONENT-IREF>
  <EXECUTABLE-REF DEST="RUNNABLE-ENTITY">
    /Components/Actuator/Actuator_IB/Actuator_Step
  </EXECUTABLE-REF>
</EOC-EXECUTABLE-ENTITY-REF>
</ORDERED-ELEMENTS>
</EXECUTION-ORDER-CONSTRAINT>
</TIMING-REQUIREMENTS>
<COMPONENT-REF DEST="COMPOSITION-SW-COMPONENT-TYPE">
  /Components/TPC_Composition
</COMPONENT-REF>
</VFB-TIMING>

```

See Also

Diagnostic Service Component | NVRAM Service Component | **Schedule Editor**

Related Examples

- “Generate and Package AUTOSAR Composition XML Descriptions and Component Code” on page 8-43
- “Author AUTOSAR Classic Compositions and Components in Architecture Model” on page 8-50
- “Model AUTOSAR Basic Software Service Calls” on page 7-12
- “Design and Simulate AUTOSAR Components and Generate Code” on page 4-77
- “Configure AUTOSAR Runnable Execution Order” on page 4-181
- “Using the Schedule Editor”
- “Create Partitions”
- “Configure AUTOSAR Architecture Model Programmatically” on page 8-67

Generate and Package AUTOSAR Composition XML Descriptions and Component Code

If you have Simulink Coder and Embedded Coder software, from an AUTOSAR architecture model, you can:

- Export composition and component AUTOSAR XML (ARXML) descriptions and generate component code.
- Optionally, create a ZIP file to package build artifacts from the model hierarchy, for example, for relocation and integration.
- Optionally, for AUTOSAR classic architecture models with ECU configuration, export an ECU extract that maps the software components in a composition to an AUTOSAR ECU.

You can export an entire architecture model, a nested composition, or a single component. If you initiate an export that encompasses a composition, the export includes XML descriptions of the composition, component prototypes, and composition ports and connectors.

Configure Composition XML Options

To prepare for exporting ARXML files, examine and modify XML options. XML options specified at the architecture model level are inherited during export by each component in the model.

To view XML options:

- 1 Open an AUTOSAR architecture model.

For example, to open the example classic model `autosar_tpc_composition`, enter:

```
openExample('autosar_tpc_composition')
```

- 2 From the **Modeling** tab, select **Export > Configure XML Options**. The **XML Options** dialog box opens.

This dialog box shows XML options for the classic architecture model. Modifications you make to these options are inherited by every component in the hierarchy.

The **System Package** option applies only to the composition level. If you export an ECU extract for a composition in a classic architecture model, **System Package** specifies the system package path to generate in the composition ARXML. For more information, see “Export Composition ECU Extract” on page 8-47.

XML option **Exported XML File Packaging** is supported for exporting architecture models.

Setting the **Exported XML File Packaging** parameter allows you to specify the granularity of XML file packaging for AUTOSAR elements created in Simulink. Selecting **Single file** exports XML into a single file. Selecting **Modular** exports XML into multiple files, named according to the type of information contained.

For more information about each XML option, see “Configure AUTOSAR XML Options” on page 4-43 for classic architecture model options and “Configure AUTOSAR Adaptive XML Options” on page 6-33 for adaptive architecture options.

Export Composition XML and Component Code

To export ARXML files and generate code for an architecture model:

- 1 Open an architecture model.

For this example, open the example classic model `autosar_tpc_composition`, enter:

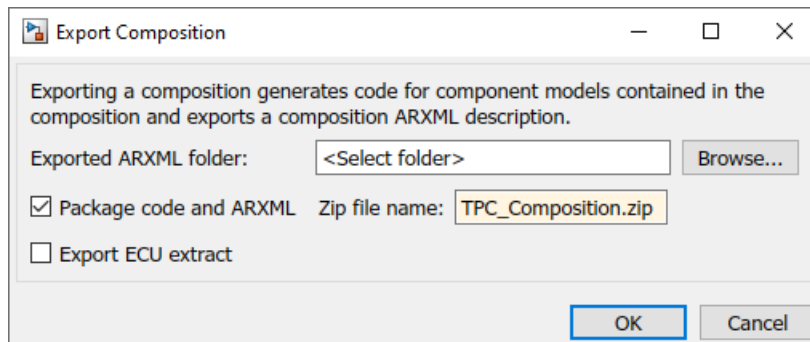
```
openExample('autosar_tpc_composition')
```

- 2 Export the architecture model.

From the **Modeling** tab, select **Export > Generate Code and ARXML**. In the Export Composition dialog box:

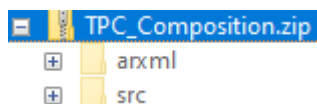
- Specify the name of the ZIP file to package the generated files in.
- Optionally, specify a path to a folder to contain the exported ARXML files.
- To export an ECU extract from a classic composition, select **Export ECU extract**. For more information, see “Export Composition ECU Extract” on page 8-47.

To begin the export, click **OK**.



As the architecture model builds, you can view the build log in the Diagnostic Viewer. First the component models build, each as a standalone top-model build. Finally, composition ARXML is exported. When the build is complete, the current folder contains build folders for the architecture model and each component model in the hierarchy, and the specified ZIP file.

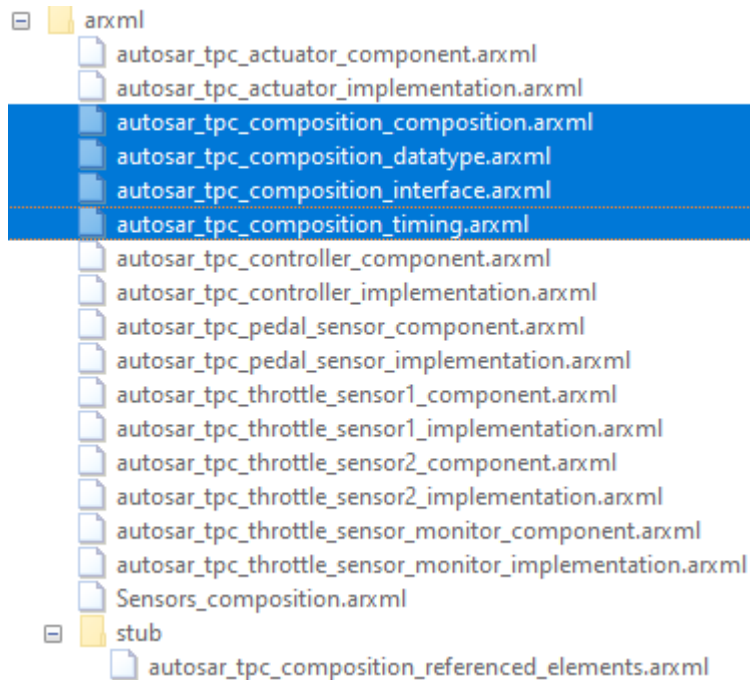
- 3 Expand the ZIP file. Its content is organized in `arxml` and `src` folders.



- 4 Examine the `arxml` folder.

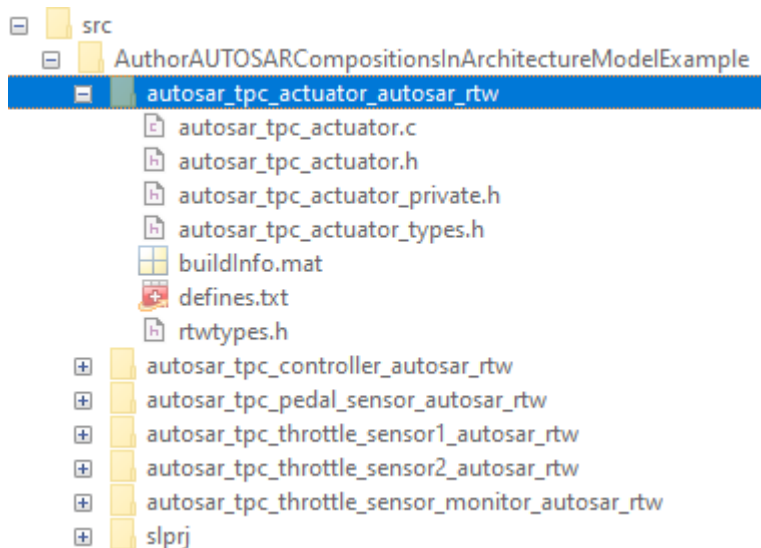
Each AUTOSAR component has component and implementation description files, while the architecture model has composition, datatype, interface, and timing description files. The composition file includes XML descriptions of the composition, component prototypes, and

composition ports and connectors. The datatype, interface, and timing files aggregate elements from the entire architecture model hierarchy.



5 Examine the src folder.

Each component model has a build folder that contains artifacts from a standalone model build.



To export a nested composition or a single component in an architecture model, use composition or component block cues or right-click options. For example, right-click a component block and select **Export Component**. Components exported from an architecture model inherit the XML options specified at the architecture model level.

When exporting an architecture model, AUTOSAR schema versions must match between the architecture model and the component models in the hierarchy. If export flags a version difference, fix the discrepancy in the component model or in the architecture model. To view the architecture model schema version, open the Configuration Parameters dialog box. In the **Modeling** tab, select **Model Settings**. In the dialog box, navigate to the AUTOSAR code generation options pane.

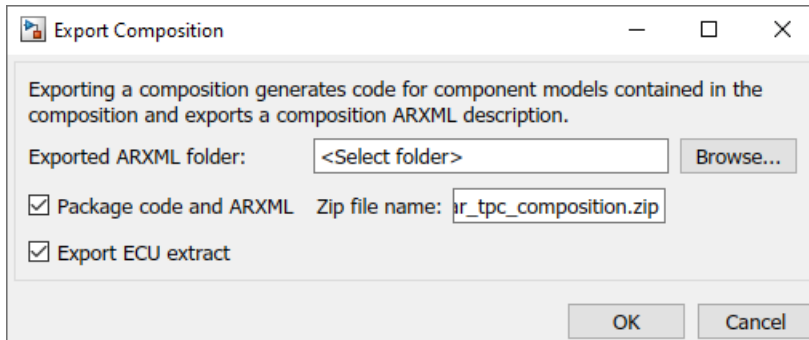
To export from an architecture model hierarchy programmatically, use the architecture function `export`. For example, to generate and package ARXML files and code for example model `autosar_tpc_composition`:

```
% Load AUTOSAR architecture model
archModel = autosar.arch.loadModel('autosar_tpc_composition');
% Export ARXML descriptions and code into ZIP file
export(archModel, 'PackageCodeAndARXML', 'myArchModel.zip');
```

Export Composition ECU Extract

You can export ECU extracts from compositions in an AUTOSAR classic architecture model. ECU extracts are an important input to AUTOSAR ECU configuration. In an AUTOSAR classic architecture, a top-level composition can model the software components mapped to one AUTOSAR ECU. To create a software description of the ECU-scoped system, you export an ECU extract from the composition.

In an open architecture model, you can export ARXML by using the Simulink Toolstrip, the software architecture canvas, or the `export` function. For example, from the **Modeling** tab, select **Export > Generate Code and ARXML**. In the Export Composition dialog box, select the option **Export ECU extract**. To begin the export, click **OK**.



To generate the ECU extract, the software automatically maps the software components in the composition to an ECU. If the composition contains nested compositions, the software uses a flattened version of the composition hierarchy, containing only components. For example, these function calls export an ECU extract for the AUTOSAR example architecture model `autosar_tpc_composition`, which contains a nested composition.

```
% Open and export AUTOSAR architecture model, generating ECU extract
archModel = 'autosar_tpc_composition';
openExample(archModel);
export(archModel, 'ExportECUExtract', true);
```

The `export` function call generates the ECU extract into the file `System.arxml`, which is located in the composition folder. The ECU extract for `autosar_tpc_composition` maps components from both the top-level composition and a nested `Sensors` composition to one ECU.

```
<SYSTEM UUID="...">
  <SHORT-NAME>EcuExtract</SHORT-NAME>
  <CATEGORY>ECU_EXTRACT</CATEGORY>
```

```

<MAPPINGS>
  <SYSTEM-MAPPING UUID="...">
    <SHORT-NAME>SystemMapping</SHORT-NAME>
    <SW-MAPPINGS>
      <SWC-TO-ECU-MAPPING UUID="...">
        <SHORT-NAME>SwcToEcuMapping</SHORT-NAME>
        <COMPONENT-IREFS>
          <COMPONENT-IREF>
            <TARGET-COMPONENT-REF DEST="SW-COMPONENT-PROTOTYPE">
              /Components/TPC_Composition/Ctrl
            </TARGET-COMPONENT-REF>
          </COMPONENT-IREF>
          ...
          <COMPONENT-IREF>
            <TARGET-COMPONENT-REF DEST="SW-COMPONENT-PROTOTYPE">
              /Components/TPC_Composition/PedalSensor
            </TARGET-COMPONENT-REF>
          </COMPONENT-IREF>
        </COMPONENT-IREFS>
        <ECU-INSTANCE-REF DEST="ECU-INSTANCE">
          /System/EcuInstance
        </ECU-INSTANCE-REF>
      </SWC-TO-ECU-MAPPING>
    </SW-MAPPINGS>
  </SYSTEM-MAPPING>
</MAPPINGS>
<ROOT-SOFTWARE-COMPOSITIONS>
  <ROOT-SW-COMPOSITION-PROTOTYPE UUID="...">
    <SHORT-NAME>RootSwCompositionPrototype</SHORT-NAME>
    <SOFTWARE-COMPOSITION-TREF DEST="COMPOSITION-SW-COMPONENT-TYPE">
      /Components/TPC_Composition
    </SOFTWARE-COMPOSITION-TREF>
  </ROOT-SW-COMPOSITION-PROTOTYPE>
</ROOT-SOFTWARE-COMPOSITIONS>
</SYSTEM>

<ECU-INSTANCE UUID="...">
  <SHORT-NAME>EcuInstance</SHORT-NAME>
</ECU-INSTANCE>

```

To specify the AUTOSAR package path for the system package that contains the ECU extract, use the composition XML option **System Package**. To view the **System Package** path value, from the **Modeling** tab, select **Export > Configure XML Options**.

System Package:

Alternatively, configure the AUTOSAR system package path by using the AUTOSAR property functions `get` and `set`.

```

openExample('autosar_tpc_composition');
arProps = autosar.api.getAUTOSARProperties('autosar_tpc_composition');
set(arProps, 'XmlOptions', 'SystemPackage', '/System');
systemPackage = get(arProps, 'XmlOptions', 'SystemPackage');

```

For more information about the hierarchical AUTOSAR package structure, see “Configure AUTOSAR Packages” on page 4-84.

See Also

Software Composition | Software Component | export

Related Examples

- “Author AUTOSAR Classic Compositions and Components in Architecture Model” on page 8-50

- “Configure AUTOSAR XML Options” on page 4-43
- “Generate AUTOSAR C Code and XML Descriptions” on page 5-2
- “Configure AUTOSAR Architecture Model Programmatically” on page 8-67

Author AUTOSAR Classic Compositions and Components in Architecture Model

Develop AUTOSAR compositions and components for the Classic Platform by using an architecture model.

An AUTOSAR architecture model provides resources and a canvas for developing AUTOSAR composition and component models. From the architecture model, you can:

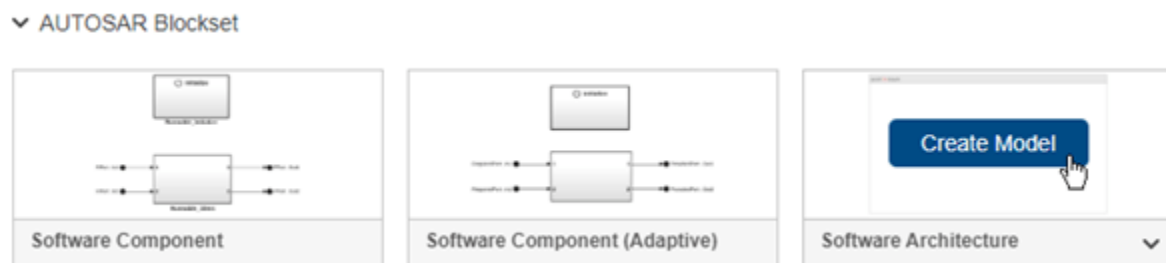
- Add and connect AUTOSAR compositions and components.
- Create architecture views for analysis.
- Link components to requirements (requires Requirements Toolbox).
- Define component behavior by creating, importing, or linking Simulink models.
- Configure scheduling and simulation.
- Export composition and component ARXML descriptions and generate component code (requires Embedded Coder).

Architecture models provide an end-to-end AUTOSAR software design workflow. In Simulink, you can author a high-level application design, implement behavior for application components, add Basic Software (BSW) service calls and service implementations, and simulate the application.

Create Architecture Model

To begin developing AUTOSAR compositions and components in a software architecture canvas, create an AUTOSAR architecture model (requires System Composer).

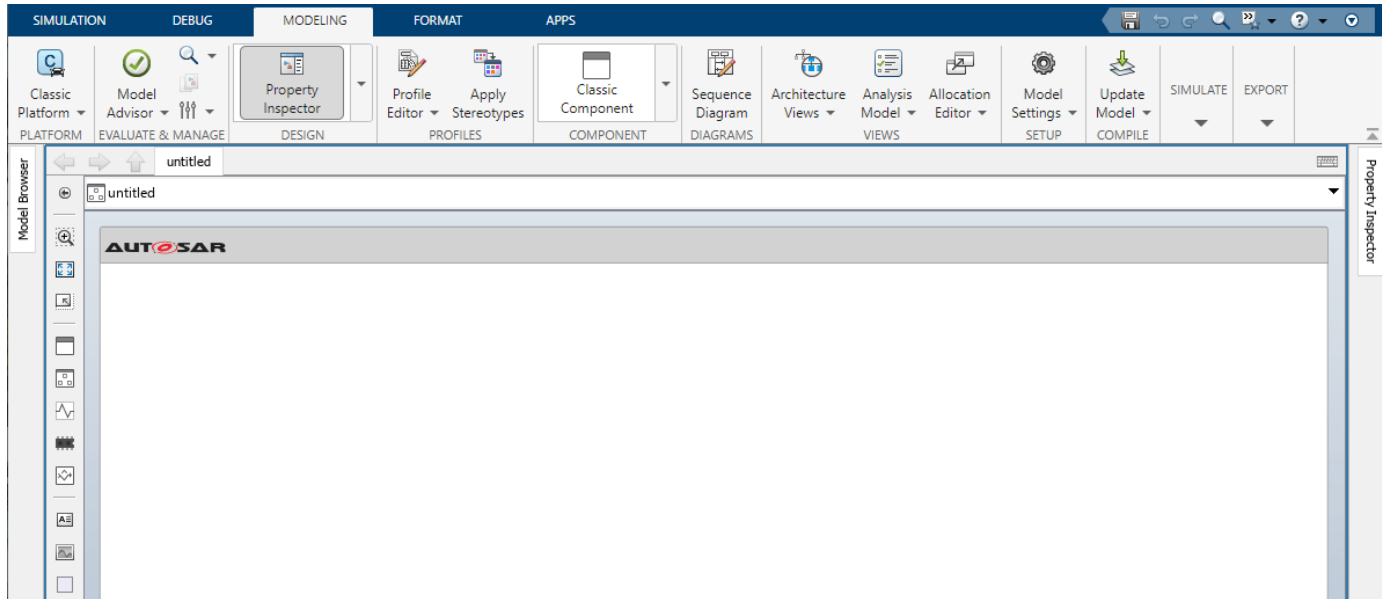
1. Open the Simulink Start Page by entering the MATLAB command `simulink`.
2. On the **New** tab, scroll down to AUTOSAR Blockset and expand the list of model templates. Place your cursor over the **Software Architecture** template and click **Create Model**.



A new AUTOSAR architecture model opens.

3. Explore the controls and content in the software architecture canvas.
 - In the Simulink Toolstrip, the **Modeling** tab supports common tasks for architecture modeling.
 - To the left of the model window, the palette includes icons for adding different types of AUTOSAR components to the model. For classic architectures, supported component blocks include Classic Component, Software Composition, and for Basic Software (BSW) modeling, Diagnostic Service Component and NVRAM Service Component.

- The composition editor provides a view of AUTOSAR software architecture based on the AUTOSAR Virtual Function Bus (VFB). The model canvas initially is empty.



You select the architecture platform, **Classic Platform** or **Adaptive Platform**, from the **Modeling** tab. The default is classic. Mixing classic and adaptive components in the same architecture model is not supported.

This example constructs a throttle position control application. Perform the steps in a new classic architecture model or refer to example model `autosar_tpc_composition`, which shows the end result.

```
% Open example model autosar_tpc_composition for reference
open_system('autosar_tpc_composition')
```

Add Compositions and Components and Link Implementation Models

After you create an AUTOSAR architecture model, use the composition editor and the Simulink Toolstrip **Modeling** tab to add and connect compositions and components.

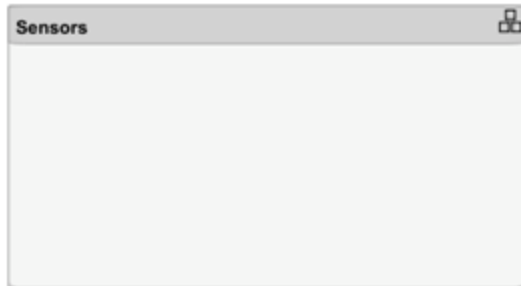
The behavior of the AUTOSAR application is defined by its AUTOSAR components, which you link to Simulink implementation models. For convenience, this example provides a Simulink implementation model for each AUTOSAR component:

- `autosar_tpc_throttle_sensor1.slx` for component `TPS_Primary`
- `autosar_tpc_throttle_sensor2.slx` for component `TPS_Secondary`
- `autosar_tpc_throttle_sensor_monitor.slx` for component `Monitor`
- `autosar_tpc_pedal_sensor.slx` for component `PedalSensor`
- `autosar_tpc_controller.slx` for component `Ctrl`
- `autosar_tpc_actuator.slx` for component `Actuator`

Four of the throttle position control components are sensor components, which this example places in a `Sensors` composition.

In your architecture model:

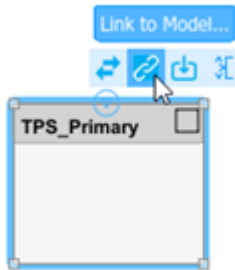
1. To create a nested **Sensors** composition, add a Software Composition block. For example, on the **Modeling** tab, select **Software Composition** and insert a Software Composition block in the canvas. In the highlighted name field, enter **Sensors**.



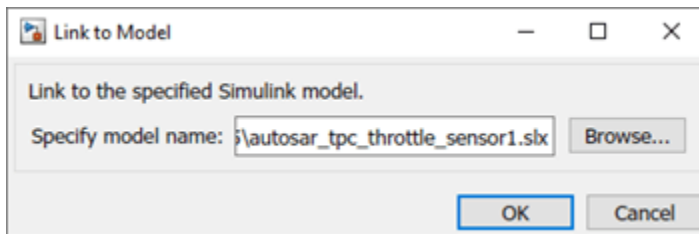
2. Open the **Sensors** block so that the model canvas shows the composition content. Inside the composition, add Classic Component blocks to represent AUTOSAR components named **TPS_Primary**, **TPS_Secondary**, **Monitor**, and **PedalSensor**. For example, on the **Modeling** tab, you can select **Classic Component** to create each one.



3. Link each AUTOSAR sensor component to a Simulink model that implements its behavior. For example, select the **TPS_Primary** component block, place your cursor over the displayed ellipsis, and select the cue **Link to Model**.



In the Link to Model dialog box, browse to the implementation model `autosar_tpc_throttle_sensor1.slx`.

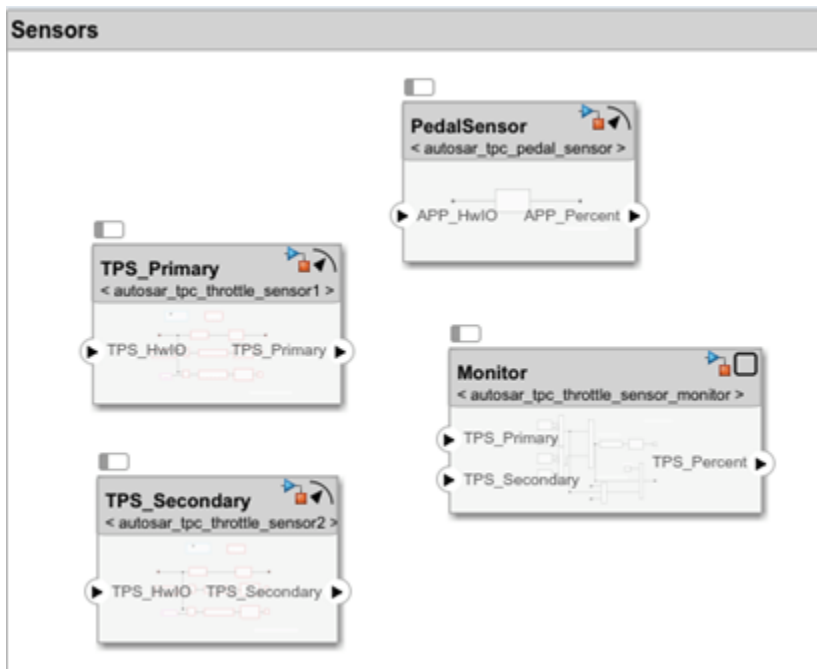


To link the component to the implementation model, click **OK**.

In an architecture model, when you initiate linking of a component block to an implementation model, the software verifies whether the specified model meets linking requirements. For example, the implementation model must use the same target as the architecture model, use a fixed-step solver, and use root-level bus ports. If the implementation model does not meet one or more of the linking requirements, the software opens the AUTOSAR Model Linker app, which offers fixes for the unmet requirements. For more information, see “Link to Implementation Model” on page 8-30.

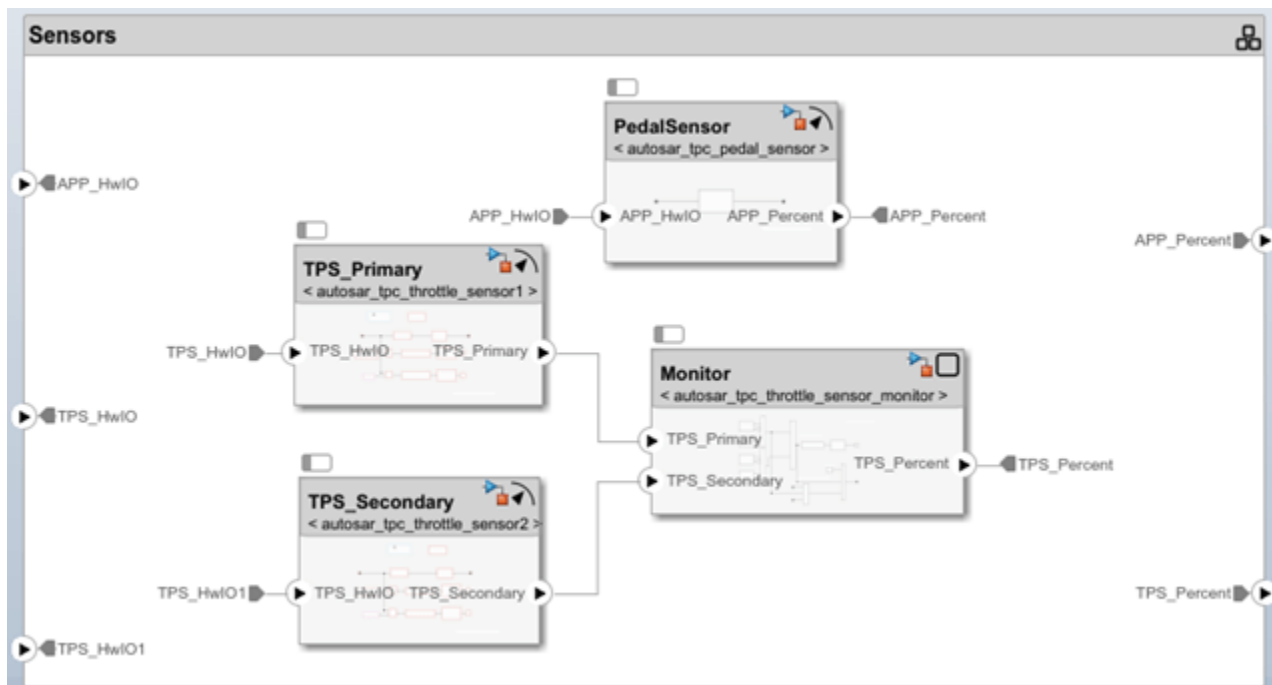
The implementation models provided for this example meet the linking requirements.

4. After you link each model, you can resize the associated component block to better display the component ports.



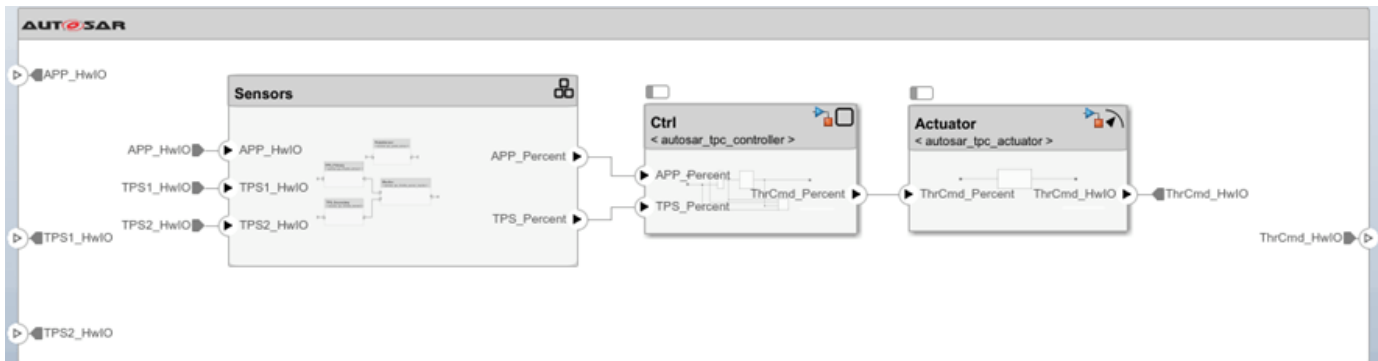
5. Connect the components to each other and to composition root ports.

- To interconnect components, drag a line from a component provider port to another component receiver port.
- To connect components to Sensors composition root ports, drag from a component port to the Sensors composition boundary.



Optionally, to exactly match the root port naming in example model `autosar_tpc_composition`, rename ports `TPS_HwIO` and `TPS_HwIO1` to `TPS1_HwIO` and `TPS2_HwIO`.

6. Return to the top level of the architecture model. To complete the application, add two Classic Component blocks and name them `Ctrl` and `Actuator`. Link the AUTOSAR components to their Simulink implementation models, `autosar_tpc_controller.slx` and `autosar_tpc_actuator.slx`. Connect the `Sensors` composition, `Ctrl` component, and `Actuator` component to each other and to the architecture model boundary.



7. To check for interface or data type issues, update the architecture model. On the **Modeling** tab, select **Update Model**. If issues are found, compare your model with example model `autosar_tpc_composition.slx`.

8. Save the model with a unique name, such as `myTPC_Composition.slx`.

Optional: Create Architecture Views for Analysis

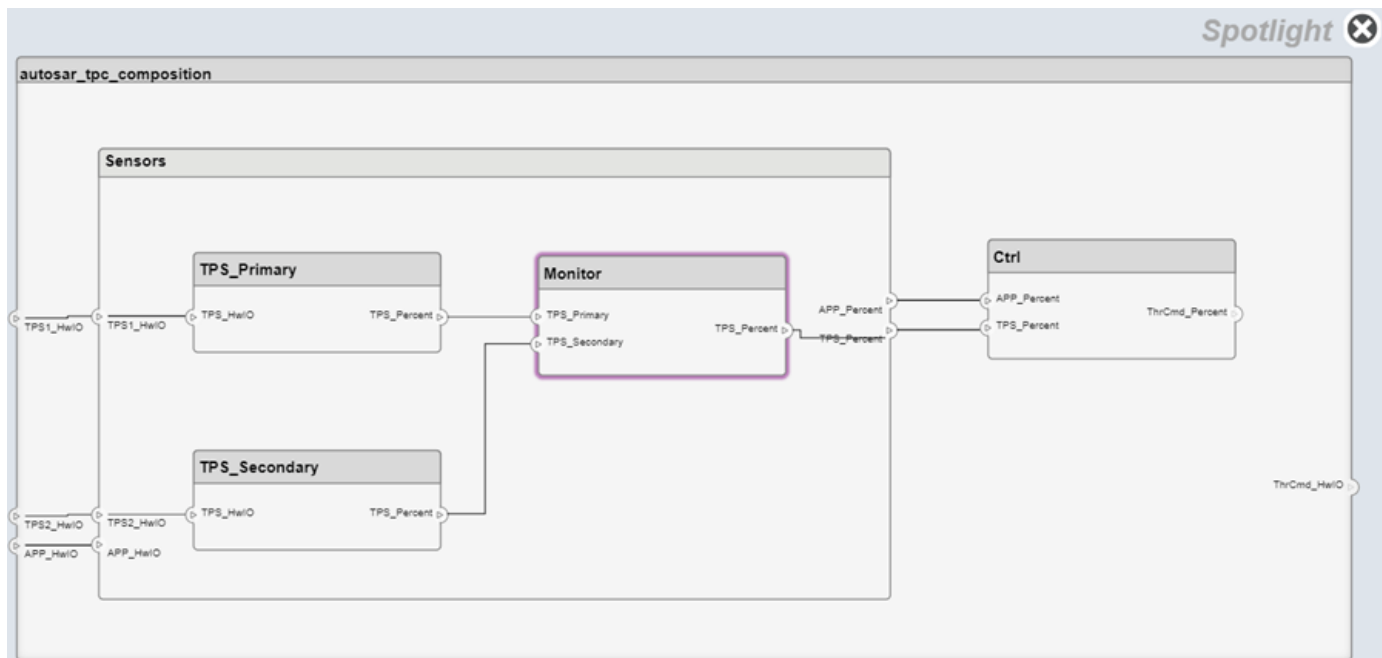
To help analyze structural and functional aspects of an AUTOSAR architecture model, you can create a filtered view of the model hierarchy. On the **Modeling** tab, in the **Architecture Views** menu:

- Select **Spotlight** to create a spotlight view.
- Select **Architecture Views** to create a custom view with grouping criteria.

To help analyze component or composition dependencies, create a spotlight view. A spotlight view is a simplified view of an architecture component or composition that captures its upstream and downstream dependencies.

For this example, select the component `Monitor`, either in the example model `autosar_tpc_composition` or in the architecture model that you created and saved. On the **Modeling** tab, select **Architecture Views > Spotlight**.

The spotlight view opens and shows the model elements to which the component or composition connects in a hierarchy. The spotlight diagram is laid out automatically and cannot be edited.



Optionally, you can create spotlight views in separate, persistent model windows. Updating the architecture model diagram with changes refreshes open spotlight views. While in spotlight view, you can move the spotlight focus.

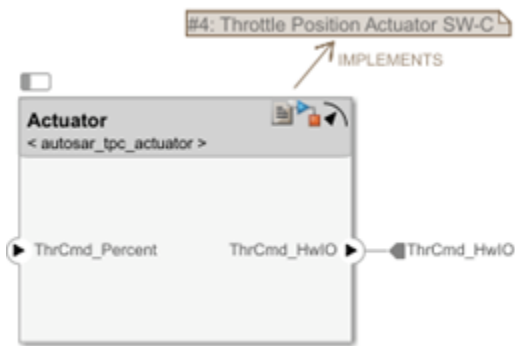
To create a custom view with more sophisticated filtering conditions, use the Architecture Views Gallery. On the **Modeling** tab, select **Architecture Views**. Custom views can be saved with the architecture model, then accessed and shared by collaborating users. For more information, see “Create Profiles Stereotypes and Views for AUTOSAR Architecture Analysis” on page 8-20.

Optional: Link Components to Requirements (Requirements Toolbox)

If you have Requirements Toolbox software, you can link components in the architecture model to requirements. The example folder provides sample requirements file `TPC_Requirements.slreqx`. The file contains requirements for four of the throttle position control application components.

To link a component to a requirement:

1. Open the **Requirements Manager** app. In the architecture model window, the **Requirements** tab opens, with the Requirements Browser docked at the bottom.
2. In the Requirements Browser, open requirements set `TPC_Requirements.slreqx`. The requirements set contains requirements for four components in the model.
3. To link an AUTOSAR component to a requirement, drag the requirement from the Requirements Browser to the component block. For example, drag requirement 4 to the `Actuator` component block.



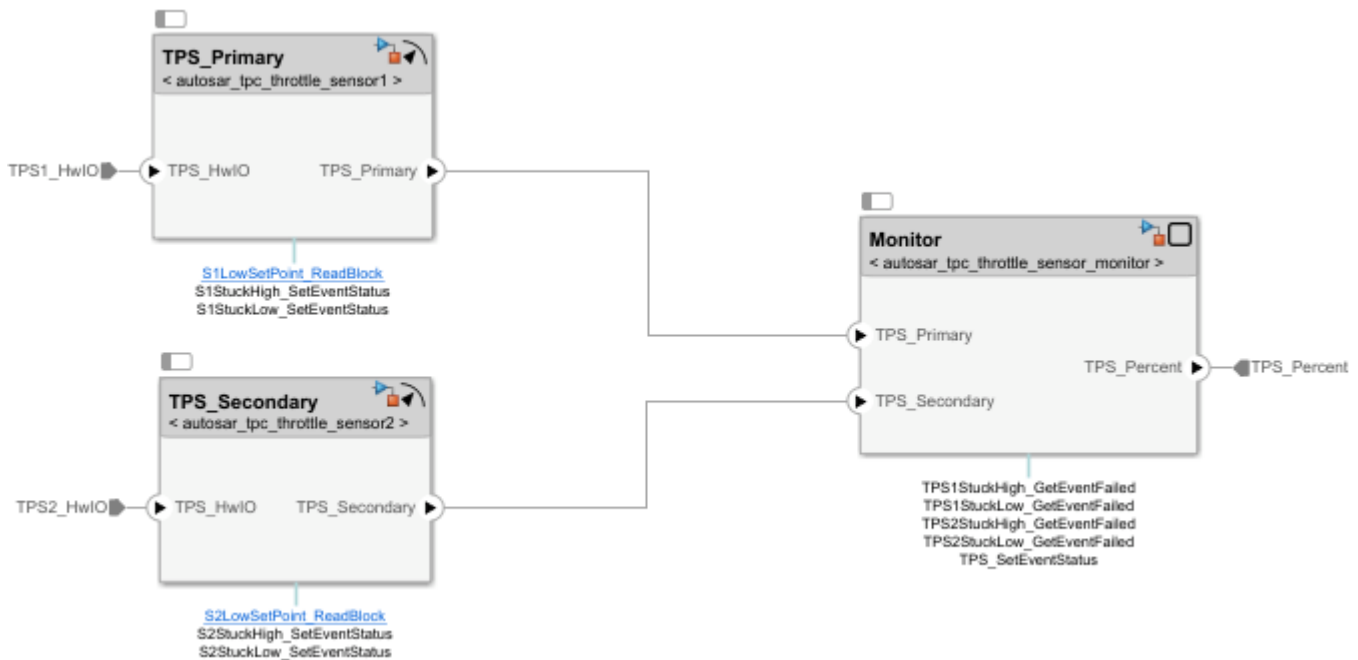
For more information, see “Link AUTOSAR Components to Requirements” on page 8-25.

Configure and Run Simulation

To simulate the behavior of the aggregated components in an AUTOSAR architecture model, click **Run**.

If you try to run the classic architecture model constructed in this example, an error message reports that a function definition was not found for a Basic Software (BSW) function caller block. Three of the component implementation models contain BSW function calls that require BSW service implementations.

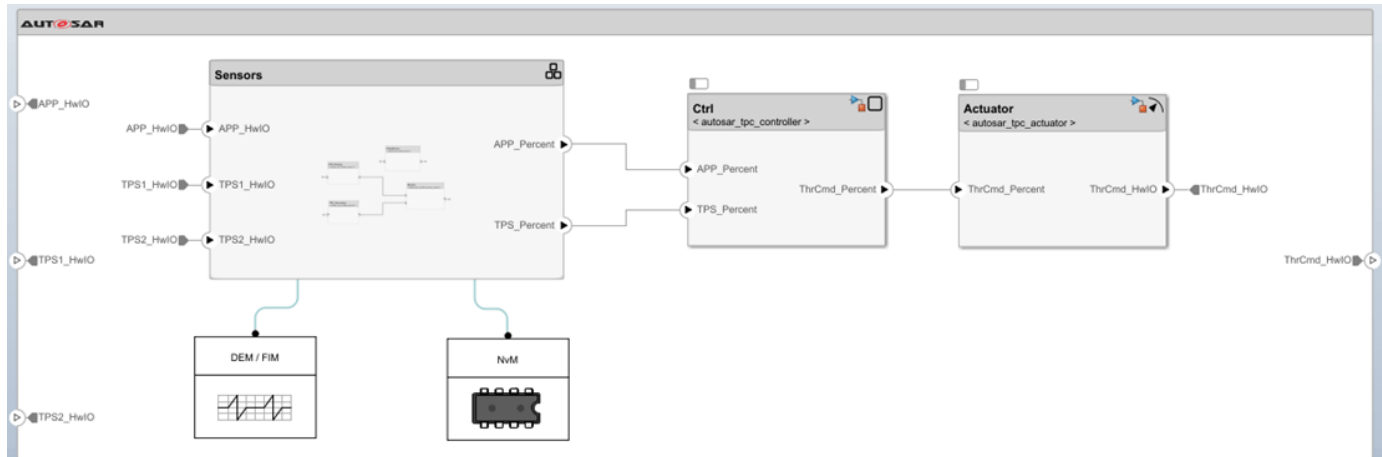
To view those function calls, open your architecture model, for example, `myTPC_Composition.slx`. On the **Debug** tab, select **Information Overlays > Function Connectors**. This selection lists function connectors for each model with functions. To see the models with BSW function calls, open the Sensors composition.



The models contain function calls to Diagnostic Event Manager (Dem) and NVRAM Manager (NvM) services. Before the application can be simulated, you must add Diagnostic Service Component and NVRAM Service Component blocks to the top model.

To add and configure the service implementation blocks:

1. Return to the top level of the architecture model and select the **Modeling** tab. Select and place an instance of **Diagnostic Service Component** and an instance of **NVRAM Service Component**. To wire the function callers to the BSW service implementations, update the model.



2. Check the mapping of the BSW function-caller client ports to BSW service IDs. Dem client ports map to Dem service event IDs and NvM client ports map to NvM service block IDs.

For this example, update the Dem mapping. Open the DEM/FIM block dialog box, select the **RTE** tab, and enter the event ID values shown. Click **OK**. For more information about BSW ID mapping, see “Simulate AUTOSAR Basic Software Services and Run-Time Environment” on page 7-36.

Diagnostic Service Component

Configure AUTOSAR Diagnostic Services and Runtime Environment (RTE) for emulation.

RTE Dem FIM

Update diagram to refresh table.

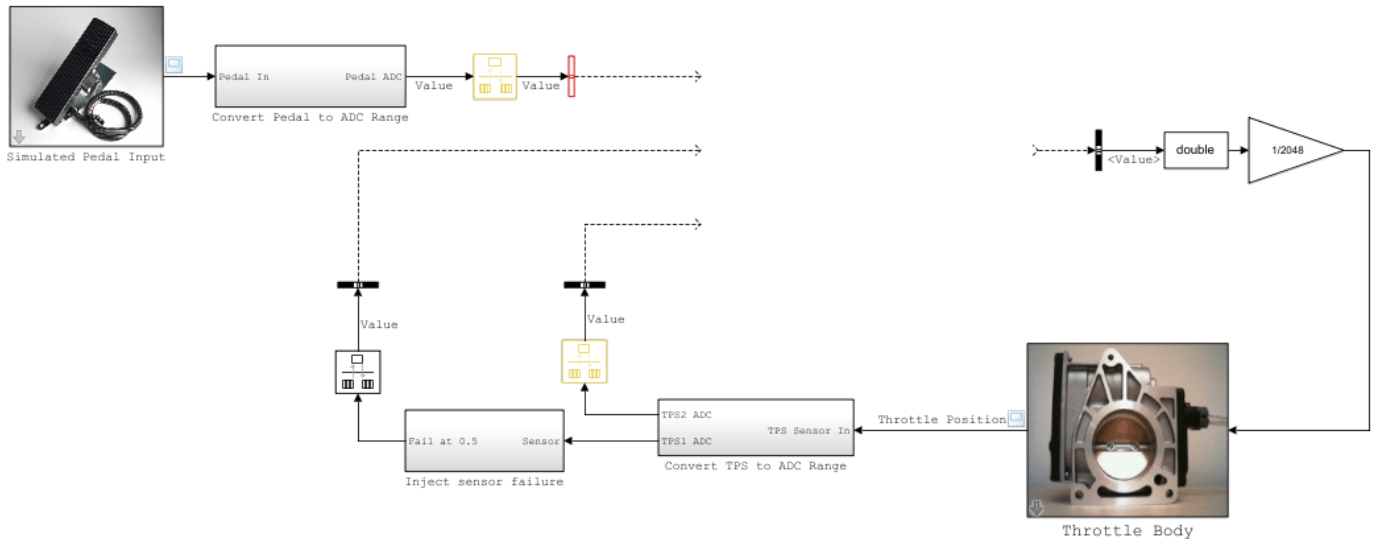
Filter contents

Client Port	ID	ID Type
S1StuckHigh	1	EventId
S1StuckLow	2	EventId
S2StuckHigh	3	EventId
S2StuckLow	4	EventId
TPS	5	EventId
TPS1StuckHigh	1	EventId
TPS1StuckLow	2	EventId
TPS2StuckHigh	3	EventId
TPS2StuckLow	4	EventId

The architecture model is now ready to be simulated. Click **Run**.

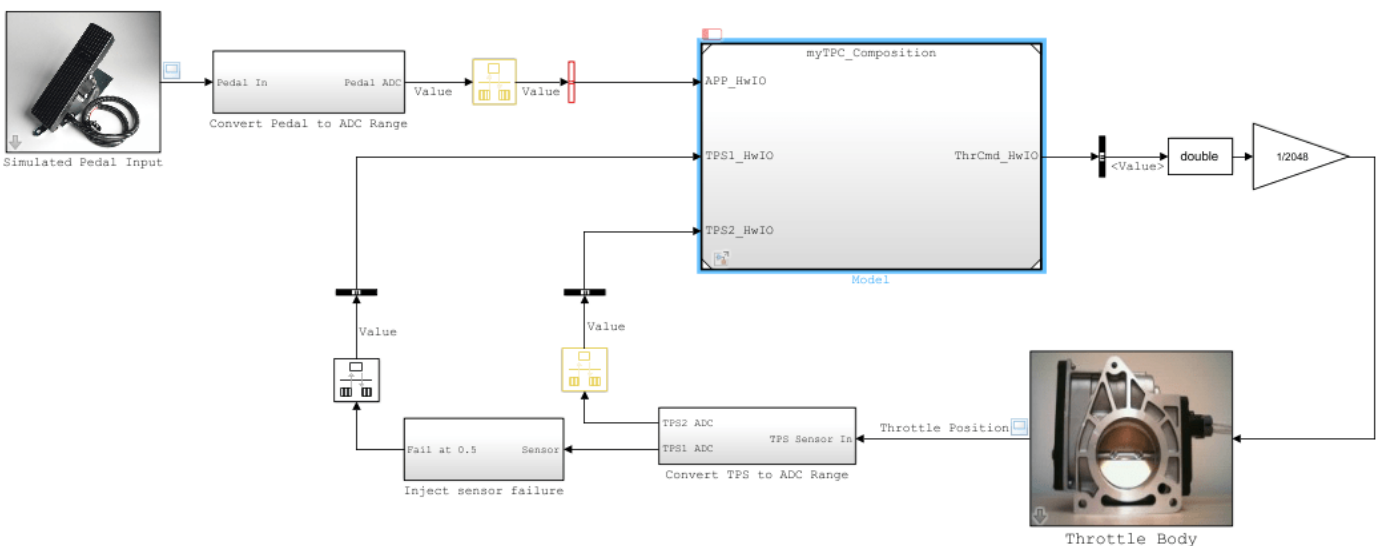
Connect Architecture Model to Test Harness Containing Plant Model and Pedal Input

To provide simulated pedal input to the throttle position control simulation, you can place the architecture model in a test harness model. The test harness can provide a plant model with a pedal input block. Refer to example test-harness model `autosar_tpc_system.slx`.



To connect the architecture model to the test harness:

1. Insert a Model block.
2. Configure the Model block to reference your architecture model, for example, `myTPC_Composition.slx`.
3. In the Model block dialog box, select the option **Schedule rates**. For the associated parameter **Schedule rates with**, select `Schedule Editor`. The throttle position control components have explicit partitions that you can schedule with the Schedule Editor.
4. Connect the architecture model ports to the test harness signals.



The test harness model is now ready to be simulated. Click **Run**. When you simulate the application, the throttle position scope indicates how well the throttle-position control algorithms in the architecture model are tracking the accelerator pedal input.

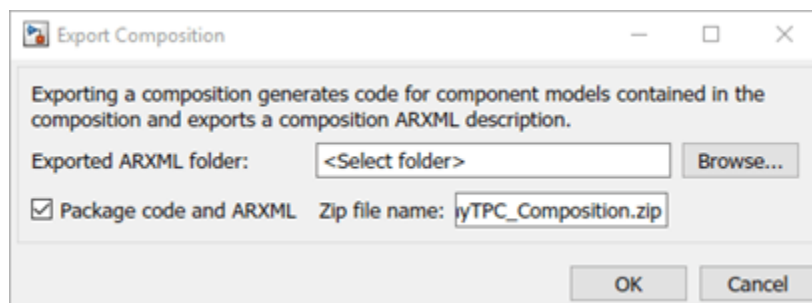
In a test harness model, from the Model block for a referenced AUTOSAR architecture model, you can use the Schedule Editor to schedule rates for component runnables. To open the Schedule Editor, click the Schedule Editor badge immediately above the Model block. In the Schedule Editor display, you can visualize and control the order of execution of the runnables (partitions) in the application components. For more information, see “Using the Schedule Editor”, “Configure AUTOSAR Runnable Execution Order” on page 4-181, and “Configure AUTOSAR Scheduling and Simulation” on page 8-38.

Generate and Package Composition ARXML Descriptions and Component Code (Embedded Coder)

If you have Simulink Coder and Embedded Coder software, you can export composition and component AUTOSAR XML (ARXML) descriptions and generate component code from an AUTOSAR architecture model. Optionally, create a ZIP file to package build artifacts for the model hierarchy, for example, for relocation and integration.

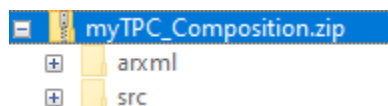
To export ARXML files and generate code:

1. Open the architecture model constructed in this example or open example model `autosar_tpc_composition.slx`.
2. To prepare for exporting ARXML, examine and modify XML options. On the **Modeling** tab, select **Export > Configure XML Options**. The AUTOSAR Dictionary opens in the XML Options view. XML options specified at the architecture model level are inherited during export by each component in the model.
3. To generate and package code for the throttle position control application, on the **Modeling** tab, select **Export > Generate Code and ARXML**. In the Export Composition dialog box, specify the name of the ZIP file in which to package the generated files. To begin the export, click **OK**.

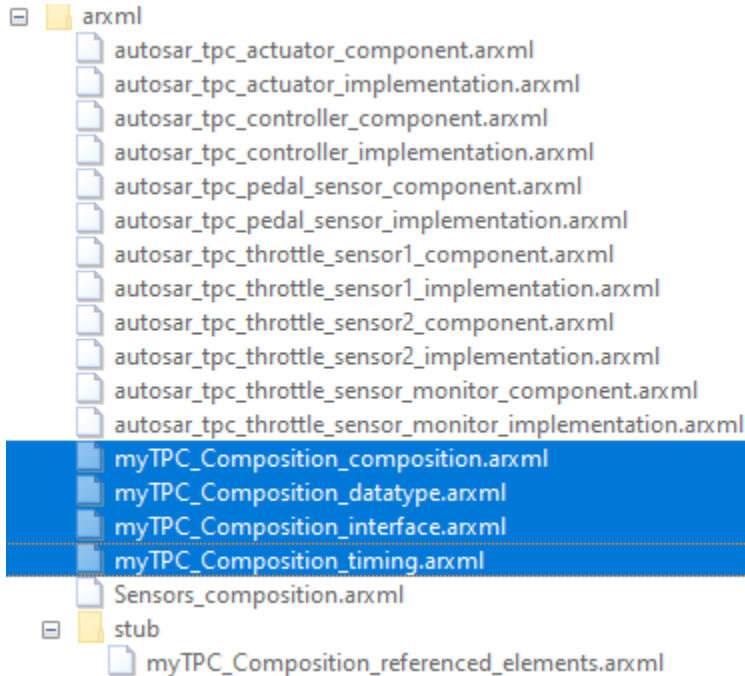


As the architecture model builds, you can view the build log in the Diagnostic Viewer. First the component models build, each as a standalone top-model build. Finally, composition ARXML is exported. When the build is complete, the current folder contains build folders for the architecture model and each component model in the hierarchy, and the specified ZIP file.

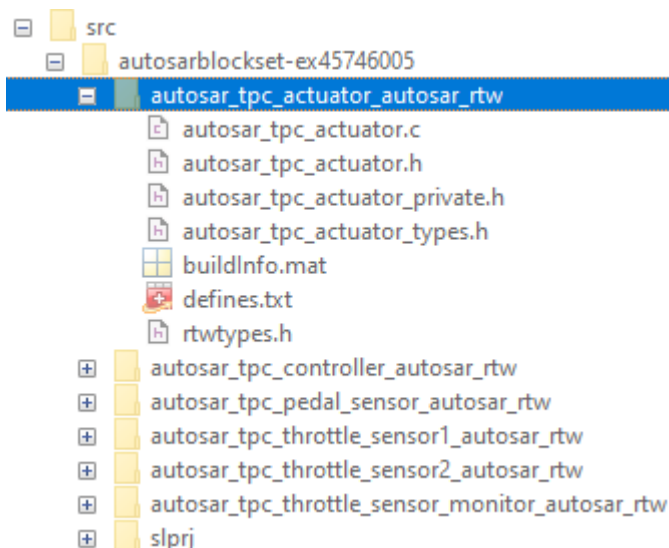
4. Expand the ZIP file. Its content is organized in `arxml` and `src` folders.



5. Examine the arxml folder. In this example, because XML Option **Exported XML File Packaging** is set to **Modular**, XML is exported into multiple files, named according to the type of information contained. Each AUTOSAR component has component and implementation description files, while the architecture model has composition, datatype, interface, and timing description files. The composition file includes XML descriptions of the composition, component prototypes, and composition ports and connectors. The datatype, interface, and timing files aggregate elements from the entire architecture model hierarchy. Nonfunctional properties captured in stereotypes and profiles are not included in the description files.



6. Examine the src folder. Each component model has a build folder that contains artifacts from a standalone model build.



Related Links

- “Create AUTOSAR Architecture Models” on page 8-2
- “Add and Connect AUTOSAR Classic Components and Compositions” on page 8-4
- “Create Profiles Stereotypes and Views for AUTOSAR Architecture Analysis” on page 8-20
- “Link AUTOSAR Components to Requirements” on page 8-25
- “Define AUTOSAR Component Behavior by Creating or Linking Models” on page 8-27
- “Configure AUTOSAR Scheduling and Simulation” on page 8-38
- “Generate and Package AUTOSAR Composition XML Descriptions and Component Code” on page 8-43

Import AUTOSAR Composition into Architecture Model

Import ARXML description of AUTOSAR software composition into architecture model.

After you create an AUTOSAR architecture model (requires System Composer), develop the top-level AUTOSAR software design. The composition editor provides a view of AUTOSAR software architecture based on the AUTOSAR Virtual Function Bus (VFB).

Import AUTOSAR Composition from ARXML File

If you have an ARXML description of an AUTOSAR software composition, you can import the composition into an AUTOSAR architecture model. The import creates a Simulink representation of the composition at the top level of the architecture model. Composition import requires an open AUTOSAR architecture model with no functional content.

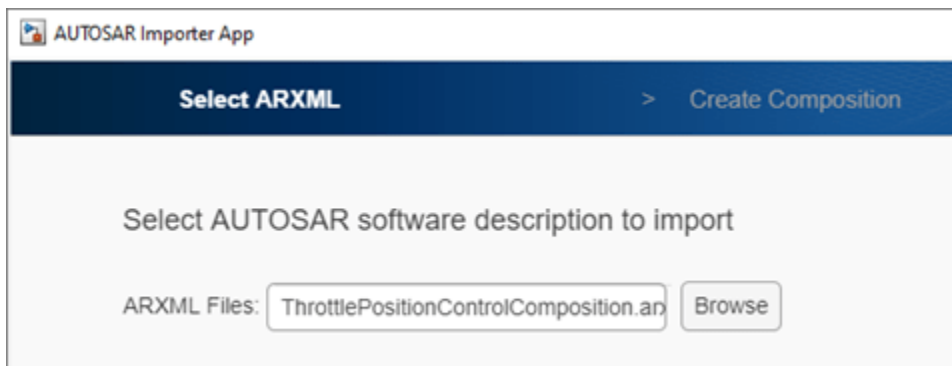
To import an AUTOSAR software composition from ARXML files into an architecture model:

1. Create or open an AUTOSAR architecture model that has no functional content. For example, enter this MATLAB® command.

```
% Create AUTOSAR architecture model
modelName = "myArchModel";
archModel = autosar.arch.createModel(modelName);
```

2. In the open architecture model, on the **Modeling** tab, in the **Component** menu, select **Import from ARXML**.

3. In the AUTOSAR Importer app, in the **Select ARXML** pane, in the **ARXML Files** field, enter the names of one or more ARXML files (comma separated) that describe an AUTOSAR software composition. For this example, enter `ThrottlePositionControlComposition.arxml`.



Click **Next**. The app parses the specified ARXML file.

4. In the **Create Composition** pane, the **Composition name** menu lists the compositions found in the parsed ARXML file. Select the composition `/Company/Components/ThrottlePositionControlComposition`.

Optionally, to view additional modeling options for composition creation, select **Configure Modeling Options**.

You can specify:

- Whether to include or exclude AUTOSAR software components, which define composition behavior. By default, the import includes components within the composition.
- Simulink data dictionary in which to place data objects for imported AUTOSAR data types.
- Names of existing Simulink behavior models to link to imported AUTOSAR software components.
- Component options to apply when creating Simulink behavior models for imported AUTOSAR software components. For example, how to model periodic runnables, or a `PredefinedVariant` or `SwSystemconstantValueSets` with which to resolve component variation points.

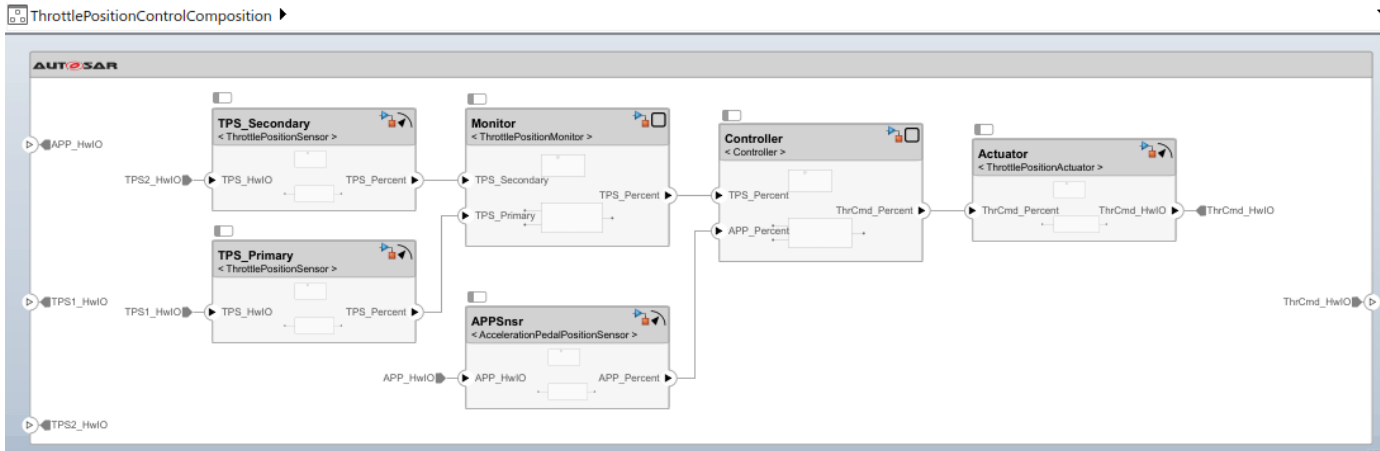
For more information about modeling options and behavior, see the `importFromARXML` reference page.

5. To finish importing the composition into the architecture model, click **Finish**. The Diagnostic Viewer displays the progress of the composition creation. On completion, the imported composition appears in the software architecture canvas.

To perform steps 2 through 5 programmatically, run these commands.

```
% Import composition from file ThrottlePositionControlComposition.arxml
importerObj = arxml.importer("ThrottlePositionControlComposition.arxml"); % Parse ARXML
importFromARXML(archModel,importerObj,...
    "/Company/Components/ThrottlePositionControlComposition");
```

```
Created model 'ThrottlePositionSensor' for component 1 of 5: /Company/Components/ThrottlePositionControlComposition
Created model 'ThrottlePositionMonitor' for component 2 of 5: /Company/Components/ThrottlePositionControlComposition
Created model 'Controller' for component 3 of 5: /Company/Components/ThrottlePositionControlComposition
Created model 'AccelerationPedalPositionSensor' for component 4 of 5: /Company/Components/ThrottlePositionControlComposition
Created model 'ThrottlePositionActuator' for component 5 of 5: /Company/Components/ThrottlePositionControlComposition
Importing composition 1 of 1: /Company/Components/ThrottlePositionControlComposition
```



Because this composition import was configured to include AUTOSAR software components (modeling option **Exclude internal behavior from import** was cleared), the import created Simulink models for each component in the composition.

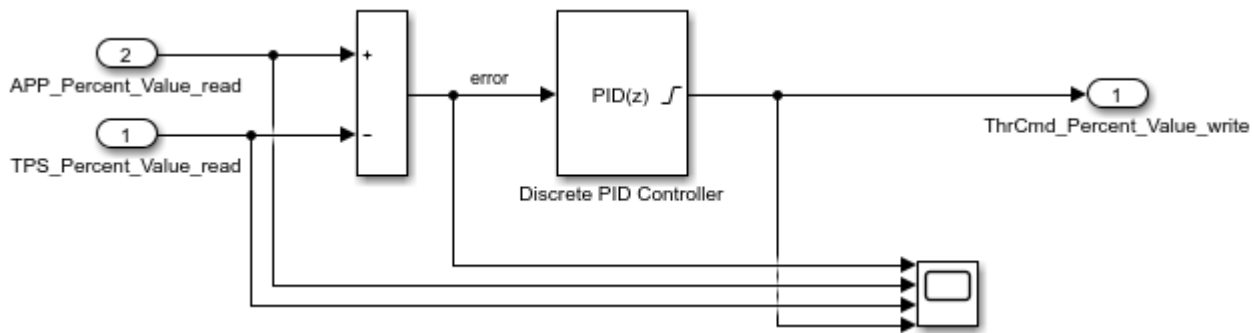
Develop AUTOSAR Component Algorithms

After creating an initial Simulink representation of the AUTOSAR composition, you develop each component in the composition. For each component, you refine the AUTOSAR configuration and create algorithmic model content.

For example, the *Controller* component model in the *ThrottlePositionControlComposition* composition model contains an atomic subsystem *Runnable_Step_sys*, which represents an AUTOSAR periodic runnable. The *Runnable_Step_sys* subsystem contains the initial stub implementation of the controller behavior.



Here is a possible implementation of the throttle position controller behavior. (To explore this implementation, see the model *autosar_swc_controller*, which is provided with the example “Design and Simulate AUTOSAR Components and Generate Code” on page 4-77.) The component takes as inputs an APP sensor percent value from a pedal position sensor and a TPS percent value from a throttle position sensor. Based on these values, the controller calculates the *error*. The error is the difference between where the operator wants the throttle, based on the pedal sensor, and the current throttle position. In this implementation, a Discrete PID Controller block uses the error value to calculate a throttle command percent value to provide to a throttle actuator. A scope displays the error value and the Discrete PID Controller block output value over time.



As you develop AUTOSAR components, you can:

- Simulate component models individually or as a group within the architecture model.
- Generate ARXML description files and algorithmic C code for testing in Simulink or integration into an AUTOSAR run-time environment. (AUTOSAR code generation requires Simulink Coder and Embedded Coder.)

For more information on developing, simulating, and building AUTOSAR components, see example “Design and Simulate AUTOSAR Components and Generate Code” on page 4-77.

Related Links

- `importFromARXML`
- “Import AUTOSAR Composition from ARXML” on page 8-16
- “Author AUTOSAR Classic Compositions and Components in Architecture Model” on page 8-50
- “Design and Simulate AUTOSAR Components and Generate Code” on page 4-77

Configure AUTOSAR Architecture Model Programmatically

An AUTOSAR architecture model provides resources and a canvas for developing AUTOSAR composition and component models. You develop the software architecture by using graphical user interfaces, equivalent architecture modeling functions, or both. AUTOSAR Blockset provides functions for these architecture related tasks.

Tasks	Functions
Create, load, open, save, or close an AUTOSAR architecture model	<code>autosar.arch.createModel</code> <code>autosar.arch.loadModel</code> <code>close</code> <code>open</code> <code>save</code>
Specify the platform kind of an AUTOSAR architecture model	<code>setPlatform</code>
Add, connect, or remove AUTOSAR components, composition, and ports	<code>addComponent</code> <code>addComposition</code> <code>addPort</code> <code>connect</code> <code>destroy</code> <code>importFromARXML</code> <code>layout</code>
Find AUTOSAR elements and modify properties	<code>find</code> <code>get</code> <code>set</code>
Define component behavior by creating or linking Simulink models	<code>createModel</code> <code>linkToModel</code>
Add Basic Software (BSW) service component blocks for simulating BSW service calls	<code>addBSWService</code>
Export composition and component ARXML descriptions and generate component code (requires Embedded Coder®)	<code>export</code> <code>getXmlOptions</code> <code>setXmlOptions</code>

Programmatically Create and Configure Architecture Model

This example script shows:

- 1 Creates and opens an AUTOSAR architecture model.
- 2 Sets the platform kind of the architecture model to the Classic Platform explicitly.
- 3 Adds a composition and components.
- 4 Adds architecture, composition, and component ports.
- 5 Connects architecture, composition, and component ports.
- 6 Creates and links Simulink® implementation models for components.
- 7 Arranges architecture model layout based on heuristics.
- 8 Sets component and port properties.

- 9 Removes a component.
- 10 Searches for elements at different levels of the architecture model hierarchy.
- 11 Lists property values for composition ports.

To run the script, copy commands from the MATLAB® script listing below to the MATLAB® command window.

```
% configAutosarArchModel.m
%
% Configure AUTOSAR architecture model.
% This script creates models Controller1.slx and Actuator.slx.
% To rerun the script, remove the models from the working folder.

% Create and open AUTOSAR architecture model
modelName = 'myArchModel';
archModel = autosar.arch.createModel(modelName);

% autosar.arch.createModel creates a classic architecture model by default
% Use setPlatform to explicitly set the platform kind
setPlatform(archModel, 'Classic');

% Add a composition
composition = addComposition(archModel, 'Sensors');

% Add 2 components inside Sensors composition
names = {'PedalSnsr', 'ThrottleSnsr'};
sensorSWCs = addComponent(composition, names, 'Kind', 'SensorActuator');
layout(composition); % Auto-arrange composition layout

% Add components at architecture model top level
controller = addComponent(archModel, 'Controller');
actuator = addComponent(archModel, 'Actuator');
set(actuator, 'Kind', 'SensorActuator');
layout(archModel);

% Add ports to architecture model and the Sensors composition
addPort(archModel, 'Receiver', {'APP_HwIO', 'TPS_HwIO'});
addPort(archModel, 'Sender', 'ThrCmd_HwIO');
addPort(composition, 'Receiver', {'TPS_HwIO', 'APP_HwIO'});
addPort(composition, 'Sender', {'APP_Percent', 'TPS_Percent'});

% Link components to implementation models
% Add path to implementation model
pedalSnsr = find(composition, 'Component', 'Name', 'PedalSnsr');
linkToModel(pedalSnsr, 'autosar_tpc_pedal_sensor');
throttleSnsr = find(composition, 'Component', 'Name', 'ThrottleSnsr');
%linkToModel(throttleSnsr, 'autosar_tpc_throttle_sensor1');
linkToModel(actuator, 'autosar_tpc_actuator');
linkToModel(controller, 'autosar_tpc_controller')

% add ports to throttle sensor component and create behavior model
addPort(throttleSnsr, 'Sender', 'TPS_Percent');
addPort(throttleSnsr, 'Receiver', 'TPS_HwIO');
createModel(throttleSnsr);

% implement internal behavior for throttle sensor. Here, we simply adjust
```

```

% datatypes on the ports.
set_param('ThrottleSnr/In Bus Element', 'OutDataTypeStr', 'uint16');
set_param('ThrottleSnr/Out Bus Element', 'OutDataTypeStr', 'single');

% connect composition and components based on matching port names
connect(archModel,composition,controller);
connect(archModel,controller,actuator);
connect(archModel,[],composition);
connect(archModel,actuator,[]);
connect(composition, [], pedalSnr);
connect(composition, [], throttleSnr);
connect(composition, pedalSnr, []);
connect(composition, throttleSnr, []);
connect(archModel,composition, controller);

% can also use API to connect specific ports
ThrCmd_Percent_pport = find(controller, 'Port', 'Name', 'ThrCmd_Percent');
ThrCmd_Percent_rport = find(actuator, 'Port', 'Name', 'ThrCmd_Percent');
connect(archModel, ThrCmd_Percent_pport, ThrCmd_Percent_rport);

layout(archModel); % Auto-arrange layout

% Find components in architecture model top level only
components_in_arch_top_level = find(archModel,'Component');
% Find components in all hierarchy
components_in_all_hierarchy = find(archModel,'Component','AllLevels',true);
% Find ports for composition block only
composition_ports = find(composition,'Port');

% List Kind and Name property values for composition ports
for ii=1:length(composition_ports)
    Port = composition_ports(ii);
    portName = get(Port,'Name');
    portKind = get(Port,'Kind');
    fprintf('%s port %s\n',portKind,portName);
end

Receiver port TPS_HwIO
Receiver port APP_HwIO
Sender port APP_Percent
Sender port TPS_Percent

% simulate the architecture model
sim(modelName);

```

See Also

Software Component | Software Composition | Diagnostic Service Component | NVRAM Service Component

Related Examples

- “Create AUTOSAR Architecture Models” on page 8-2
- “Add and Connect AUTOSAR Classic Components and Compositions” on page 8-4
- “Define AUTOSAR Component Behavior by Creating or Linking Models” on page 8-27

- “Configure AUTOSAR Scheduling and Simulation” on page 8-38
- “Generate and Package AUTOSAR Composition XML Descriptions and Component Code” on page 8-43
- “Author AUTOSAR Classic Compositions and Components in Architecture Model” on page 8-50

Manage Shared Interfaces and Data Types for AUTOSAR Architecture Models

Interface dictionaries enable interfaces, data types, and AUTOSAR-specific design data to be authored, managed, and shared between AUTOSAR components and compositions modeled in Simulink. Interface dictionaries provide scalability for system-level and multicomponent designs by containing these shared elements in a central location.

You can programmatically or graphically configure the attributes and contents of an interface dictionary and apply them to an architecture model using this basic workflow:

- 1 Create an interface dictionary.
- 2 Design interface and data types with the interface dictionary API or the standalone Interface Editor.
- 3 Link the interface dictionary to an architecture model.
- 4 Apply the interfaces to the architecture model in the Simulink environment.
- 5 Deploy the interface dictionary shared interface and data type content in the final application.

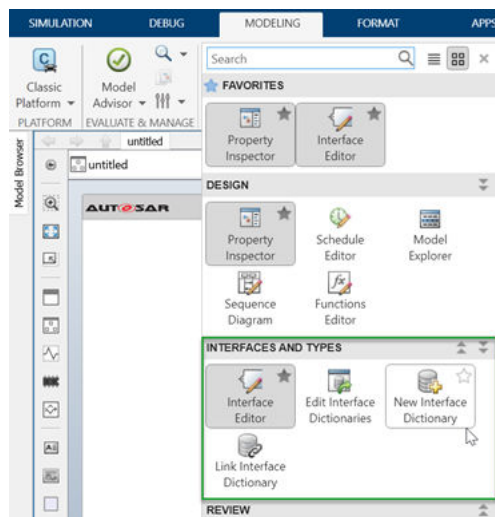
To migrate data stored in the base workspace or in a data dictionary hierarchy to the interface dictionary associated with an architecture model, use the interface dictionary Migrator object.

Create Interface Dictionary

To create an interface dictionary programmatically, use the `Simulink.interface.dictionary.create` function.

```
% create new interface dictionary
dictName = 'MyInterfaces.sldd';
dictAPI = Simulink.interface.dictionary.create(dictName);
```

Alternatively, to create an interface dictionary from the AUTOSAR architecture model toolstrip, on the **Modeling** tab, open the **Design** menu and select **New Interface Dictionary** from the **Interfaces and Types** section.



Design Data Types and Interfaces by Using Interface Dictionary

Once you create your interface dictionary, you can add design data programmatically by using the interface dictionary API or interactively by using the standalone Interface Editor. These tools allow you to author shared elements outside of the context of a particular component or composition and allow multiple team members to share in the definition and management of these elements.

Add Design Data Programmatically

To programmatically create, configure, and manage interfaces and data types in your interface dictionary, use the functions for the `Simulink.interface.Dictionary` object.

Here, use type-specific functions to add alias types, numeric types, value types, structured types, and enumerations to the interface dictionary.

```
% add DataTypes
%% AliasTypes
myAliasType1 = dictAPI.addAliasType('aliasType', BaseType='single');
myAliasType1.Name = 'myAliasType1';
myAliasType1.BaseType = 'fixdt(1,32,16)';

myAliasType2 = dictAPI.addAliasType('myAliasType2');
% can also use interface dict type objs
myAliasType2.BaseType = myAliasType1;

%% EnumType
myEnumType1 = dictAPI.addEnumType('myColor');
myEnumType1.addEnumeral('RED', '0', 'RED BLOOD');
myEnumType1.addEnumeral('BLUE', '1', 'Blue Skies');
myEnumType1.DefaultValue = 'BLUE';
myEnumType1.Description = 'I am a Simulink Enumeration';
myEnumType1.StorageType = 'int16';

% set base type of an alias type to be this enum object
myAliasType3 = dictAPI.addAliasType('myAliasType3');
myAliasType3.BaseType = myEnumType1;

%% NumericType
myNumericType1 = addNumericType(dictAPI, 'myNumericType1');
myNumericType1.DataTypeMode = "Single";

%% ValueType
myValueType1 = dictAPI.addValueType('myValueType1');
myValueType1.DataType = 'int32';
myValueType1.Dimensions = '[2 3]';
myValueType1.Description = 'I am a Simulink ValueType';
myValueType1.DataType = myEnumType1; % can also use interface dict type objs

%% StructType
myStructType1 = dictAPI.addStructType('myStructType1');
structElement1 = myStructType1.addElement('Element1');
structElement1.Type.DataType = 'single';
structElement1.Type.Dimensions = '3';
structElement2 = myStructType1.addElement('Element2');
structElement2.Type = myValueType1;
% or
structElement2.Type = 'ValueType: myValueType1';

%% Nested StructType
myStructType2 = dictAPI.addStructType('myStructType2');
myStructType2.Description = 'I am a nested structure';
structElement = myStructType2.addElement('Element');
structElement.Dimensions = '5';
structElement.Type = myStructType1;
% or
structElement.Type = 'Bus: myStructType1';
```

Add communication interfaces and their data elements.

```
nvInterface1 = dictAPI.addDataInterface('NV1');

dataElm1 = nvInterface1.addElement('DE1');
dataElm1.Type = myValueType1;
```

```

dataElm2 = nvInterface1.addElement('DE2');
dataElm2.Type = myStructType2;
dataElm2.Dimensions = '4';
dataElm2.Description = 'I am a data element with datatype = array of struct type';

% data element with owned type
dataElm3 = nvInterface1.addElement('DE3');
dataElm3.Type.DataType = 'single';
dataElm3.Type.Dimensions = '10';
dataElm3.Type.Minimum = '-5';

srInterface2 = dictAPI.addDataInterface('SRI');

```

Then, create, configure, and manage platform-specific properties.

```

% now add AUTOSARClassic mapping
platformMapping = dictAPI.addPlatformMapping('AUTOSARClassic');

% set platform properties
platformMapping.setPlatformProperty(nvInterface1,...
    'Package', '/Interface2', 'InterfaceKind', 'NvDataInterface');

% get the platform properties
[pNames, pValues] = platformMapping.getPlatformProperties(nvInterface1);

% managing AUTOSAR Classic platform related elements (these don't have mapping to Simulink)
arObj = autosar.api.getAUTOSARProperties(dictName);
arObj.addPackageableElement('SwAddrMethod', '/SwAddressMethods', 'VAR1', 'SectionType', 'Var');

platformMapping.setPlatformProperty(dataElm1,...
    'SwAddrMethod', 'VAR1', 'SwCalibrationAccess', 'ReadWrite', 'DisplayFormat', '%.3f');

```

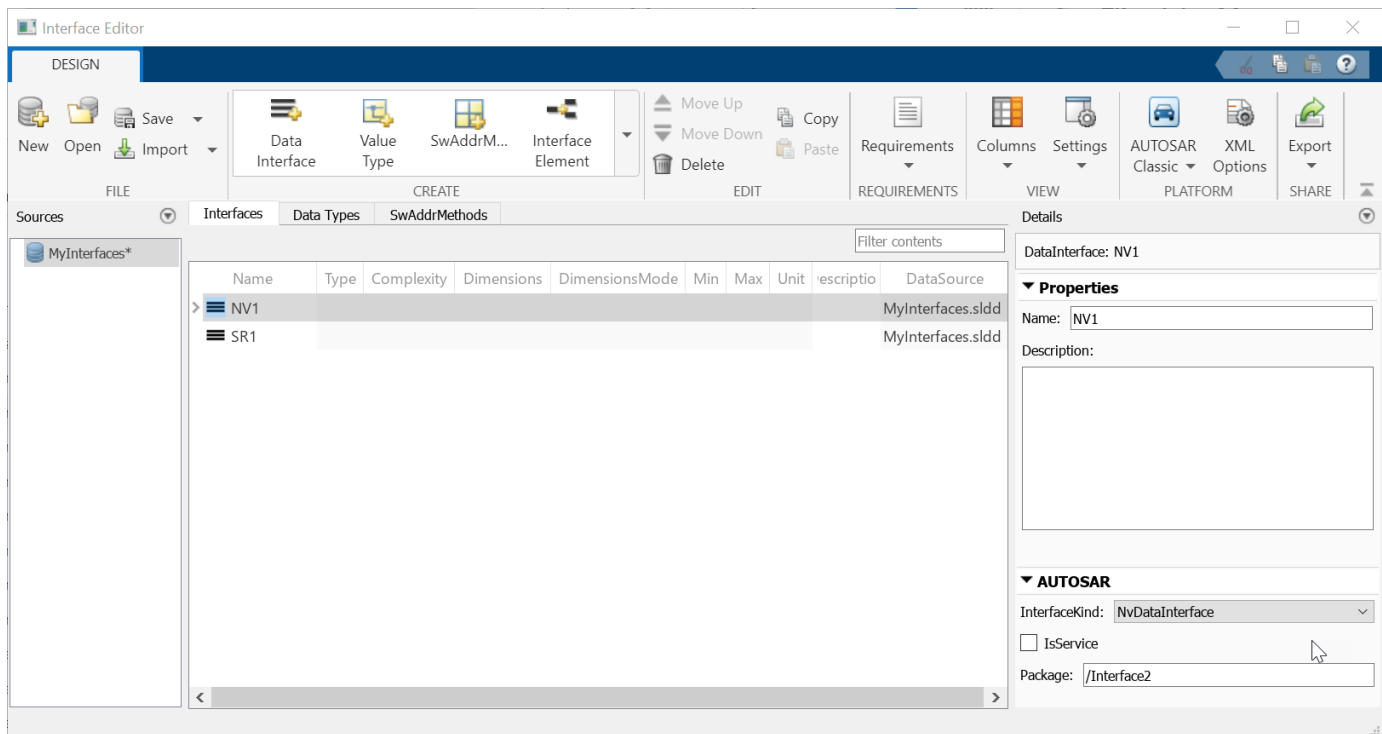
Add Design Data Using Standalone Editor

Alternatively, you can add design data by using the standalone Interface Editor. To open the editor from outside of the context of a model:

- Double-click the `.sldd` file from the MATLAB Current Folder browser.
- Use the show function of the interface dictionary object.
- Enter `interfaceeditor` at the MATLAB command prompt.

To open the standalone editor from a model:

- In the Simulink editor of an AUTOSAR architecture model, on the **Modeling** tab, open the **Design** menu and select **Edit Interface Dictionary**.
- In the Model Explorer, under the **External Data** node for the model, select the interface dictionary, then click **Open Interface Editor** from the Dialog pane.



With the standalone Interface Editor, you can create, configure, and manage design data.

- **Create** — On the toolstrip, in the **Create** section, add data type definitions and interfaces. Data types and interfaces each have a dedicated tab for data management.
- **Configure** — In the right panel, use the **Details** pane to configure your data. The **Details** pane can also display platform-specific properties. For example, when you set the deployment platform to AUTOSAR Classic, **Details** displays AUTOSAR interface communication properties such as **InterfaceKind**, **IsService**, and **Package**. Setting these properties under **Details** sets them in the generated interface dictionary `.sldd` file.
- **Manage** — You can filter, sort, and search data on the **Interfaces** and **Data Types** tabs.

In addition to the **Interfaces** and **Data Types** tabs, the Interface Editor displays platform-specific software address method data for the AUTOSAR Classic Platform in the **SwAddrMethods** tab.

For more information on using the standalone editor, see **Interface Editor**.

Link Interface Dictionary to Architecture Model

Once you have a saved interface dictionary, you can link it to your architecture model. Link a dictionary to a model programmatically as follows.

```
% open a previously defined interface dictionary
dictName = 'MyInterfaces.sldd';
dictAPI = Simulink.interface.dictionary.open(dictName);

% create AUTOSAR arch model and link interface dictionary
archModel = autosar.arch.createModel('myTopArchModel');
archModel.linkDictionary(dictName);
```


Alternatively, to link an existing interface dictionary from the AUTOSAR architecture model toolstrip, on the **Modeling** tab, open the **Design** menu and select **Link Interface Dictionary** from the **Interfaces and Types** section. When you create a component model from an AUTOSAR architecture model, Simulink automatically links it to the interface dictionary.

Use Data Dictionary with Interface Dictionary

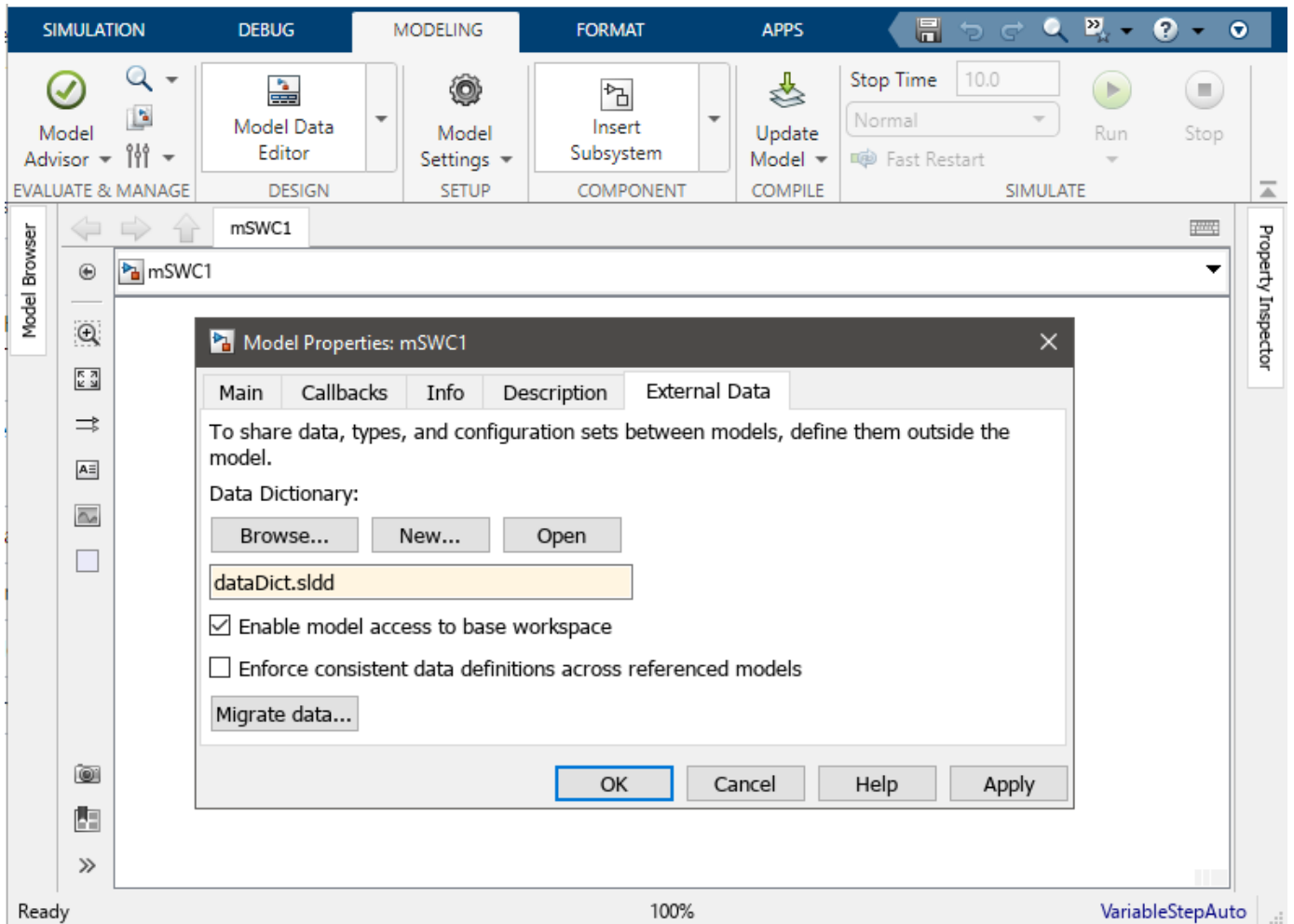
The design also allows regular data dictionaries to coexist with an interface dictionary. This approach allows for proper scoping and encapsulation of the data in your model hierarchy.

- Model workspace — Contains parameters and signal definitions that are scoped to the model.
- Data dictionary — Contains configuration sets, values, and variants that can be shared with other components but should be separate from interface definitions.
- Interface dictionary — Contains interface and data type definitions that can be shared across components.

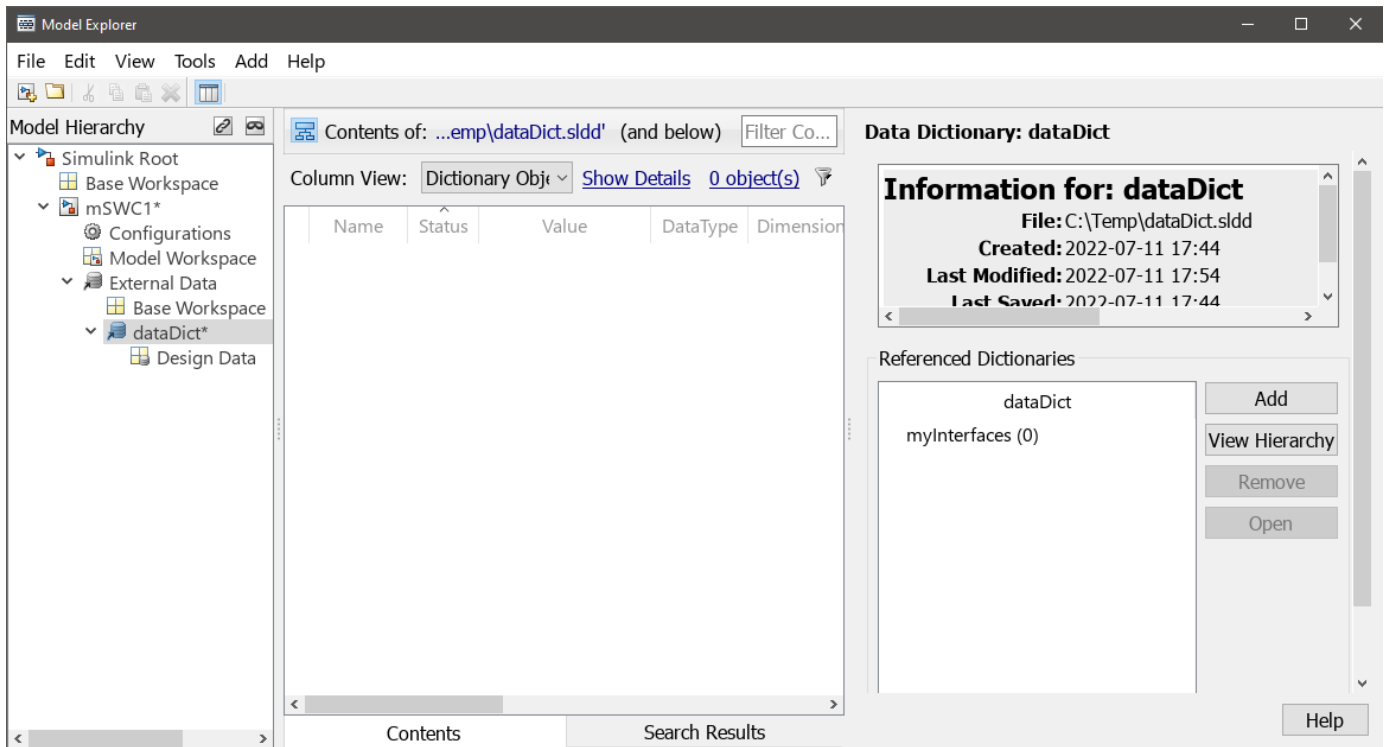


To use both dictionaries, first link a data dictionary to a component model, then reference the interface dictionary from the data dictionary.

- 1 Create an empty model.
- 2 In the Simulink Editor, on the model canvas, right-click and select **Model Properties**.
- 3 In the Model Properties dialog box, create a new data dictionary or link to an existing dictionary.



- 4 Click the model data badge in the bottom left corner of the model, then click the **External Data** link.
- 5 In the Model Explorer **Model Hierarchy** pane, under the **External Data** node, select the node for the data dictionary.
- 6 In the **Dialog** pane, in the **Referenced Dictionaries** section, add your interface dictionary as a referenced dictionary.



Apply Interfaces to Architecture Model in Simulink Environment

Once your interface dictionary is linked to an AUTOSAR architecture model, you can apply the interfaces to your modeled AUTOSAR application programmatically by using the AUTOSAR architecture model API, or by using the Interface Editor in the Simulink editor window.

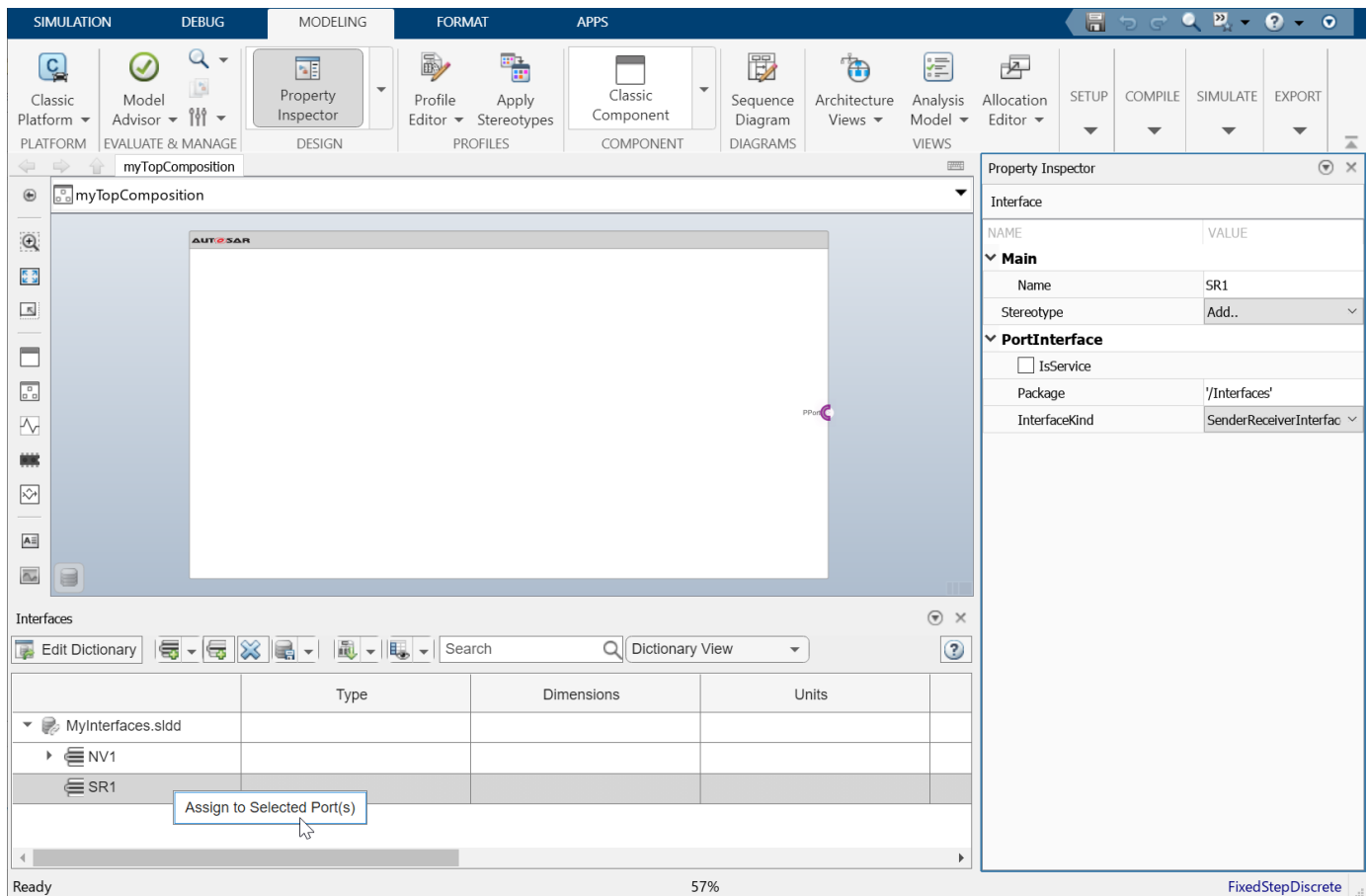
Here, you link a dictionary with an AUTOSAR Classic platform mapping to an architecture model then map a SenderPort to an interface in that dictionary.

```
% open previously created interface dictionary
dictName = 'MyInterfaces.sldd';
dictAPI = Simulink.interface.dictionary.open(dictName);

% link interface dictionary to an AUTOSAR arch model
archModel = autosar.arch.createModel('myTopComposition');
archModel.linkDictionary(dictName);

pport = archModel.addPort("Sender", 'PPort');
pport.setInterface(srInterface2);
```

Alternatively, you can apply the interfaces to your AUTOSAR architecture model by using the Interface Editor or Property Inspector. In the AUTOSAR architecture model toolstrip, on the **Modeling** tab, open the **Design** menu and select **Interface Editor**. The editor opens as a pane in the current Simulink Editor window.

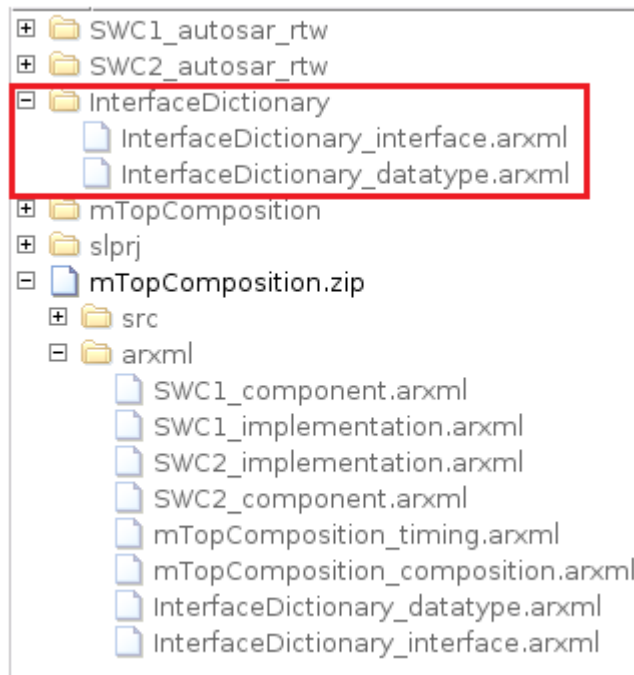


The primary focus of this model-centric editor is applying interfaces to ports. It displays the available interfaces in the linked interface dictionary. By using Interface Editor you can:

- Right-click on an interface to assign the interface to a selected port on the canvas.
- Trace between ports and interfaces
- Focus on a particular interface by using the **Port Interface View**
- Use the Property Inspector to view and configure a selected interface
- Add data interfaces in the AUTOSAR architecture model. These interfaces can be mapped to `SenderReceiverInterface`, `ModeSwitchInterface`, or `NvDataInterface` by using the Property Inspector.

Deploy Interface Dictionary

Finally, to deploy an interface dictionary to a particular platform, you must provide a mapping of the dictionary elements to the platform. When you build an AUTOSAR architecture model, the build process exports an interface dictionary that is linked to the model as ARXML into a folder with the interface dictionary name. This ensures that the interfaces and types defined in the dictionary are included in the ARXML. In addition, the created ZIP file includes the ARXML files that come out of the dictionary.



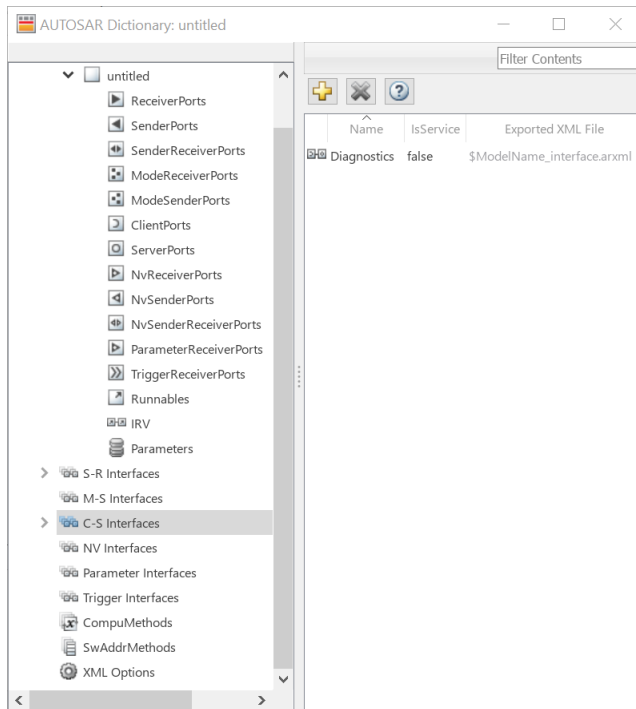
You can also export an interface dictionary independently of an architecture model by using the standalone Interface Editor or the interface dictionary API. In the Interface Editor, in the **Share** section, select **Export > Export to ARXML**. To export programmatically, use the following commands.

```
platformMapping = dictAPI.getPlatformMapping('AUTOSARClassic');
exportDictionary(platformMapping);
```

Limitations

Some limitations for the interface dictionary include:

- An AUTOSAR classic component model cannot reference multiple interface dictionaries.
- Interface dictionary reference hierarchies are not supported for interface dictionaries mapped to the AUTOSAR Classic Platform.
- The editor for the interface dictionary can only view and edit data interfaces. To author and view other kinds of interfaces for AUTOSAR workflows, such as client/server, parameter and trigger interfaces, open the AUTOSAR component dictionary.



- The interface dictionary does not support the import of AUTOSAR information from an ARXML.

See Also

[Simulink.interface.Dictionary](#) | [autosar.dictionary.ARClassicPlatformMapping](#) | [exportDictionary](#) | [getPlatformProperties](#) | [getPlatformProperty](#) | [setPlatformProperty](#) | [Simulink.interface.dictionary.create](#) | [Simulink.interface.dictionary.open](#) | [Migrator](#) | **Interface Editor**

Related Examples

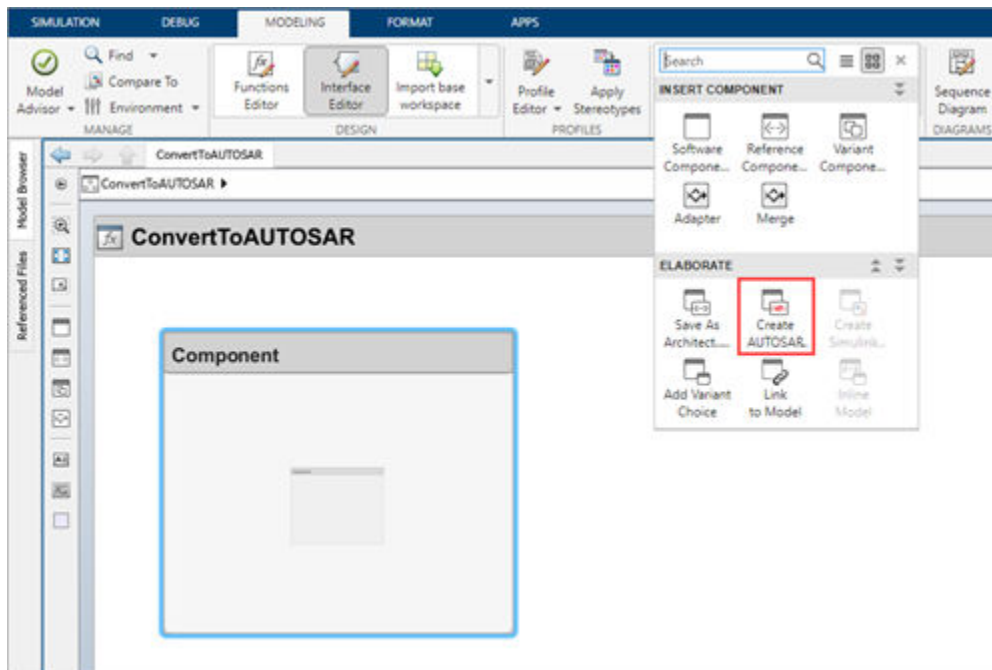
- “Create AUTOSAR Architecture Models” on page 8-2
- “Add and Connect AUTOSAR Classic Components and Compositions” on page 8-4
- “Define AUTOSAR Component Behavior by Creating or Linking Models” on page 8-27
- “Generate and Package AUTOSAR Composition XML Descriptions and Component Code” on page 8-43
- “Author AUTOSAR Classic Compositions and Components in Architecture Model” on page 8-50

Create AUTOSAR Architecture from a Component in System Composer Model

You can convert a System Composer Component (System Composer) to an AUTOSAR architecture model. During conversion, you can select Classic or Adaptive Platform. For classic architecture modeling, data interfaces are supported. For adaptive architecture modeling, data interfaces and service interfaces are supported.

To convert a System Composer component to an AUTOSAR architecture model, use one of these methods:

- In your System Composer architecture model, right-click the component and select **Create AUTOSAR Architecture Model**.
- Select the component. In the toolbar, on the **Modeling** tab, in the **Component** section, click **Create AUTOSAR Architecture Model**.



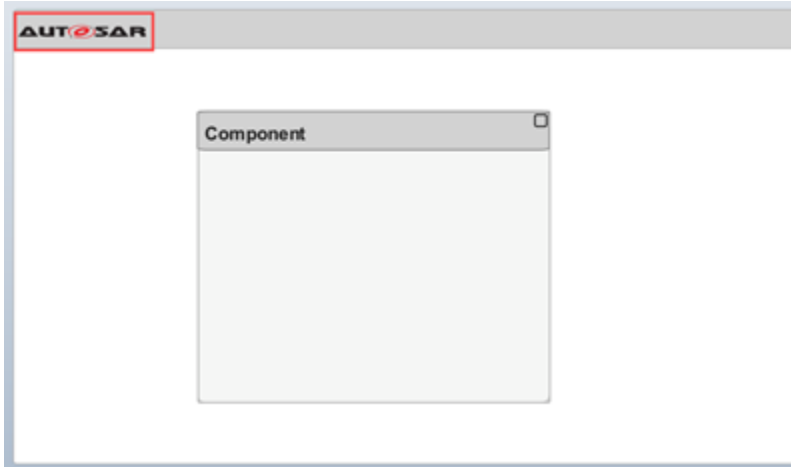
- Use the `createArchitectureModel` function with the `ClassicAUTOSARArchitecture` or `AdaptiveAUTOSARArchitecture` option for the `modelType` argument.

During the conversion process, inline subcomponents will be created as:

- A Classic Component or an Adaptive Component, if the subcomponent does not contain other components
- A Software Composition, if the subcomponent does contain other components

If the component being converted has a defined behavior, the conversion process updates configuration parameters for the linked implementation model to support AUTOSAR modeling. Under **Solver**, the **Type** parameter changes to `Fixed-step` and the **Solver** parameter changes to `auto`. Under **Code Generation**, the **System target file** changes to `autosar.tlc` for classic architectures or `autosar_adaptive.tlc` for adaptive architectures.

After the conversion, observe the AUTOSAR icon in the upper left corner. The new AUTOSAR software architecture contains the elements from the component, including the previously applied stereotypes.



Modeling elements in an existing System Composer component that are not supported in an AUTOSAR architecture are removed from your model during the conversion. These elements include:

- Adapter blocks with applied interface conversions
- Functions defined in the top level of the architecture

See Also

Related Examples

- “Author AUTOSAR Classic Compositions and Components in Architecture Model” on page 8-50
- “Add and Connect AUTOSAR Classic Components and Compositions” on page 8-4
- “Add and Connect AUTOSAR Adaptive Components and Compositions” on page 8-10
- “Design Software Architectures” (System Composer)