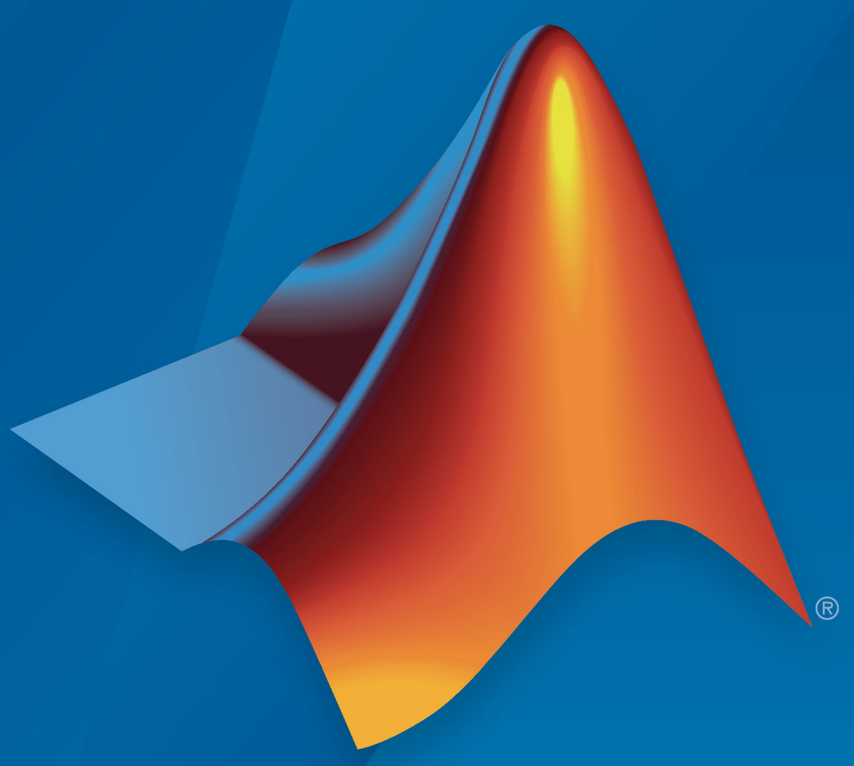
AUTOSAR Blockset

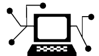
User's Guide



MATLAB® & SIMULINK®

R2019a

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 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)

Contents

1 Overview of AUTOSAR Support

AUTOSAR Standard **1-2**

Comparison of AUTOSAR Classic and Adaptive Platforms **1-4**
Classic Platform **1-4**
Adaptive Platform **1-6**

AUTOSAR Software Components and Compositions **1-11**

Workflows for AUTOSAR **1-13**
Round-Trip Workflow **1-13**

Simulink Originated (Bottom-Up) Workflow **1-16**

Workflow Samples **1-17**

Develop Mapped Simulink Representation of AUTOSAR Software Component **1-18**
Prerequisites **1-18**
Example Model **1-18**
What You Will Learn **1-19**

Create Algorithmic Model Content That Represents AUTOSAR Software Component Behavior **1-20**

Configure Elements of AUTOSAR Software Component for Simulink Modeling Environment **1-23**
Set Up Initial Component Configuration **1-23**
Customize Component Configuration **1-25**
Configure AUTOSAR Software Component Elements from AUTOSAR Standard Perspective **1-27**

Simulate AUTOSAR Software Component **1-29**

Optional: Generate AUTOSAR Software Component Code (Requires Embedded Coder) **1-30**

Develop Mapped Simulink Representation of AUTOSAR Adaptive Software Component **1-35**
Prerequisites **1-35**
Example Model **1-35**
What You Will Learn **1-36**

Create Algorithmic Model Content That Represents AUTOSAR Adaptive Software Component Behavior **1-37**

Configure Elements of AUTOSAR Adaptive Software Component for Simulink Modeling Environment **1-41**
Set Up an Initial Component Configuration **1-41**
Customize Component Configuration **1-43**
Configure AUTOSAR Adaptive Software Component Elements from AUTOSAR Standard Perspective **1-44**

Simulate AUTOSAR Adaptive Software Component **1-47**

Optional: Generate AUTOSAR Adaptive Software Component Code (Requires Embedded Coder) **1-48**

2 Modeling Patterns for AUTOSAR

Simulink Modeling Patterns for AUTOSAR **2-2**

Model AUTOSAR Software Components **2-3**
About AUTOSAR Software Components **2-3**
Implementation Considerations **2-4**
Rate-Based Components **2-7**
Function-Call Based Components **2-9**
Multi-Instance Components **2-11**
Startup, Reset, and Shutdown **2-11**

Modeling Patterns for AUTOSAR Runnables **2-13**

Model AUTOSAR Communication **2-26**
About AUTOSAR Communication **2-26**
Sender-Receiver Interface **2-27**
Client-Server Interface **2-28**
Mode-Switch Interface **2-30**
Nonvolatile Data Interface **2-35**
Parameter Interface **2-35**
Trigger Interface **2-36**
Service Interface (Adaptive Platform) **2-36**

Model AUTOSAR Component Behavior **2-38**
AUTOSAR Elements for Modeling Component Behavior **2-38**
Runnables **2-39**
Inter-Runnable Variables **2-39**
System Constants **2-40**
Per-Instance Memory **2-41**
Static and Constant Memory **2-42**
Shared and Per-Instance Parameters **2-42**

Model AUTOSAR Variants **2-44**
Variants for Ports and Runnables **2-44**
Variants for Array Sizes **2-45**
Variants for Runnable Implementations **2-46**
Predefined Variants and System Constant Value Sets **2-46**

Model AUTOSAR Calibration Parameters and Lookup Tables **2-48**
AUTOSAR Calibration Parameters **2-48**
Calibration Parameters for STD_AXIS and COM_AXIS Lookup Tables **2-48**

Model AUTOSAR Data Types **2-52**
About AUTOSAR Data Types **2-52**
Enumerated Data Types **2-54**
Structure Parameters **2-55**
Release 2.x and 3.x Data Types **2-55**
Release 4.x Data Types **2-55**
CompuMethod Categories for Data Types **2-59**

Model AUTOSAR Adaptive Software Components **2-62**

Model AUTOSAR Basic Software Service Calls **2-67**

3 AUTOSAR Component Creation

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 **3-6**

Create and Configure AUTOSAR Software Component **3-9**

Create and Configure AUTOSAR Adaptive Software Component **3-17**

AUTOSAR arxml Importer **3-25**

Import AUTOSAR Software Component **3-27**

Import AUTOSAR Software Component with Multiple Runnables **3-31**

Import AUTOSAR Component to Simulink **3-32**

Import AUTOSAR Software Composition with Atomic Software Components (Classic Platform) **3-38**

Import AUTOSAR Software Component Updates **3-39**
Update Model with AUTOSAR Software Component Changes **3-39**
AUTOSAR Update Report Section Examples **3-42**

Round-Trip Preservation of AUTOSAR XML File Structure and Element Information **3-46**

Reuse AUTOSAR Element Descriptions **3-49**

Reuse AUTOSAR Elements in Component Model **3-52**

Limitations and Tips **3-58**
Cannot Save Importer Objects in MAT-Files **3-58**
ApplicationRecordDataType and ImplementationDataType Element Names Must Match **3-58**

4 AUTOSAR Component Development

AUTOSAR Component Configuration **4-3**

Configure AUTOSAR Elements and Properties **4-8**
AUTOSAR Elements Configuration Workflow **4-8**
Configure AUTOSAR Atomic Software Components **4-10**
Configure AUTOSAR Ports **4-13**
Configure AUTOSAR Runnables **4-28**
Configure AUTOSAR Inter-Runnable Variables **4-34**
Configure AUTOSAR Parameters **4-35**
Configure AUTOSAR Communication Interfaces **4-37**
Configure AUTOSAR Computation Methods **4-58**
Configure AUTOSAR SwAddrMethods **4-60**
Configure AUTOSAR XML Options **4-62**

Map AUTOSAR Elements for Code Generation **4-68**
Simulink to AUTOSAR Mapping Workflow **4-68**
Map Inports and Outports to AUTOSAR Sender-Receiver Ports **4-70**
Map Entry-Point Functions to AUTOSAR Runnables **4-72**
Map Data Transfers to AUTOSAR Inter-Runnable Variables . **4-74**
Map Function Callers to AUTOSAR Client-Server Ports and Operations **4-74**
Map Model Workspace Parameters to AUTOSAR Component Internal Parameters **4-75**
Map Block Signals and States to AUTOSAR Variables **4-79**
Map Data Stores to AUTOSAR Variables **4-81**
Specify C Type Qualifiers for AUTOSAR Static and Constant Memory **4-84**

Configure AUTOSAR Adaptive Elements and Properties **4-87**
AUTOSAR Elements Configuration Workflow **4-87**
Configure AUTOSAR Adaptive Software Components **4-88**
Configure AUTOSAR Required and Provided Ports **4-91**
Configure AUTOSAR Service Communication Interfaces **4-94**
Configure AUTOSAR Adaptive XML Options **4-98**

Map AUTOSAR Adaptive Elements for Code Generation . . . **4-103**
Simulink to AUTOSAR Mapping Workflow **4-103**
Map Inports and Outports to AUTOSAR Required and Provided Service Ports **4-105**

Incrementally Update AUTOSAR Mapping after Model Changes **4-107**

Configure AUTOSAR Adaptive Software Components **4-111**

Design AUTOSAR Components, Simulate, and Generate Code **4-121**

Configure Parameters and Signals for AUTOSAR Calibration and Measurement **4-129**

Model AUTOSAR Runnables Using Exported Functions . . . **4-133**

Configure AUTOSAR Packages **4-138**
AR-PACKAGE Structure **4-138**
Configure AUTOSAR Packages and Paths **4-140**
Control AUTOSAR Elements Affected by Package Path Modifications **4-143**
Export AUTOSAR Packages **4-144**
AR-PACKAGE Location in Exported ARXML Files **4-146**

Configure AUTOSAR Package for Component, Interface, CompuMethod, or SwAddrMethod **4-150**

Configure AUTOSAR Sender-Receiver Communication **4-153**
Configure AUTOSAR Sender-Receiver Interface **4-153**
Configure AUTOSAR Provide-Require Port **4-155**
Configure AUTOSAR Receiver Port for IsUpdated Service . . **4-157**
Configure AUTOSAR Sender-Receiver Data Invalidation . . . **4-159**
Configure AUTOSAR S-R Interface Port for End-To-End Protection **4-163**
Configure AUTOSAR Receiver Port for DataReceiveErrorEvent **4-165**
Configure AUTOSAR Sender-Receiver Port ComSpecs **4-168**

Configure AUTOSAR Queued Sender-Receiver Communication **4-172**
Simulink Workflow for Modeling AUTOSAR Queued Send and Receive **4-173**
Configure AUTOSAR Sender and Receiver Components for Queued Communication **4-174**
Implement AUTOSAR Queued Send and Receive Messaging **4-177**

Configure Simulation of AUTOSAR Queued Sender-Receiver Communication **4-183**
Simulate N-to-1 AUTOSAR Queued Sender-Receiver Communication **4-184**
Simulate Event-Driven AUTOSAR Queued Sender-Receiver Communication **4-188**

Configure AUTOSAR Client-Server Communication **4-197**
Configure AUTOSAR Server **4-197**
Configure AUTOSAR Client **4-207**
Configure AUTOSAR Client-Server Error Handling **4-216**
Concurrency Constraints for AUTOSAR Server Runnables . **4-221**
Configure and Map AUTOSAR Server and Client Programmatically **4-223**

Configure AUTOSAR Mode-Switch Communication **4-225**
Configure Mode Receiver Port and Mode-Switch Event for Mode User **4-225**
Configure Mode Sender Port and Mode Switch Point for Application Mode Manager **4-230**

Configure AUTOSAR Nonvolatile Data Communication **4-234**

Configure Receiver for AUTOSAR Parameter Communication **4-237**

Configure Receiver for AUTOSAR External Trigger Event Communication **4-240**

Configure AUTOSAR Adaptive Service Communication **4-245**

Configure Lookup Tables for AUTOSAR Measurement and Calibration **4-250**
Configure STD_AXIS Lookup Tables By Using Lookup Table Objects **4-250**
Configure COM_AXIS Lookup Tables By Using Lookup Table and Breakpoint Objects **4-255**

Configure AUTOSAR Data for Measurement and Calibration **4-263**
About Software Data Definition Properties (SwDataDefProps) **4-263**
Configure SwCalibrationAccess **4-264**

Configure DisplayFormat **4-266**
Configure SwAddrMethod **4-270**
Configure SwAlignment **4-274**
Export SwImplPolicy **4-274**
Export SwRecordLayout for Lookup Table Data **4-275**

Configure AUTOSAR Runnables and Events **4-277**

Configure AUTOSAR Initialize, Reset, or Terminate Runnables **4-281**

Add Top-Level Asynchronous Trigger to Periodic Rate-Based System **4-288**

Configure AUTOSAR Initialization Runnable (R4.1) **4-291**

Configure Disabled Mode for AUTOSAR Runnable Event . . **4-294**

Configure AUTOSAR Per-Instance Memory **4-295**
Configure Block Signals and States as AUTOSAR Typed Per-Instance Memory **4-295**
Configure Data Stores as AUTOSAR Typed Per-Instance Memory **4-297**

Configure AUTOSAR Static Memory **4-300**
Configure Block Signals and States as AUTOSAR Static Memory **4-300**
Configure Data Stores as AUTOSAR Static Memory **4-302**

Configure AUTOSAR Constant Memory **4-305**

Configure AUTOSAR Shared or Per-Instance Parameters . . **4-308**
Configure Model Workspace Parameters as AUTOSAR Shared Parameters **4-308**
Configure Model Workspace Parameters as AUTOSAR Per-Instance Parameters **4-310**

Configure AUTOSAR Release 4.x Data Types **4-314**
Control Application Data Type Generation **4-314**
Configure DataTypeMappingSet Package and Name **4-315**
Initialize Data with ApplicationValueSpecification **4-317**

Automatic AUTOSAR Data Type Generation **4-318**

Configure AUTOSAR CompuMethods **4-320**
Configure AUTOSAR CompuMethod Properties **4-320**
Create AUTOSAR CompuMethods **4-322**
Configure CompuMethod Direction for Linear Functions . . **4-323**
Export CompuMethod Unit References **4-325**
Modify Linear Scaling for SCALE_LINEAR_AND_TEXTTABLE CompuMethod **4-325**
Configure Rational Function CompuMethod for Dual-Scaled Parameter **4-327**

Configure AUTOSAR Internal Data Constraints Export **4-333**

Configure Variants for AUTOSAR Ports and Runnables **4-335**

Configure Variants for AUTOSAR Array Sizes **4-339**

Configure Variants for AUTOSAR Runnable Implementations **4-341**

Control AUTOSAR Variants with Predefined Value Combinations **4-344**

Configure and Map AUTOSAR Component Programmatically **4-347**
AUTOSAR Property and Map Functions **4-347**
Tree View of AUTOSAR Configuration **4-348**
Properties of AUTOSAR Elements **4-349**
Specify AUTOSAR Element Location **4-353**

AUTOSAR Property and Map Function Examples **4-355**
Configure AUTOSAR Software Component **4-356**
Configure AUTOSAR Interfaces **4-368**
Configure AUTOSAR XML Export **4-376**

Limitations and Tips **4-378**
AUTOSAR Client Block in Referenced Model **4-378**
Use the Merge Block for Inter-Runnable Variables **4-378**

5 AUTOSAR Code Generation

Generate AUTOSAR C or C++ Code and XML Descriptions . . . 5-2

Configure AUTOSAR Code Generation . . . 5-12
Select an AUTOSAR Schema . . . **5-12**
Specify Maximum SHORT-NAME Length . . . **5-13**
Configure AUTOSAR Compiler Abstraction Macros . . . **5-14**
Root-Level Matrix I/O . . . **5-15**
Inspect AUTOSAR XML Options . . . **5-16**
Generate AUTOSAR C and XML Files . . . **5-16**

Configure AUTOSAR Adaptive Code Generation . . . 5-18
Inspect AUTOSAR Adaptive XML Options . . . **5-18**
Generate AUTOSAR Adaptive C++ and XML Files . . . **5-18**

Code Generation with AUTOSAR Code Replacement Library . . . 5-20
Code Replacement Library for AUTOSAR Code Generation . . **5-20**
Find Supported AUTOSAR Library Routines . . . **5-21**
Configure Code Generator to Use AUTOSAR 4.0 Code Replacement Library . . . **5-21**
Code Replacement Library Checks . . . **5-21**
AUTOSAR Code Replacement Library example for IFX/IFL Function Replacement . . . **5-21**

Verify AUTOSAR C or C++ Code with SIL and PIL . . . 5-23

Import and Simulate AUTOSAR Code from Previous Releases . . . 5-24

Limitations and Tips . . . 5-25
Generate Code Only Check Box . . . **5-25**
AUTOSAR Compiler Abstraction Macros . . . **5-25**
Relative File Paths in AUTOSAR Code Descriptors (Schema Versions 3.x and Earlier) . . . **5-26**
Schedule Editor Explicit Partitions . . . **5-26**

6 AUTOSAR Composition and ECU Software Simulation

Import AUTOSAR Composition to Simulink **6-2**

Combine and Simulate AUTOSAR Software Components **6-7**
Import AUTOSAR Composition as Model (Classic Platform) . . **6-7**
Create Composition Model for Simulating AUTOSAR Components **6-8**
Alternatives for AUTOSAR System-Level Simulation **6-10**

Configure Calls to AUTOSAR Diagnostic Event Manager Service **6-13**

Configure Calls to AUTOSAR NVRAM Manager Service **6-20**

Configure AUTOSAR Basic Software Service Implementations for Simulation **6-27**

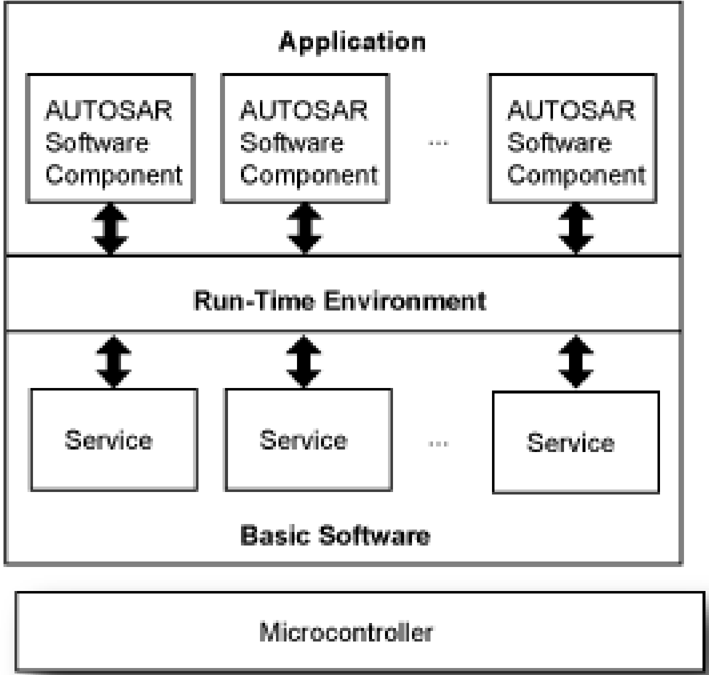
Simulate AUTOSAR Basic Software Services and Run-Time Environment **6-32**

Overview of AUTOSAR Support

- “AUTOSAR Standard” on page 1-2
- “Comparison of AUTOSAR Classic and Adaptive Platforms” on page 1-4
- “AUTOSAR Software Components and Compositions” on page 1-11
- “Workflows for AUTOSAR” on page 1-13
- “Simulink Originated (Bottom-Up) Workflow” on page 1-16
- “Workflow Samples” on page 1-17
- “Develop Mapped Simulink Representation of AUTOSAR Software Component” on page 1-18
- “Create Algorithmic Model Content That Represents AUTOSAR Software Component Behavior” on page 1-20
- “Configure Elements of AUTOSAR Software Component for Simulink Modeling Environment” on page 1-23
- “Simulate AUTOSAR Software Component” on page 1-29
- “Optional: Generate AUTOSAR Software Component Code (Requires Embedded Coder)” on page 1-30
- “Develop Mapped Simulink Representation of AUTOSAR Adaptive Software Component” on page 1-35
- “Create Algorithmic Model Content That Represents AUTOSAR Adaptive Software Component Behavior” on page 1-37
- “Configure Elements of AUTOSAR Adaptive Software Component for Simulink Modeling Environment” on page 1-41
- “Simulate AUTOSAR Adaptive Software Component” on page 1-47
- “Optional: Generate AUTOSAR Adaptive Software Component Code (Requires Embedded Coder)” on page 1-48

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 1-4.
- 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 1-4
- "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

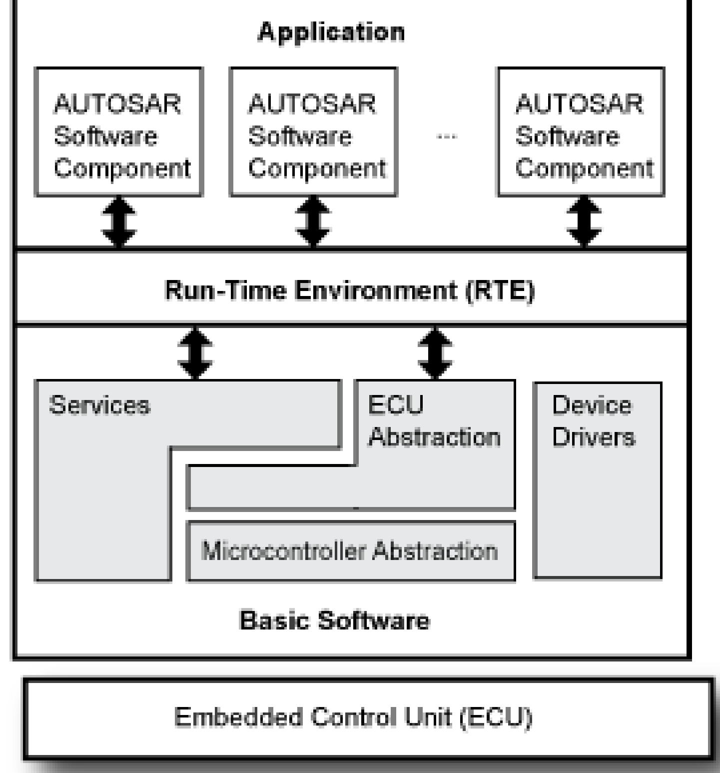
Classic Platform

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

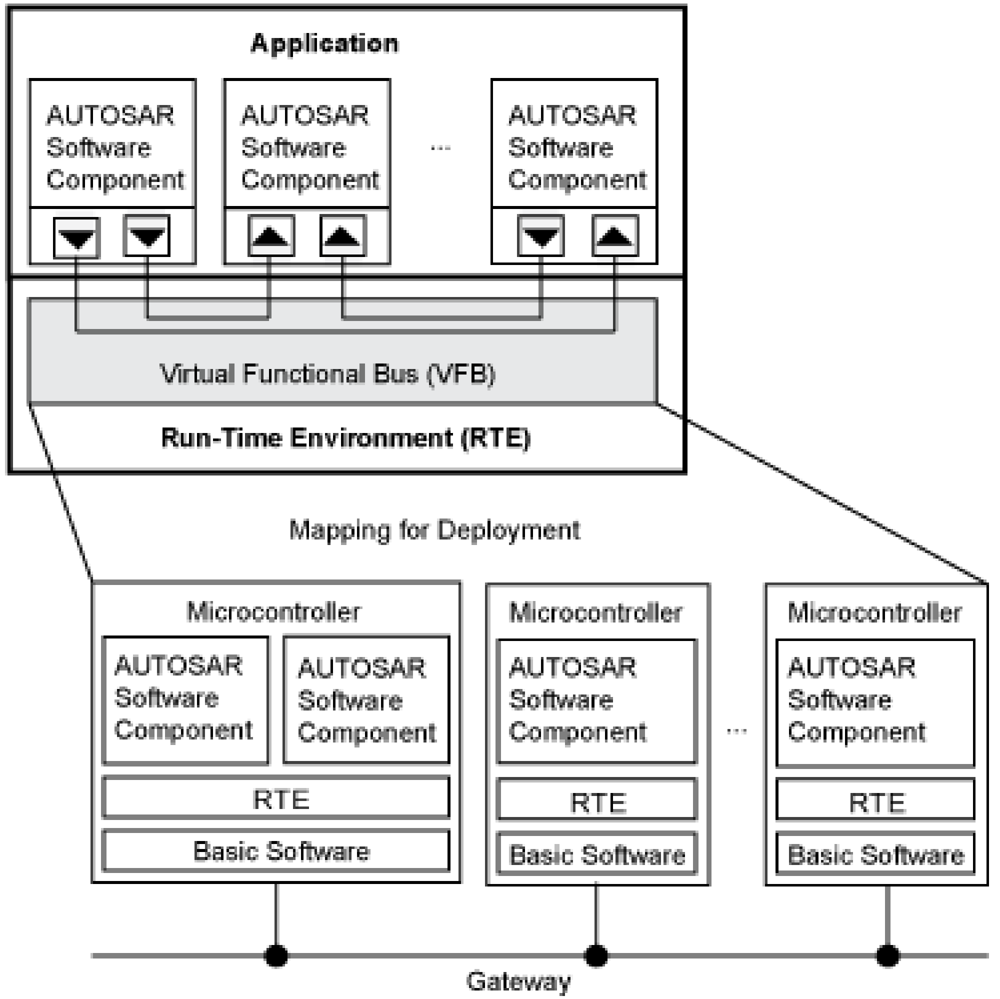
between AUTOSAR software components and services provided by the Basic Software layer. The Basic Software layer consists of:

- 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).

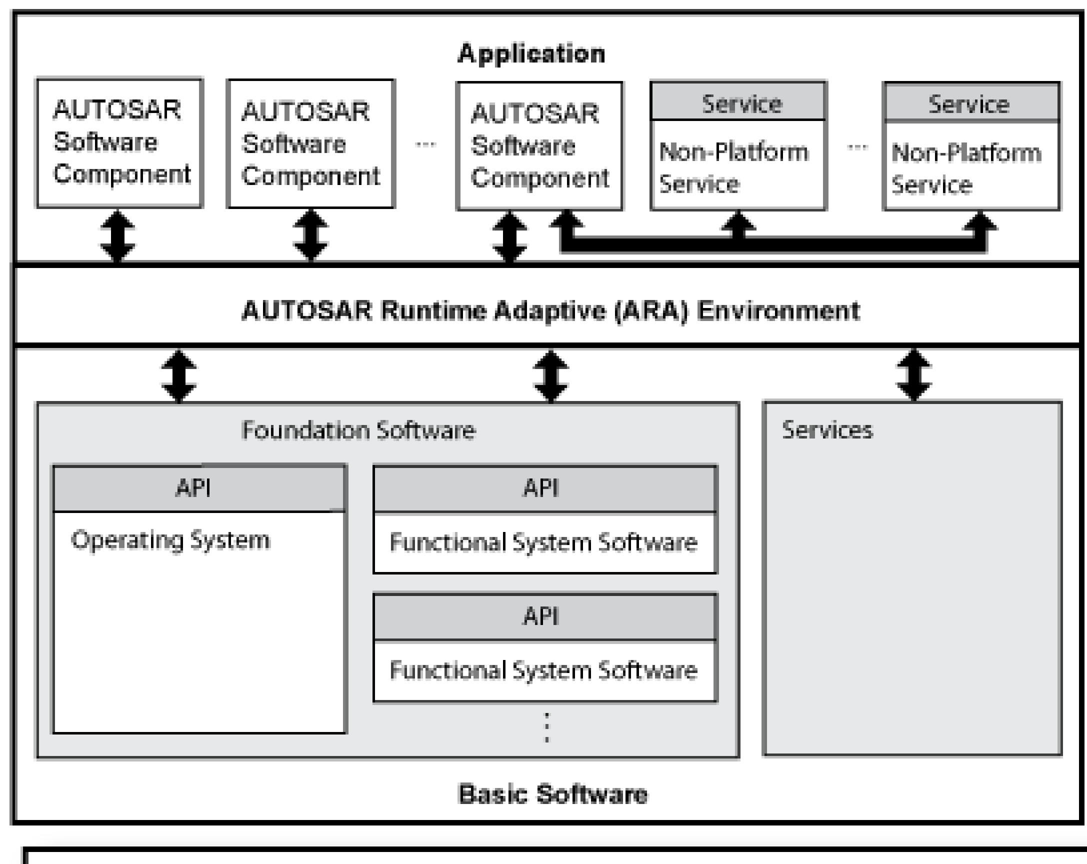
Foundation software includes the POSIX operating system and software for system management tasks, such as:

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

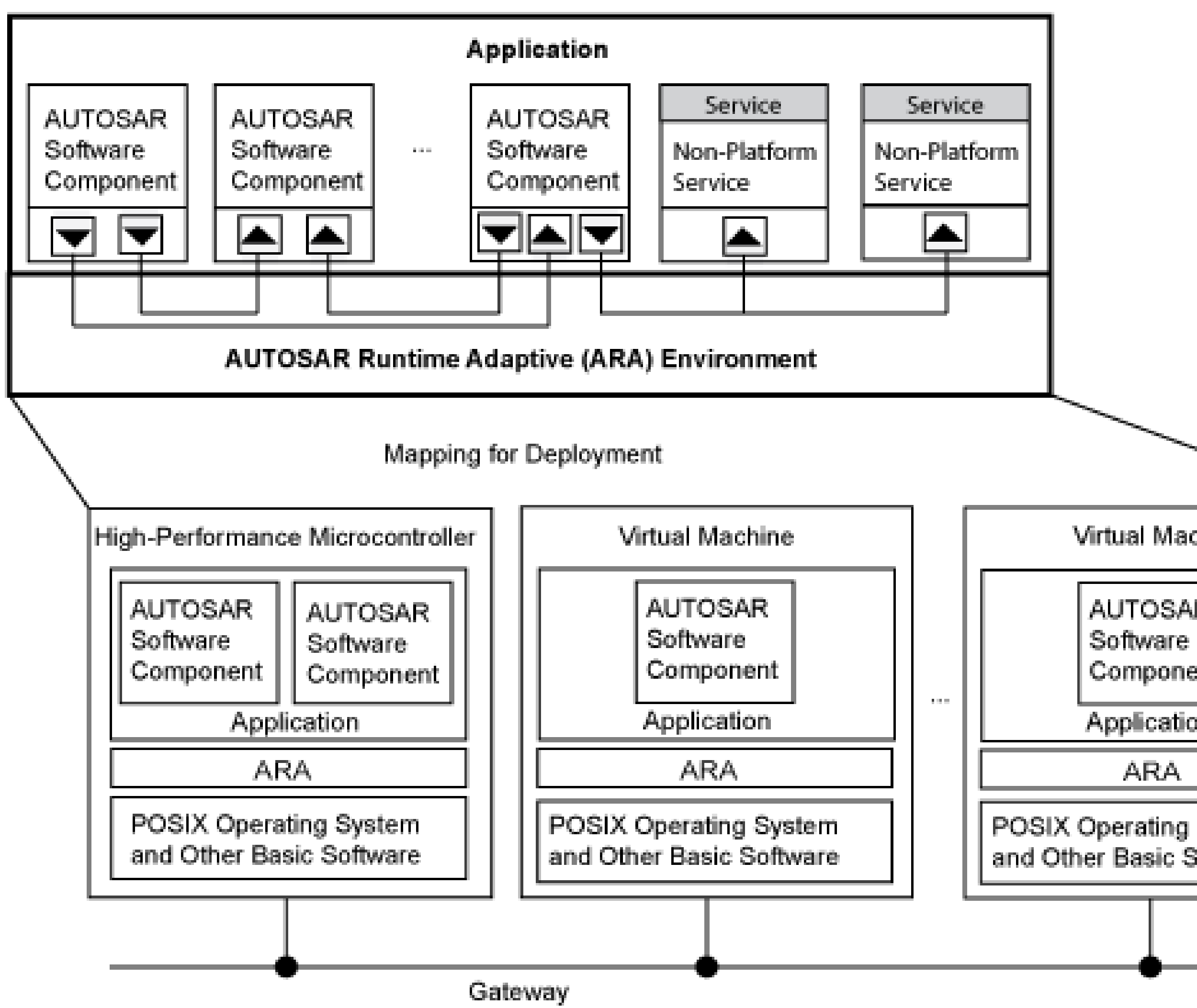
Examples of services include:

- 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 sevices communicate by using dedicated ports. You configure an application by mapping component ports to the RTE representations of the system ECUs.


Application
AUTOSAR Software Component
AUTOSAR Software Component
...
AUTOSAR Software Component
Service
Non-Platform Service
Service
Non-Platform Service
AUTOSAR Runtime Adaptive (ARA) Environment
Mapping for Deployment
High-Performance Microcontroller
AUTOSAR Software Component
AUTOSAR Software Component
Application
ARA
POSIX Operating System and Other Basic Software
Virtual Machine
AUTOSAR Software Component
Application
ARA
POSIX Operating System and Other Basic Software
...
ARA
Gateway

See Also

More About

- https://www.autosar.org

AUTOSAR Software Components and Compositions

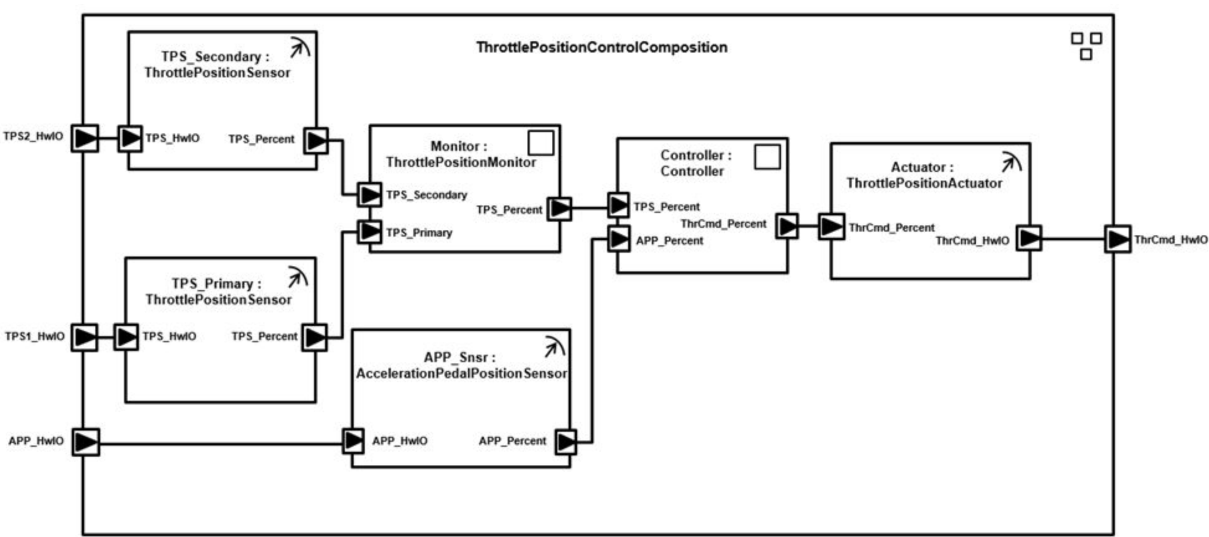
AUTOSAR software components are reusable building blocks of AUTOSAR software. An AUTOSAR software component encapsulates one or more algorithms and communicates with its environment through well-defined ports. For example, a throttle application might include AUTOSAR software components that represent 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 create a Simulink representation of an AUTOSAR composition by importing the composition into Simulink from an `arxml` file.

See Also

More About

- https://www.autosar.org
- “Develop Mapped Simulink Representation of AUTOSAR Software Component” on page 1-18
- “Develop Mapped Simulink Representation of AUTOSAR Adaptive Software Component” on page 1-35
- “Model AUTOSAR Software Components” on page 2-3
- “Import AUTOSAR Software Component” on page 3-27
- “Create AUTOSAR Software Component in Simulink” on page 3-2
- “Import AUTOSAR Composition to Simulink” on page 6-2
- “Import AUTOSAR Software Composition with Atomic Software Components (Classic Platform)” on page 3-38

Workflows for AUTOSAR

This topic describes how to use AUTOSAR Blockset and Embedded Coder® software to configure a Simulink representation of an AUTOSAR application for model-based design, and subsequently generate code from the model that complies with the AUTOSAR standard.

Two typical workflows are:

- The *round-trip* workflow, in which you import AUTOSAR software components created by an AUTOSAR authoring tool (AAT) into the Simulink model-based design environment, and later export XML descriptions and C code for merging back into the AAT environment.
- The Simulink originated, or *bottom-up*, workflow, in which you take a model-based design that originated in Simulink, configure and evolve it for AUTOSAR code generation, and export XML descriptions and C code for use in the AUTOSAR environment.

Round-Trip Workflow

This diagram shows the round-trip workflow.

Simulink Model-Based Design Environment

Export SWC Description/
Generate SWC C Code

Import or Update
SWC Description

Merge
SWC Description

Export
SWC Description

SW-C 1
Internal Behavior
Runnable 1
IRV 1
Runnable 2

AUTOSAR Authoring Tool

In the round-trip workflow, you perform the following tasks:

1. Import previously specified AUTOSAR software components, including definitions of calibration parameters, into Simulink. See:
   - “Import AUTOSAR Software Component” on page 3-27
   - “Import AUTOSAR Component to Simulink” on page 3-32
   - “Import AUTOSAR Composition to Simulink” on page 6-2
   - “Model AUTOSAR Calibration Parameters and Lookup Tables” on page 2-48
2. Develop the model in Simulink. Configure AUTOSAR software component elements, map Simulink model elements to AUTOSAR software component elements, and validate the AUTOSAR configuration. See:
   - “Model AUTOSAR Software Components” on page 2-3
   - “AUTOSAR Component Configuration” on page 4-3
   - “Design AUTOSAR Components, Simulate, and Generate Code” on page 4-121

- "Modeling Patterns for AUTOSAR Runnables" on page 2-13

3. Export the AUTOSAR software component from Simulink, generating XML description files and C code files. See:

   - "Generate AUTOSAR C and XML Files" on page 5-16
   - "Generate AUTOSAR C or C++ Code and XML Descriptions" on page 5-2

   You can verify the generated code in a simulation. See "Verify AUTOSAR C or C++ Code with SIL and PIL" on page 5-23.

4. Merge generated code and description files with other systems by using an AUTOSAR authoring tool.

   You can use the authoring tool to export specifications, which you can import back into Simulink. If the `arxml` code contains AUTOSAR software component changes, you can update the model to reflect the changes. See "Import AUTOSAR Software Component Updates" on page 3-39.

Simulink Originated (Bottom-Up) Workflow

In the Simulink originated (*bottom-up*) workflow, you perform the same tasks as with the round-trip workflow, except that rather than importing AUTOSAR software components from an AAT (step 1), you start with a Simulinkmodel and use Simulink to create a customized AUTOSAR software component. See “Create AUTOSAR Software Component in Simulink” on page 3-2. Subsequent tasks in the workflow are as listed above.

Workflow Samples

Example	How to ...
"Generate AUTOSAR C or C++ Code and XML Descriptions" on page 5-2	Generate AUTOSAR-compliant C or C++ code and export AUTOSAR XML (`arxml`) descriptions from a Simulink model.
"Design AUTOSAR Components, Simulate, and Generate Code" on page 4-121	Develop AUTOSAR components by implementing behavior algorithms, simulating components and compositions, and generating component code.
"Import AUTOSAR Component to Simulink" on page 3-32	Create Simulink representation of AUTOSAR component imported from AUTOSAR authoring tool `arxml` file.
"Import AUTOSAR Composition to Simulink" on page 6-2	Create Simulink representation of AUTOSAR composition imported from AUTOSAR authoring tool `arxml` file.
"Modeling Patterns for AUTOSAR Runnables" on page 2-13	Use Simulink models, subsystems, and functions to model AUTOSAR atomic software components and their runnable entities (runnables).
"Simulate AUTOSAR Basic Software Services and Run-Time Environment" on page 6-32	Simulate AUTOSAR component calls to Basic Software memory and diagnostic services using reference implementations.
"AUTOSAR Property and Map Function Examples" on page 4-355	Programmatically add AUTOSAR elements to a model, configure AUTOSAR properties, and map Simulink elements to AUTOSAR elements.

See Also

More About

- https://www.autosar.org
- "Modeling Patterns"

Develop Mapped Simulink Representation of AUTOSAR Software Component

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.

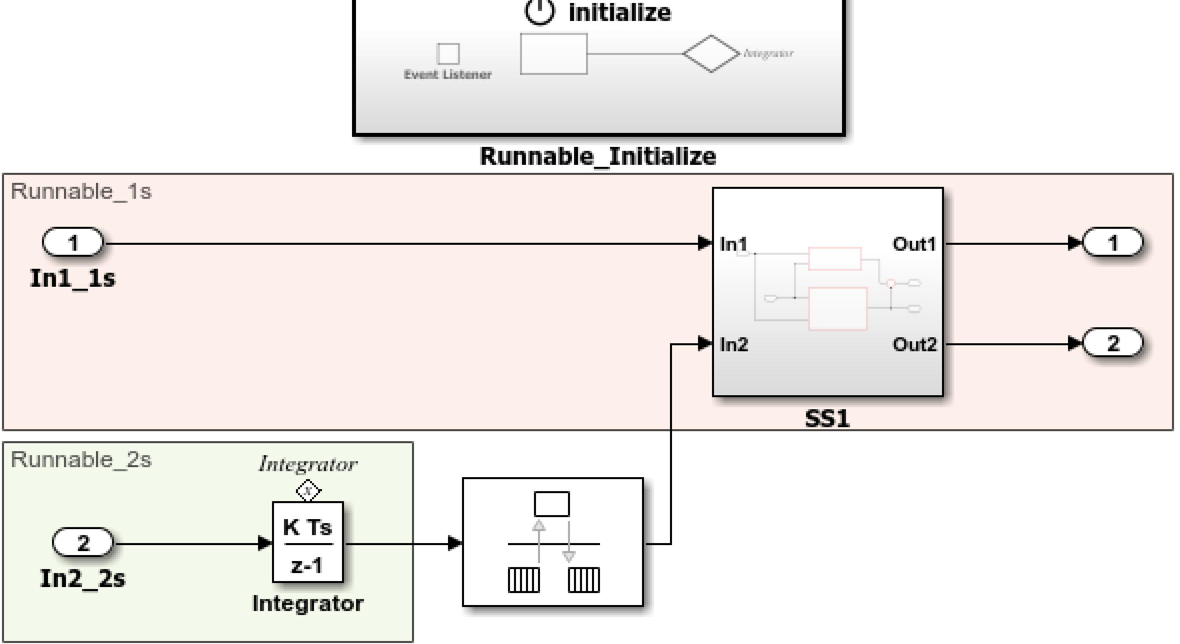
To complete this tutorial, you must have:

- MATLAB®
- Simulink

To complete the optional part of the tutorial, you must have access to Simulink Coder™ and Embedded Coder software.

Example Model

The tutorial uses example models `swc` and `autosar_swc`. Model `swc` is accessible in the folder *`matlabroot`*`/help/toolbox/autosar/examples`. Model `autosar_swc` is available on the MATLAB search path.



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 1-20.

Create Algorithmic Model Content That Represents AUTOSAR Software Component Behavior

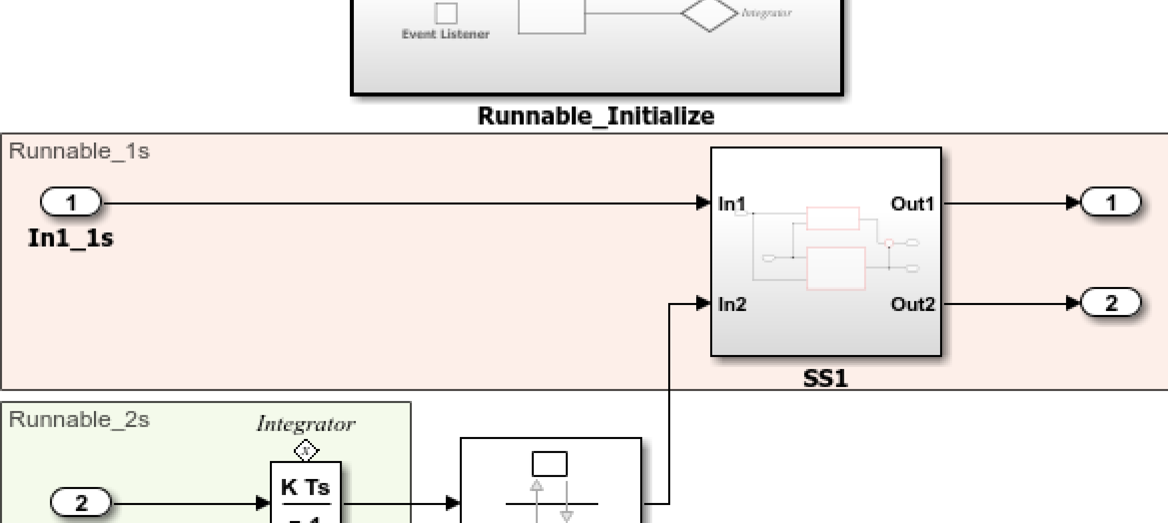
AUTOSAR Blockset software supports AUTOSAR software component modeling for the AUTOSAR Classic Platform. To develop an AUTOSAR software component in Simulink, create a Simulink model that represents the AUTOSAR software component. Initiate the model creation in one of these ways:

- 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_swc` to show a sample model representation of an AUTOSAR software component.

1. Open model `autosar_swc`.



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 Blocksetsoftware.

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 **Display > Sample Time > 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

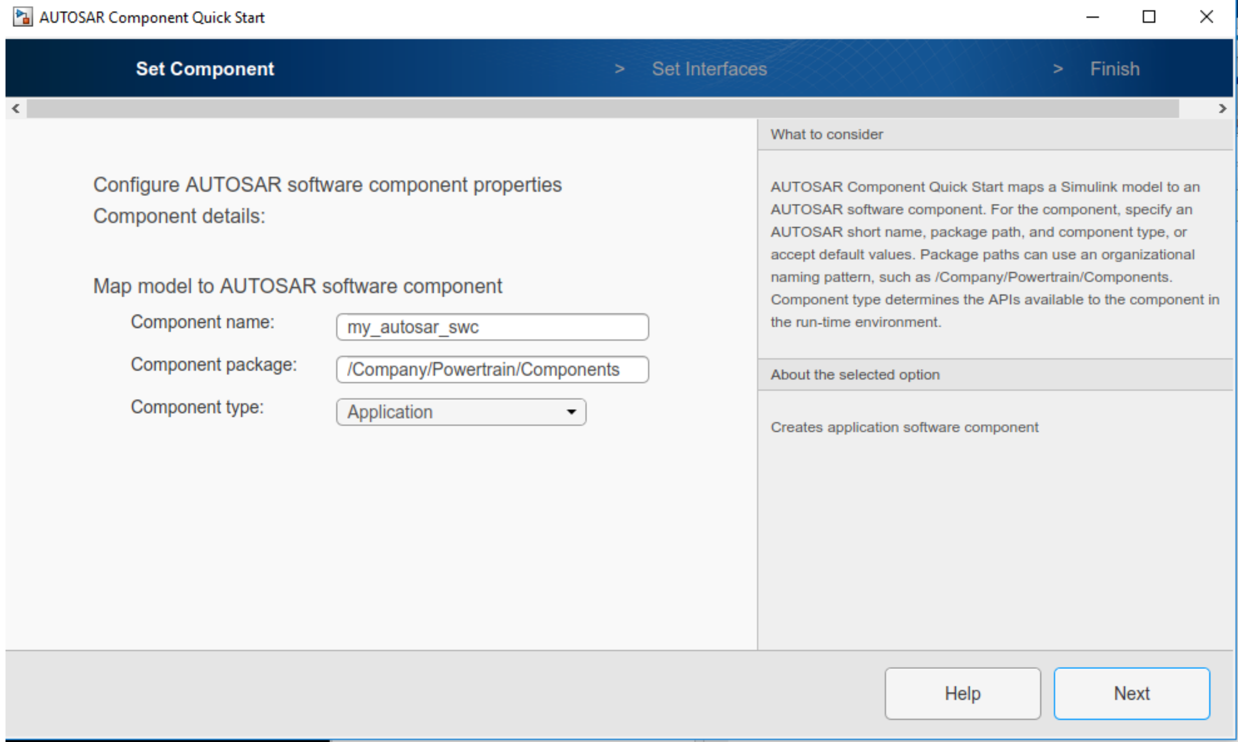
After you create a representation of the AUTOSAR software component in the Simulink Editor, configure elements of the software component for use in Simulink. The configuration maps AUTOSAR software component elements to Simulink modeling elements.

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 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. In the Simulink Editor, open the Code perspective by selecting **Code > C/C++ Code > Configure Model in Code Perspective**. When you open the Code perspective for a 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.

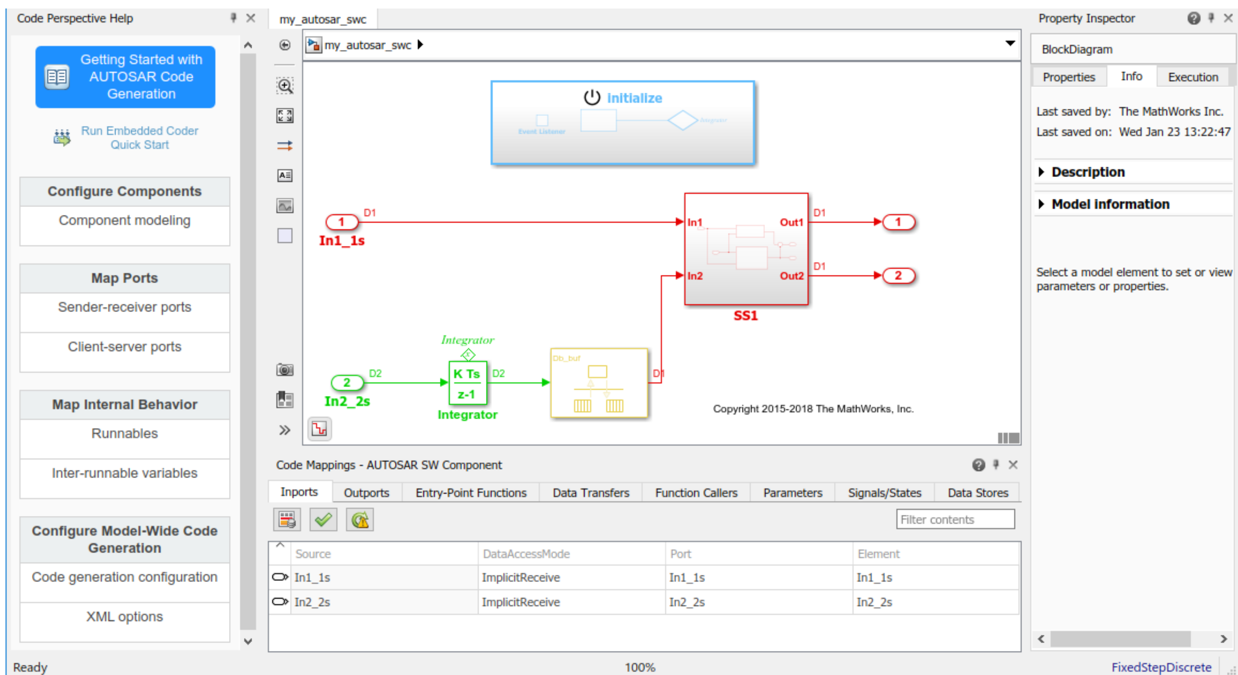
   For this tutorial, use the defaults.

   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 a help panel, a Property Inspector panel,

and, directly below the model, the Code Mappings editor, which you can use to customize the configuration.

- Displays the AUTOSAR software component mappings in the Code Mappings editor.



6 Save the model.

Customize Component Configuration

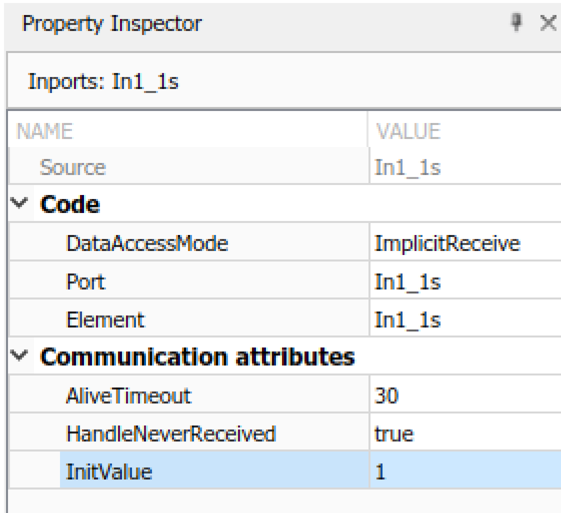
The AUTOSAR Component Quick Start tool sets up an initial configuration for an AUTOSAR software component. To refine or make changes to an existing component configuration, use the Code Mappings editor and AUTOSAR Dictionary.

In a tabbed table format, the Code Mappings editor displays Simulink model elements, such as inports, outports, entry-point functions, 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 ports, runnable entities, and inter-runnable variables (IRVs).

1. If not already open, open model `my_autosar_swc`.
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`. Attributes of the inport appear in the Property Inspector panel.

   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`. In the Property Inspector, expand the list of communication attributes and change:

   - **AliveTimeout** from 0 to 30
   - **HandleNeverReceived** from `false` to `true`
   - **InitValue** from 0 to 1

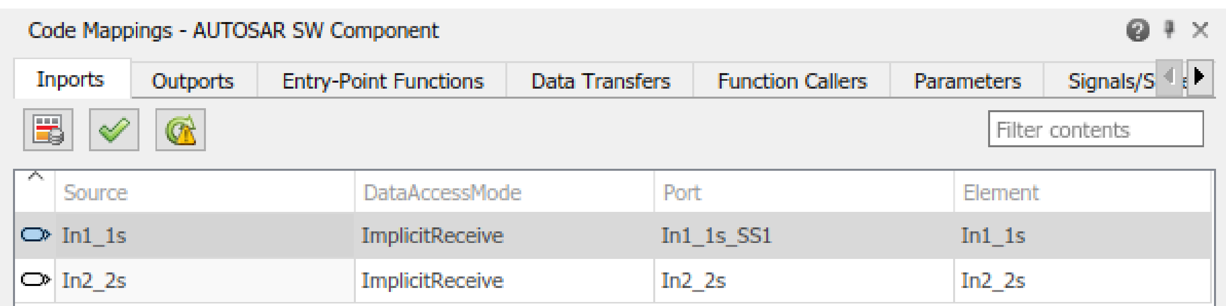
5. Save the model.

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_swc`.
2. Open the AUTOSAR Dictionary. In the Code Mappings editor, click the AUTOSAR Dictionary button . 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 ReceiverPorts view and displays the AUTOSAR port that you mapped the inport to.

   In a tree format, 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.



   If the Update Code Mappings button appears with a warning symbol , click the button to update the Code Mappings to include the latest model data.
5. Save the model.

Next, simulate the AUTOSAR software component.

See Also

Related Examples

- “AUTOSAR Component Configuration” on page 4-3
- “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_swc`, which you configured in “Configure Elements of AUTOSAR Software Component for Simulink Modeling Environment” on page 1-23.

1. If not already open, open your configured version of model `my_autosar_swc`.
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.

See Also

Related Examples

- “Simulate Model and View Results” (Simulink)

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_swc`.
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_swc_autosar_rtw` folder contains the primary files listed in this table.

Generated Code Files

Files	Description
my_autosar_swc.c	Contains entry points for the code that implements the model algorithm. This file includes rate scheduling code.
my_autosar_swc.h	Declares model data structures and a public interface to the model entry points and data structures.
rtwtypes.h	Defines data types, structures, and macros that the generated code requires.
my_autosar_swc_component.arxml my_autosar_swc_datatype.arxml my_autosar_swc_implementation.arxml my_autosar_swc_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.

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.

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

Entry-point functions:

- Initialization entry-point function — `void Runnable_Init(void)`. At startup, call this function once.
- Output and update entry-point function — `void Runnable_Step(void)`. Call this function periodically at the fastest rate in the model. For this model, call the

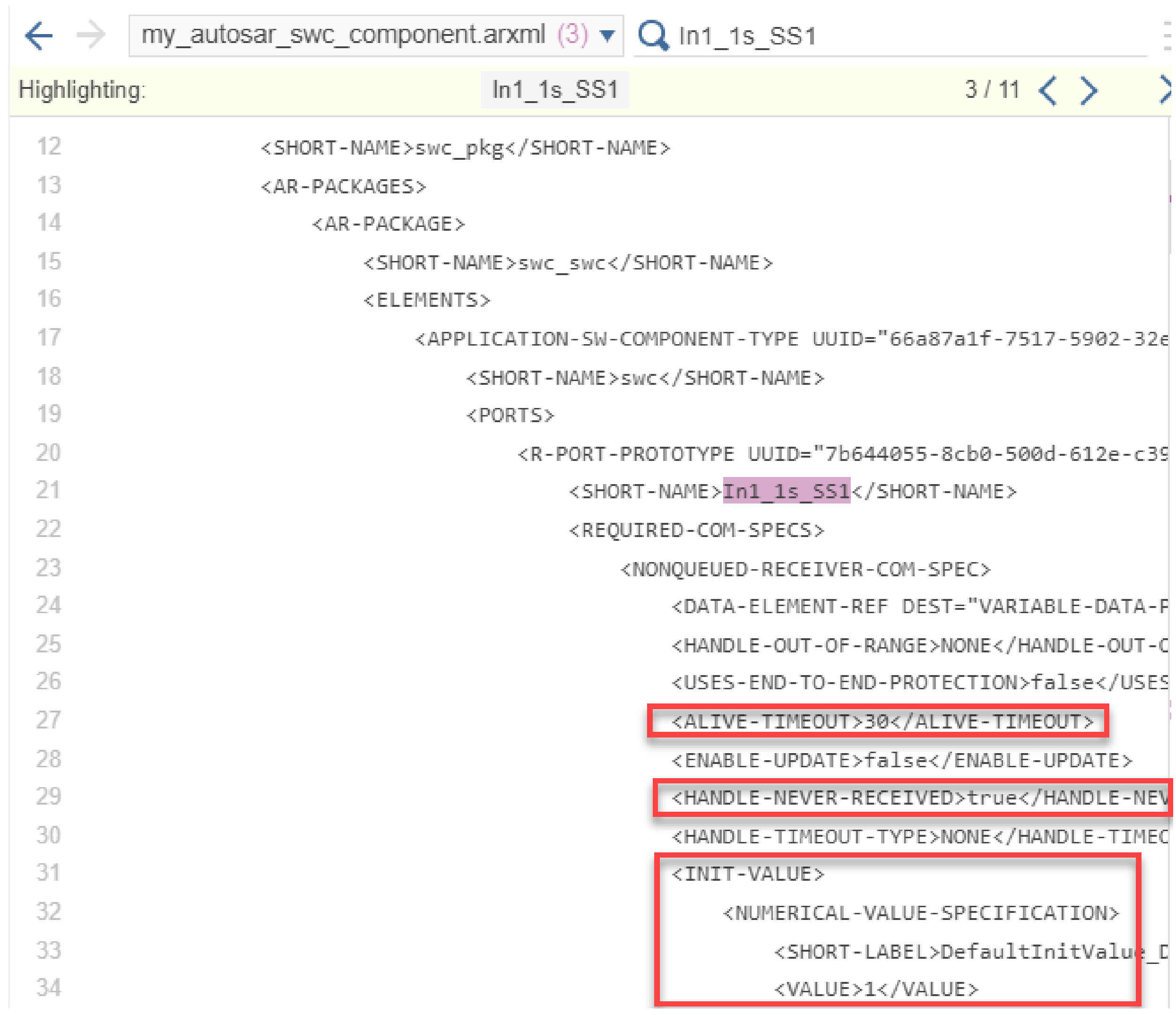
function every second. To achieve real-time execution, attach this function to a timer.
- Output and update entry-point function — `void Runnable_Step1(void)`. Call this function periodically at the second fastest rate in the model. For this model, call the function every 2 seconds. To achieve real-time execution, attach this function to a timer.

The entry-point functions are also accessible in the Code Mappings editor, on the **Entry-Point 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.

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, in the Simulink Editor, select **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 the third instance of the name in the `arxml` file `my_autosar_swc_component.arxml`. Verify that the settings of communication attributes that you modified for the AUTOSAR software component port appear correctly.



6 Use the Code perspective Code panel to explore other aspects of the generated code. For example, if you select file `my_autosar_swc.c`, and then click in the search field, a list of links to code elements, including entry-point functions `Runnable_Init`, `Runnable_Step`, and `Runnable_Step1`, 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 5-12

Develop Mapped Simulink Representation of AUTOSAR Adaptive Software Component

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.

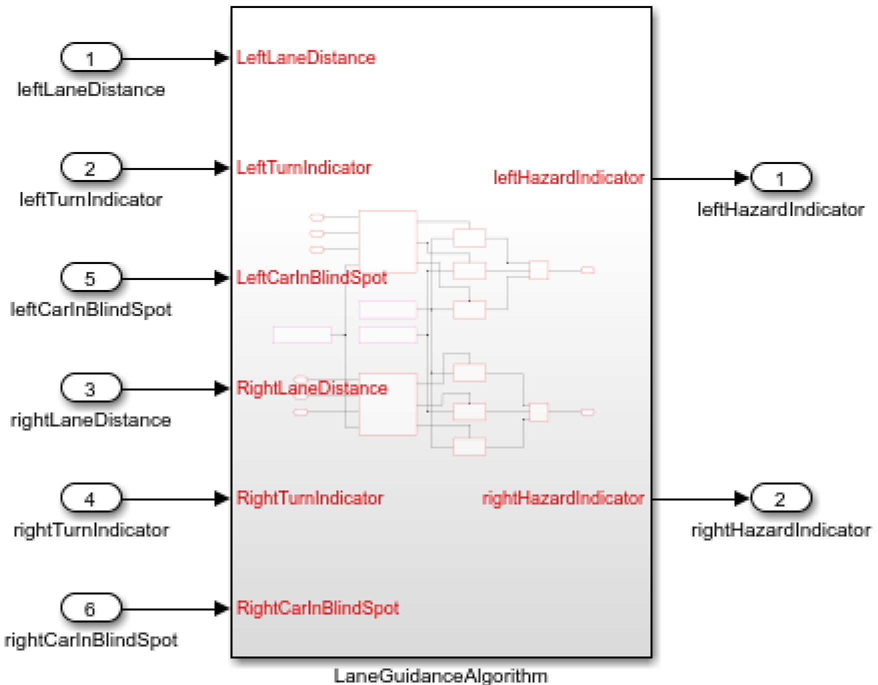
To complete this tutorial, you must have:

- MATLAB
- Simulink

To complete the optional part of the tutorial, you must have access to 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 1-37.

Create Algorithmic Model Content That Represents AUTOSAR Adaptive Software Component Behavior

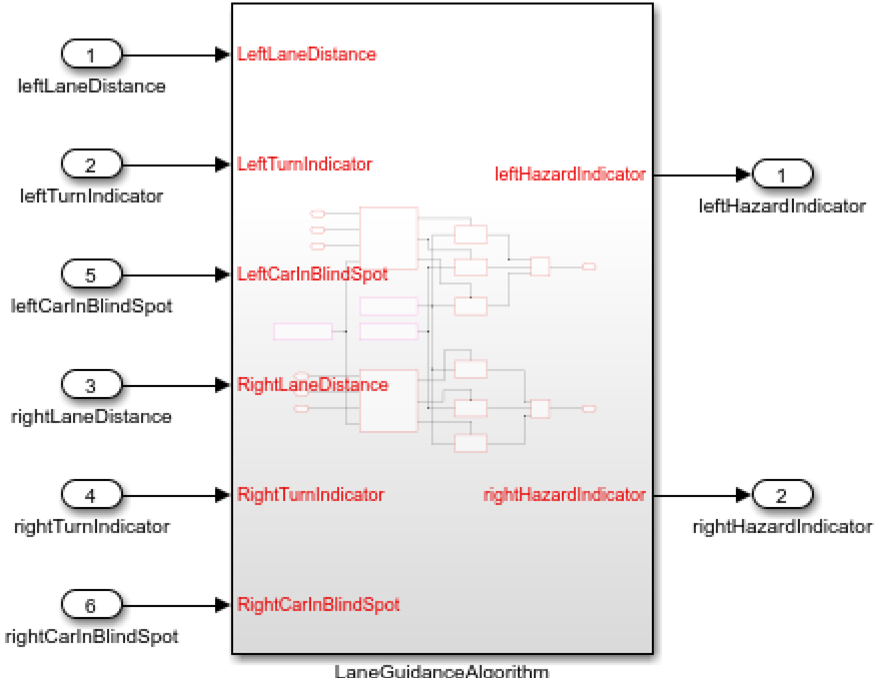
AUTOSAR Blockset software supports AUTOSAR software component modeling for the AUTOSAR Adaptive Platform. To develop an AUTOSAR adaptive software component in Simulink, create a Simulink model that represents the AUTOSAR adaptive software component. Initiate the model creation in one of these ways:

- 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 Blocksetsoftware 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`.

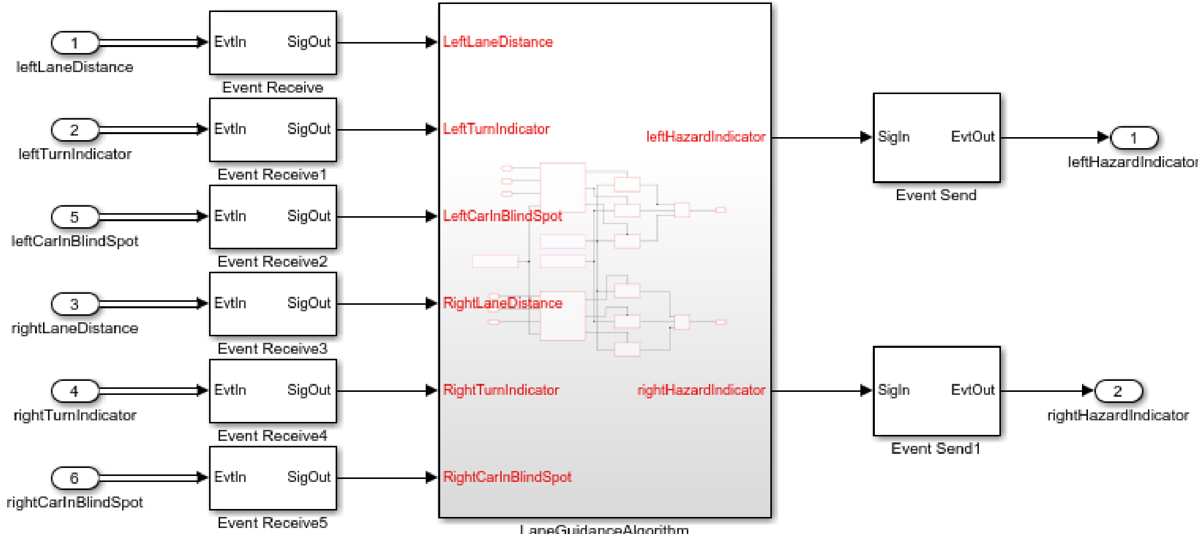
4 At the top level of the model, set up event-based communication. An AUTOSAR adaptive software component provides and consumes services. Each component contains:

- 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 **Display > Sample Time > 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

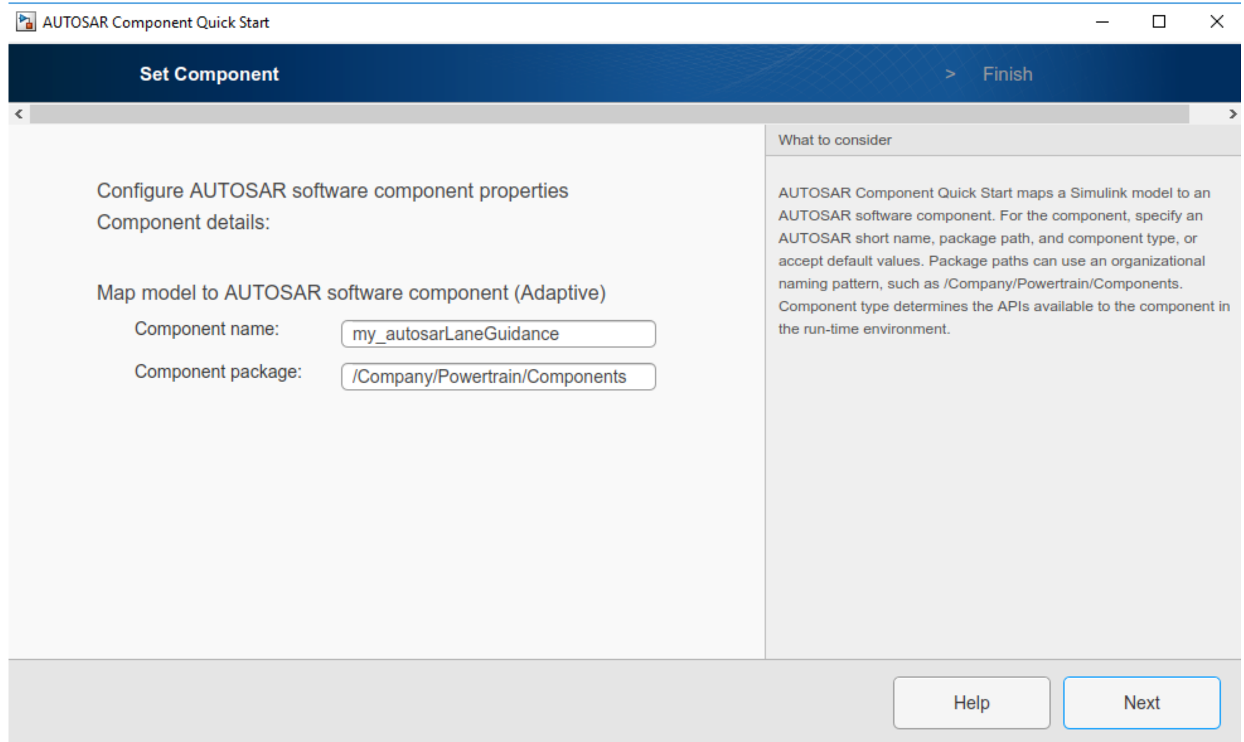
After you create a representation of the AUTOSAR adaptive software component in the Simulink Editor, configure elements of the software component for use in Simulink. The configuration maps AUTOSAR adaptive software component elements to Simulink modeling elements.

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 Code Mappings editor and 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.

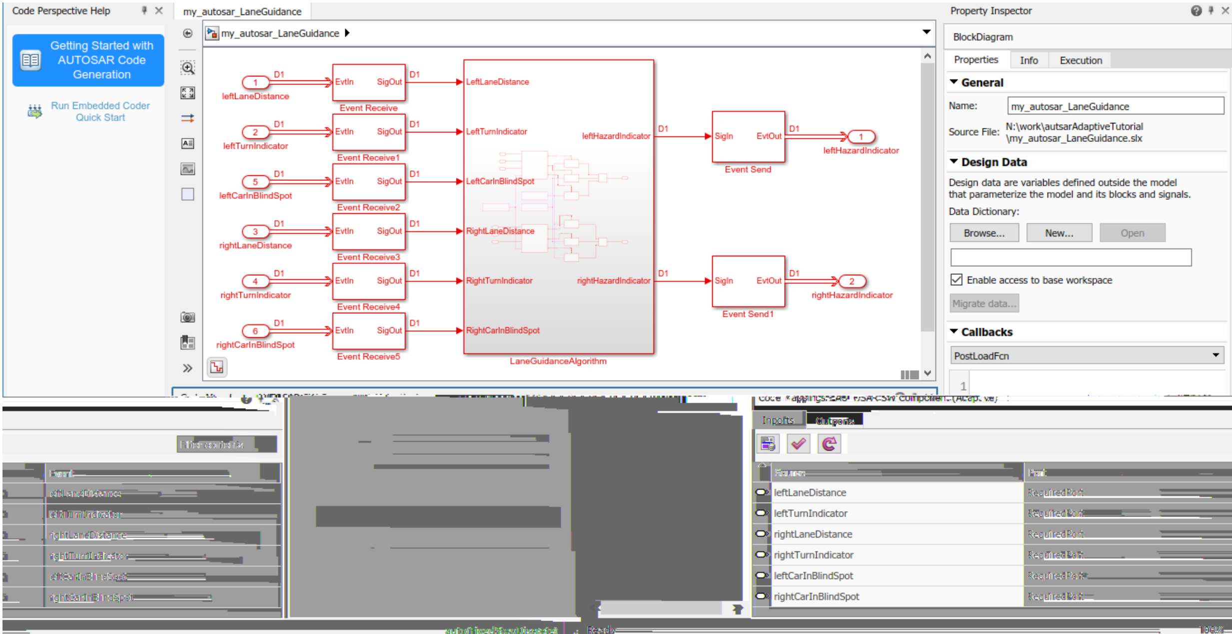
1. Open your saved version of the example model `my_autosar_LaneGuidance`.
2. Run the AUTOSAR Component Quick Start tool. In the Simulink Editor, open the Code perspective by selecting **Code > C/C++ Code > Configure Model in Code Perspective**. When you open the Code perspective for a model that is configured with an AUTOSAR system target file, the AUTOSAR Component Quick Start tool runs automatically.



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 a help panel, a Property Inspector panel, and directly below the model, the Code Mappings editor, which you can use to customize the configuration.
   - Displays the AUTOSAR software component mappings in the Code Mappings editor.



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 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 Model Data Editor, 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. To view data in the Model Data Editor, in the Simulink Editor, select **View > Model Data Editor**. For this tutorial, make no changes.

3. In the Code Mappings editor, examine the mapping of Simulink inports and outports to AUTOSAR ports and events. To view the mappings, exit the Model Data Editor and open the Code Mappings editor. Select **View > Model Data Editor**, removing the check mark next to the **Model Data Editor** menu item.
4. In the Code Mappings editor, 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`. For this tutorial, make no changes.

   If the Update Code Mappings button appears with a warning symbol, click the button to update the Code Mappings to include the latest model data.

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. 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, AUTOSAR Dictionary displays the mapped AUTOSAR software component and its elements, service 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`.

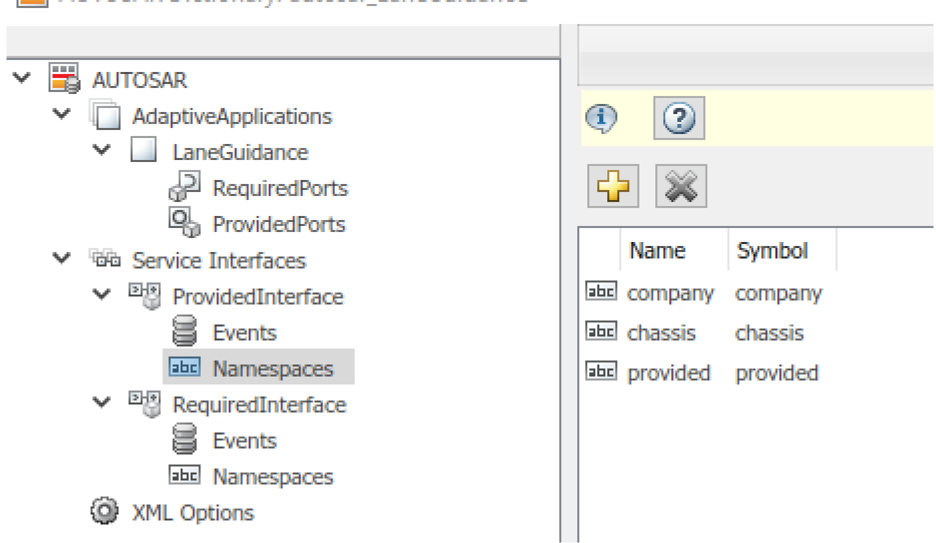
   a In the left pane of the dictionary, expand the **Service Interfaces** and **ProvidedInterface** nodes.

   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 After making configuration changes, in the Code Mappings editor, the Update Code Mappings button appears with a warning symbol , click the button to update the Code Mappings to include the latest model data.

7 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 4-103
- “Configure AUTOSAR Adaptive Elements and Properties” on page 4-87

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 1-41.

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.

See Also

Related Examples

- "Simulate Model and View Results" (Simulink)

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
`my_autosar_LaneGuidance.cpp`	Contains entry points for the code that implements the model algorithm. This file includes the rate scheduling code.
`my_autosar_LaneGuidance.h`	Declares model data structures and a public interface to the model entry points and data structures.
`rtwtypes.h`	Defines data types, structures, and macros that the generated code requires.

`my_autosar_LaneGuidance_component.arxml`
`my_autosar_LaneGuidance_datatype.arxml`
`my_autosar_LaneGuidance_implementation.arxml`
`my_autosar_LaneGuidance_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.

`rtmodel.h`	Declares model classes.
`main.cpp` `MainUtils.hpp`	Provide a framework unning adaptive software component service code.

- 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

Entry-point function — Output entry-point function, `void step(void)`. Call this function periodically at every 0.1 seconds.

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, in the Simulink Editor, select **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.

my_autosarLaneGuidance.cpp (4) company

Highlighting: company 1 / 12

```
455
456     // End of If: '<S9>/If1'
457
458     // Chart: '<S8>/Event Send'
459     sf_msg_send_EvtOut_e();
460   }
461
462   // Model initialize function
463   void mObjectDetectionModelClass::initialize()
464   {
465     {
466       ara::com::ServiceHandleContainer< company::chassis::required::proxy::
467         RequiredInterfaceProxy::HandleType > handles;
468       ProvidedPort = std::make_unique< company::chassis::provided::skeleton::
469         ProvidedInterfaceSkeleton >(ara::com::InstanceIdentifier("ProvidedPort"),
470         ara::com::MethodCallProcessingMode::kPoll);
471       ProvidedPort->OfferService();
472       handles = company::chassis::required::proxy::RequiredInterfaceProxy::
473         FindService();
474       if (handles.size() > 0U) {
475         RequiredPort = std::make_shared< company::chassis::required::proxy::
476           RequiredInterfaceProxy >(*handles.begin());
477       }
```

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 5-18

Modeling Patterns for AUTOSAR

- "Simulink Modeling Patterns for AUTOSAR" on page 2-2
- "Model AUTOSAR Software Components" on page 2-3
- "Modeling Patterns for AUTOSAR Runnables" on page 2-13
- "Model AUTOSAR Communication" on page 2-26
- "Model AUTOSAR Component Behavior" on page 2-38
- "Model AUTOSAR Variants" on page 2-44
- "Model AUTOSAR Calibration Parameters and Lookup Tables" on page 2-48
- "Model AUTOSAR Data Types" on page 2-52
- "Model AUTOSAR Adaptive Software Components" on page 2-62
- "Model AUTOSAR Basic Software Service Calls" on page 2-67

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 AUTOSAR-compliant code generation.

- "Model AUTOSAR Software Components" on page 2-3
- "Modeling Patterns for AUTOSAR Runnables" on page 2-13
- "Model AUTOSAR Communication" on page 2-26
- "Model AUTOSAR Component Behavior" on page 2-38
- "Model AUTOSAR Variants" on page 2-44
- "Model AUTOSAR Calibration Parameters and Lookup Tables" on page 2-48
- "Model AUTOSAR Data Types" on page 2-52
- "Model AUTOSAR Adaptive Software Components" on page 2-62
- "Model AUTOSAR Basic Software Service Calls" on page 2-67

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 2-4
“Rate-Based Components” on page 2-7
“Function-Call Based Components” on page 2-9
“Multi-Instance Components” on page 2-11
“Startup, Reset, and Shutdown” on page 2-11

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 cannot be split into smaller software components, and runs on exactly one automotive electronic control unit (ECU).
- *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 2-48.

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.

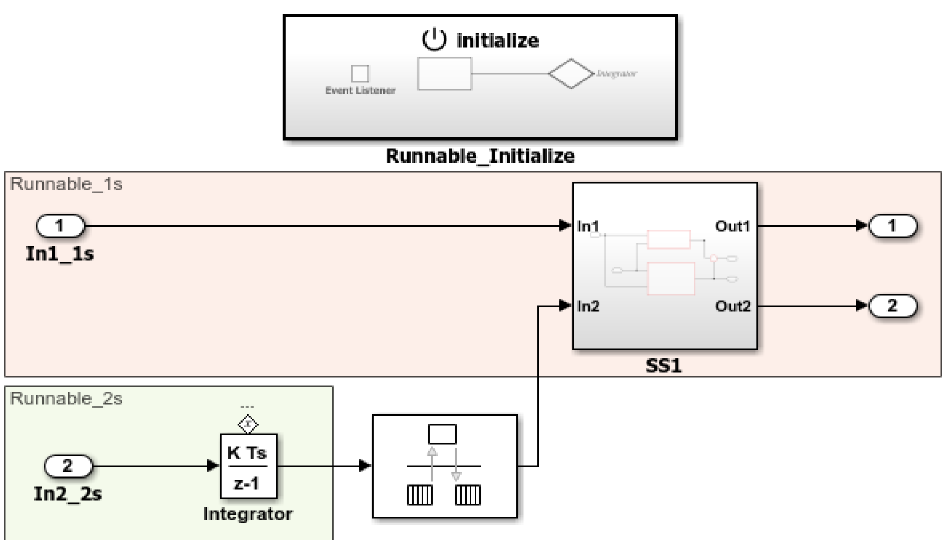
Implementation Considerations

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

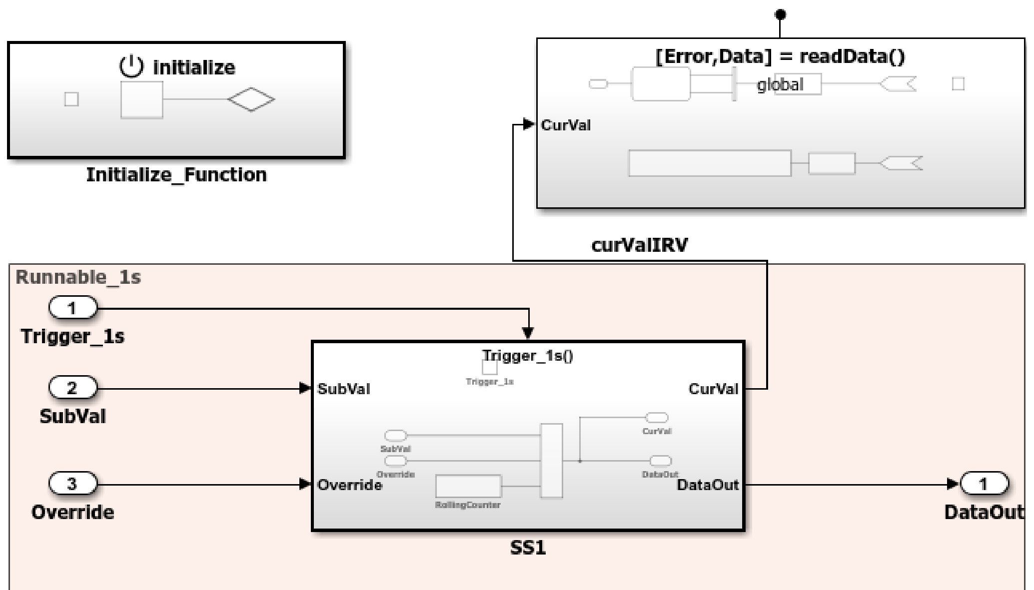
Usually, the Simulink representation of an AUTOSAR component is a rate-based model, in which periodic runnables are modeled as atomic subsystems with periodic rates.

Consider AUTOSAR example model `autosar_swc`. 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, the Simulink representation can be a function-call based model. The model can contain Simulink Function blocks or function-call subsystems with periodic rates.

Consider AUTOSAR example model `autosar_swc_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, designate how to model periodic runnables:

- If you are importing an AUTOSAR component description from `arxml` files using `arxml.importer` object function `createComponentAsModel`, specify the property `ModelPeriodicRunnablesAs` as `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, at the top level of a model, create function-call subsystems — or (for client-server modeling) 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. *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.

Sometimes, conditions in your AUTOSAR software component can prevent use of rate-based modeling. For example:

- The AUTOSAR software component contains a server runnable.
- The AUTOSAR software component contains an inter-runnable variable (IRV) that multiple runnables read or write.
- The AUTOSAR software component contains a periodic runnable with a rate that is not a multiple of the fastest rate.
- The AUTOSAR software component contains multiple runnables that access the same read or write data at different rates.
- The AUTOSAR software component contains a periodic runnable that other events also trigger.
- The AUTOSAR software component contains multiple periodic runnables that are triggered at the same period.

If your AUTOSAR software component supports multiple instantiation (that is, `SwcInternalBehavior` attribute `supportsMultipleInstantiation` is set to `true`), you cannot model periodic runnables as function-call subsystems. Either use rate-based modeling and model periodic runnables as atomic subsystems, or set `supportsMultipleInstantiation` to `false`.

For examples of different ways to model AUTOSAR software components, see "Rate-Based Components" on page 2-7, "Function-Call Based Components" on page 2-9, and "Modeling Patterns for AUTOSAR Runnables" on page 2-13.

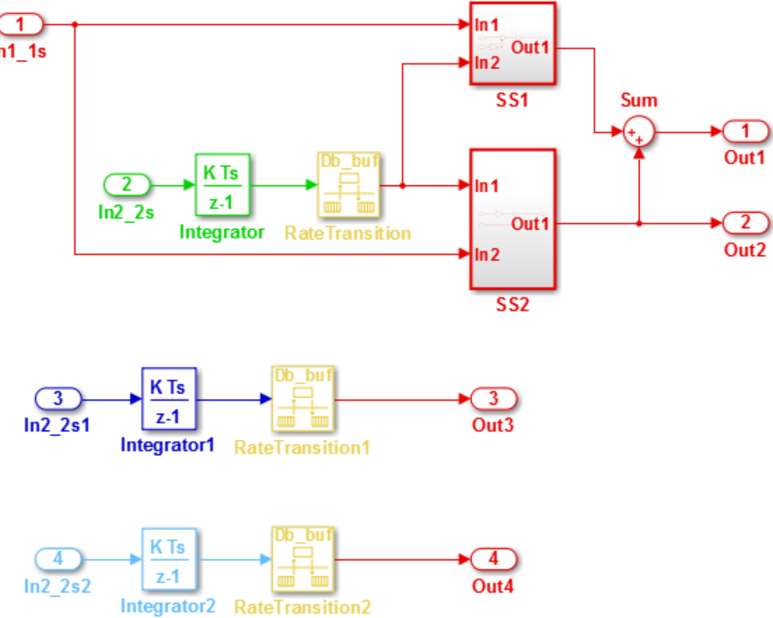
Rate-Based Components

You can model AUTOSAR multi-runnables 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 model *`matlabroot`*`/help/toolbox/autosar/examples/mMultitasking_4rates.slx`.) The model represents an AUTOSAR software component. The four colors displayed when you update the model (if **Display** > **Sample Time** > **Colors** is selected) represent the different periodic rates present. The Rate Transition blocks represent three AUTOSAR IRVs.

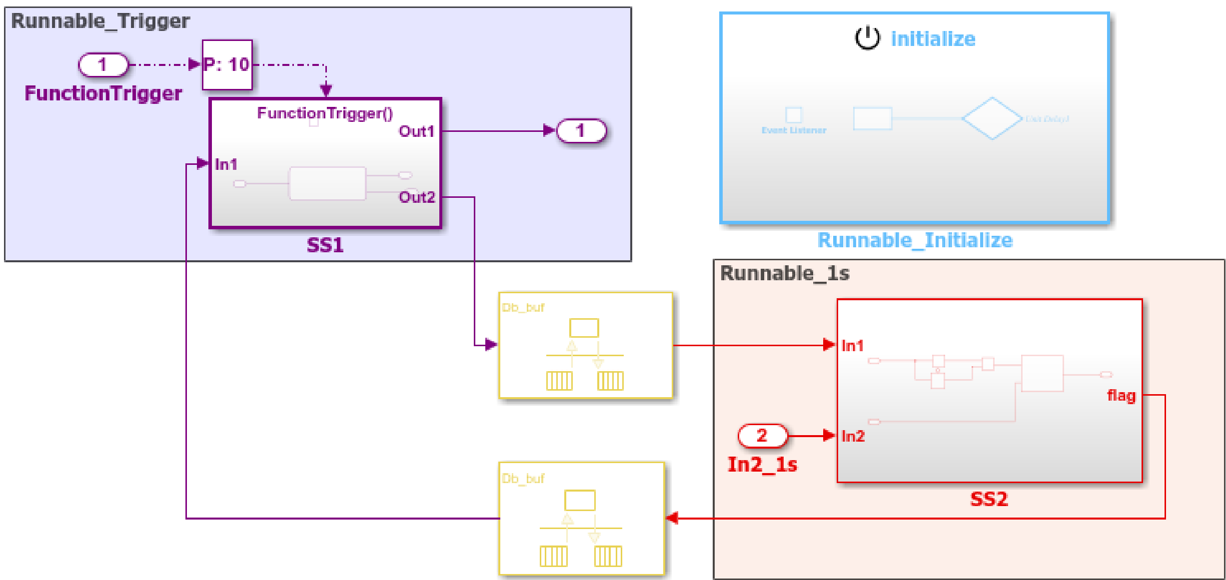


When you generate code, the model C code contains rate-grouped model step functions corresponding to AUTOSAR runnables, one for each discrete rate in the model. (The

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 2-13.

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 AUTOSAR example model `autosar_swc_fcncalls`. 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 4-288.

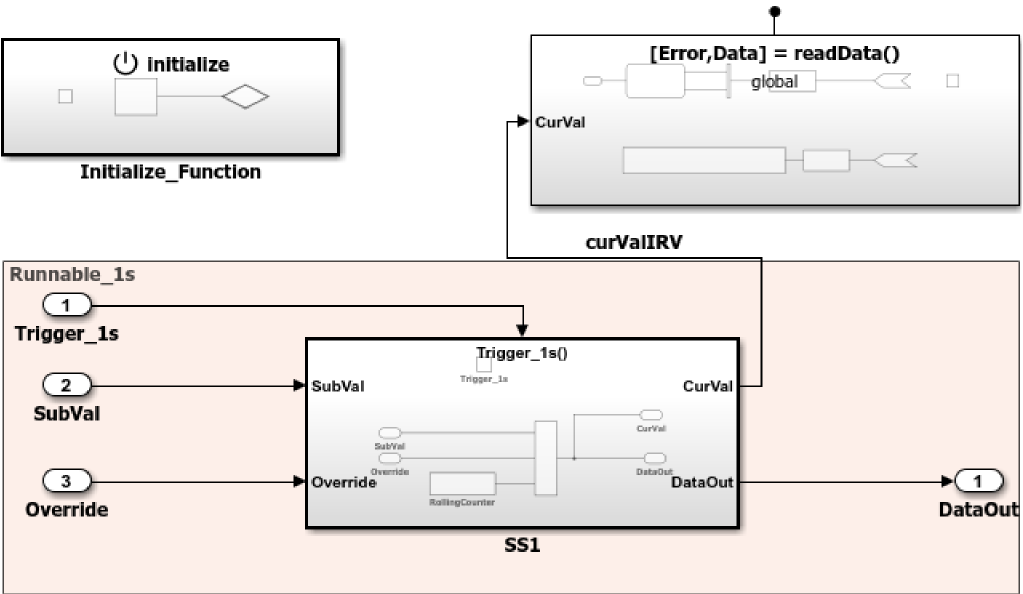
Function-Call Based Components

You can model AUTOSAR multi-runnables 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 2-13.

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 SIL and PIL simulations.
- Import and export multi-instance AUTOSAR SWC description XML files.

Note Configuring a model as a multi-instance AUTOSAR SWC is not supported when the model contains either of the following blocks:

- Simulink Function
- Model-level Inport configured to output a periodic function-call

---

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, Initialize Function block initializes block state with the State Writer block. Similarly, the Terminate Function block saves 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 4-281.

See Also

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

Related Examples

- "Import AUTOSAR Software Component" on page 3-27
- "Modeling Patterns for AUTOSAR Runnables" on page 2-13
- "Configure AUTOSAR Runnables and Events" on page 4-277
- "Configure AUTOSAR Initialize, Reset, or Terminate Runnables" on page 4-281
- "Add Top-Level Asynchronous Trigger to Periodic Rate-Based System" on page 4-288
- "Configure AUTOSAR Code Generation" on page 5-12

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

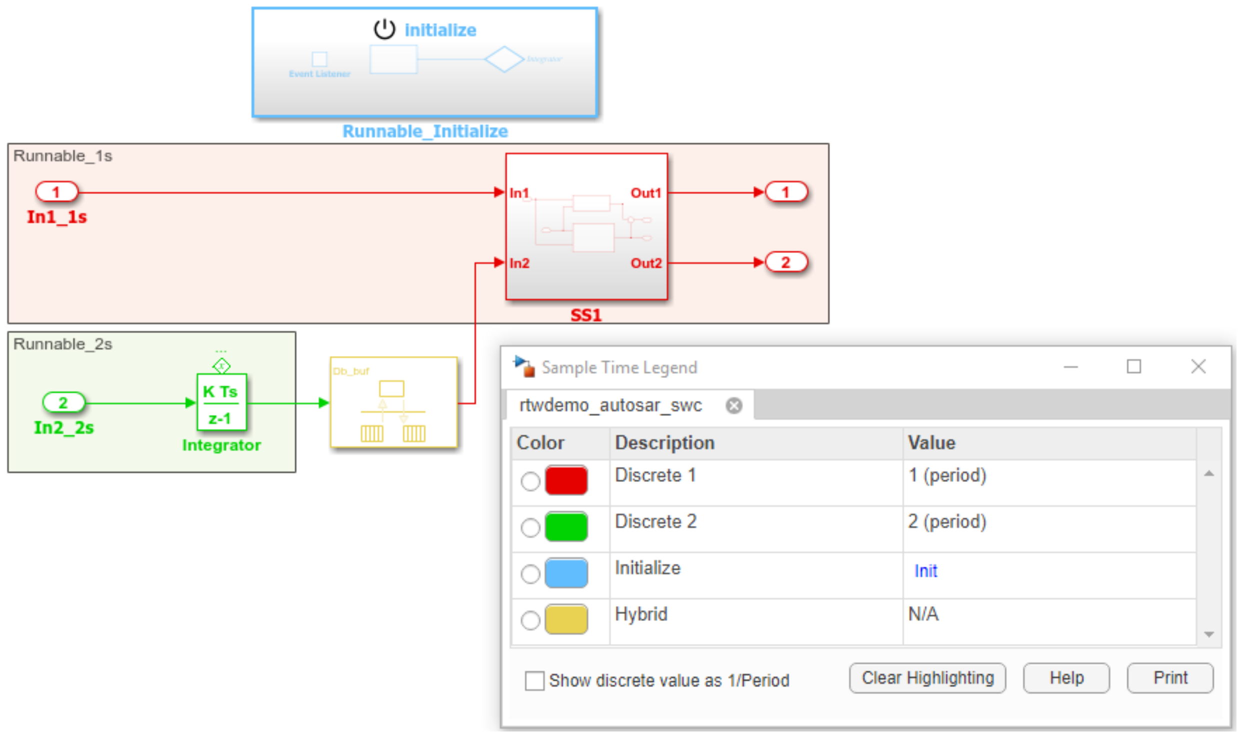
Open the example model `autosar_swc.slx`.

```
open_system('autosar_swc')
```

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, select **Display > Sample Time > 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 **Display > Sample Time > 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 are licensed for Simulink Coder and Embedded Coder, 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.c` contains entry points for the code that implements the model algorithm. This file includes the rate scheduling code.
- `autosar_swc.h` declares model data structures and a public interface to the model entry points and data structures.
- `autosar_swc_private.h` contains local `define` constants and local data required by the model and subsystems.
- `autosar_swc_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_component.arxml`, `autosar_swc_datatype.arxml`, `autosar_swc_implementation.arxml`, and `autosar_swc_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')
```

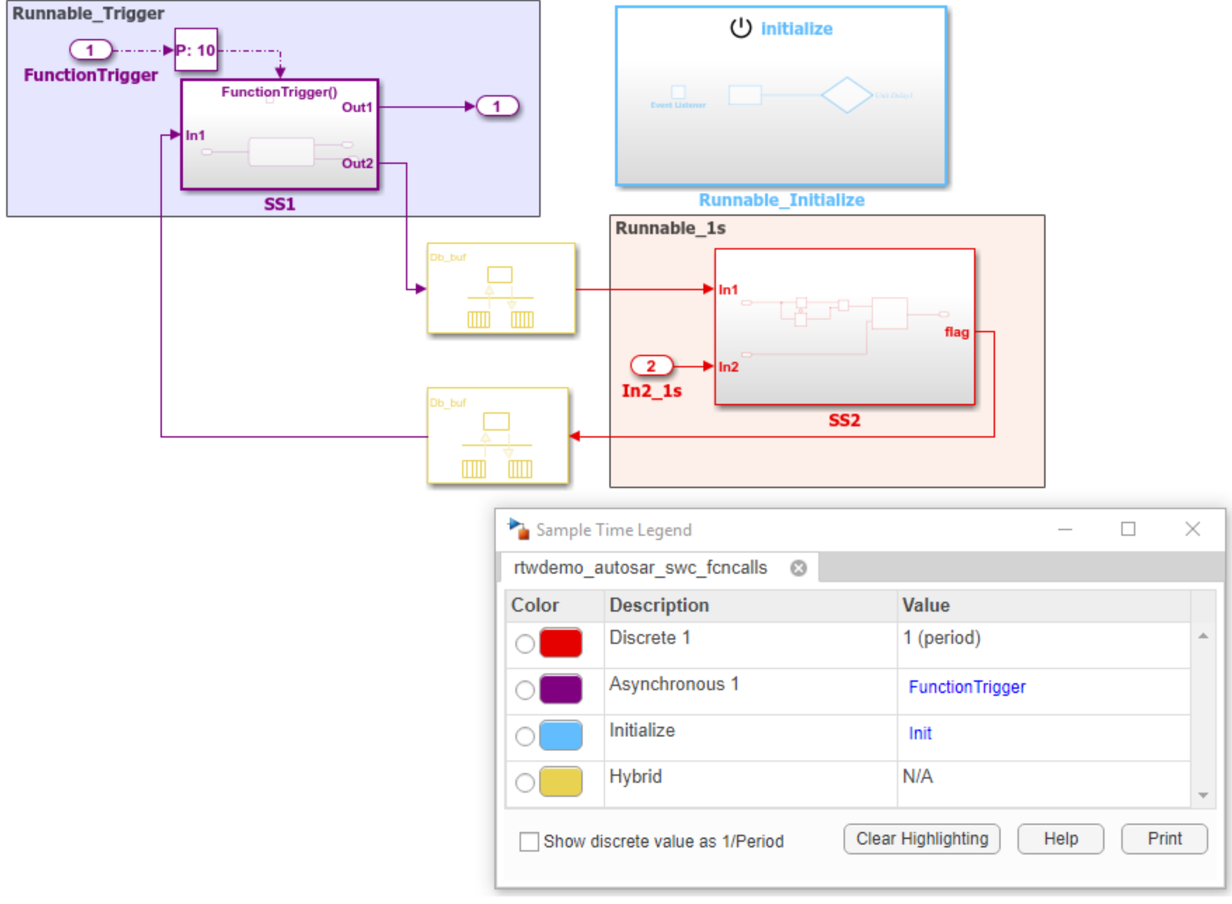
The model shows the implementation of an AUTOSAR atomic software component (ASWC). The model uses an asynchronous function-call runnable, Runnable_Trigger,

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, select **Display > Sample Time > 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 **Display > Sample Time > 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 are licensed for Simulink Coder and Embedded Coder, 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

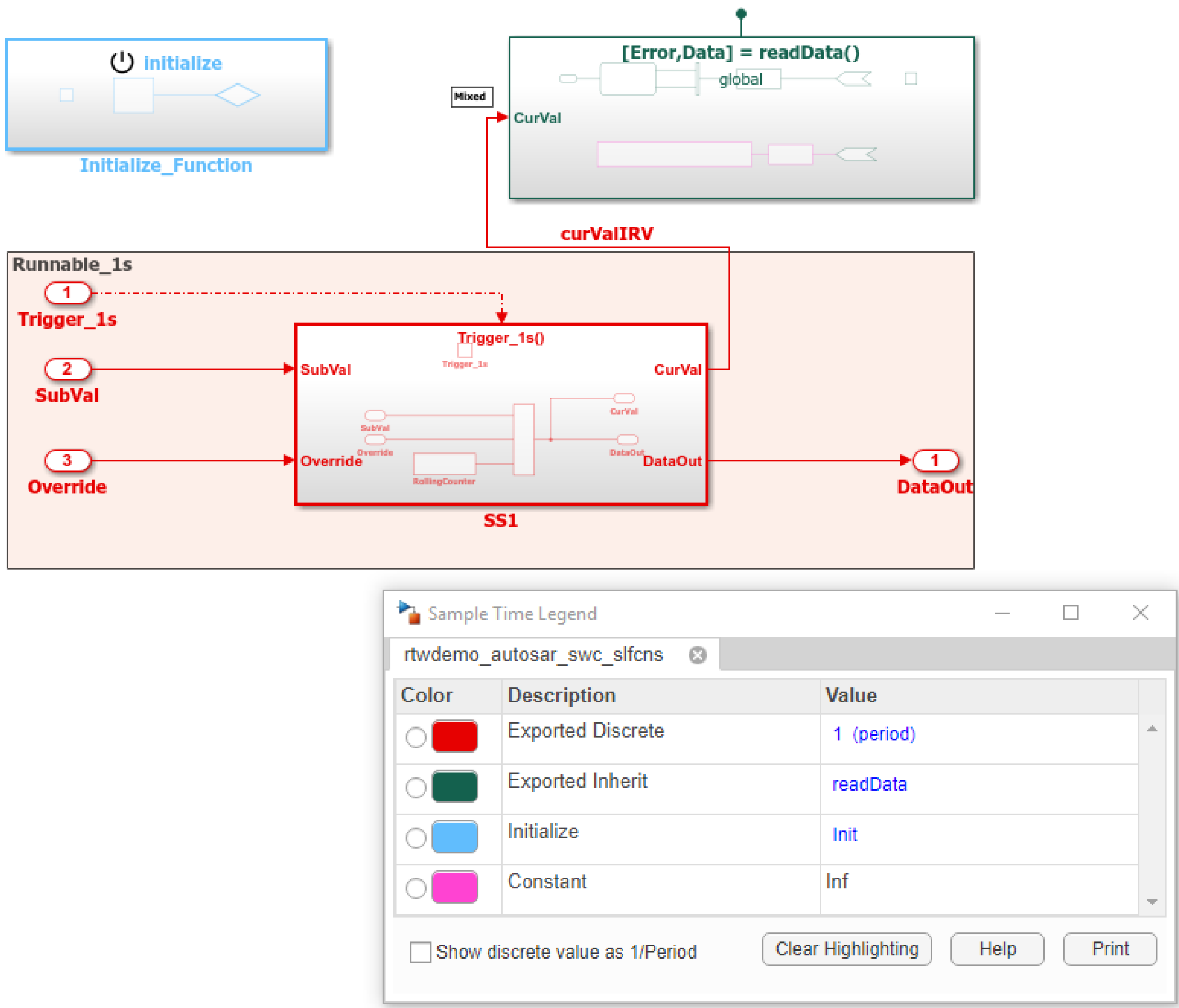
Open the example model autosar_swc_slfcns.slx.

```
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, select **Display > Sample Time > 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 clicking **Display > Sample Time > 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 are licensed for Simulink Coder and Embedded Coder, 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.c contains code for the Simulink function.
- readData_private.h contains local define constants and local data required by the function.
- readData.h declares data structures and a public interface for calling the 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:

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 Communication

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

In this section...

"About AUTOSAR Communication" on page 2-26

"Sender-Receiver Interface" on page 2-27

"Client-Server Interface" on page 2-28

"Mode-Switch Interface" on page 2-30

"Nonvolatile Data Interface" on page 2-35

"Parameter Interface" on page 2-35

"Trigger Interface" on page 2-36

"Service Interface (Adaptive Platform)" on page 2-36

About AUTOSAR Communication

AUTOSAR software components provide well-defined connection points, 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:

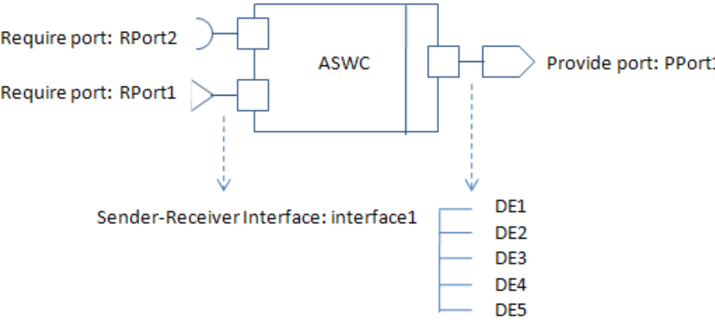
- Sender-Receiver
- Client-Server
- Mode-Switch
- Nonvolatile Data
- Parameter
- Trigger
- Service (Adaptive Platform)

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.

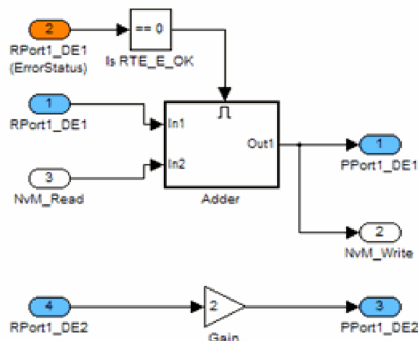
Sender-Receiver Interface

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 4-155.

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 AUTOSAR Dictionary and Code Mappings editor to specify the AUTOSAR settings for each inport and outport. For more information, see “Configure AUTOSAR Sender-Receiver Communication” on page 4-153.

Client-Server Interface

AUTOSAR allows client-server communication between:

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

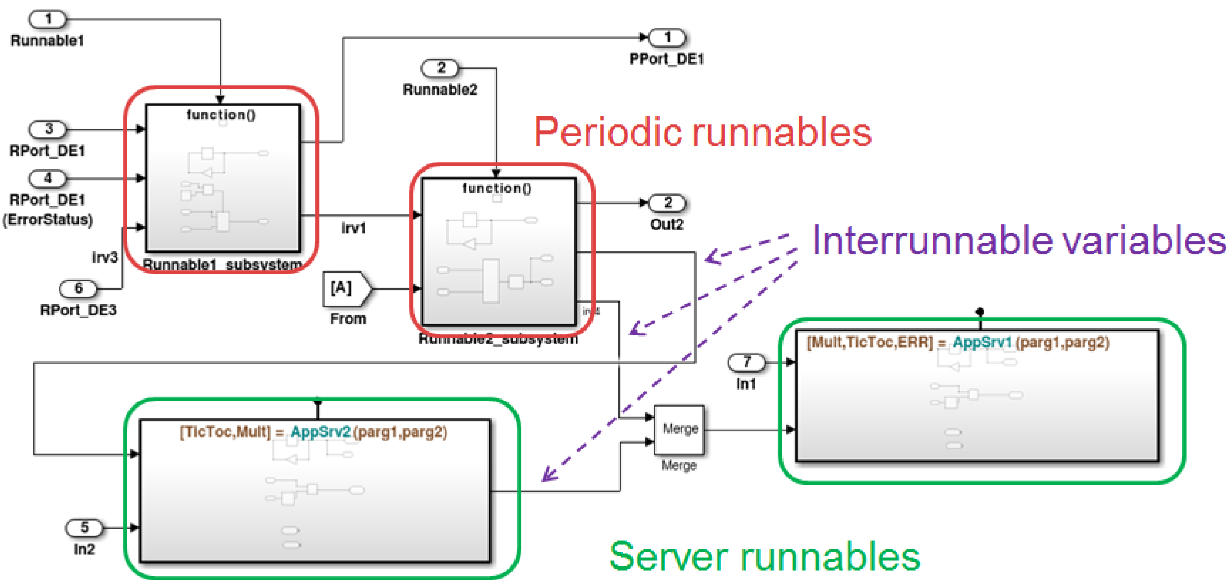
An AUTOSAR Client-Server Interface defines the interaction between a software component that *provides* the interface and a software component that *requires* the

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 and `arxml` code 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 and `arxml` code from the model.
4. Integrate the generated C code into a test framework for testing, for example, with SIL simulation. (Ultimately, the generated C and `arxml` code are integrated into the AUTOSAR Runtime Environment (RTE).)

For more information, see “Configure AUTOSAR Client-Server Communication” on page 4-197.

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 2-30
- “Application Mode Manager” on page 2-33

Mode User

To model an AUTOSAR mode user software component in Simulink:

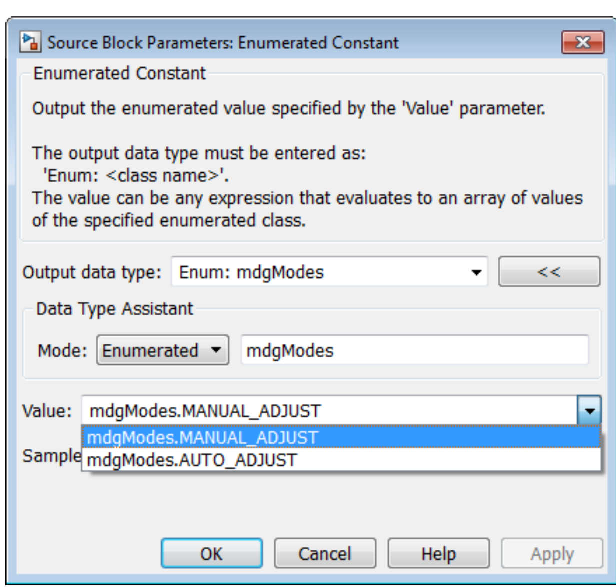
- Create an AUTOSAR mode-switch interface.
- Create an AUTOSAR mode receiver port and map it to a Simulink inport.
- For an initialization or other AUTOSAR runnable in the model, specify a mode-switch event to trigger the runnable.

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

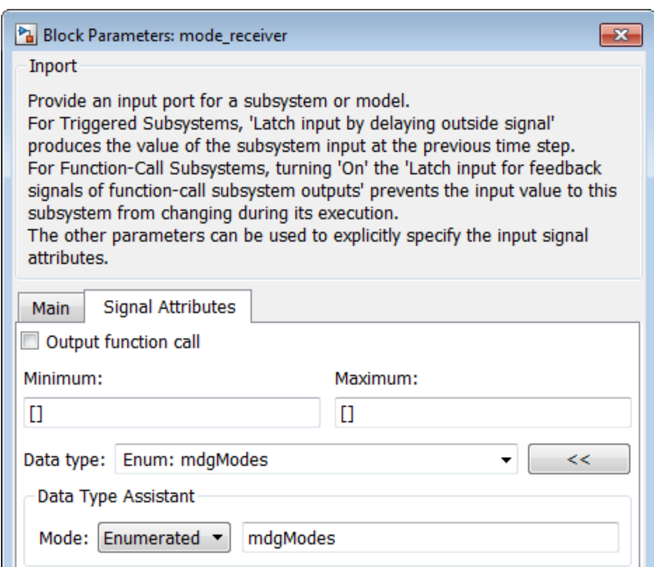
1. Declare a mode declaration group — a group of mode values — using Simulink enumeration. For example, you could create an enumerated type `mdgModes`, with

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'...
);
```

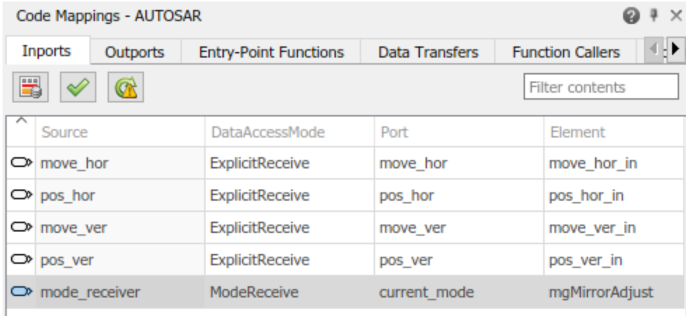


2 Apply the enumeration data type to a Simulink inport that represents an AUTOSAR mode-receiver port. In this Inport block dialog box, enumerated type `mdgModes` is specified as the inport data type.



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

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

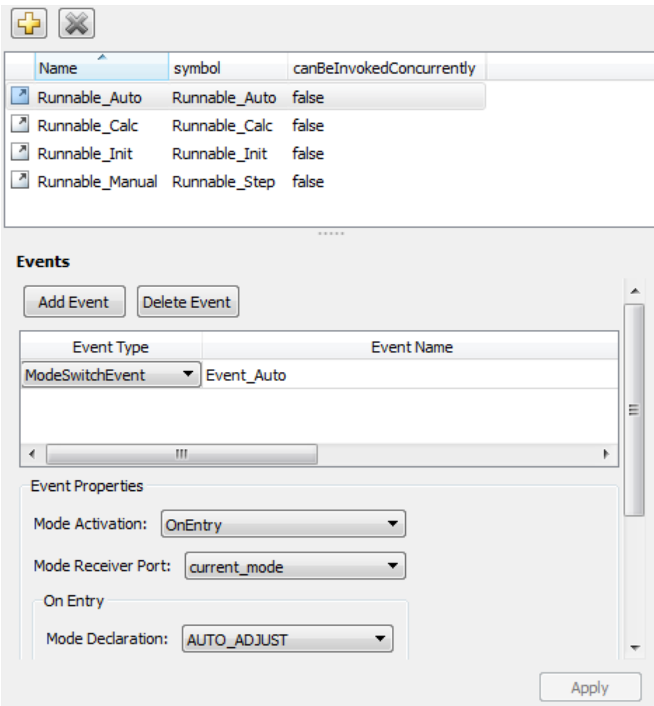


To specify a mode-switch event to trigger an initialize runnable or exported runnable, general steps can include:

1. To edit, add, or remove AUTOSAR mode-switch interfaces and mode-receiver ports, use 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 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.

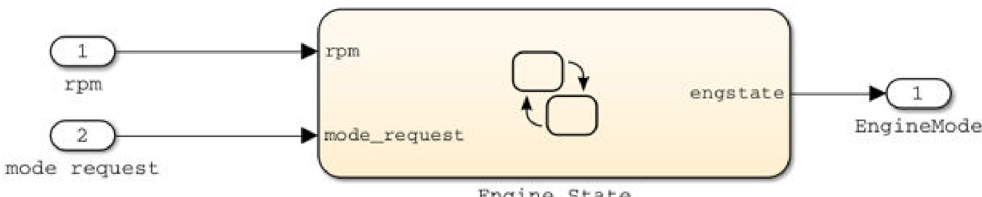


For more information, see “Configure AUTOSAR Mode-Switch Communication” on page 4-225.

Application Mode Manager

To model an application mode manager software component in Simulink, use an AUTOSAR mode sender port (as defined in AUTOSAR Release 4). 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 outport, which is mapped to an AUTOSAR mode sender port and a mode-switch (M-S) interface. The outport 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` code.
  - The software imports ModeSwitchPoints, ModeSwitchInterfaces, and ModeDeclarationGroups.
  - For each AUTOSAR provider port that references an M-S interface, the importer creates a root outport with `ModeSend` data access and with an AUTOSAR mode declaration group enumeration class.
  - The importer maps the model outport to an AUTOSAR mode sender port with an M-S interface.
- Create AUTOSAR mode-switch communication elements.
  - Create a model root outport, and set the outport 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 Code Mappings editor, set the outport data access mode to `ModeSend`, and map the outport to the AUTOSAR mode sender port.
- Generate `arxml` and C code for AUTOSAR mode sender ports and related AUTOSAR M-S communication elements.
  - The `arxml` code includes 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 4-225.

Nonvolatile Data Interface

AUTOSAR Release 4.0 introduced 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.

In Simulink, you can:

- Import AUTOSAR NV data interfaces and ports from `arxml` code.
- 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 2-27.
- Generate C and `arxml` code for AUTOSAR NV data interfaces and ports.

For more information, see "Configure AUTOSAR Nonvolatile Data Communication" on page 4-234.

Parameter Interface

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 parameter software component 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 Receiver for AUTOSAR Parameter Communication" on page 4-237.

Trigger Interface

AUTOSAR Release 4.0 introduced 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 4-240.

Service Interface (Adaptive Platform)

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++ namepaces.
- Create root-level inports and outports and map them to AUTOSAR required and provided ports and service interface events.
- If you are licensed for Simulink Coder and Embedded Coder, generate C++ code and `arxml` descriptions for AUTOSAR service communication.

For more information, see “Configure AUTOSAR Adaptive Service Communication” on page 4-245.

See Also

Related Examples

- “Configure AUTOSAR Provide-Require Port” on page 4-155
- “Configure AUTOSAR Client-Server Communication” on page 4-197
- “Configure AUTOSAR Mode-Switch Communication” on page 4-225
- “Configure AUTOSAR Nonvolatile Data Communication” on page 4-234
- “Configure Receiver for AUTOSAR Parameter Communication” on page 4-237
- “Configure Receiver for AUTOSAR External Trigger Event Communication” on page 4-240
- “Configure AUTOSAR Adaptive Service Communication” on page 4-245

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.

In this section...

"AUTOSAR Elements for Modeling Component Behavior" on page 2-38
"Runnables" on page 2-39
"Inter-Runnable Variables" on page 2-39
"System Constants" on page 2-40
"Per-Instance Memory" on page 2-41
"Static and Constant Memory" on page 2-42
"Shared and Per-Instance Parameters" on page 2-42

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:

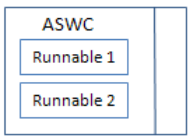
- Runnables and the events to which they respond
- Inter-runnable variables, used to communicate data between runnables in the same component
- System constants, used to specify system-level constant values that are available for reference in component algorithms
- Per-instance memory, used to specify 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 memory, for access to component internal parameters.

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.



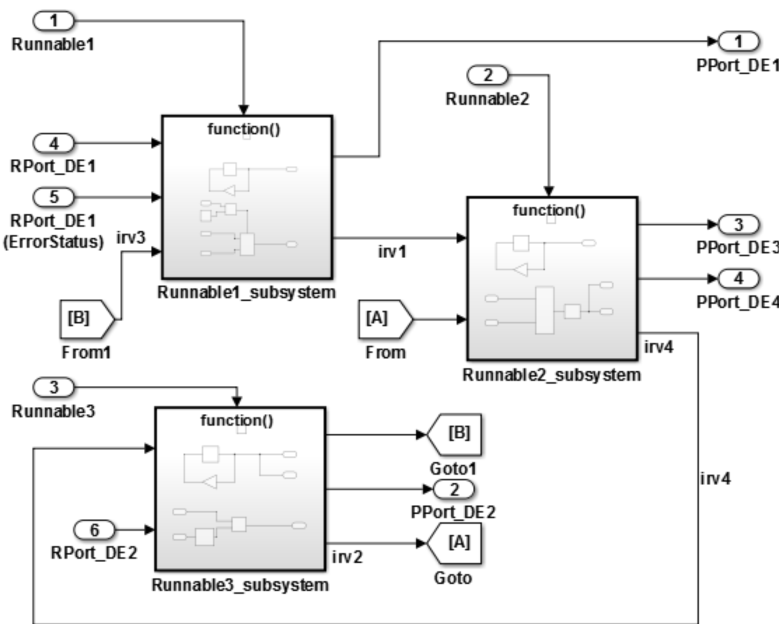
A component also can contain a single runnable, represented by a model, and can be single-rate or multirate.

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 4-277.

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.

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, using `AUTOSAR.Parameter` objects with **Storage class** set to `SystemConstant`.

You can then reference the AUTOSAR system constants in Simulink algorithms. For example, you could 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 code contains 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 used to represent a conditional value associated with variant condition logic, see "Configure Variants for AUTOSAR Runnable Implementations" on page 4-341.

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), introduced in AUTOSAR schema version 4.0, is described using AUTOSAR data types rather than C types. When exported in arxml code, arTypedPerInstanceMemory allows the use of measurement and calibration 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.

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 Code Mappings editor, **Signals/States** tab, to select a signal or state and map it to arTypedPerInstanceMemory. Property Inspector displays code and calibration attributes for the static memory, which you can modify.

- To use data stores, use Code Mappings editor, **Data Stores** tab, to select a data store and map it to `arTypedPerInstanceMemory`. Property Inspector displays code and calibration attributes for the static memory, which you can modify.

For more information, see “Configure AUTOSAR Per-Instance Memory” on page 4-295.

Static and Constant Memory

AUTOSAR supports static memory (StaticMemory) and constant memory (ConstantMemory) data, introduced in AUTOSAR schema version 4.0. 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` code, static memory and constant memory allow the use of measurement and calibration tools to monitor the internal memory data.

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

To model AUTOSAR constant memory in Simulink, use Code Mappings editor, **Parameters** tab, to select a parameter and map it to `ConstantMemory`. Property Inspector displays code and calibration attributes for the constant memory, which you can modify.

For more information, see “Configure AUTOSAR Static Memory” on page 4-300 and “Configure AUTOSAR Constant Memory” on page 4-305.

Shared and Per-Instance Parameters

AUTOSAR supports shared parameters (`SharedParameters`) and per-instance parameters (`PerInstanceParameters`) for use in 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 Simulink, you can import and export `arxml` descriptions of AUTOSAR shared and per-instance parameters. When exported in `arxml` code, shared and per-instance parameters allow the use of measurement and calibration 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 Code Mappings editor, **Parameters** tab, select the parameter and map it to parameter type `SharedParameter`. Property Inspector displays code and calibration attributes for the shared parameter, which you can modify.

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 Code Mappings editor, **Parameters** tab, select the parameter and map it to parameter `PerInstanceParameter`. Property Inspector displays code and calibration attributes for the per-instance parameter, which you can modify.

For more information, see “Configure AUTOSAR Shared or Per-Instance Parameters” on page 4-308.

See Also

Data Store Memory

Related Examples

- “Configure AUTOSAR Runnables and Events” on page 4-277
- “Configure Variants for AUTOSAR Runnable Implementations” on page 4-341
- “Configure AUTOSAR Per-Instance Memory” on page 4-295
- “Configure AUTOSAR Static Memory” on page 4-300
- “Configure AUTOSAR Constant Memory” on page 4-305
- “Configure AUTOSAR Shared or Per-Instance Parameters” on page 4-308

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. Components:

- Enable or disable an AUTOSAR port or runnable.
- Vary the array size of an AUTOSAR port.
- Vary the implementation of an AUTOSAR runnable.
- 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 array size of an AUTOSAR port, use Simulink symbolic dimensions.
  - To vary the implementation of an AUTOSAR runnable, use Variant Subsystem blocks.
- Resolve modeled variants by using predefined variants and system constant value sets imported from `arxml` files.

In this section...
“Variants for Ports and Runnables” on page 2-44
“Variants for Array Sizes” on page 2-45
“Variants for Runnable Implementations” on page 2-46
“Predefined Variants and System Constant Value Sets” on page 2-46

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` code.
- Export AUTOSAR ports and runnables with variation points.

For more information, see “Configure Variants for AUTOSAR Ports and Runnables” on page 4-335.

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 array size for an AUTOSAR element, reference an `AUTOSAR.Parameter` data object.
- Modify array size values in system constants and simulate the model, without regenerating code for simulation.
- Generate C and `arxml` code with symbols corresponding to variant array sizes.

For more information, see "Configure Variants for AUTOSAR Array Sizes" on page 4-339.

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 4-341.

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` code.

For more information, see “Control AUTOSAR Variants with Predefined Value Combinations” on page 4-344.

See Also

Related Examples

- “Configure Variants for AUTOSAR Ports and Runnables” on page 4-335
- “Configure Variants for AUTOSAR Array Sizes” on page 4-339
- “Configure Variants for AUTOSAR Runnable Implementations” on page 4-341
- “Control AUTOSAR Variants with Predefined Value Combinations” on page 4-344

More About

- “Variant Systems” (Embedded Coder)
- “AUTOSAR Component Configuration” on page 4-3

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 measurement and calibration tools.

In this section...
“AUTOSAR Calibration Parameters” on page 2-48
“Calibration Parameters for STD_AXIS and COM_AXIS Lookup Tables” on page 2-48

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 AUTOSAR code perspective and use Code Mappings editor, **Parameters** tab. Use the Property Inspector to configure parameter code and calibration attributes. For more information, see “Map Model Workspace Parameters to AUTOSAR Component Internal Parameters” on page 4-75.

Calibration Parameters for STD_AXIS and COM_AXIS Lookup Tables

You can model standard axis (STD_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 measurement and calibration 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 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 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 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` code defines 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 and COM_AXIS lookup tables and map them to AUTOSAR parameters. You map lookup table objects to AUTOSAR parameters using 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 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 AUTOSAR code perspective and use Code Mappings editor, **Parameters** tab. Use the Property Inspector to configure parameter code and calibration attributes. For more information, see “Map Model Workspace Parameters to AUTOSAR Component Internal Parameters” on page 4-75

- In the Simulink 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 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 Measurement and Calibration" on page 4-250.

See Also

Curve | Curve Using Prelookup | Map | Map Using Prelookup | Prelookup | `Simulink.Breakpoint` | `Simulink.LookupTable` | `getParameter` | `mapParameter`

Related Examples

- "Map Model Workspace Parameters to AUTOSAR Component Internal Parameters" on page 4-75
- "Configure Lookup Tables for AUTOSAR Measurement and Calibration" on page 4-250

More About

- "Code Generation with AUTOSAR Code Replacement Library" on page 5-20

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.

In this section...

"About AUTOSAR Data Types" on page 2-52
"Enumerated Data Types" on page 2-54
"Structure Parameters" on page 2-55
"Release 2.x and 3.x Data Types" on page 2-55
"Release 4.x Data Types" on page 2-55
"CompuMethod Categories for Data Types" on page 2-59

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:

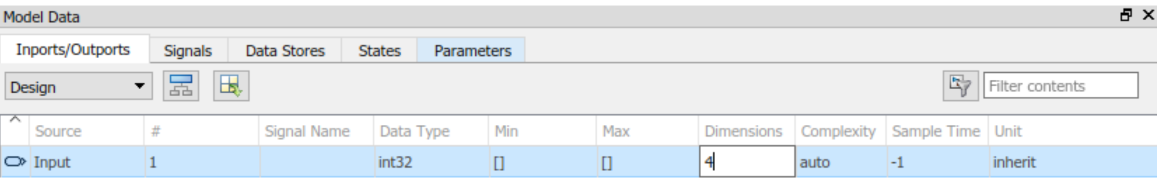
- Platform (primitive) data types, which allow a direct mapping to C intrinsic types.
- Composite data types, which map to C arrays and structures.

To model AUTOSAR data types, use corresponding Simulink data types.

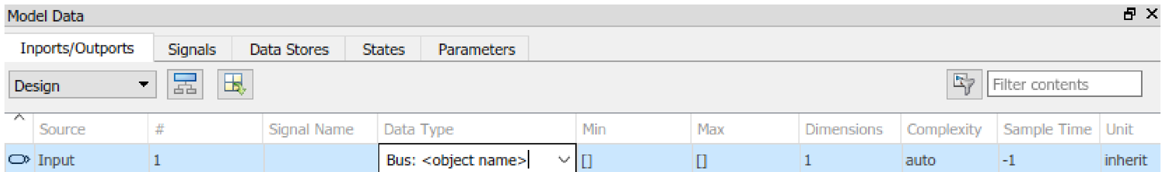
AUTOSAR Data Type		Simulink Data Type
R4.x Platform Type	**R2.x/3.x Primitive Type**	
`boolean`	`Boolean`	`boolean`
`float32`	`Float`	`single`
`float64`	`Double`	`double`
`sint8`	`Sint8`	`int8`
`sint16`	`Sint16`	`int16`
`sint32`	`Sint32`	`int32`
`uint8`	`Uint8`	`uint8`
`uint16`	`Uint16`	`uint16`
`uint32`	`Uint32`	`uint32`

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 Outport blocks, use the Model Data Editor. In the model, select **View** > **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.

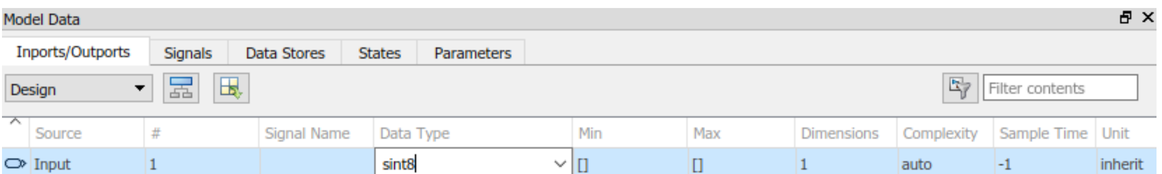
The following figure shows how to specify a wide signal, which corresponds to an AUTOSAR composite array.



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 Outport 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" (Simulink). 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.

Release 2.x and 3.x Data Types

The following table shows how the software translates AUTOSAR R2.x and R3.x data types to Simulink data types. For information about Release 4.x data types, see "Release 4.x Data Types" on page 2-55.

AUTOSAR	Simulink

Primitive types (excluding fixed point), for example, `myInt16`
 	
Covers Boolean, integer, real
	`myInt16 = Simulink.AliasType;`
	`myInt16.BaseType = 'int16';`
	`myInt16.HeaderFile = 'Rte_Type.h';`

Primitive type (fixed point), for example, `myFixPt`	`myFixPt = Simulink.NumericType;`
	`myFixPt.DataTypeMode = …;`
	`myFixPt.IsAlias = true;`
	`myFixPt.HeaderFile = 'Rte_Type.h';`

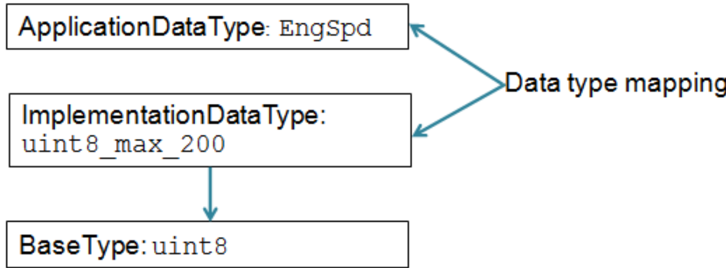
Enumerations, for example, `myEnum`	`Simulink.defineIntEnumType('myEnum',...`
	` {'Red','Green','Blue'},...`
	` [1;2;3],…);`

Record types, for example, `myRecord`	`myRecord = Simulink.Bus;`

Release 4.x Data Types

AUTOSAR Release 4.0 introduced a new 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 R4.x compliant 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 schema 4.x elements that relate to data types, including application data types, software base types, data type mapping sets, system constants, and units.
  - Schema 4.x implementation data types are aggregated in the main data types package.

  For more information, see “Configure AUTOSAR Packages” on page 4-138.
- 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 2-59.

For AUTOSAR R4.x 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 Release 4.x Data Types” on page 4-314.

- “R4.x Data Types in Simulink Originated Workflow” on page 2-57
- “R4.x Data Types in Round-Trip Workflow” on page 2-58
- “Application Data Type Physical Constraint Mapping” on page 2-59

R4.x 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.

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, `myInt16`
 	
Covers Boolean, integer, real
 	
`myInt16 = Simulink.AliasType;`
`myInt16.BaseType = 'int16';`
	`<IMPLEMENTATION-DATA-TYPE>`
	`<SHORT-NAME>myInt16</SHORT-NAME>`
	`<CATEGORY>VALUE</CATEGORY>`
	…
		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

R4.x 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 3-27 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:

```
ImplDT1
```

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. Specifically:

- If you import `arxml` files, `PhysConstr` values on `ApplicationDataTypes` in the `arxml` files are imported to `Min` and `Max` values on the corresponding Simulink data objects and root-level I/O signals.
- When you export `arxml` code from a model, the `Min` and `Max` values specified on Simulink data objects and root-level I/O signals are exported to the corresponding `ApplicationDataType` `PhysConstrs` in the `arxml` files.

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 measurement and calibration.

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 measurement and calibration 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_TEXTABLE	Yes	No
Floating-point	IDENTICAL SCALE_LINEAR_AND_TEXTABLE	Yes	No
Integer	IDENTICAL SCALE_LINEAR_AND_TEXTABLE	Yes	No

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 4-320.

See Also

Related Examples

- "Organize Data into Structures in Generated Code" (Embedded Coder)
- "Application Data Type Physical Constraint Mapping" on page 2-59

- “Configure AUTOSAR Release 4.x Data Types” on page 4-314
- “Automatic AUTOSAR Data Type Generation” on page 4-318
- “Configure AUTOSAR CompuMethods” on page 4-320

More About

- “AUTOSAR Data Types”

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.

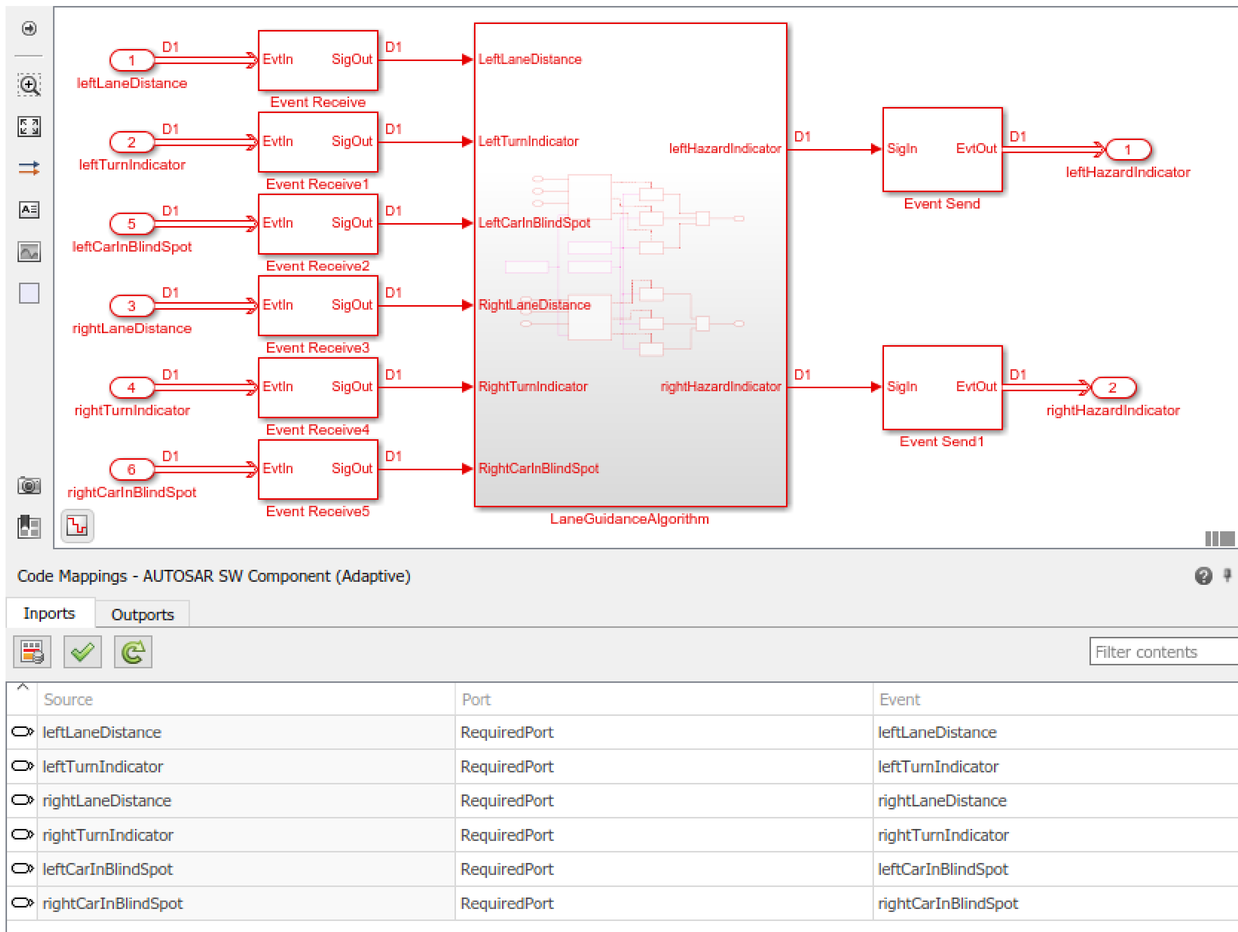
To model an AUTOSAR adaptive software component in Simulink, you start with a model that contains an automotive algorithm. From that model, you generate an AUTOSAR Dictionary that defines service interfaces, and an AUTOSAR code perspective that maps Simulink model elements to AUTOSAR component elements. 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.

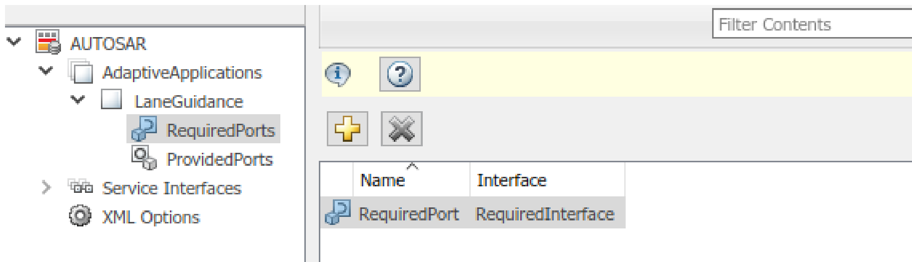
1. Open a Simulink model that either is empty or contains a functional algorithm.
2. Using the Model Configuration Parameters dialog box, configure the model for adaptive AUTOSAR code generation. Set **System target file** to `autosar_adaptive.tlc`.
3. Develop the model algorithmic content for use in 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 algorithmic model to an AUTOSAR adaptive software component. For example, in the model window, select **Code > C/C++ Code > Configure Model in Code Perspective**. An AUTOSAR Component Quick Start dialog box opens. When a Quick Start pane offers default or delayed component creation, select **Create default mapping**. To generate an AUTOSAR code perspective and an AUTOSAR Dictionary, click the Quick Start **Finish** button.



6 Using 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 a model class header file.
- C++ files that implement a main program module.
- `CMakeLists.txt` file that supports CMake generation of executables.

For more information, see “Configure AUTOSAR Adaptive Software Components” on page 4-111.

See Also

Event Receive | Event Send

Related Examples

- “Configure AUTOSAR Adaptive Software Components” on page 4-111
- “Create and Configure AUTOSAR Adaptive Software Component” on page 3-17
- “Map AUTOSAR Adaptive Elements for Code Generation” on page 4-103
- “Configure AUTOSAR Adaptive Elements and Properties” on page 4-87
- “Configure AUTOSAR Adaptive Service Communication” on page 4-245
- “Configure AUTOSAR Adaptive Code Generation” on page 5-18

More About

- “Comparison of AUTOSAR Classic and Adaptive Platforms” on page 1-4

Model AUTOSAR Basic Software Service Calls

The AUTOSAR standard defines important services as part of Basic Software (BSW) that runs in the AUTOSAR Runtime Environment (RTE). Examples include the NVRAM Manager (NvM) and the Diagnostic Event Manager (Dem). 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 and DiagnosticMonitorCaller.
- 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.

To configure the added blocks in the AUTOSAR software component, click the **Update** button in 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 6-13 and "Configure Calls to AUTOSAR NVRAM Manager Service" on page 6-20.

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 and Dem 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 and Dem service operations. To support simulation of component

calls to the NvM and Dem 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 6-27 and “Simulate AUTOSAR Basic Software Services and Run-Time Environment” on page 6-32.

See Also

Diagnostic Service Component | DiagnosticInfoCaller | DiagnosticMonitorCaller | NVRAM Service Component | NvMAdminCaller | NvMServiceCaller

Related Examples

- “Configure Calls to AUTOSAR Diagnostic Event Manager Service” on page 6-13
- “Configure Calls to AUTOSAR NVRAM Manager Service” on page 6-20
- “Configure AUTOSAR Basic Software Service Implementations for Simulation” on page 6-27
- “Simulate AUTOSAR Basic Software Services and Run-Time Environment” on page 6-32

More About

- “Model AUTOSAR Communication” on page 2-26

AUTOSAR Component Creation

- "Create AUTOSAR Software Component in Simulink" on page 3-2
- "Create and Configure AUTOSAR Software Component" on page 3-9
- "Create and Configure AUTOSAR Adaptive Software Component" on page 3-17
- "AUTOSAR arxml Importer" on page 3-25
- "Import AUTOSAR Software Component" on page 3-27
- "Import AUTOSAR Software Component with Multiple Runnables" on page 3-31
- "Import AUTOSAR Component to Simulink" on page 3-32
- "Import AUTOSAR Software Composition with Atomic Software Components (Classic Platform)" on page 3-38
- "Import AUTOSAR Software Component Updates" on page 3-39
- "Round-Trip Preservation of AUTOSAR XML File Structure and Element Information" on page 3-46
- "Reuse AUTOSAR Element Descriptions" on page 3-49
- "Reuse AUTOSAR Elements in Component Model" on page 3-52
- "Limitations and Tips" on page 3-58

Create AUTOSAR Software Component in Simulink

To create an initial Simulink representation of an AUTOSAR software component, you either import an AUTOSAR software component description from `arxml` files into a new Simulink model or create an AUTOSAR software component using an existing Simulink model. To create an AUTOSAR software component using 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 component development.

If you are licensed for Simulink Coder and Embedded Coder, to prepare your AUTOSAR model for code generation, you can use Embedded Coder Quick Start. Select **Code > C/C++ Code > Embedded Coder Quick Start**. 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 3-6

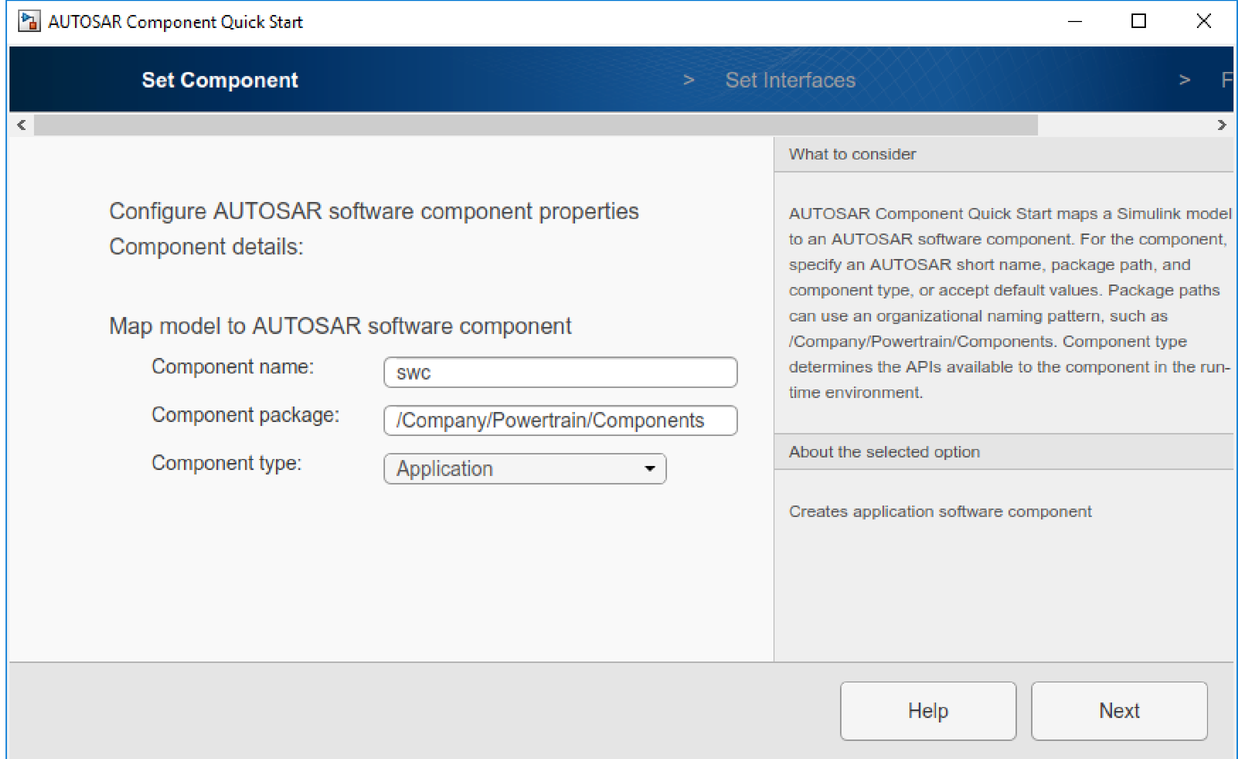
Create Mapped AUTOSAR Component with Quick Start

To create a mapped AUTOSAR software component using 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. To open AUTOSAR Component Quick Start, select **Code > C/C++ Code > Configure Model in Code Perspective**. The AUTOSAR Component Quick Start opens.

3 To configure the model for AUTOSAR software component development, work through the quick-start procedure.



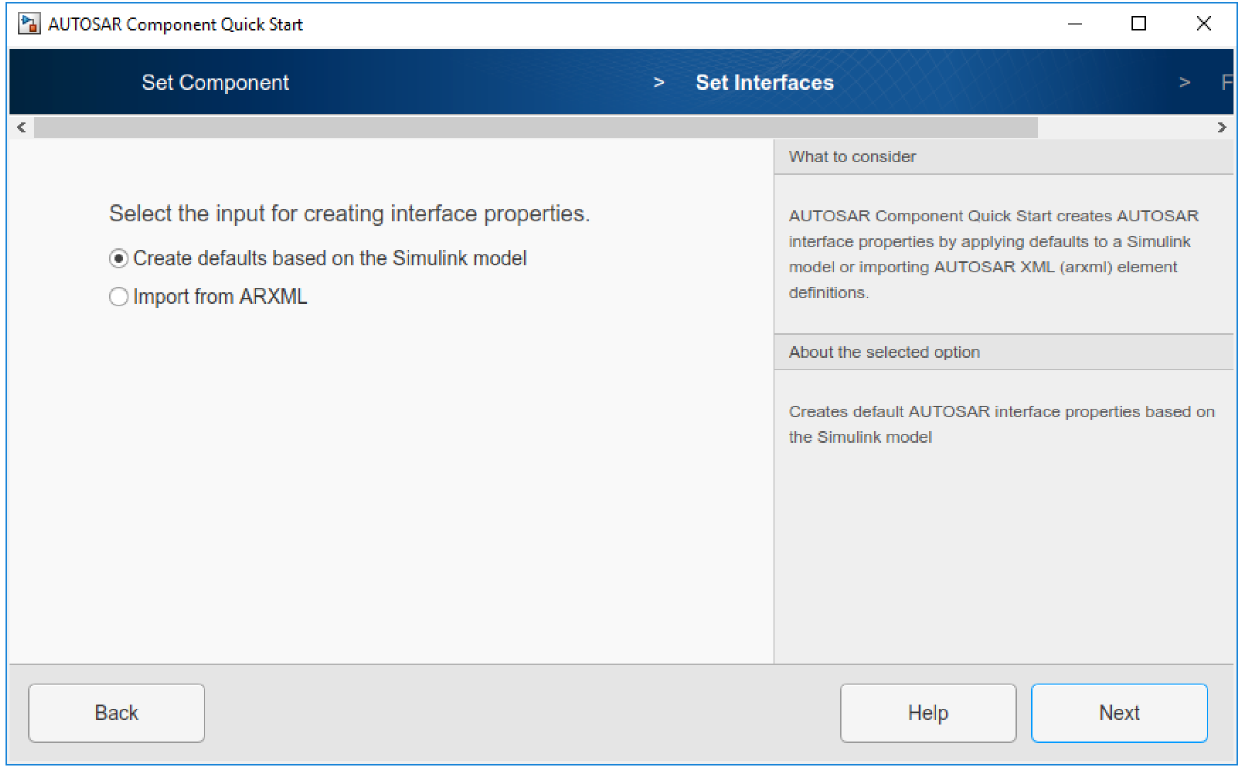
In the **Set Component** pane:

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

  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 3-38.
- For an Adaptive Platform software component, specify an AUTOSAR short name and package path.

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 shared AUTOSAR element definitions (not AUTOSAR software component descriptions). For more information, see “Reuse AUTOSAR Element Descriptions” on page 3-49.

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

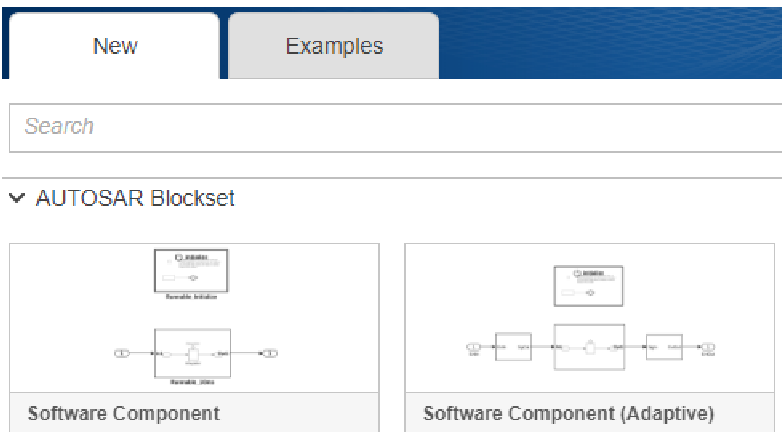
The Simulink Start Page provides AUTOSAR Blockset model templates as a starting point for AUTOSAR software component development. You can select either a Classic Platform or Adaptive Platform template and click **Create Model**.

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, you use AUTOSAR Component Quick Start (or potentially Embedded Coder Quick Start) to map the model to an AUTOSAR software component. For example:

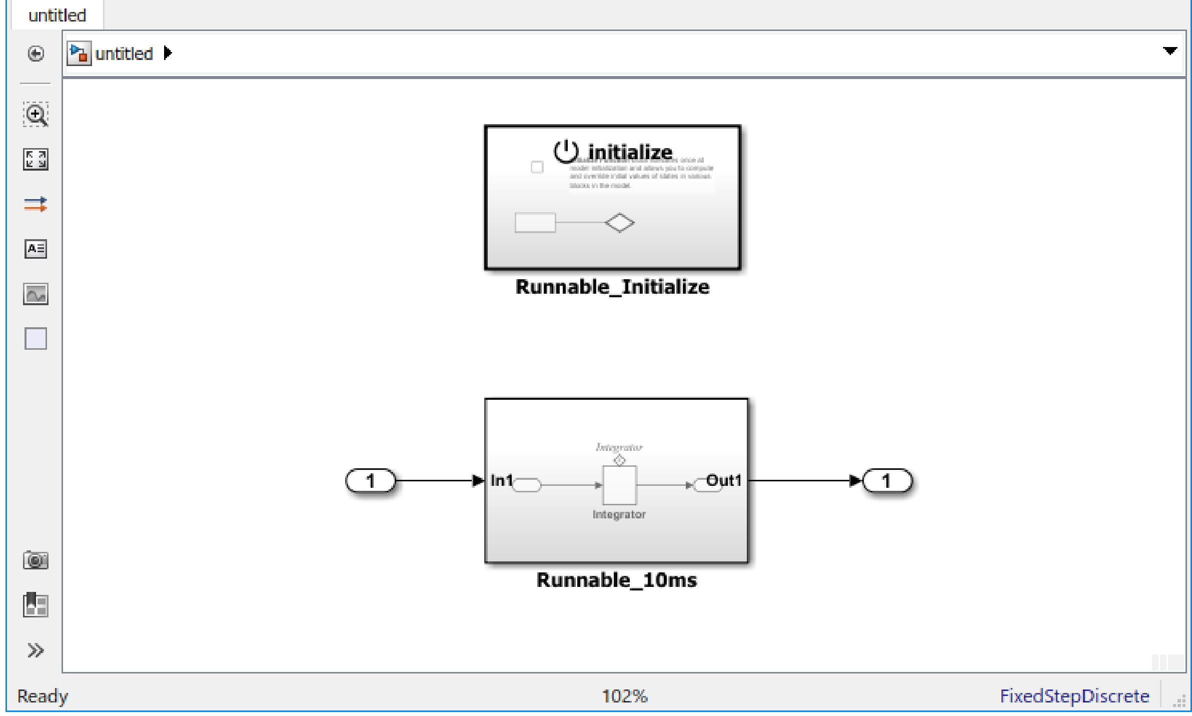
1. Open the Simulink Start Page. For example:

   - Enter the MATLAB command `simulink`.
   - In the MATLABtoolstrip, select **Home** > **New** > **Simulink Model**.
   - In a model window, select **File** > **New** > **Model**.

   The Start Page opens. Scroll down to **AUTOSAR Blockset** and expand the product row.

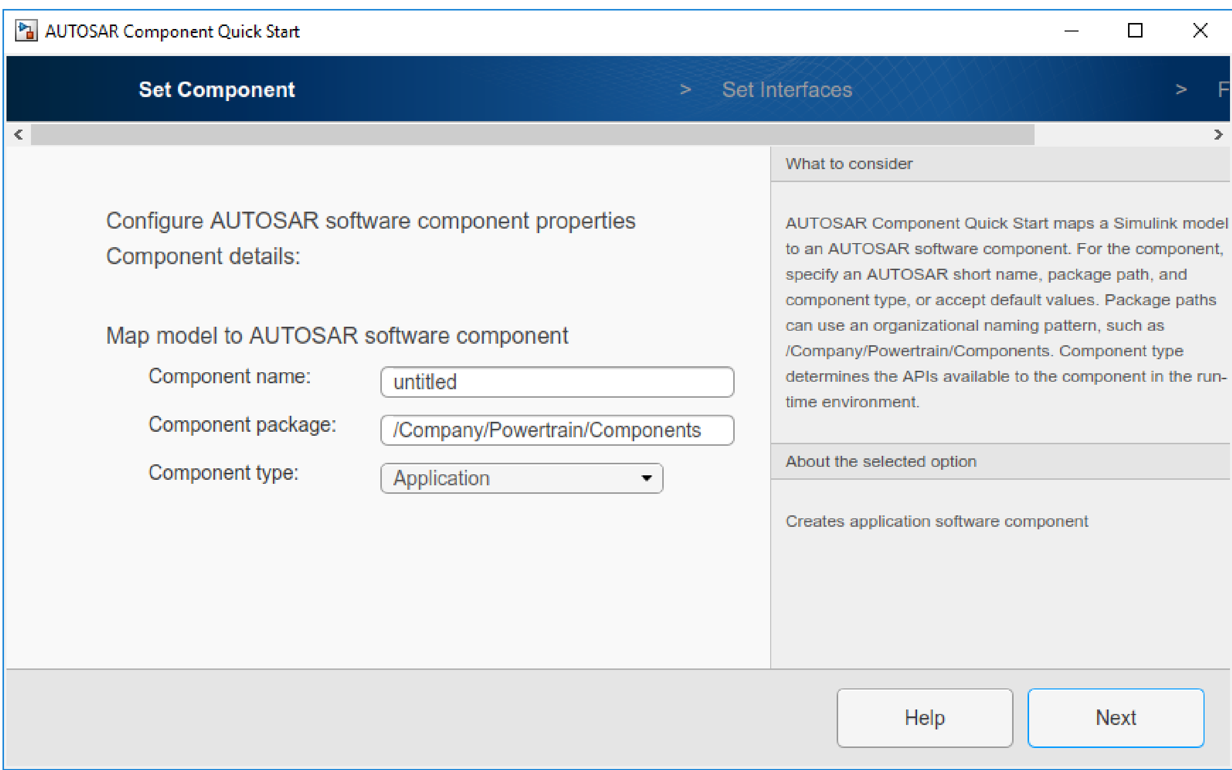
   

2. Hover over the template you want to use and click **Create Model**. A model based on the template opens. (The Simulink Start Page closes.)



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

3 Explore the model and refine the configuration to match your requirements. Optionally, you can develop the component behavior. To map the model to an AUTOSAR software component, use AUTOSAR Component Quick Start. In the model window, select **Code** > **C/C++ Code** > **Configure Model in Code Perspective**. 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

Create and Configure AUTOSAR Software Component

Create a mapped Simulink® representation of an AUTOSAR software component from a 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.)

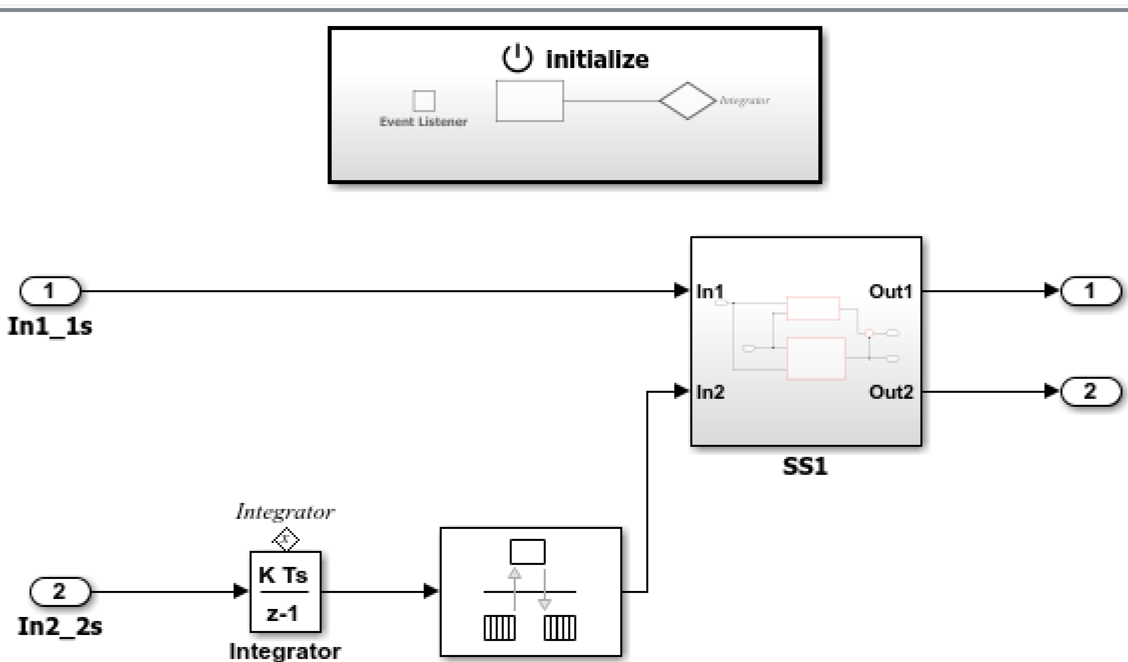
Create AUTOSAR Software Component in Simulink

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

- Import an AUTOSAR software component description from `arxml` files into a new Simulink model. (See example “Import AUTOSAR Component to Simulink” on page 3-32.)
- Create an AUTOSAR software component using an existing Simulink model.

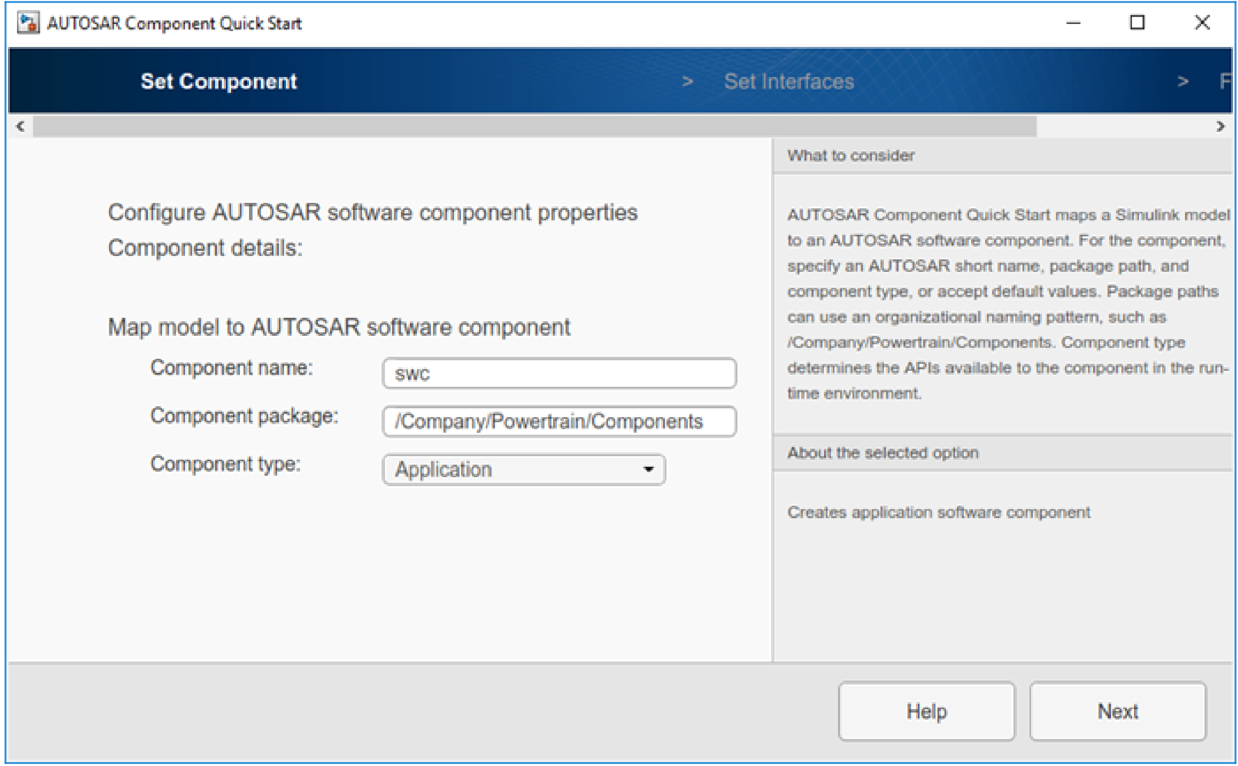
To create an AUTOSAR software component using an existing model, first open a Simulink component model for which an AUTOSAR software component is not mapped. This example uses AUTOSAR example model `swc`.

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



In the model window, open the Configuration Parameters dialog box, **Code Generation** pane, and set the system target file to `autosar.tlc`. Click **OK**.

To configure the model as a mapped AUTOSAR software component, open AUTOSAR Component Quick Start. Select **Code** > **C/C++ Code** > **Configure Model in Code Perspective**. The AUTOSAR Component Quick Start opens.

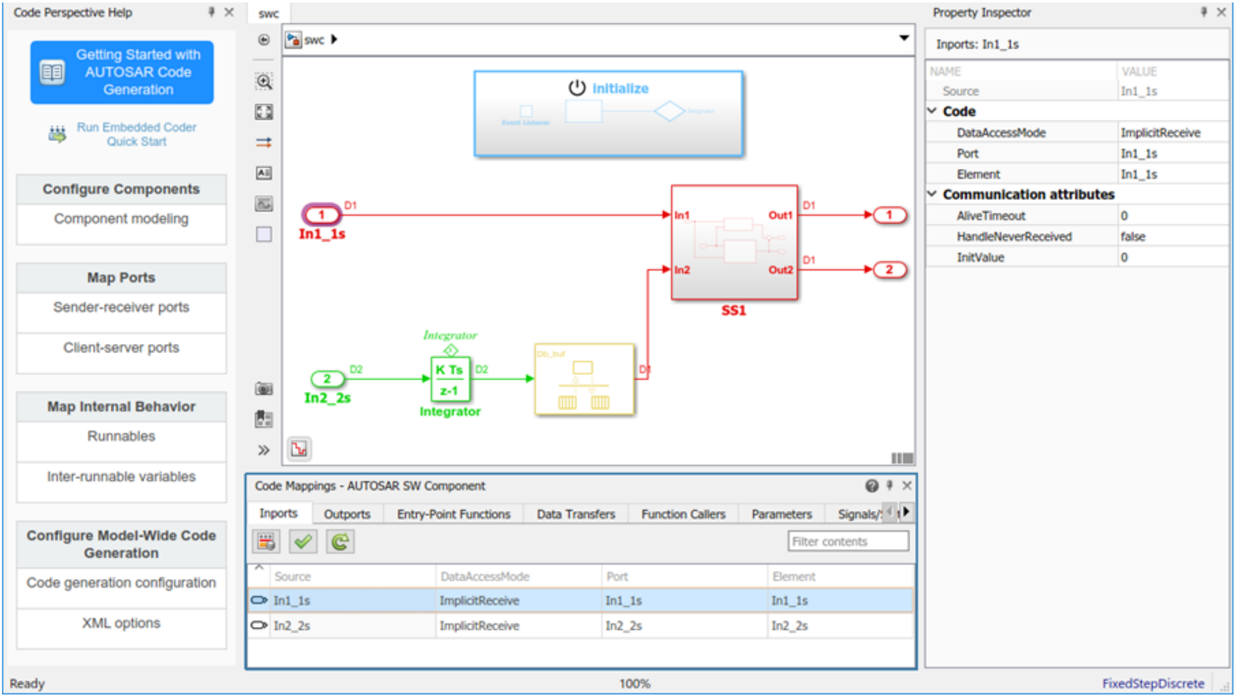


To configure the model for AUTOSAR software component development, work through the quick-start procedure. This example accepts default settings for the options in the Quick Start **Set Component** and **Set Interfaces** panes.

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

Configure AUTOSAR Software Component in Simulink

After you create an initial Simulink representation of an AUTOSAR software component, AUTOSAR code perspective displays the model, a Property Inspector panel, and directly below the model, the Code Mappings editor. To additionally display a help panel, in the model window, select **View > Code Perspective Help**.



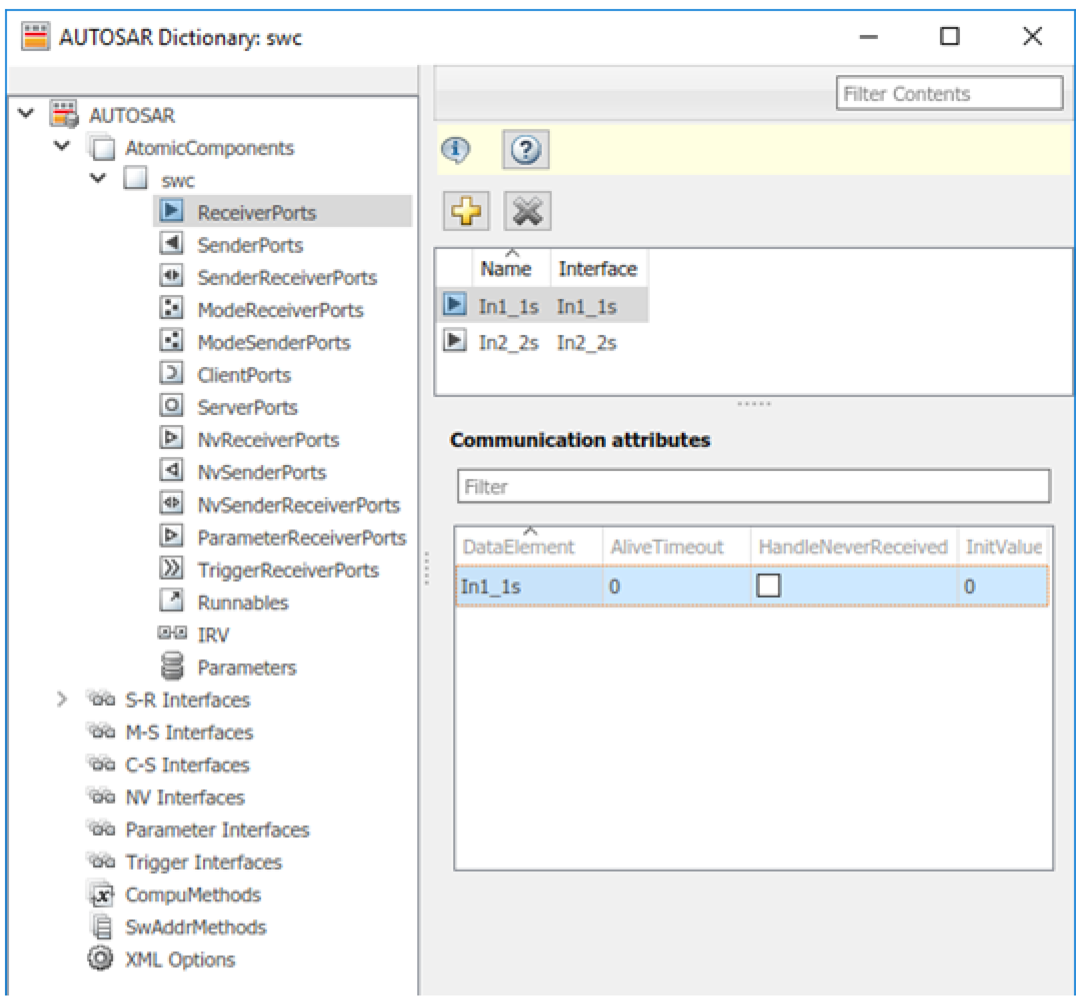
Next you use the Code Mappings editor and the AUTOSAR Dictionary to further develop the AUTOSAR component.

Code Mappings editor displays model inports, outports, entry-point functions, data transfers, 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 ports, runnable entities, 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, and Property Inspector displays its code and communication attributes.

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 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 are licensed for Simulink Coder and Embedded Coder, 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 click the **Code** menu and select **C/C++ Code > Build Model**.

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.

Code Generation Report

Find: Match Case

Contents

Summary

Subsystem Report

Code Interface Report

Traceability Report

Static Code Metrics Report

Code Replacements Report

Coder Assumptions

Generated Code

[-] **Model files**

swc.c

swc.h

swc_private.h

swc_types.h

[+] **Utility files (1)**

[+] **Interface files (4)**

[+] **RTE files (6)**

```
25  /* Model step function for TID0 */
26  void Runnable_Step(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_Step_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_Step_In1_1s_In1_1s();
40
41    /* End of Outputs for SubSystem: '<S2>/SS2' */
42
43    /* Outputs for Atomic SubSystem: '<S2>/SS1' */
44    /* Outport: '<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_Step_Out1_Out1(5.0 *
52      (Rte_IRead_Runnable_Step_In1_1s_In1_1s() + 3.0 * rtb_RateTransition) +
53      rtb_Sum_n);
54
55    /* End of Outputs for SubSystem: '<S2>/SS1' */
56
57    /* Outport: '<Root>/Out2' */
58    Rte_IWrite_Runnable_Step_Out2_Out2(rtb_Sum_n);
59  }
```

OK Help

Related Links

- “AUTOSAR Component Configuration” on page 4-3
- “Code Generation”

- "AUTOSAR Blockset"

Create and Configure AUTOSAR Adaptive Software Component

Create a mapped Simulink® representation of an AUTOSAR adaptive software component from a model.

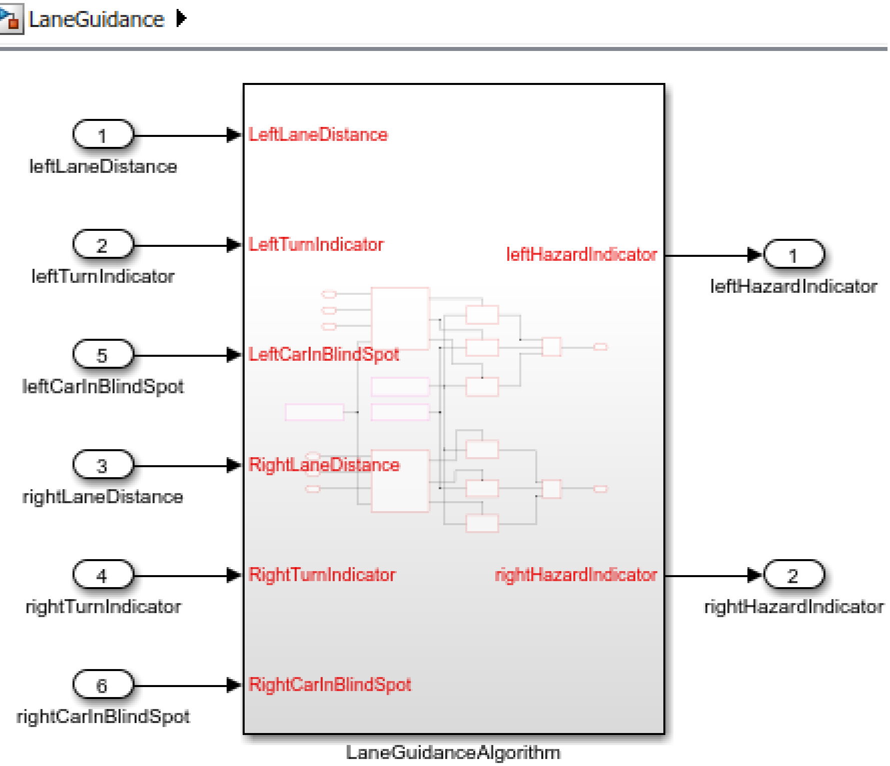
AUTOSAR Blockset software supports AUTomotive Open System ARchitecture (AUTOSAR), an open and standardized automotive software architecture. Automobile manufacturers, suppliers, and tool developers jointly develop AUTOSAR components. To develop AUTOSAR adaptive components in Simulink, follow this general 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 use an existing Simulink model. 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, open the Configuration Parameters dialog box, **Code Generation** pane, and 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:

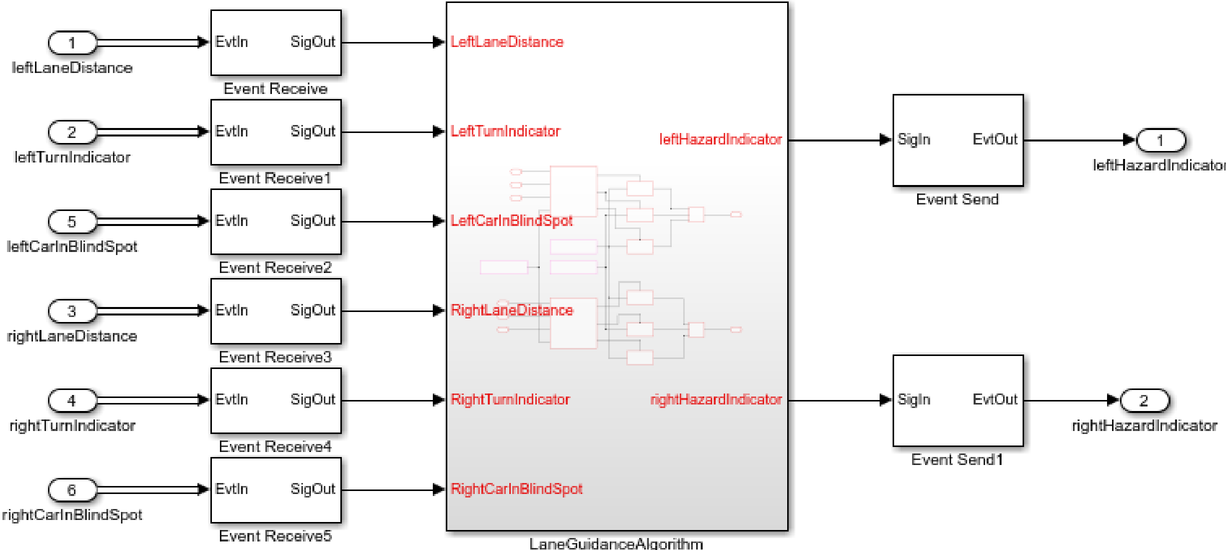
- An algorithm that performs tasks in response to received events
- Required and provided ports, each associated with a service interface
- Service interfaces, with associated events and associated namespaces

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

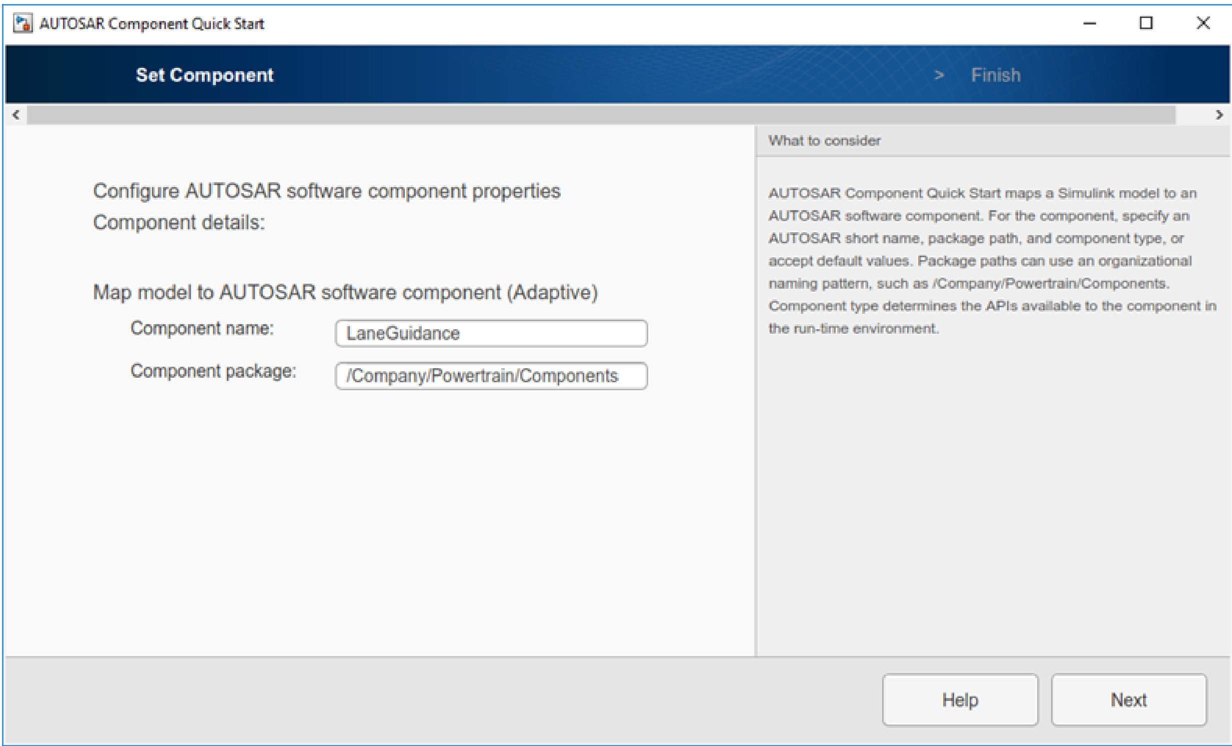
- After each root inport, add an Event Receive block, which converts an input event to a signal while preserving the signal values and data type.

- 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 AUTOSAR example model `autosar_LaneGuidance`.)



To configure the model as a mapped AUTOSAR adaptive software component, open AUTOSAR Component Quick Start. Select **Code** > **C/C++ Code** > **Configure Model in Code Perspective**. The AUTOSAR Component Quick Start opens.

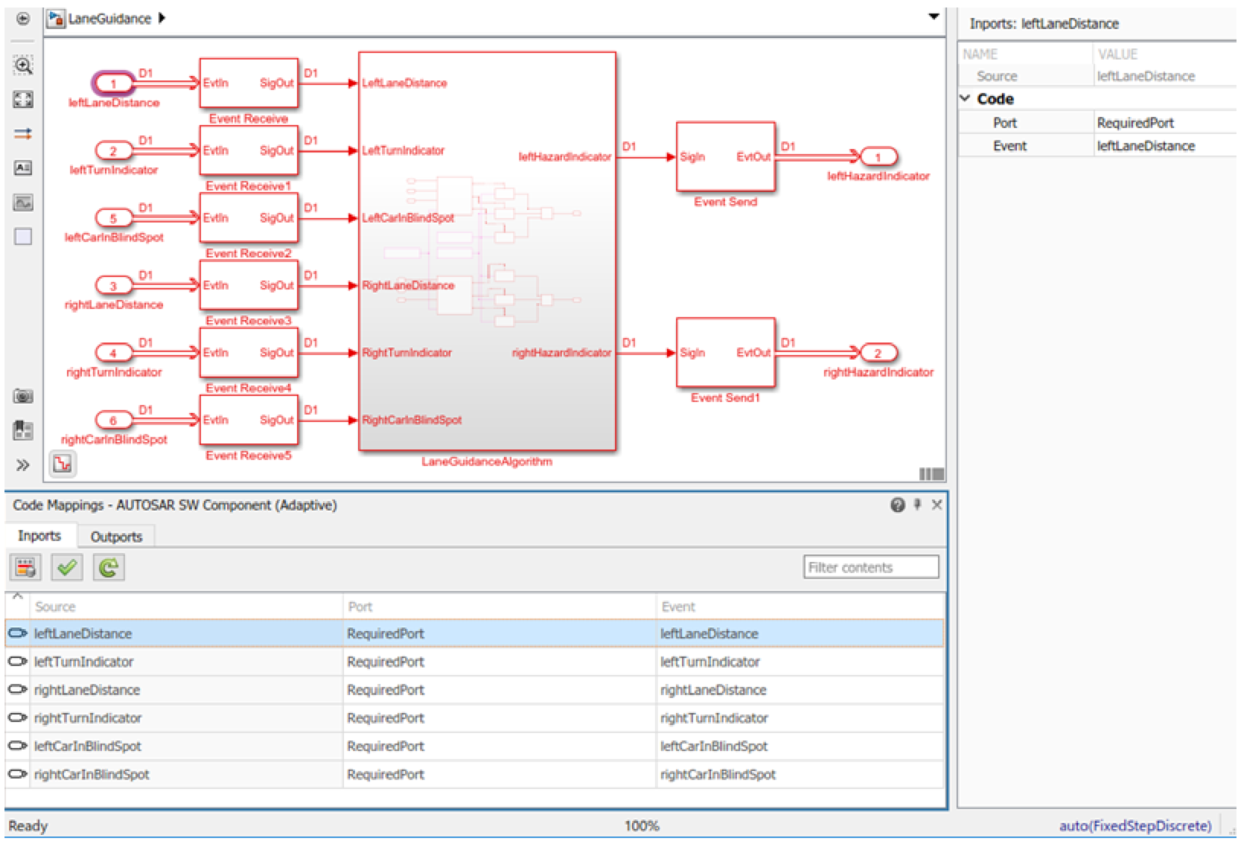


To configure the model for AUTOSAR adaptive software component development, work through the quick-start procedure. This example accepts default settings for the options in the Quick Start **Set Component** pane.

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

Configure AUTOSAR Adaptive Software Component in Simulink

After you create an initial Simulink representation of an AUTOSAR adaptive software component, AUTOSAR code perspective displays the model, a Property Inspector panel, and directly below the model, the Code Mappings editor.



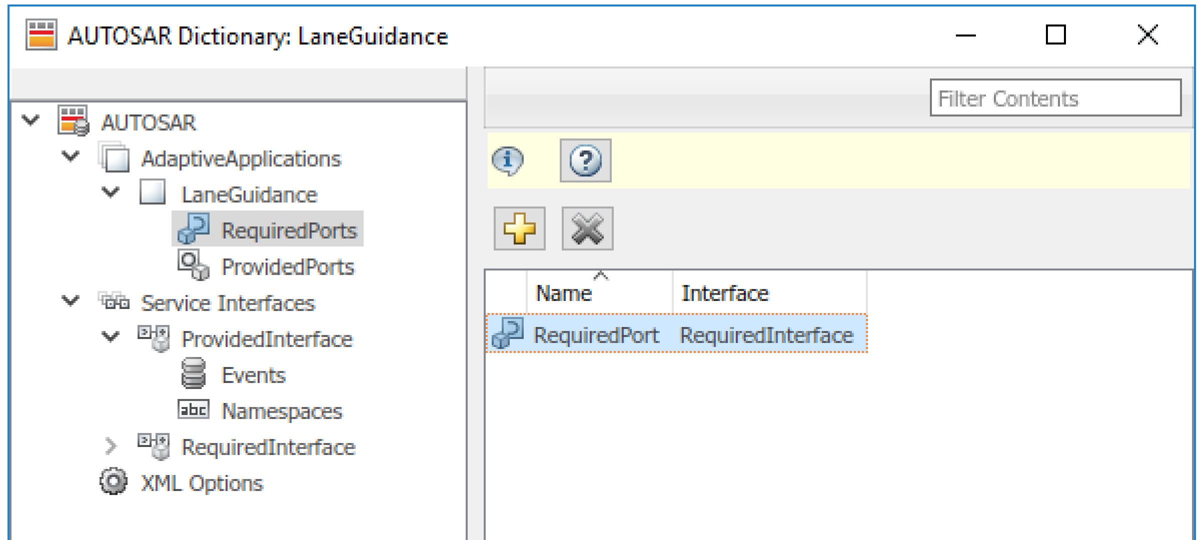
Next you use the Code Mappings editor and the AUTOSAR Dictionary to further develop the AUTOSAR adaptive component.

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, and Property Inspector displays its code attributes.

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 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.

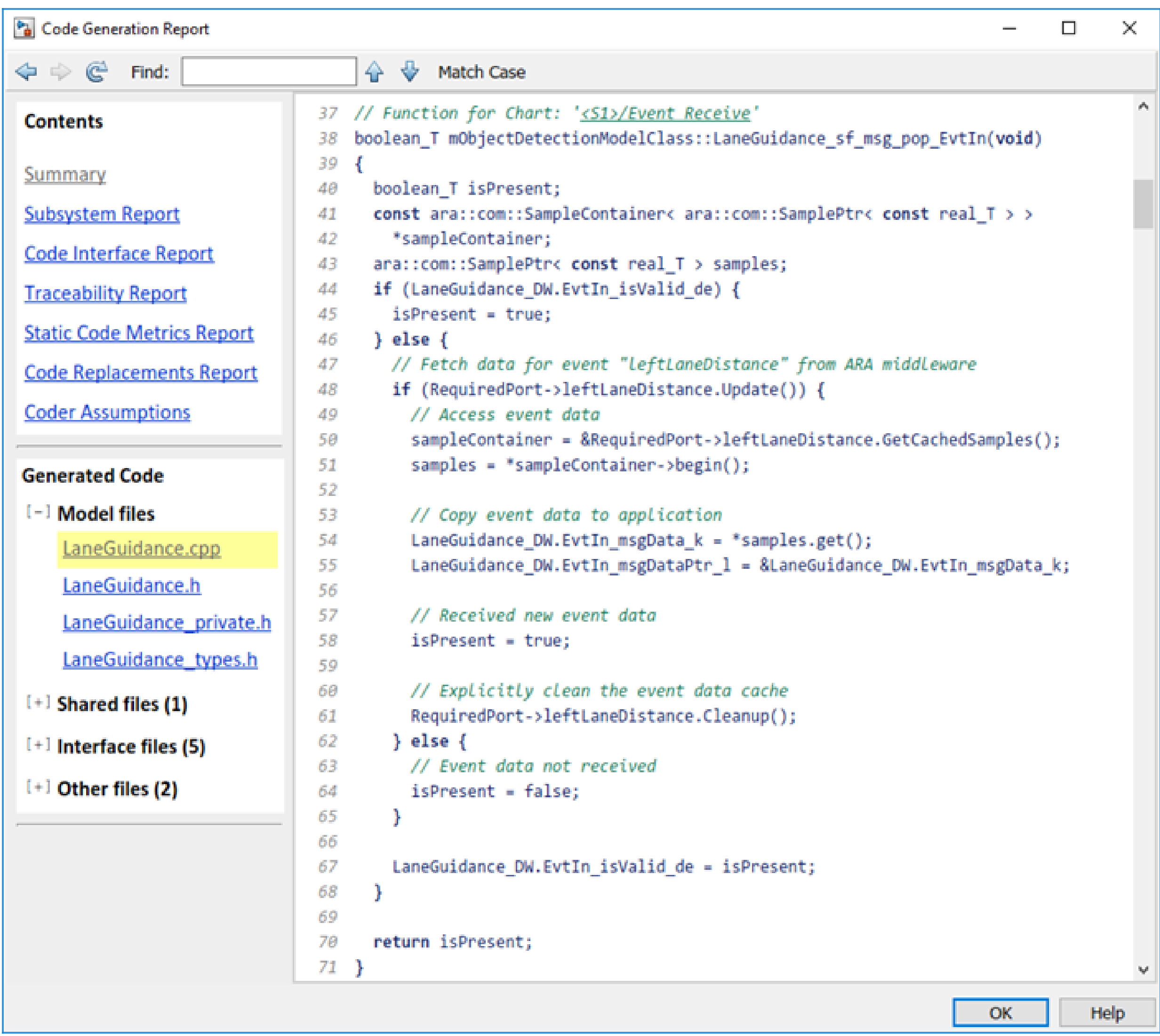
Open each node and examine its AUTOSAR elements. To modify an AUTOSAR element, select an element and modify its associated properties. AUTOSAR XML and AUTOSAR-compliant C code generated from the model reflect your modifications.

Generate C++ Code and ARXML Descriptions (Embedded Coder)

If you are licensed for Simulink Coder and Embedded Coder, 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 click the **Code** menu and select **C/C++ Code** > **Build Model**.

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"
- "AUTOSAR Blockset"

AUTOSAR arxml Importer

The AUTOSAR `arxml` importer imports AUTOSAR software component description files produced by an AUTOSAR authoring tool (AAT) into a Simulink model. The importer first parses `arxml` code that describes an AUTOSAR software component or composition. 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.

The importer creates an initial Simulink representation of each imported AUTOSAR software 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 UUIDs for `arxml` export. For more information, see “Round-Trip Preservation of AUTOSAR XML File Structure and Element Information” on page 3-46.
- 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 3-39.

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 Software Component" on page 3-27
- "Import AUTOSAR Component to Simulink" on page 3-32
- "Import AUTOSAR Composition to Simulink" on page 6-2
- "Configure AUTOSAR Code Generation" on page 5-12

More About

- "Workflows for AUTOSAR" on page 1-13

Import AUTOSAR Software Component

In Simulink, you can import an AUTOSAR software component description from `arxml` files and create a model representing the AUTOSAR software component. You use the AUTOSAR `arxml` importer, which is implemented as an `arxml.importer` object. For more information, see “AUTOSAR arxml Importer” on page 3-25.

Use `arxml.importer` functions in the following order:

1. Call the `arxml.importer` function to create an importer object that represents the software component information in the specified XML file or files. For example, this call specifies a main AUTOSAR software component file, `mr_component.arxml`, and related dependent files containing data type, implementation, and interface information that completes the software component description.

   ```
   ar = arxml.importer({'mr_component.arxml','mr_datatype.arxml',...
                        'mr_implementation.arxml','mr_interface.arxml'})
   ```

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

   ```
   addpath(fullfile(matlabroot,'examples','autosarblockset'));
   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 "matlabroot/examples/autosarblockset/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'

>>
```

2. To import a parsed atomic software component or composition into a Simulink model, call function `createComponentAsModel`, `createCompositionAsModel`, or `updateModel`. If you have not specified all dependencies for the components, you will see errors.

   - `createComponentAsModel` — Create Simulink representation of AUTOSAR `arxml` atomic software component.

     For example:

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

     The `ModelPeriodicRunnablesAs` property 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 3-31.

     To import Simulink data objects for AUTOSAR data into a Simulink data dictionary, you can set the `DataDictionary` property 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` property on the model creation.

     For more information, see the `createComponentAsModel` reference page and the live-script example “Import AUTOSAR Component to Simulink” on page 3-32.
   - `createCompositionAsModel` — Create Simulink representation of AUTOSAR `arxml` software composition.

     For example:

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

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

For more information, see the `createCompositionAsModel` reference page and the live-script example “Import AUTOSAR Composition to Simulink” on page 6-2.

- `updateModel` — Update AUTOSAR model with `arxml` changes.

  For example:

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

  For more information, see the `updateModel` reference page, “Import AUTOSAR Software Component Updates” on page 3-39, and the live-script example “Import AUTOSAR Component to Simulink” on page 3-32.

After you import your 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 AUTOSAR code generation options and XML export options, see “Configure AUTOSAR Code Generation” on page 5-12.

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 UUIDs for `arxml` export. For more information, see “Round-Trip Preservation of AUTOSAR XML File Structure and Element Information” on page 3-46.

See Also

`arxml.importer`

Related Examples

- “Import AUTOSAR Component to Simulink” on page 3-32
- “Import AUTOSAR Composition to Simulink” on page 6-2
- “Import AUTOSAR Software Component Updates” on page 3-39
- “Round-Trip Preservation of AUTOSAR XML File Structure and Element Information” on page 3-46
- “Configure AUTOSAR Code Generation” on page 5-12

More About

- "AUTOSAR arxml Importer" on page 3-25
- "Workflows for AUTOSAR" on page 1-13

Import AUTOSAR Software Component with Multiple Runnables

The AUTOSAR arxml importer functions createComponentAsModel and createCompositionAsModel can import AUTOSAR software components with multiple runnable entities into a new Simulink model. Use the ModelPeriodicRunnablesAs property 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 and map it into a new rate-based model:

```
addpath(fullfile(matlabroot,'examples','autosarblockset'))
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

Related Examples

- “Import AUTOSAR Component to Simulink” on page 3-32
- “Import AUTOSAR Composition to Simulink” on page 6-2

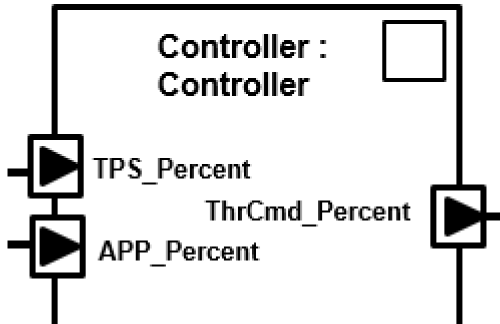
Import AUTOSAR Component to Simulink

Create Simulink® representation of AUTOSAR component imported from AUTOSAR authoring tool `arxml` file

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.

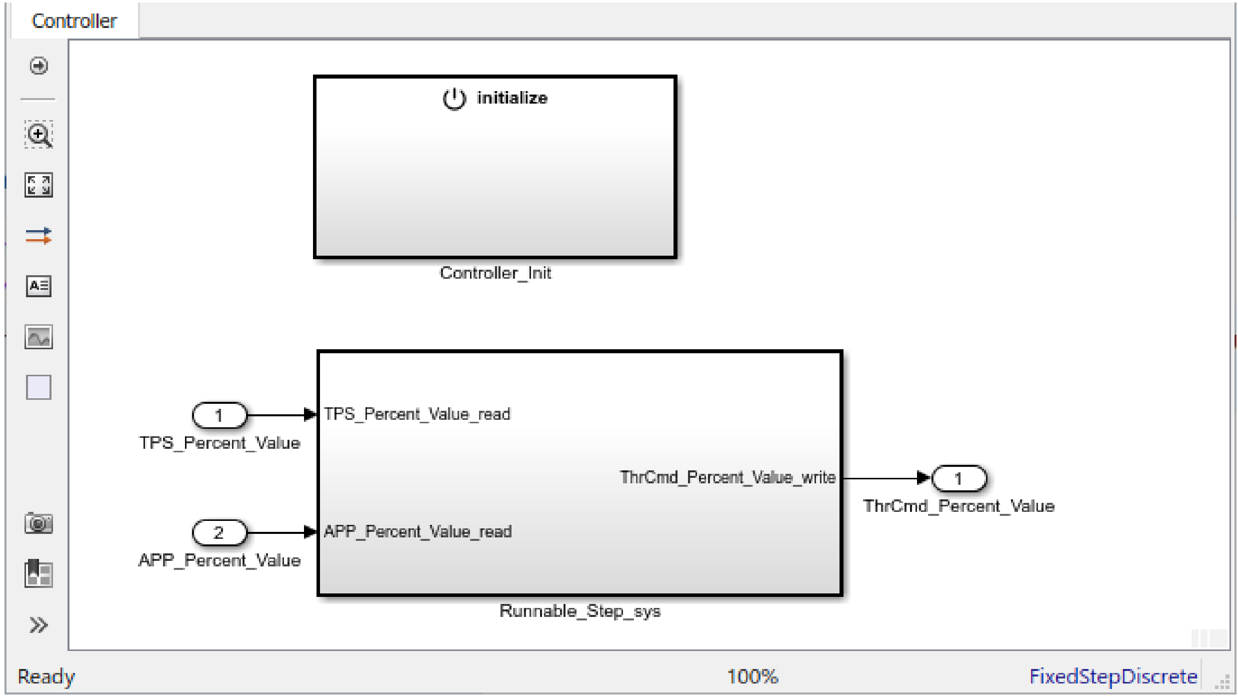
The component was created in an AUTOSAR authoring tool and exported to the file `ThrottlePositionControlComposition.arxml`.



Use the MATLAB function `createComponentAsModel` to import the `arxml` description and create an initial Simulink representation of the AUTOSAR component.

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



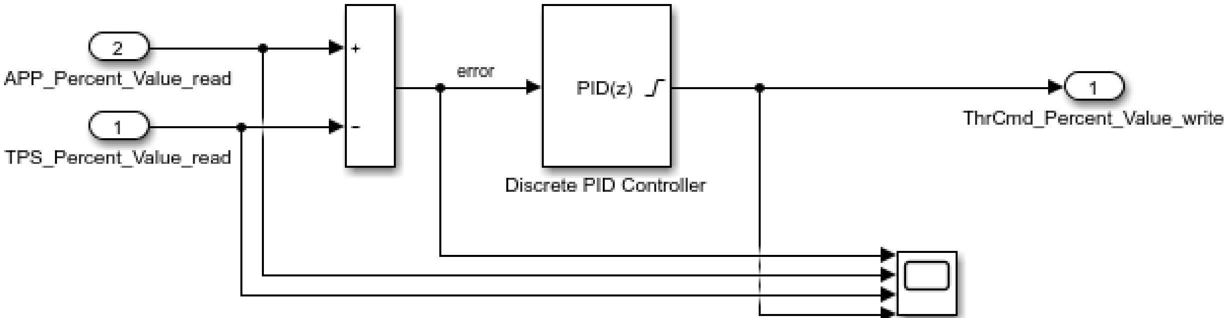
Develop AUTOSAR Component Algorithm, Simulate, and Generate Code

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, open model `autosar_swc_controller.slx` at *`matlabroot/`* `examples/autosarblockset`.) 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:

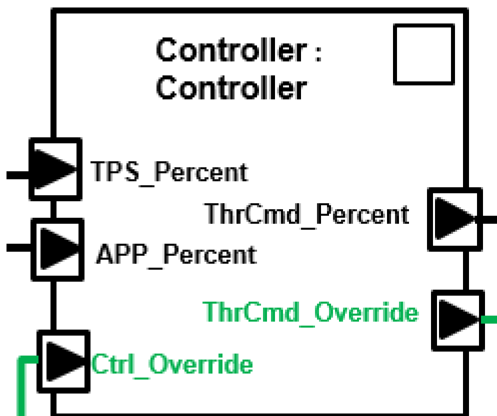
- Simulate the component model individually or in a containing composition or test harness.
- Generate `arxml` component description files and algorithmic C code for testing in Simulink or integration into an AUTOSAR run-time environment. (AUTOSAR code generation requires Simulink Coder and Embedded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

For more information on developing, simulating, and building AUTOSAR components, see example “Design AUTOSAR Components, Simulate, and Generate Code” on page 4-121.

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`.

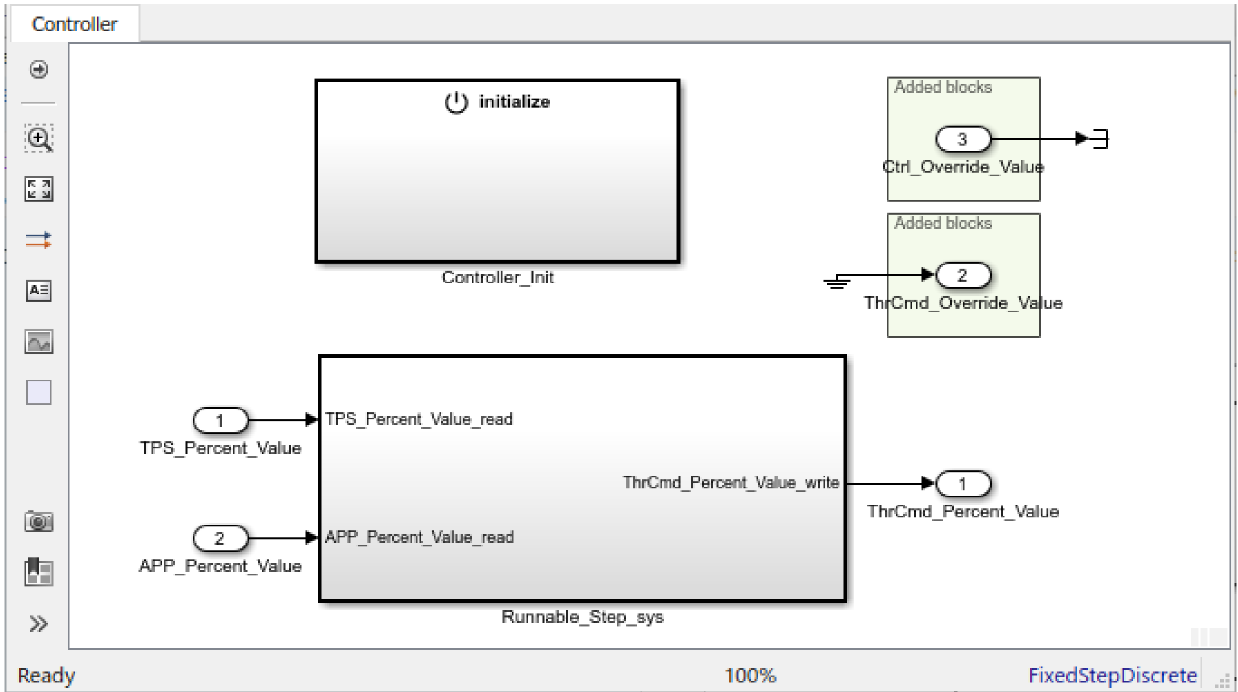


Use the MATLAB function `updateModel` to import the architectural revisions from the `arxml` file. The function updates the AUTOSAR component model with the changes and reports the results.

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

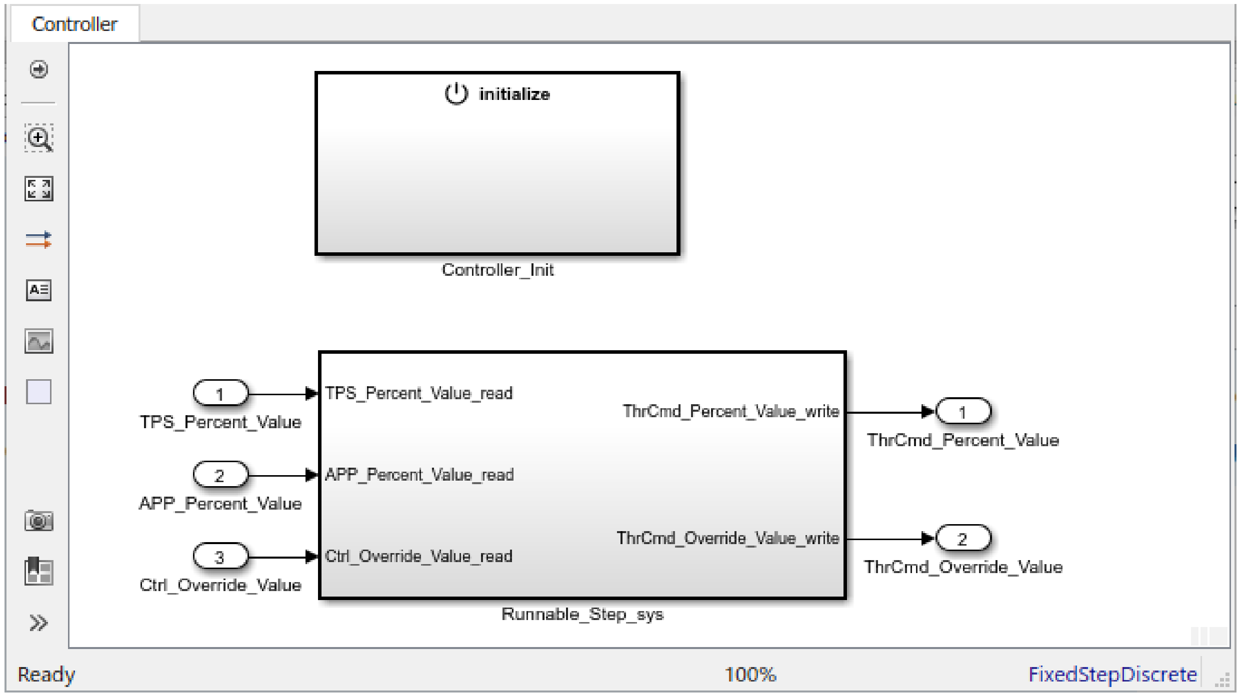
```
### Updating model Controller
### Saving original model as Controller_backup.slx
### Creating HTML report Controller_update_report.html
```

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 imports and outports inside the subsystem, and update the model diagram. For example:



Related Links

- `createComponentAsModel`
- `updateModel`
- "Component Creation"
- "Import AUTOSAR Software Component Updates" on page 3-39
- "Design AUTOSAR Components, Simulate, and Generate Code" on page 4-121

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` function `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 6-2

Import AUTOSAR Software Component Updates

After you create a Simulink model that represents an AUTOSAR software component, the `arxml` description of the component can change independently. Using `arxml.importer` function `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 3-39
"AUTOSAR Update Report Section Examples" on page 3-42

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 `Controller` model created by live script "Import AUTOSAR Component to Simulink" on page 3-32.

```
% Create and open AUTOSAR controller component model
addpath(fullfile(matlabroot,'examples','autosarblockset'))
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.

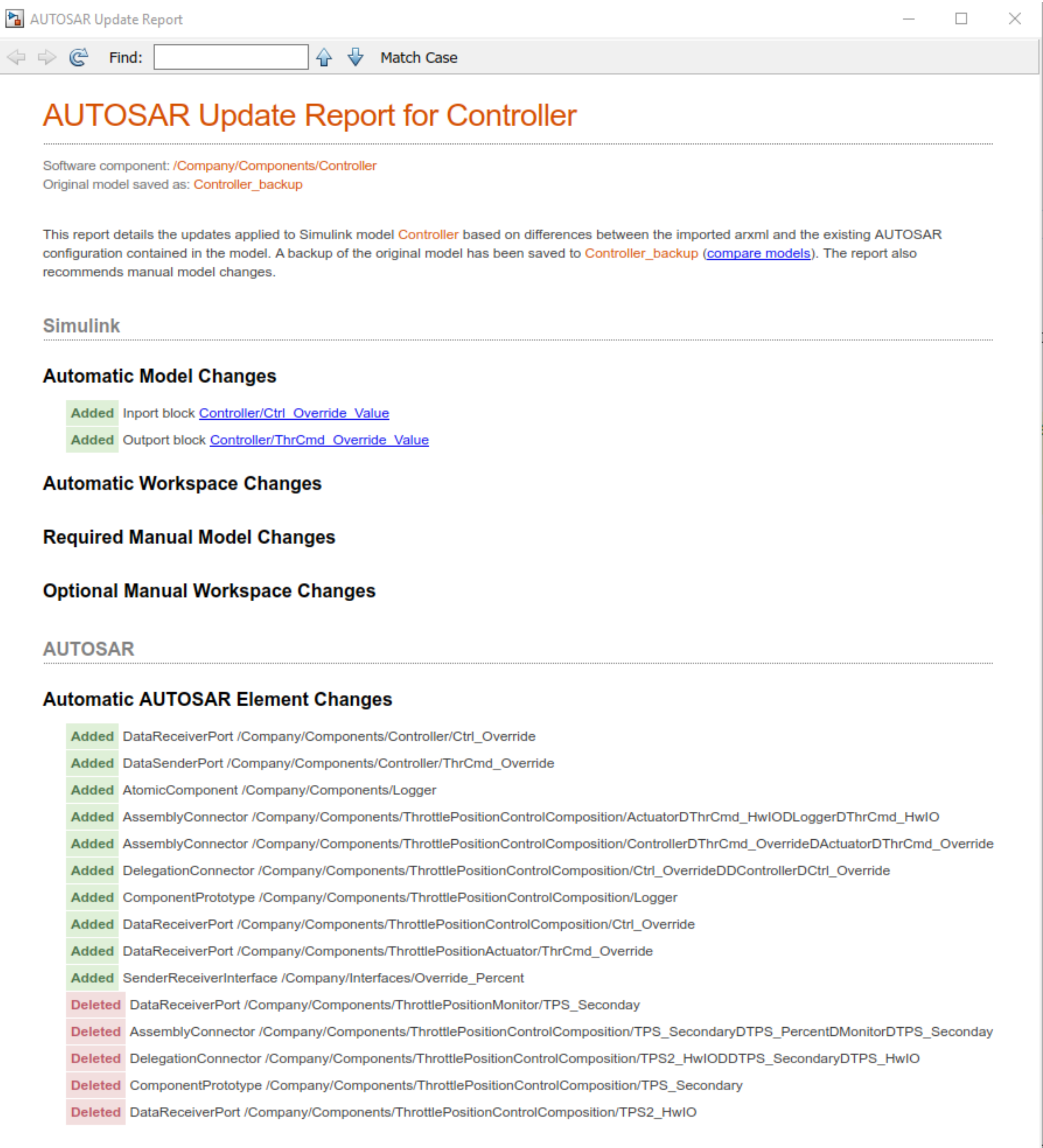
Note The imported `arxml` descriptions must contain the AUTOSAR software component mapped by the model.

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


AUTOSAR Update Report
Find:
Match Case
AUTOSAR Update Report for Controller
Software component: /Company/Components/Controller
Original model saved as: Controller_backup
This report details the updates applied to Simulink model Controller based on differences between the imported arxml and the existing AUTOSAR configuration contained in the model. A backup of the original model has been saved to Controller_backup (compare models). The report also recommends manual model changes.
Simulink
Automatic Model Changes
Added Inport block Controller/Ctrl_Override_Value
Added Outport block Controller/ThrCmd_Override_Value
Automatic Workspace Changes
Required Manual Model Changes
Optional Manual Workspace Changes
AUTOSAR
Automatic AUTOSAR Element Changes
Added DataReceiverPort /Company/Components/Controller/Ctrl_Override
Added DataSenderPort /Company/Components/Controller/ThrCmd_Override
Added AtomicComponent /Company/Components/Logger
Added AssemblyConnector /Company/Components/ThrottlePositionControlComposition/ActuatorDThrCmd_HwIODLoggerDThrCmd_HwIO
Added AssemblyConnector /Company/Components/ThrottlePositionControlComposition/ControllerDThrCmd_OverrideDActuatorDThrCmd_Override
Added DelegationConnector /Company/Components/ThrottlePositionControlComposition/Ctrl_OverrideDDControllerDCtrl_Override
Added ComponentPrototype /Company/Components/ThrottlePositionControlComposition/Logger
Added DataReceiverPort /Company/Components/ThrottlePositionControlComposition/Ctrl_Override
Added DataReceiverPort /Company/Components/ThrottlePositionActuator/ThrCmd_Override
Added SenderReceiverInterface /Company/Interfaces/Override_Percent
Deleted DataReceiverPort /Company/Components/ThrottlePositionMonitor/TPS_Seconday
Deleted AssemblyConnector /Company/Components/ThrottlePositionControlComposition/TPS_SecondaryDTPS_PercentDMonitorDTPS_Seconday
Deleted DelegationConnector /Company/Components/ThrottlePositionControlComposition/TPS2_HwIODDTPS_SecondaryDTPS_HwIO
Deleted ComponentPrototype /Company/Components/ThrottlePositionControlComposition/TPS_Secondary
Deleted DataReceiverPort /Company/Components/ThrottlePositionControlComposition/TPS2_HwIO

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 3-32 and “Import AUTOSAR Composition to Simulink” on page 6-2.

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 3-42
- “Automatic Workspace Changes” on page 3-43
- “Required Manual Model Changes” on page 3-44
- “Automatic AUTOSAR Element Changes” on page 3-44

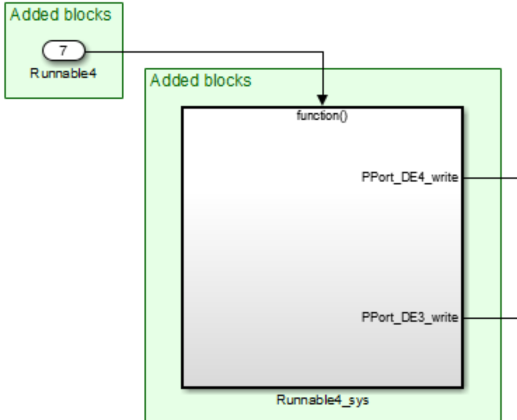
Automatic Model Changes

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 Outport mySWC/Runnable3/TicToc_irv from Inherit: auto to int8
Updated	PortDimensions of Outport mySWC/Runnable3/TicToc_irv from -1 to 1
Updated	SignalType of Outport mySWC/Runnable3/TicToc_irv from auto to real
Updated	SamplingMode of Outport mySWC/Runnable3/TicToc_irv 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	DataType of AUTOSAR.Parameter INC from UInt8 to uint8

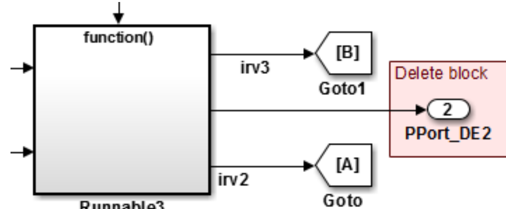
Required Manual Model Changes

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 IrvData /pkg/swc/ASWC/IB/IRV5
Added IrvData /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 Software Component” on page 3-27
- “Import AUTOSAR Component to Simulink” on page 3-32
- “Import AUTOSAR Composition to Simulink” on page 6-2
- “Configure AUTOSAR Code Generation” on page 5-12

More About

- “Workflows for AUTOSAR” on page 1-13

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.
- AUTOSAR UUIDs for identifiable elements. If an imported element does not have a UUID, none is created.

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 code. For more information, see “Configure AUTOSAR Elements and Properties” on page 4-8.

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 4-62.

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.
- 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.

Suppose that, in a working folder, you create a Simulink model named `Controller.slx` from example `arxml` file *`matlabroot`*`/help/toolbox/autosar/examples/ThrottlePositionController.arxml`.

```
% Create Controller model from AUTOSAR component
addpath(fullfile(matlabroot,'help','toolbox','autosar','examples'));
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.

```
% In AUTOSAR model, add SwAddrMethod CODE to SwAddrMethods package
arProps = autosar.api.getAUTOSARProperties('Controller');
addPackageableElement(arProps,'SwAddrMethod',...
    '/AUTOSAR_Platform/SwAddrMethods','CODE','SectionType','Code')
% Map step runnable function to SwAddrMethod CODE
slMap = autosar.api.getSimulinkMapping('Controller');
mapFunction(slMap,'StepFunction','Runnable_Step','SwAddrMethod','CODE')
% Display SwAddrMethod CODE path and step function mapping information
swAddrMethodPath = find(arProps,[],'SwAddrMethod','PathType','FullyQualified',...
    'SectionType','Code')
[arRunnableName,arRunnableSwAddrMethod] = getFunction(slMap,'StepFunction')

swAddrMethodPath =
    {'/AUTOSAR_Platform/SwAddrMethods/CODE'}

arRunnableName =
    'Runnable_Step'

arRunnableSwAddrMethod =
    'CODE'
```

You can view the modifications in the AUTOSAR Dictionary, **SwAddrMethods** view, and Code Mappings editor, **Entry-Point Functions** tab.

Build the model, for example, by using the command `rtwbuild('Controller')`. If the model has **Exported XML file packaging** set to `Modular`, the build exports three `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 4-62

More About

- "Configure AUTOSAR Elements and Properties" on page 4-8

Reuse AUTOSAR Element Descriptions

When developing an AUTOSAR software component in Simulink, you can reuse existing ARXML element definitions that are common to many components. For example, multiple product lines and teams might share elements such as interfaces and SwAddrMethods. Benefits of sharing and reusing AUTOSAR element definitions include lower risk of definition conflicts and easier code integration. You can manage shared definitions in a coordinated way.

After you create an AUTOSAR component model, you can import shared definitions from ARXML files that are dedicated to describing AUTOSAR elements, not components. 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 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 imports definitions from shared descriptions file `SwAddrMethods.arxml` into example model `autosar_swc`.

```
addpath(fullfile(matlabroot,'/examples/autosarblockset'));
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

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.

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 measurement and calibration.
4. When you 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
rtwdemo_autosar_swc_component.arxml
rtwdemo_autosar_swc_datatype.arxml
rtwdemo_autosar_swc_implementation.arxml
rtwdemo_autosar_swc_interface.arxml

See Also

`updateAUTOSARProperties`

Related Examples

- "Reuse AUTOSAR Elements in Component Model" on page 3-52

Reuse AUTOSAR Elements in Component Model

Update AUTOSAR software component model with ARXML element definitions

Add AUTOSAR Element Definitions to Model

When developing an AUTOSAR software component in Simulink, you can reuse existing ARXML element definitions that are common to many components. After you create an AUTOSAR component model, you import the definitions from ARXML files that are dedicated to describing AUTOSAR elements, not components. To help implement the component behavior, you want to reuse predefined elements such as interfaces and SwAddrMethods.

Suppose that you are developing an AUTOSAR software component model, into which 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.

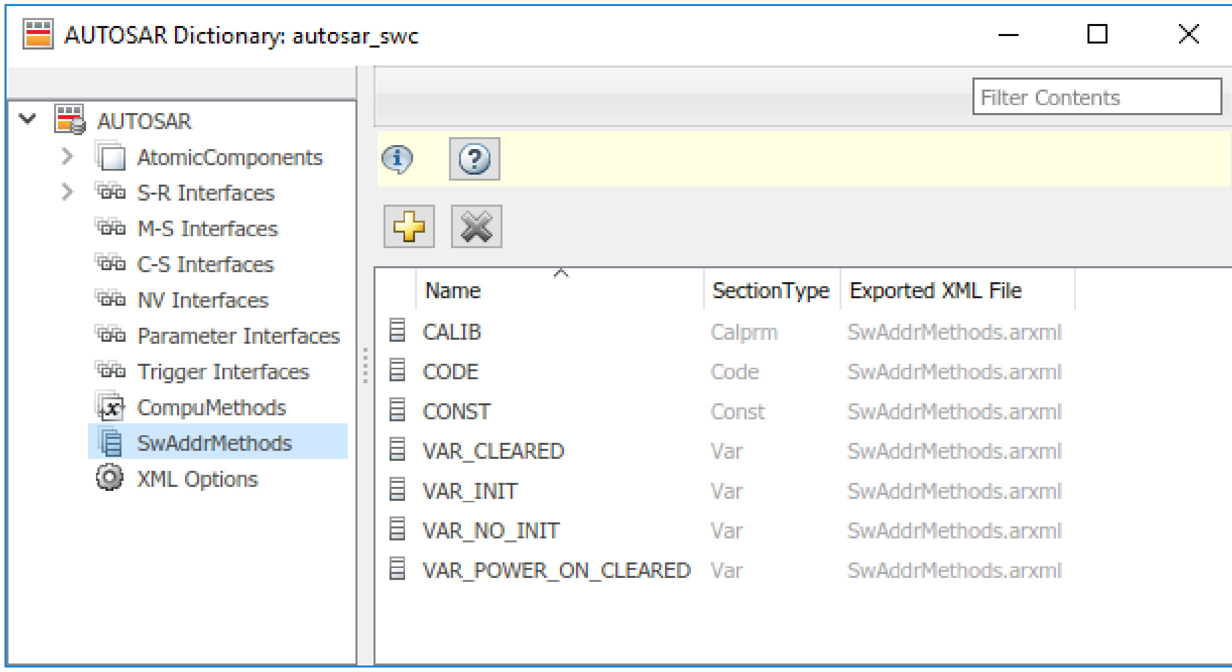
AUTOSAR

Automatic AUTOSAR Element Changes

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

You can view the added elements as read-only elements in the AUTOSAR Dictionary.

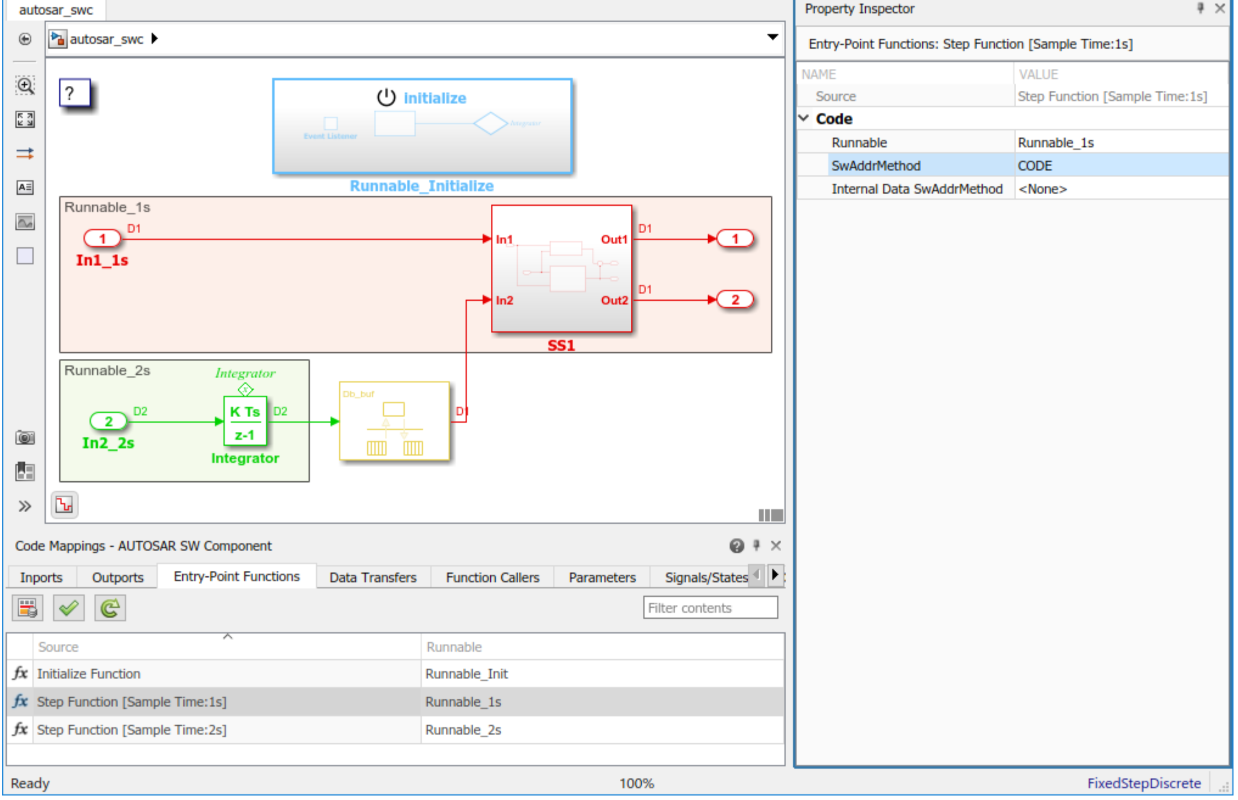
```
autosar_ui_launch(modelName);
```



Use 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 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,'StepFunction','Runnable_1s','SwAddrMethod','CODE');
```



Generate AUTOSAR C Code and XML Descriptions (Embedded Coder)

If you are licensed for Simulink Coder and Embedded Coder, you can generate AUTOSAR-compliant C code and XML descriptions from the model. To build the model, enter the command `rtwbuild(modelName);`.

Building the model generates an HTML code generation report. The C code contains a SwAddrMethod 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>
```

ARXML export preserves the file structure and content of the shared descriptions file `SwAddrMethods.arxml` from which you added SwAddrMethod definitions.

[-] **Interface files**

- SwAddrMethods.arxml
- rtwdemo_autosar_swc_component.arxml
- rtwdemo_autosar_swc_datatype.arxml
- rtwdemo_autosar_swc_implementation.arxml
- rtwdemo_autosar_swc_interface.arxml

Related Links

- `updateAUTOSARProperties`

- "Reuse AUTOSAR Element Descriptions" on page 3-49
- "Import AUTOSAR Software Component" on page 3-27
- "Import AUTOSAR Software Component Updates" on page 3-39
- "AUTOSAR arxml Importer" on page 3-25

Limitations and Tips

The following limitations apply to AUTOSAR component creation.

In this section...
"Cannot Save Importer Objects in MAT-Files" on page 3-58
"ApplicationRecordDataType and ImplementationDataType Element Names Must Match" on page 3-58

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.

4

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 4-68
- “Configure AUTOSAR Adaptive Elements and Properties” on page 4-87
- “Map AUTOSAR Adaptive Elements for Code Generation” on page 4-103
- “Incrementally Update AUTOSAR Mapping after Model Changes” on page 4-107
- “Configure AUTOSAR Adaptive Software Components” on page 4-111
- “Design AUTOSAR Components, Simulate, and Generate Code” on page 4-121
- “Configure Parameters and Signals for AUTOSAR Calibration and Measurement” on page 4-129
- “Model AUTOSAR Runnables Using Exported Functions” on page 4-133
- “Configure AUTOSAR Packages” on page 4-138
- “Configure AUTOSAR Package for Component, Interface, CompuMethod, or SwAddrMethod” on page 4-150
- “Configure AUTOSAR Sender-Receiver Communication” on page 4-153
- “Configure AUTOSAR Queued Sender-Receiver Communication” on page 4-172
- “Configure AUTOSAR Client-Server Communication” on page 4-197
- “Configure AUTOSAR Mode-Switch Communication” on page 4-225
- “Configure AUTOSAR Nonvolatile Data Communication” on page 4-234
- “Configure Receiver for AUTOSAR Parameter Communication” on page 4-237
- “Configure Receiver for AUTOSAR External Trigger Event Communication” on page 4-240
- “Configure AUTOSAR Adaptive Service Communication” on page 4-245
- “Configure Lookup Tables for AUTOSAR Measurement and Calibration” on page 4-250
- “Configure AUTOSAR Data for Measurement and Calibration” on page 4-263
- “Configure AUTOSAR Runnables and Events” on page 4-277
- “Configure AUTOSAR Initialize, Reset, or Terminate Runnables” on page 4-281

- “Add Top-Level Asynchronous Trigger to Periodic Rate-Based System” on page 4-288
- “Configure AUTOSAR Initialization Runnable (R4.1)” on page 4-291
- “Configure Disabled Mode for AUTOSAR Runnable Event” on page 4-294
- “Configure AUTOSAR Per-Instance Memory” on page 4-295
- “Configure AUTOSAR Static Memory” on page 4-300
- “Configure AUTOSAR Constant Memory” on page 4-305
- “Configure AUTOSAR Shared or Per-Instance Parameters” on page 4-308
- “Configure AUTOSAR Release 4.x Data Types” on page 4-314
- “Automatic AUTOSAR Data Type Generation” on page 4-318
- “Configure AUTOSAR CompuMethods” on page 4-320
- “Configure AUTOSAR Internal Data Constraints Export” on page 4-333
- “Configure Variants for AUTOSAR Ports and Runnables” on page 4-335
- “Configure Variants for AUTOSAR Array Sizes” on page 4-339
- “Configure Variants for AUTOSAR Runnable Implementations” on page 4-341
- “Control AUTOSAR Variants with Predefined Value Combinations” on page 4-344
- “Configure and Map AUTOSAR Component Programmatically” on page 4-347
- “AUTOSAR Property and Map Function Examples” on page 4-355
- “Limitations and Tips” on page 4-378

AUTOSAR Component Configuration

After you create an AUTOSAR software component model in Simulink, use Code Mappings editor and AUTOSAR Dictionary to further develop the AUTOSAR component. Code Mappings editor and AUTOSAR Dictionary provide mapping and properties views of the component model, which can be used separately and together to configure the AUTOSAR component:

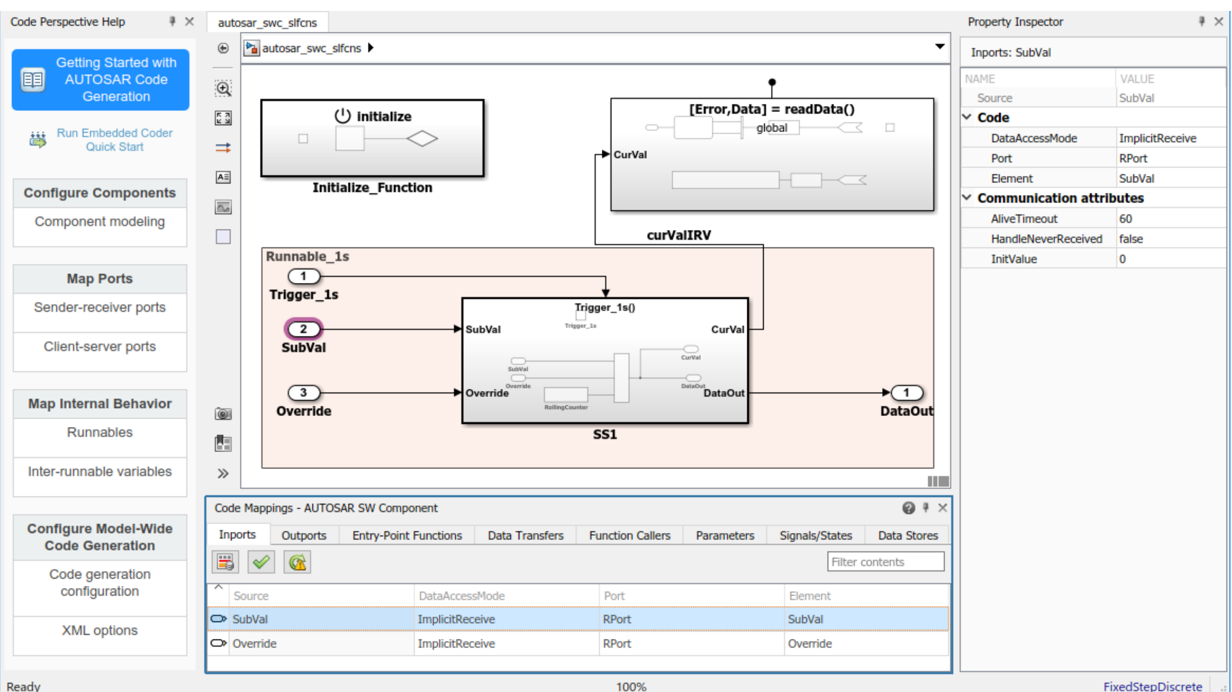
- Code Mappings editor — Using a tabbed table format, displays model inports, outports, entry-point functions, and other Simulink elements relevant to your AUTOSAR platform. Use this view to map model elements to AUTOSAR component elements from a Simulink model perspective.
- AUTOSAR Dictionary — Using 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.

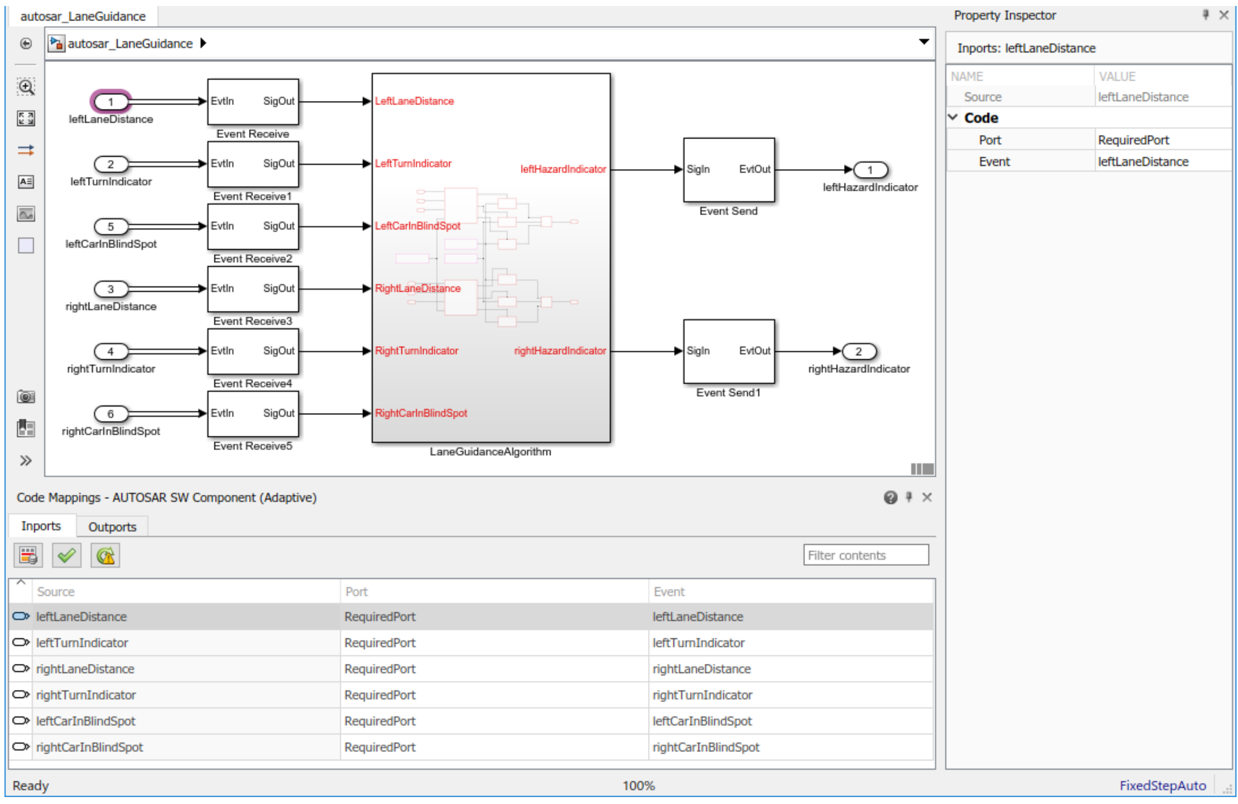
Alternatively, you can configure AUTOSAR mapping and properties programmatically. See “Configure and Map AUTOSAR Component Programmatically” on page 4-347.

In a model for which an AUTOSAR system target file (`autosar.tlc` or `autosar_adaptive.tlc`) has been selected, create or open a mapped view of the AUTOSAR model. In the model window, do one of the following:

- Select **Code > C/C++ Code > Configure Model in Code Perspective**
- Click the perspective control in the lower-right corner and select **Code**.

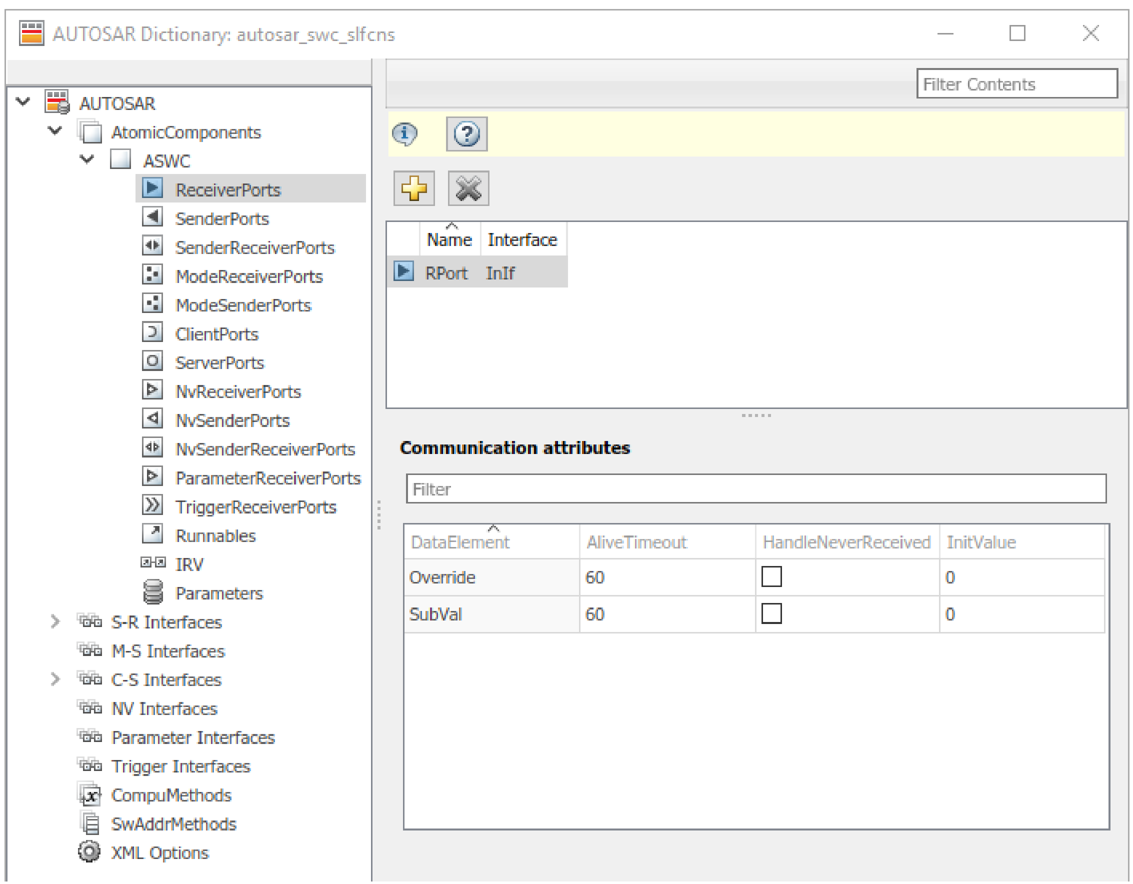
The model opens in the AUTOSAR code perspective. This perspective displays a help panel, a Property Inspector dialog box, and, directly under the model, Code Mappings editor.


Code Perspective Help
Getting Started with AUTOSAR Code Generation
Run Embedded Coder Quick Start
Configure Components
Component modeling
Map Ports
Sender-receiver ports
Client-server ports
Map Internal Behavior
Runnables
Inter-runnable variables
Configure Model-Wide Code Generation
Code generation configuration
XML options
autosar_swc_slfcns
initialize
Initialize_Function
[Error,Data] = readData()
global
CurVal
curValIRV
Runnable_1s
Trigger_1s
SubVal
Override
Trigger_1s()
CurVal
DataOut
SS1
Code Mappings - AUTOSAR SW Component
Inports
Outports
Entry-Point Functions
Data Transfers
Function Callers
Parameters
Signals/States
Data Stores
Filter contents
Source
DataAccessMode
Port
Element
SubVal
ImplicitReceive
RPort
SubVal
Override
ImplicitReceive
RPort
Override
Property Inspector
Inports: SubVal
NAME
VALUE
Source
SubVal
Code
DataAccessMode
ImplicitReceive
Port
RPort
Element
SubVal
Communication attributes
AliveTimeout
60
HandleNeverReceived
false
InitValue
0
Ready
100%
FixedStepDiscrete



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 open an AUTOSAR properties view of the component model, either click the **AUTOSAR Dictionary** button in Code Mappings editor or select **Code > C/C++ Code > Configure AUTOSAR Dictionary**. 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 **Code > C/C++ Code** 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 Code Mappings editor, click the **Update** button to update the Simulink to AUTOSAR mapping of the model with changes to Simulink data transfers, entry-point functions, and function callers.

- In Code Mappings editor, click the **Validate** button to validate the AUTOSAR component configuration.

See Also

Related Examples

- “Map AUTOSAR Elements for Code Generation” on page 4-68
- “Configure AUTOSAR Elements and Properties” on page 4-8
- “Map AUTOSAR Adaptive Elements for Code Generation” on page 4-103
- “Configure AUTOSAR Adaptive Elements and Properties” on page 4-87
- “Configure and Map AUTOSAR Component Programmatically” on page 4-347

Configure AUTOSAR Elements and Properties

In Simulink, you can use AUTOSAR Dictionary and 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 AUTOSAR Dictionary to configure AUTOSAR elements from an AUTOSAR perspective. Using a tree format, 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.

In this section...

"AUTOSAR Elements Configuration Workflow" on page 4-8
"Configure AUTOSAR Atomic Software Components" on page 4-10
"Configure AUTOSAR Ports" on page 4-13
"Configure AUTOSAR Runnables" on page 4-28
"Configure AUTOSAR Inter-Runnable Variables" on page 4-34
"Configure AUTOSAR Parameters" on page 4-35
"Configure AUTOSAR Communication Interfaces" on page 4-37
"Configure AUTOSAR Computation Methods" on page 4-58
"Configure AUTOSAR SwAddrMethods" on page 4-60
"Configure AUTOSAR XML Options" on page 4-62

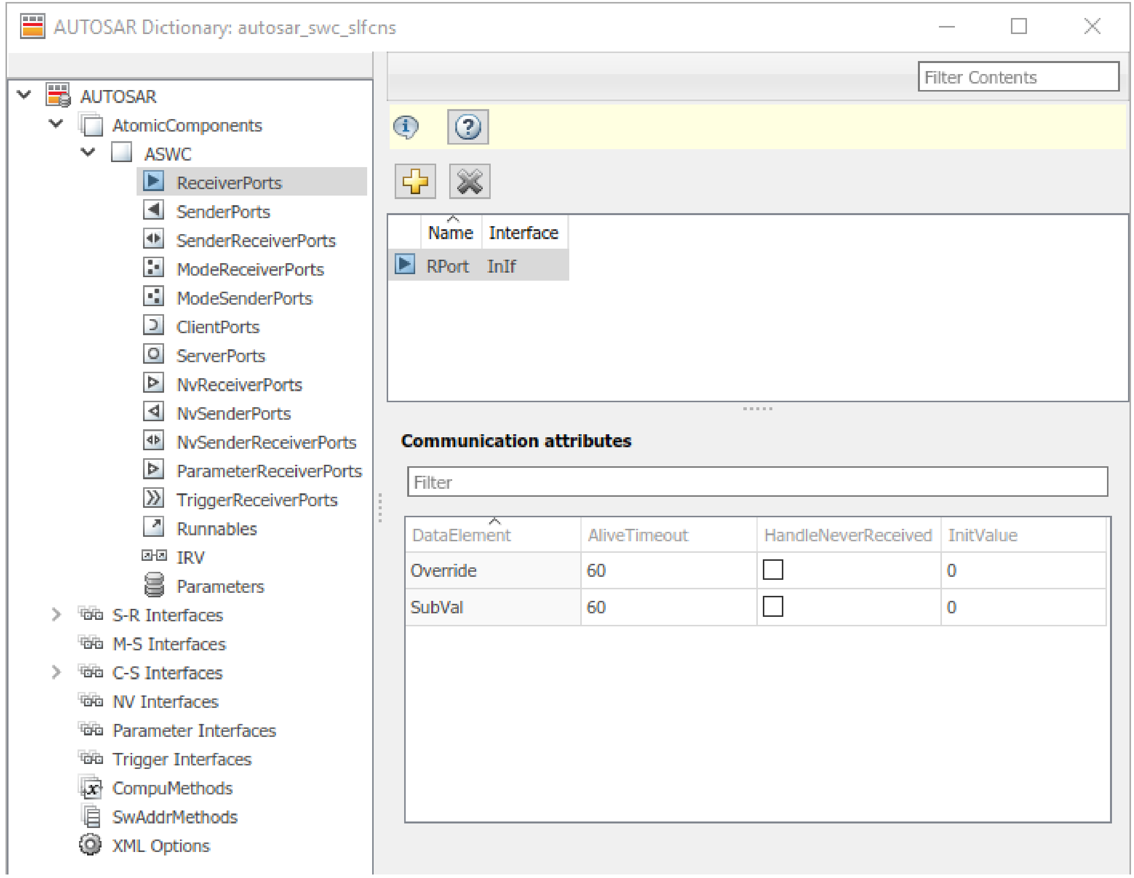
AUTOSAR Elements Configuration Workflow

To configure AUTOSAR component elements for the Classic Platform in Simulink:

1. Open a model for which AUTOSAR system target file `autosar.tlc` has been selected.
2. If the model does not yet have a mapped AUTOSAR software component, create one. In the model window, select **Code** > **C/C++ Code** > **Configure Model in Code Perspective** or click the perspective control in the lower-right corner and select **Code**. The AUTOSAR Component Quick Start opens. Work through the quick-start

procedure. When you click **Finish**, the model opens in the AUTOSAR code perspective. For more information, see “Create AUTOSAR Software Component in Simulink” on page 3-2.

3 Open AUTOSAR Dictionary. Either click the **AUTOSAR Dictionary** button in Code Mappings editor or select **Code > C/C++ Code > Configure AUTOSAR Dictionary**.



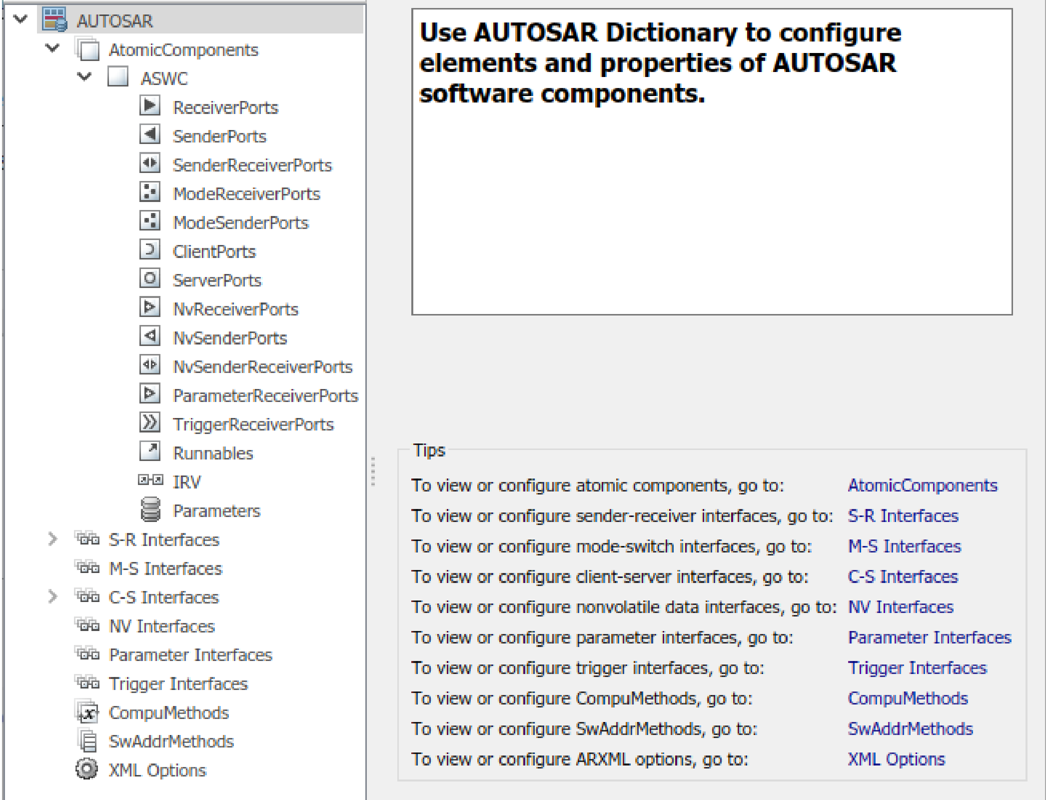
4 Navigate the AUTOSAR Dictionary tree to configure AUTOSAR elements and properties. You can add elements, remove elements, or select elements to view and modify their properties. Use the **Filter Contents** field (where available) to selectively display some elements, while omitting others, in the current view.

5 After configuring AUTOSAR elements and properties, open Code Mappings editor. Use 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 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.

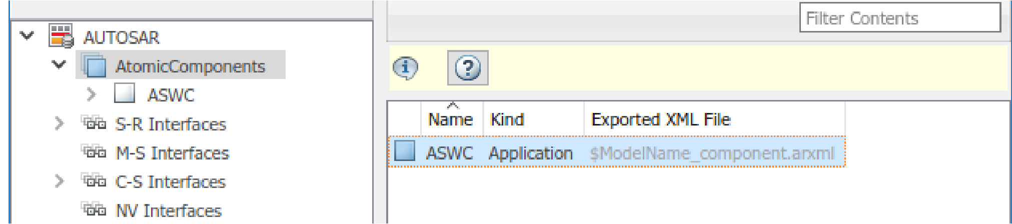


To configure AUTOSAR atomic software component elements and properties:

1. Open a model for which a mapped AUTOSAR software component has been created. For more information, see “Component Creation”.
2. Open AUTOSAR Dictionary. Either click the **AUTOSAR Dictionary** button in Code Mappings editor or select **Code > C/C++ Code > Configure AUTOSAR Dictionary**.
3. In the left-hand pane of AUTOSAR Dictionary, under **AUTOSAR**, select **AtomicComponents**.

   The atomic components view in 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
     - `SensorAccuator` for sensor or actuator component
     - `ServiceProxy` for service proxy component
   - Rename an AUTOSAR component by clicking its name and then editing the name text.

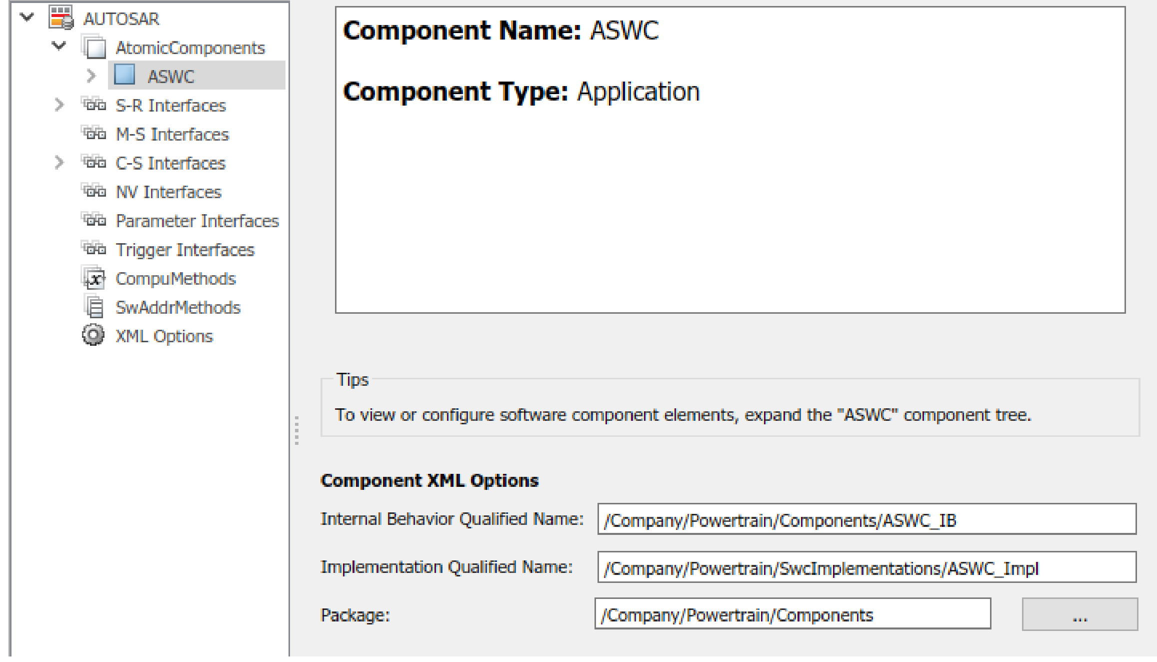
4. In the left-hand pane of AUTOSAR Dictionary, expand **AtomicComponents** and select an AUTOSAR component.

   The component view in 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 4-150.

For more information about component XML options, see "Configure AUTOSAR Packages" on page 4-138.



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 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 4-13
- “Mode-Switch Ports” on page 4-17
- “Client-Server Ports” on page 4-19
- “Nonvolatile Data Ports” on page 4-22
- “Parameter Receiver Ports” on page 4-25
- “Trigger Receiver Ports” on page 4-27

Sender-Receiver Ports

The AUTOSAR Dictionary views of sender and receiver ports support modeling AUTOSAR sender-receiver (S-R) communication in Simulink. You use 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 4-153 and “Configure AUTOSAR Queued Sender-Receiver Communication” on page 4-172.

To configure AUTOSAR S-R port elements and properties, open a model for which a mapped AUTOSAR software component has been created and open AUTOSAR Dictionary.

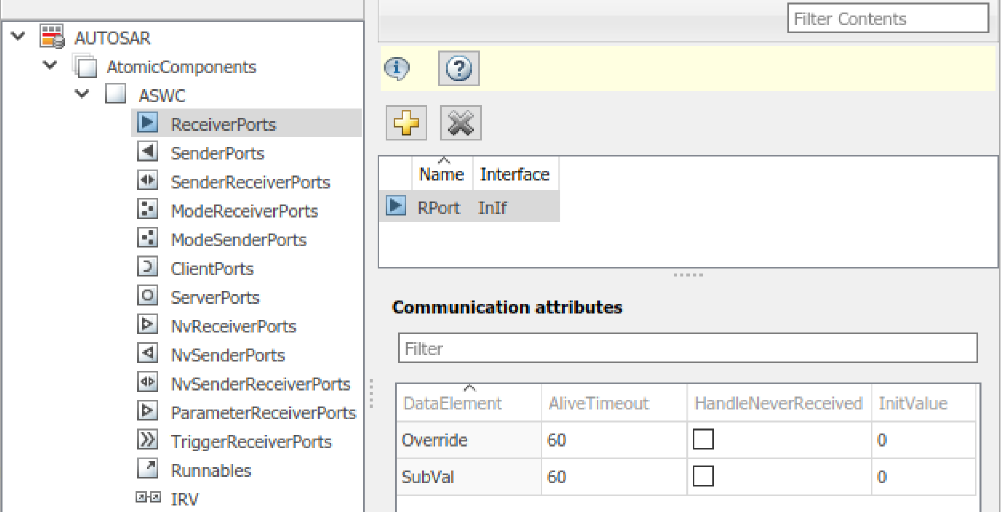
1. In the left-hand pane of AUTOSAR Dictionary, expand the component name and select **ReceiverPorts**.

   The receiver ports view in AUTOSAR Dictionary lists receiver ports and their properties. You can:

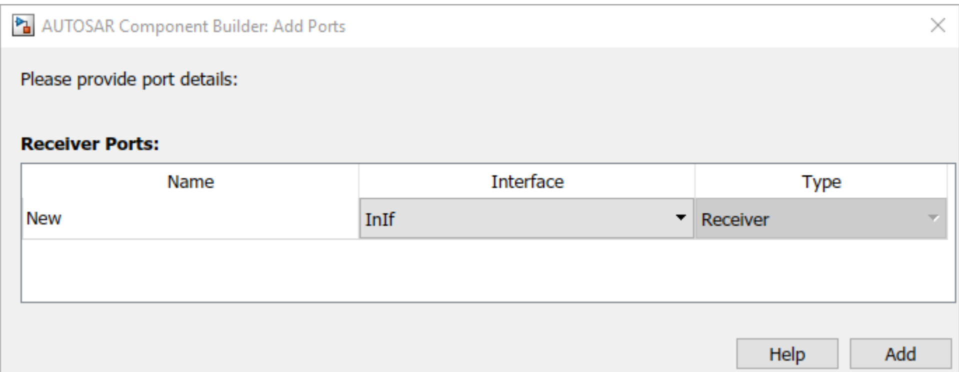
   - Select an AUTOSAR receiver port, and view and optionally reselect its associated S-R interface.
   - Rename an AUTOSAR receiver port by clicking its name and then editing the name text.
   - When you select a port that is configured for `QueuedExplicitReceive` data access, AUTOSAR Dictionary displays additional port communication specification

(ComSpec) attributes. For AUTOSAR receiver ports, you can modify ComSpec attributes `AliveTimeout`, `HandleNeverReceived`, and `InitValue`. For more information, see “Configure AUTOSAR Sender-Receiver Port ComSpecs” on page 4-168.

- Click the **Add** button to open an Add Ports dialog box to add a port.
- Select a port and then click the **Delete** button to remove it.



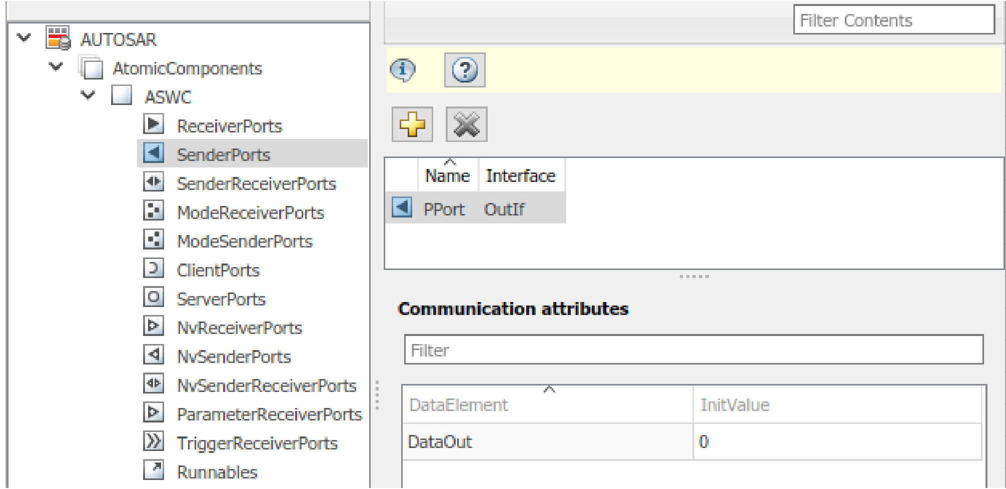
The Add Ports dialog box lets you add a receiver port and associate it with an existing S-R interface. To add the port and return to the receiver ports view, click **Add**.



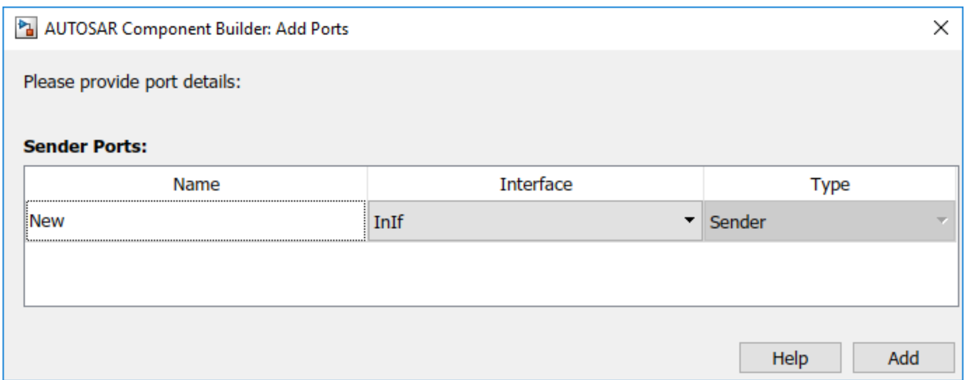
2 In the left-hand pane of AUTOSAR Dictionary, select **SenderPorts**.

The sender ports view in 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 an AUTOSAR sender port by clicking its name and then editing the name text.
- When you select a port that is configured for `QueuedExplicitSend` data access, AUTOSAR Dictionary displays additional port communication specification (ComSpec) attributes. For AUTOSAR sender ports, you can modify ComSpec attribute `InitValue`. For more information, see "Configure AUTOSAR Sender-Receiver Port ComSpecs" on page 4-168.
- Click the **Add** button to open an Add Ports dialog box to add a port.
- Select a port and then click the **Delete** button to remove it.



The Add Ports dialog box lets you add a sender port and associate it with an existing S-R interface. Click **Add** to add the port and return to the sender ports view.

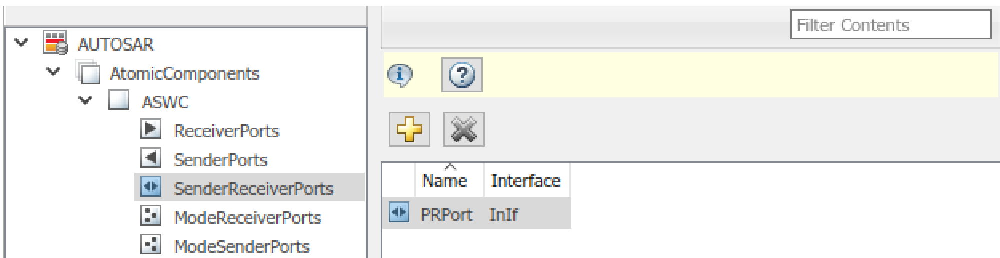


3 In the left-hand pane of AUTOSAR Dictionary, select **SenderReceiverPorts**.

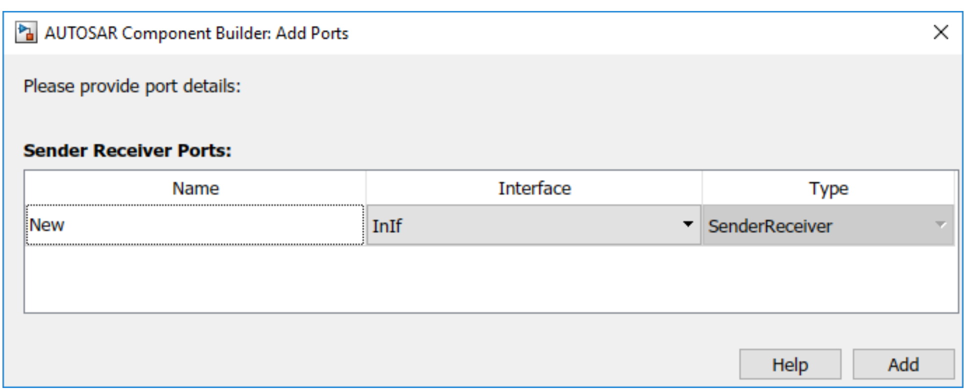
The sender-receiver ports view in AUTOSAR Dictionary lists sender-receiver ports and their properties. You can:

- Select an AUTOSAR sender-receiver port, and view and optionally reselect its associated S-R interface.
- Rename an AUTOSAR sender-receiver port by clicking its name and then editing the name text.
- Click the **Add** button to open an Add Ports dialog box to add a port.
- Select a port and then click the **Delete** button to remove it.

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.



The Add Ports dialog box lets you add a sender-receiver port and associate it with an existing S-R interface. Click **Add** to add the port and return to the sender-receiver ports view.



Mode-Switch Ports

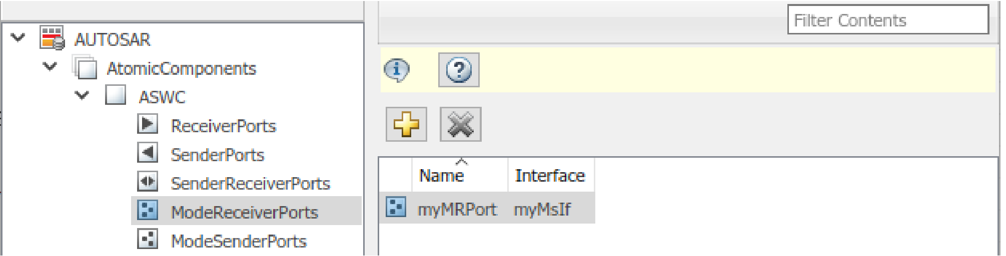
The AUTOSAR Dictionary views of mode sender and receiver ports support modeling AUTOSAR mode-switch (M-S) communication in Simulink. You use AUTOSAR Dictionary to configure AUTOSAR M-S ports and M-S interfaces in your model. For more information, see “Configure AUTOSAR Mode-Switch Communication” on page 4-225.

To configure AUTOSAR M-S port elements and properties, open a model for which a mapped AUTOSAR software component has been created and open AUTOSAR Dictionary.

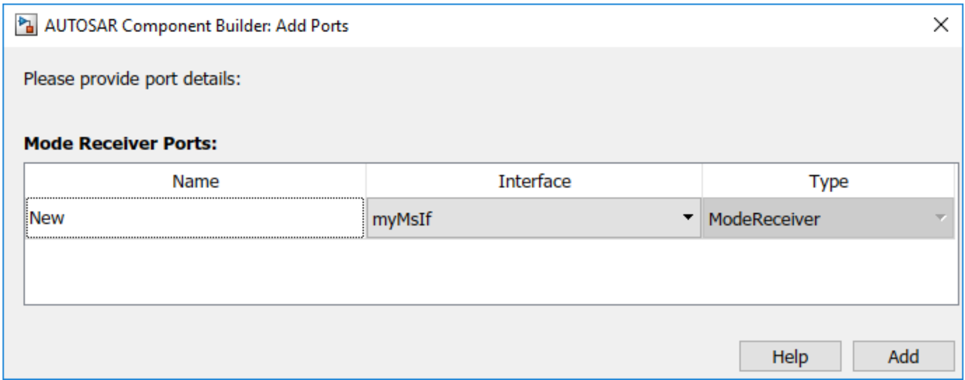
1. In the left-hand pane of AUTOSAR Dictionary, expand the component name and select **ModeReceiverPorts**.

   The mode receiver ports view in 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 an AUTOSAR mode receiver port by clicking its name and then editing the name text.
   - Click the **Add** button to open an Add Ports dialog box to add a port.
   - Select a port and then click the **Delete** button to remove it.



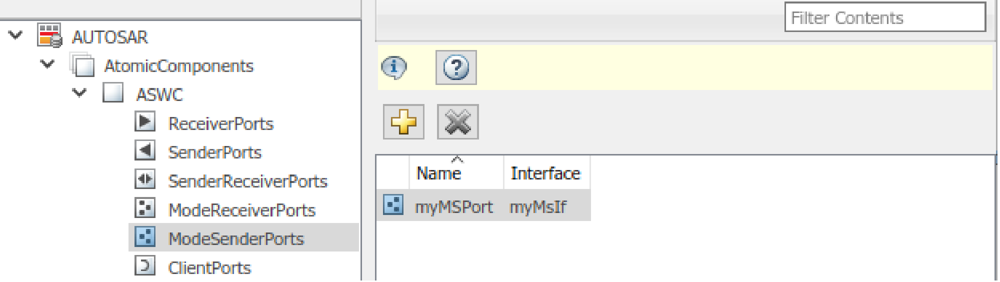
The Add Ports dialog box lets you add a mode receiver port and associate it with an existing M-S interface. If an M-S interface does not exist in the component, you must create one before adding the port. Click **Add** to add the port and return to the mode receiver ports view.



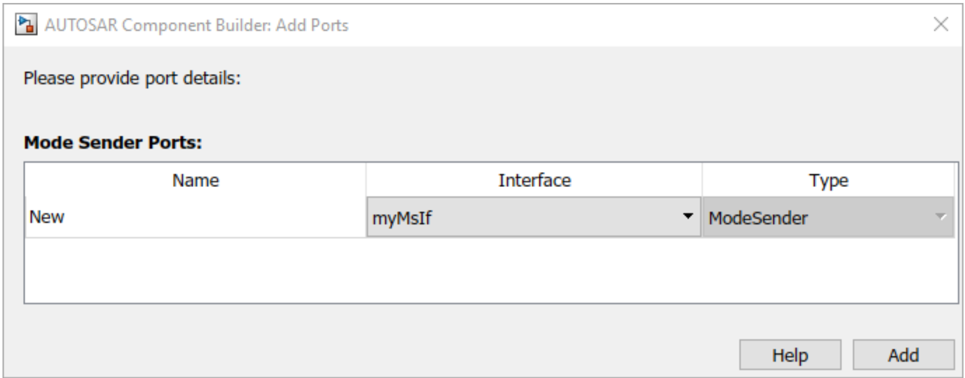
2 In the left-hand pane of AUTOSAR Dictionary, select **ModeSenderPorts**.

The mode sender ports view in 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 an AUTOSAR mode sender port by clicking its name and then editing the name text.
- Click the **Add** button to open an Add Ports dialog box to add a port.
- Select a port and then click the **Delete** button to remove it.



The Add Ports dialog box lets you add a mode sender port 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. Click **Add** to add the port and return to the mode sender ports view.



Client-Server Ports

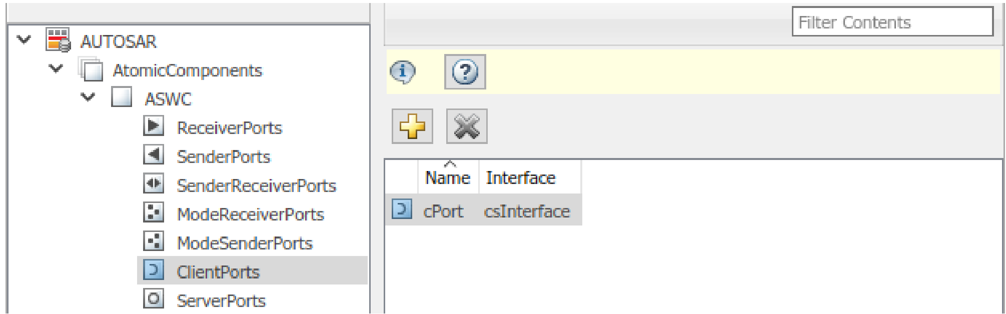
The AUTOSAR Dictionary views of client and server ports support modeling AUTOSAR client-server (C-S) communication in Simulink. You use AUTOSAR Dictionary to configure AUTOSAR C-S ports, C-S interfaces, and C-S operations in your model. For more information, see “Configure AUTOSAR Client-Server Communication” on page 4-197.

To configure AUTOSAR C-S port elements and properties, open a model for which a mapped AUTOSAR software component has been created and open AUTOSAR Dictionary.

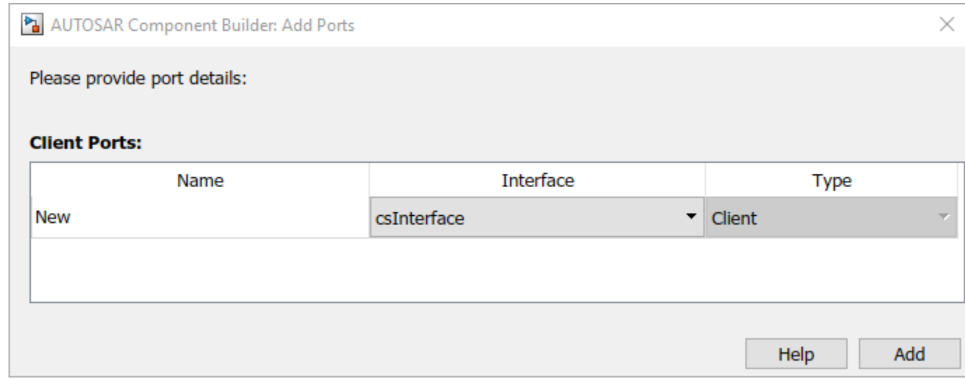
1. In the left-hand pane of AUTOSAR Dictionary, expand the component name and select **ClientPorts**.

The client ports view in 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 an AUTOSAR client port by clicking its name and then editing the name text.
- Click the **Add** button to open an Add Ports dialog box to add a client port.
- Select a port and then click the **Delete** button to remove it.



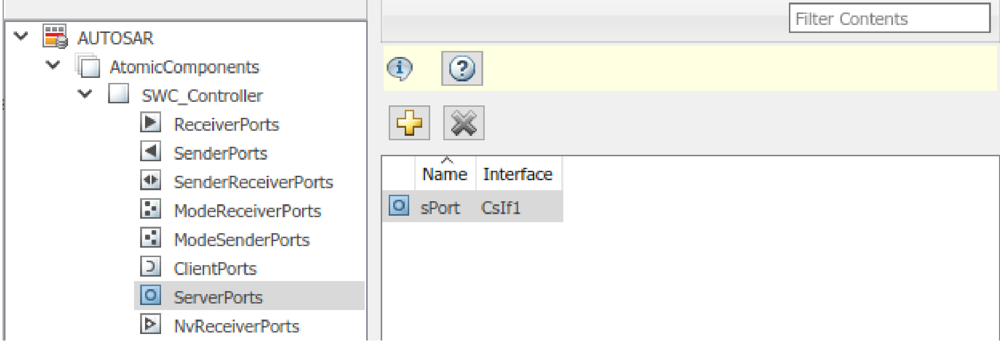
The Add Ports dialog box lets you add a client port and associate it with an existing C-S interface. If a C-S interface does not exist in the component, you must create one before adding the port. Click **Add** to add the port and return to the client ports view.



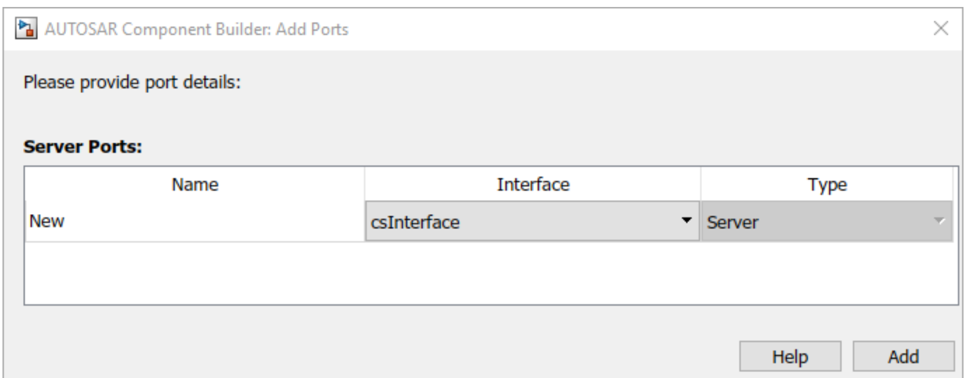
2 In the left-hand pane of AUTOSAR Dictionary, select **ServerPorts**.

The server ports view in 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 an AUTOSAR server port by clicking its name and then editing the name text.
- Click the **Add** button to open an Add Ports dialog box to add a server port.
- Select a port and then click the **Delete** button to remove it.



The Add Ports dialog box lets you add a server port 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. Click **Add** to add the port and return to the server ports view.



Nonvolatile Data Ports

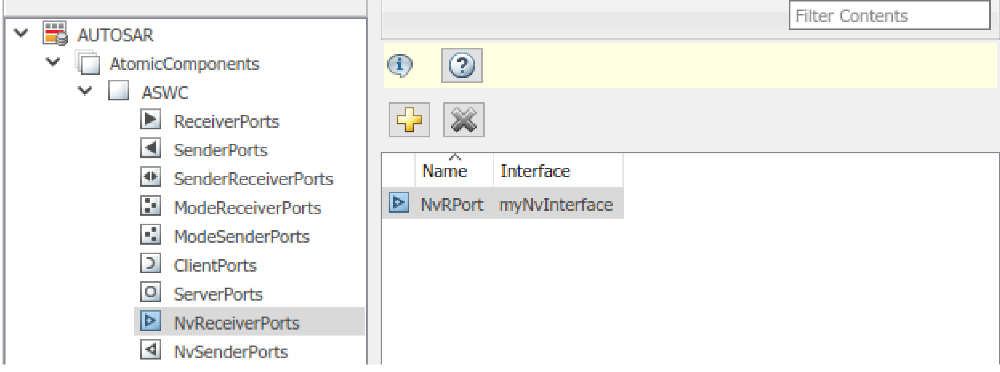
The AUTOSAR Dictionary views of nonvolatile (NV) sender and receiver ports support modeling AUTOSAR NV data communication in Simulink. You use AUTOSAR Dictionary to configure AUTOSAR NV ports, NV interfaces, and NV data elements in your model. For more information, see “Configure AUTOSAR Nonvolatile Data Communication” on page 4-234.

To configure AUTOSAR NV port elements and properties, open a model for which a mapped AUTOSAR software component has been created and open AUTOSAR Dictionary.

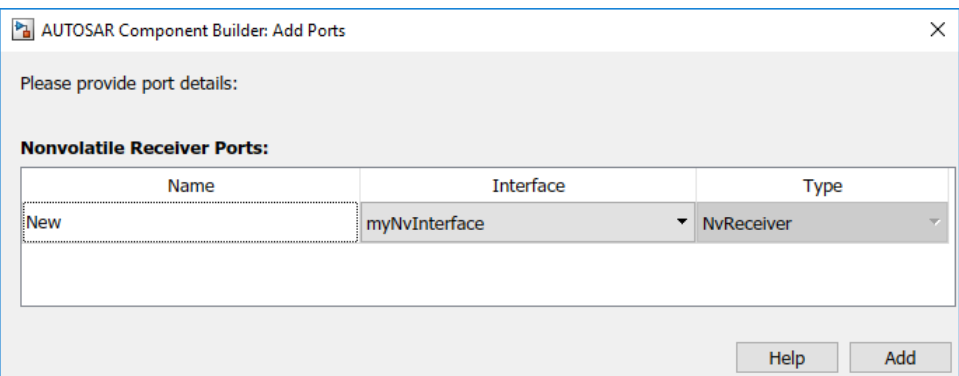
1. In the left-hand pane of AUTOSAR Dictionary, expand the component name and select **NvReceiverPorts**.

   The NV receiver ports view in 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 AUTOSAR NV receiver port by clicking its name and then editing the name text.
   - Click the **Add** button to open an Add Ports dialog box to add a port.
   - Select a port and then click the **Delete** button to remove it.

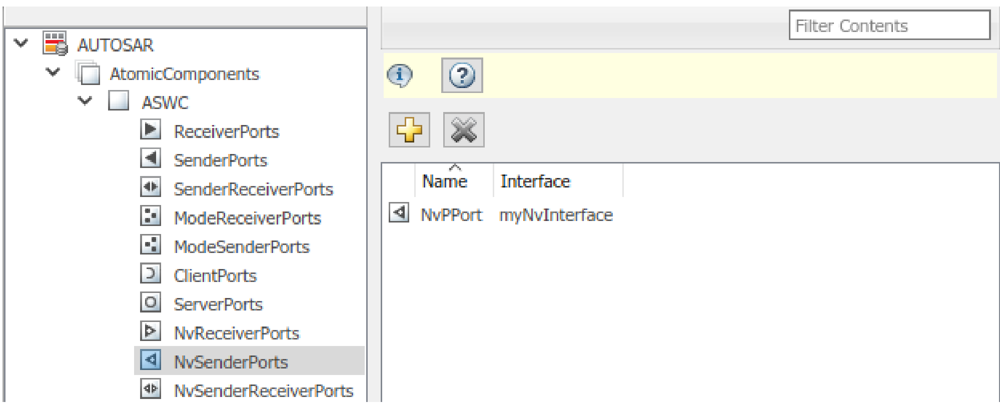
   The Add Ports dialog box lets you add an NV receiver port and associate it with an existing NV interface. Click **Add** to add the port and return to the NV receiver ports view.



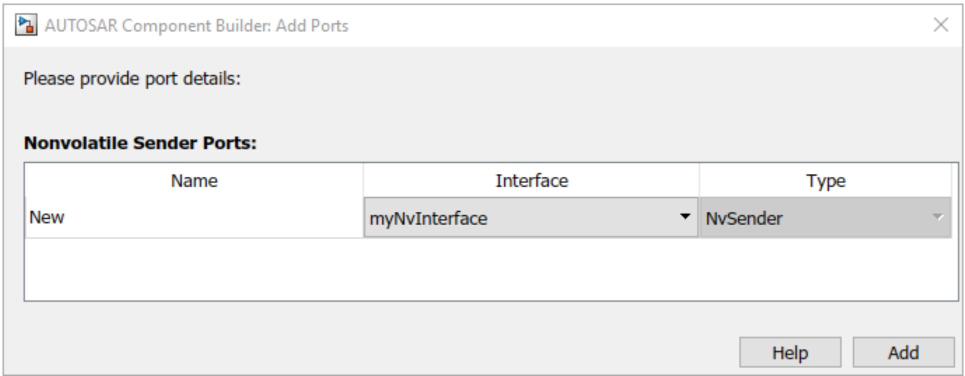
2 In the left-hand pane of AUTOSAR Dictionary, select **NvSenderPorts**.

The NV sender ports view in 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 AUTOSAR NV sender port by clicking its name and then editing the name text.
- Click the **Add** button to open an Add Ports dialog box to add a port.
- Select a port and then click the **Delete** button to remove it.



The Add Ports dialog box lets you add an NV sender port and associate it with an existing NV interface. Click **Add** to add the port and return to the NV sender ports view.

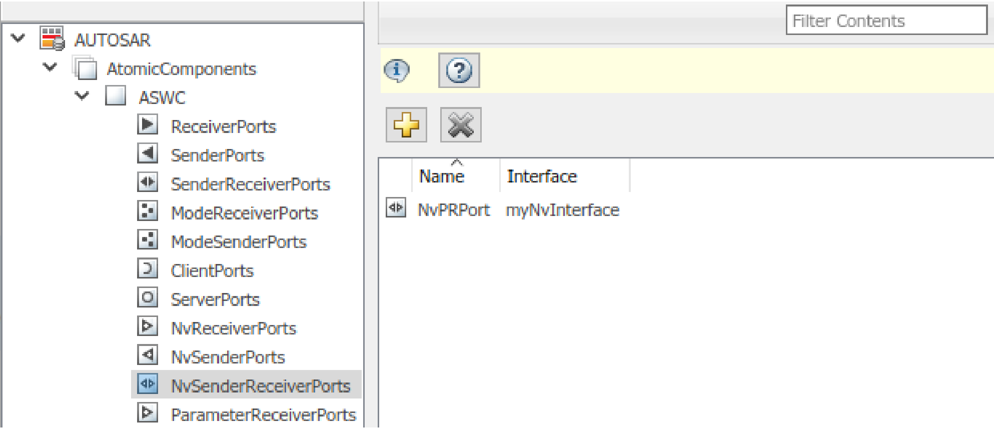


3 In the left-hand pane of AUTOSAR Dictionary, select **NvSenderReceiverPorts**.

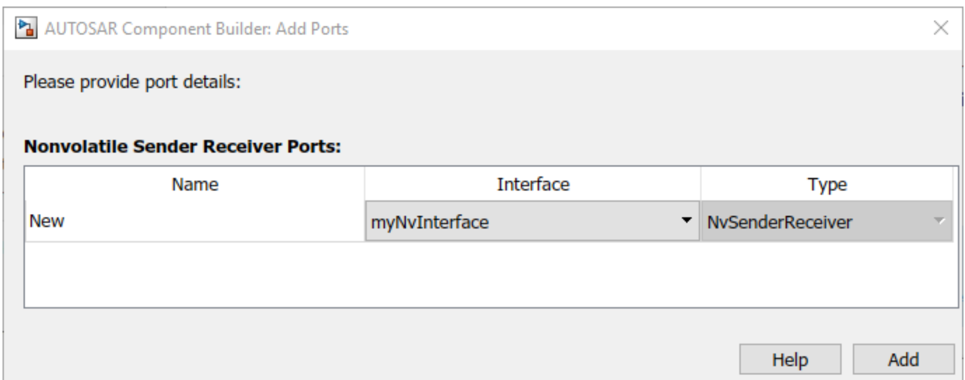
The NV sender-receiver ports view in AUTOSAR Dictionary lists NV sender-receiver ports and their properties. You can:

- Select an AUTOSAR NV sender-receiver port, and view and optionally reselect its associated NV data interface.
- Rename an AUTOSAR NV sender-receiver port by clicking its name and then editing the name text.
- Click the **Add** button to open an Add Ports dialog box to add a port.
- Select a port and then click the **Delete** button to remove it.

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.



The Add Ports dialog box lets you add an NV sender-receiver port and associate it with an existing NV interface. Click **Add** to add the port and return to the NV sender-receiver ports view.



Parameter Receiver Ports

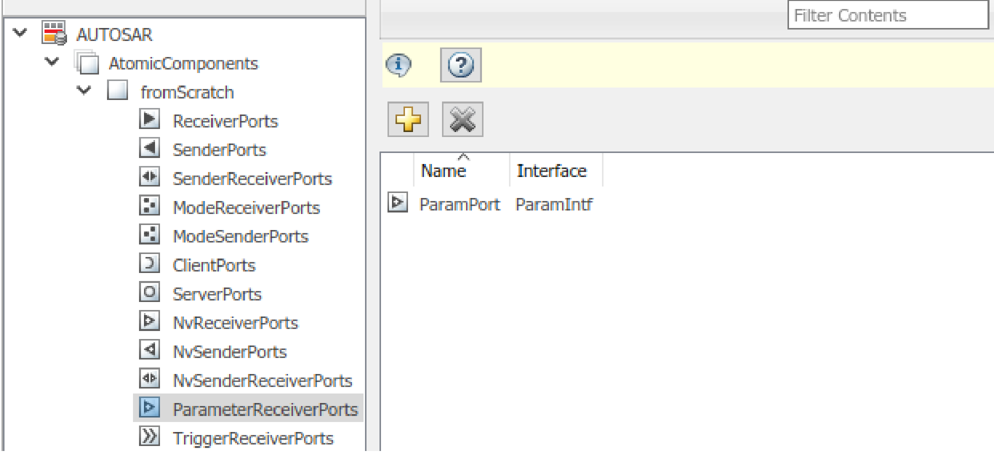
The AUTOSAR Dictionary view of parameter receiver ports supports modeling the receiver side of AUTOSAR parameter communication in Simulink. You use AUTOSAR Dictionary to configure AUTOSAR parameter receiver ports, parameter interfaces, and parameter data elements in your model. For more information, see “Configure Receiver for AUTOSAR Parameter Communication” on page 4-237.

To configure AUTOSAR parameter receiver port elements and properties, open a model for which a mapped AUTOSAR software component has been created and open AUTOSAR

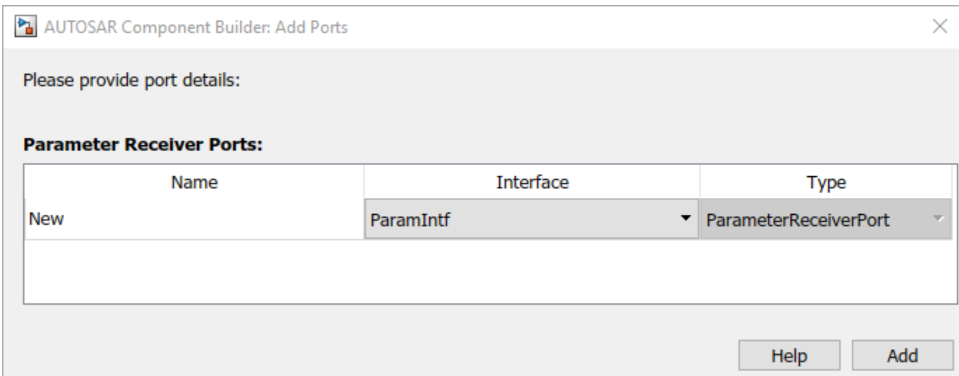
Dictionary. In the left-hand pane of AUTOSAR Dictionary, expand the component name and select **ParameterReceiverPorts**.

The parameter receiver ports view in 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 an AUTOSAR parameter receiver port by clicking its name and then editing the name text.
- Click the **Add** button to open an Add Ports dialog box to add a port.
- Select a port and then click the **Delete** button to remove it.



The Add Ports dialog box lets you specify the name of the new port and associate it with an existing parameter interface. Click **Add** to add the port and return to the parameter receiver ports view.



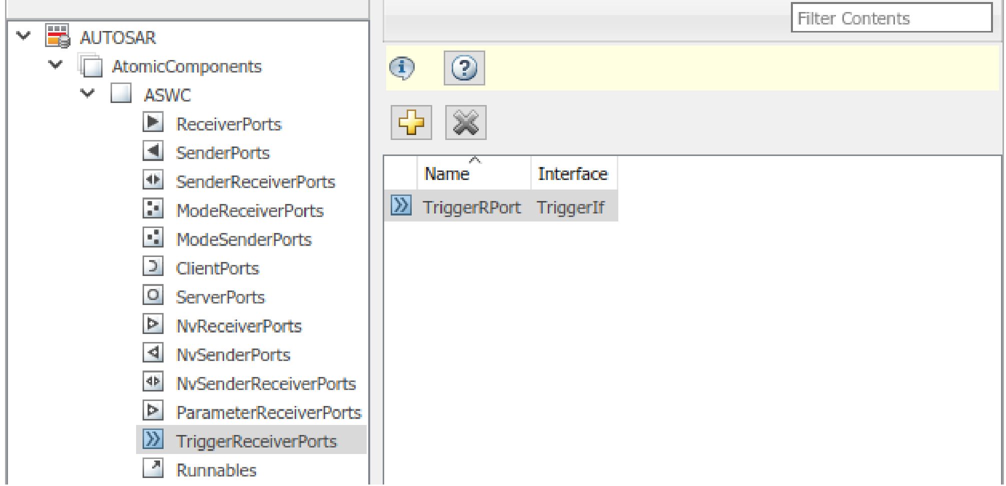
Trigger Receiver Ports

The AUTOSAR Dictionary view of trigger receiver ports supports modeling the receiver side of AUTOSAR trigger communication in Simulink. You use 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 4-240.

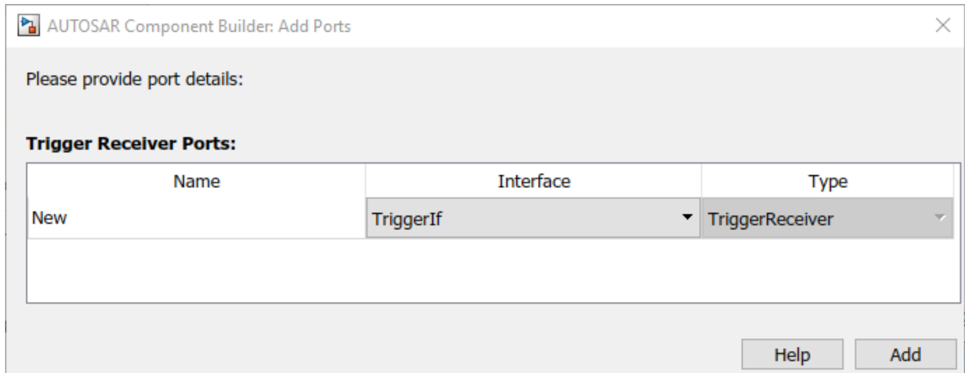
To configure AUTOSAR trigger receiver port elements and properties, open a model for which a mapped AUTOSAR software component has been created and open AUTOSAR Dictionary. In the left-hand pane of AUTOSAR Dictionary, expand the component name and select **TriggerReceiverPorts**.

The trigger receiver ports view in 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 an AUTOSAR trigger receiver port by clicking its name and then editing the name text.
- Click the **Add** button to open an Add Ports dialog box to add a port.
- Select a port and then click the **Delete** button to remove it.



The Add Ports dialog box lets you specify the name of the new port and associate it with an existing trigger interface. Click **Add** to add the port and return to the trigger receiver ports view.



Configure AUTOSAR Runnables

The **Runnables** view in AUTOSAR Dictionary supports modeling AUTOSAR runnable entities (runnables) and events, which implement aspects of internal AUTOSAR component behavior, in Simulink. You use AUTOSAR Dictionary to configure AUTOSAR runnables and associated events that activate them. For more information, see “Configure AUTOSAR Runnables and Events” on page 4-277.

In 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 AUTOSAR Dictionary. In the left-hand pane of AUTOSAR Dictionary, expand the component name and select **Runnables**.

The runnables view in AUTOSAR Dictionary lists runnables for the AUTOSAR component. You can:

- Rename an AUTOSAR runnable by clicking its name and then editing the name text.
- Modify the symbol name for a runnable. The specified AUTOSAR runnable symbol-name is exported in `arxml` and C code. For example, in the display below, 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 as shown below.

  Example 4.1. autosar_swc_expfcns.arxml

  ```
  <RUNNABLE-ENTITY UUID="...">
      <SHORT-NAME>Runnable1</SHORT-NAME>
  ...
      <SYMBOL>test_symbol</SYMBOL>
  ...
  </RUNNABLE-ENTITY>
  ```

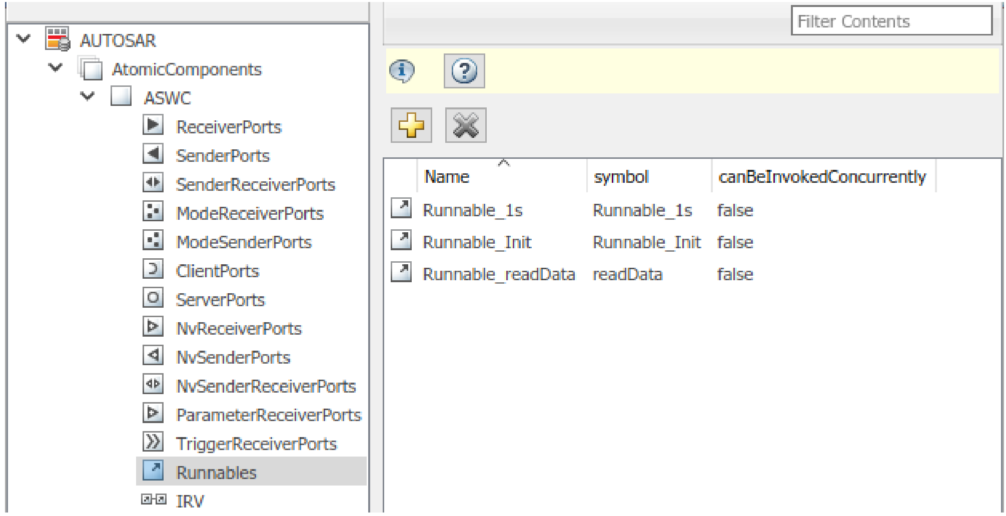
  Example 4.2. autosar__swc_expfcns.c

  ```
  /* Model step function for TID1 */
  void test_symbol(void)                /* Explicit Task: Runnable1 */
  {
  ...
  }
  ```

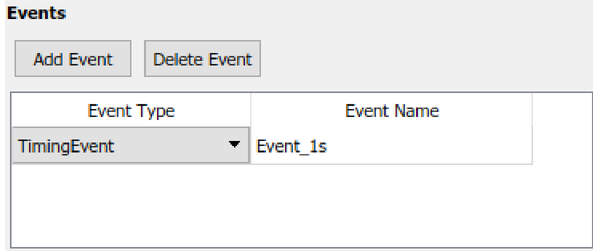
  Note For an AUTOSAR server runnable — that is, a runnable with an `OperationInvokedEvent` — the **symbol** name must match the Simulink server function name.

- 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 4-221.

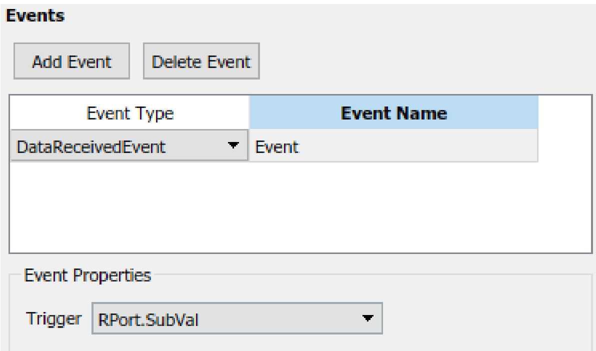
- Click the **Add** button to add an AUTOSAR runnable.
- Select an AUTOSAR runnable and then click the **Delete** button to remove it.



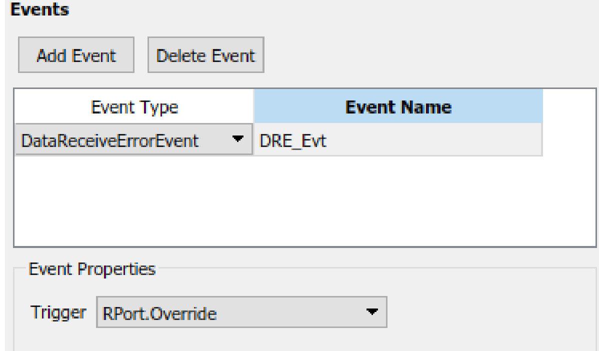
Select a runnable to see its list of associated events. The **Events** pane lists each AUTOSAR event with its type — `TimingEvent`, `DataReceivedEvent`, `ModeSwitchEvent`, `OperationInvokedEvent`, `InitEvent`, `DataReceiveErrorEvent`, or `ExternalTriggerOccurredEvent` — and name. You can rename an AUTOSAR event by clicking its name and then editing the name text. You can use the buttons **Add Event** and **Delete Event** to add or delete events from a runnable.



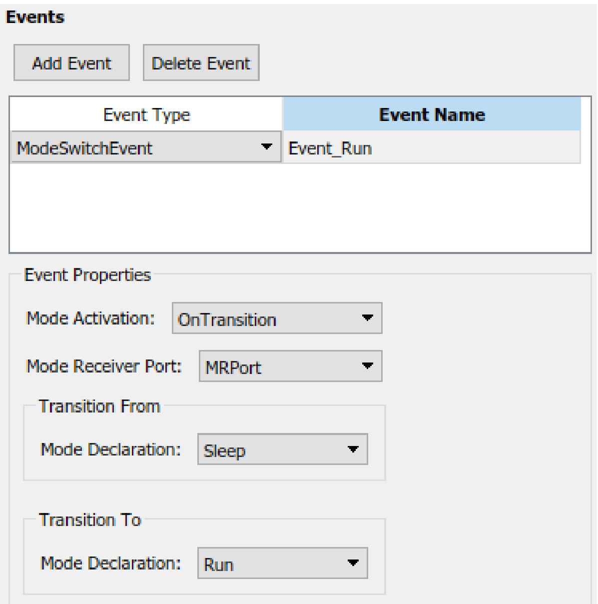
If you select an event of type `DataReceivedEvent`, the runnable is activated by a `DataReceivedEvent`. Select the event name to display its **Trigger** property. Select a trigger for the event from the list of available trigger ports.



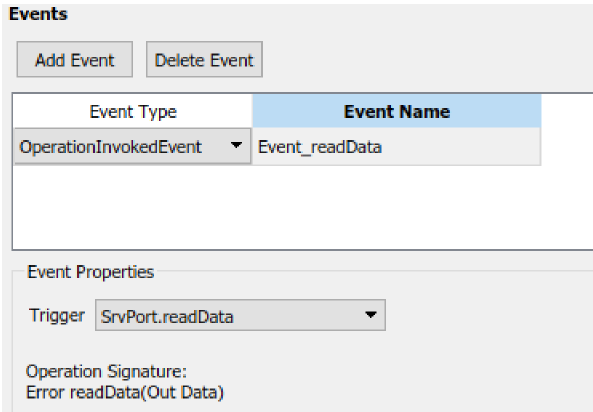
If you select an event of type `DataReceiveErrorEvent`, the runnable is activated by a `DataReceiveErrorEvent`. Select the event name to display its **Trigger** property. Select a trigger for the event from the list of available trigger ports. (For more information on using a `DataReceiveErrorEvent`, see “Configure AUTOSAR Receiver Port for DataReceiveErrorEvent” on page 4-165.)



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 4-225.)

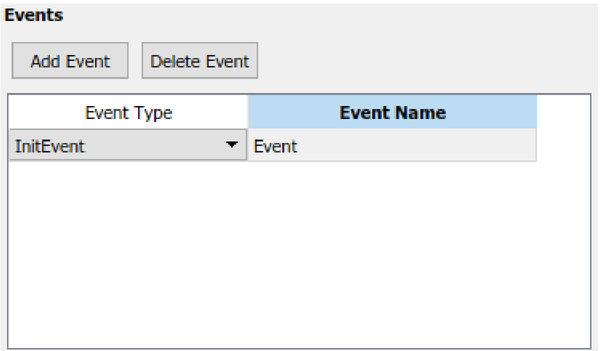


If you select an event of type `OperationInvokedEvent`, the runnable becomes an AUTOSAR server runnable. Select the event name to display its **Trigger** property. Select a trigger for the event from the list of available server port and operation combinations. The **Operation Signature** is displayed below the **Trigger** property. (For more information on using an `OperationInvokedEvent`, see “Configure AUTOSAR Client-Server Communication” on page 4-197.)

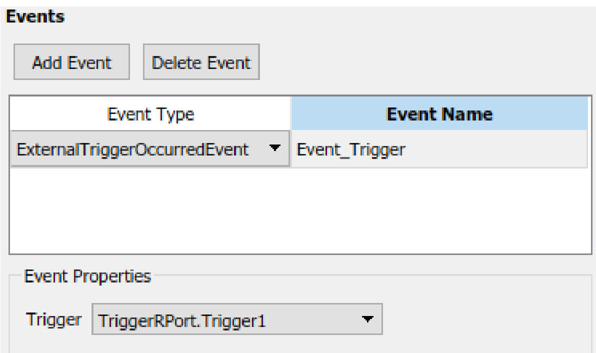


If you select an event of type `InitEvent`, you can rename the event by clicking its name and then editing the name text. (For more information on using an `InitEvent`, see “Configure AUTOSAR Initialization Runnable (R4.1)” on page 4-291.)

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.



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 4-240.)



Configure AUTOSAR Inter-Runnable Variables

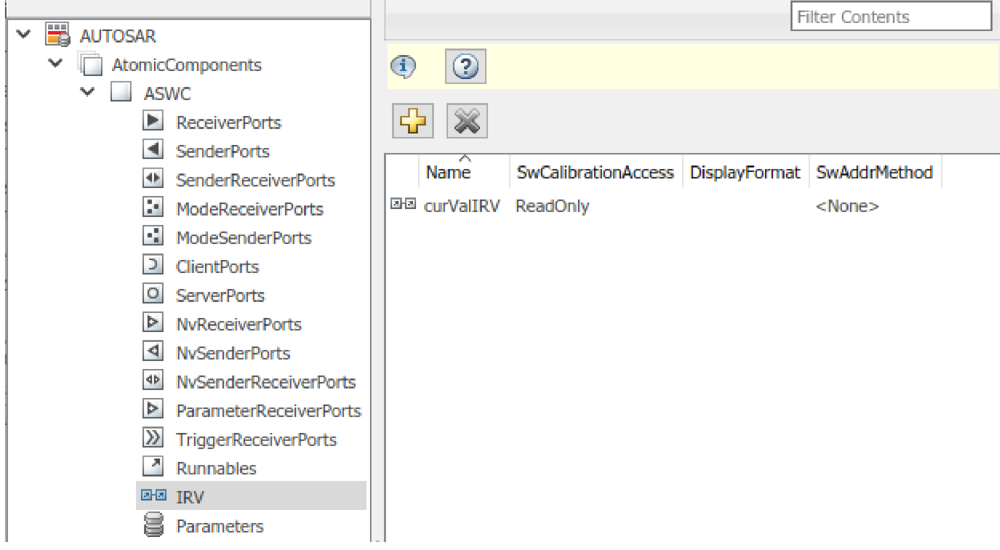
The **IRV** view in AUTOSAR Dictionary supports modeling AUTOSAR inter-runnable variables (IRVs), which connect runnables and implement aspects of internal AUTOSAR component behavior, in Simulink. You use AUTOSAR Dictionary to create AUTOSAR IRVs and configure IRV data properties. For more information, see “Configure AUTOSAR Data for Measurement and Calibration” on page 4-263.

In 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 AUTOSAR Dictionary. In the left-hand pane of AUTOSAR Dictionary, expand the component name and select **IRV**.

The IRV view in AUTOSAR Dictionary lists IRVs for the AUTOSAR component. You can:

- Rename an AUTOSAR IRV by clicking its name and then editing the name text.
- Specify the level of measurement and calibration 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 measurement and calibration 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 4-266.
- 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 measurement and calibration tools. For more information, see “Configure AUTOSAR SwAddrMethods” on page 4-60.
- Click the **Add** button to add an AUTOSAR IRV.
- Select an AUTOSAR IRV and then click the **Delete** button to remove it.



Configure AUTOSAR Parameters

The **Parameters** view in AUTOSAR Dictionary supports modeling AUTOSAR internal calibration parameters, for use with AUTOSAR integrated and distributed lookups, in Simulink. You use AUTOSAR Dictionary to create AUTOSAR internal parameters and configure parameter data properties. For port-based calibration parameters, you create “Parameter Interfaces” on page 4-50.

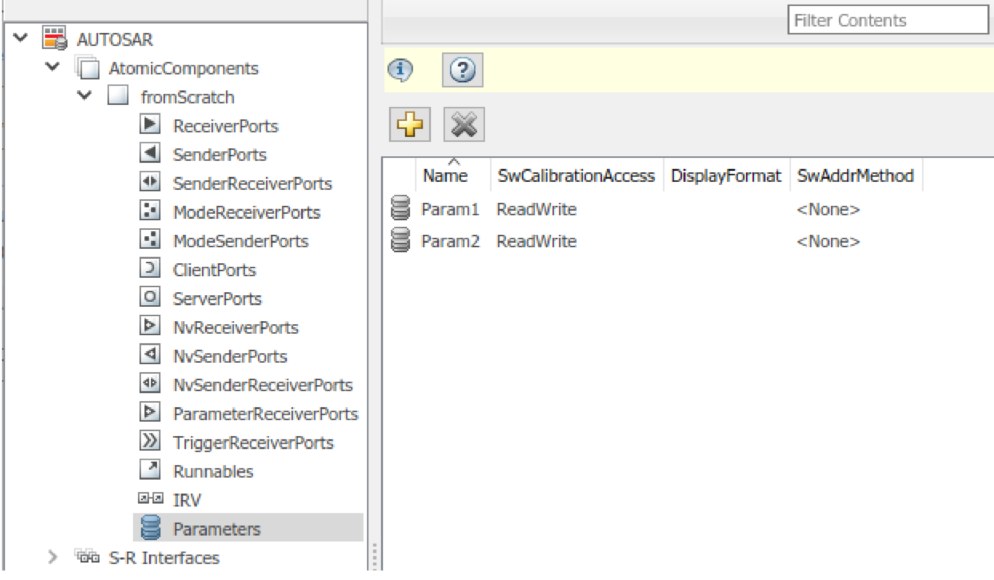
In AUTOSAR Dictionary, internal parameters appear in a tree format under the component that owns them. To access parameter elements and their properties, you expand the component name.

To configure AUTOSAR parameter elements and properties, open a model for which a mapped AUTOSAR software component has been created and open AUTOSAR Dictionary. In the left-hand pane of AUTOSAR Dictionary, expand the component name and select **Parameters**.

The parameters view in AUTOSAR Dictionary lists internal parameters for the AUTOSAR component. You can:

- Rename an AUTOSAR parameter by clicking its name and then editing the name text.

- Specify the level of measurement and calibration 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 measurement and calibration 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 4-266.
- 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 measurement and calibration tools. For more information, see “Configure AUTOSAR SwAddrMethods” on page 4-60.
- Click the **Add** button to add an AUTOSAR internal parameter.
- Select an AUTOSAR internal parameter and then click the **Delete** button to remove it.



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 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 4-37
- “Mode-Switch Interfaces” on page 4-40
- “Client-Server Interfaces” on page 4-42
- “Nonvolatile Data Interfaces” on page 4-47
- “Parameter Interfaces” on page 4-50
- “Trigger Interfaces” on page 4-53

Sender-Receiver Interfaces

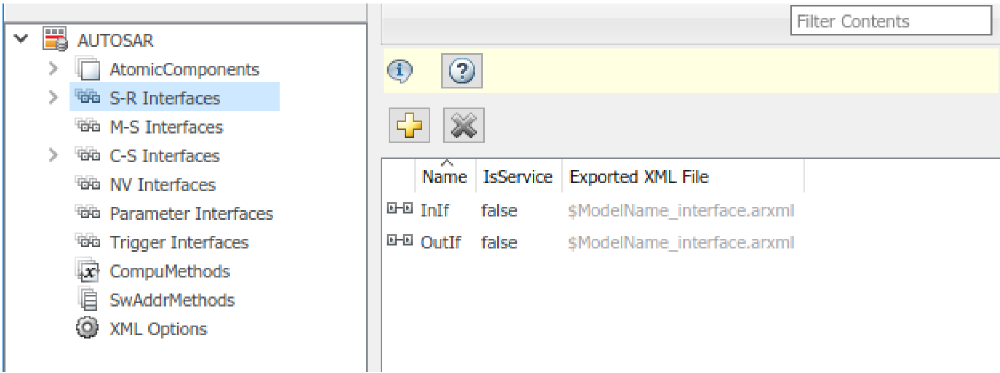
The **S-R Interfaces** view in AUTOSAR Dictionary supports modeling AUTOSAR sender-receiver (S-R) communication in Simulink. You use AUTOSAR Dictionary to configure AUTOSAR S-R ports, S-R interfaces, and S-R data elements in your model. For more information, see “Configure AUTOSAR Sender-Receiver Communication” on page 4-153 and “Configure AUTOSAR Queued Sender-Receiver Communication” on page 4-172.

To configure AUTOSAR S-R interface elements and properties, open a model for which a mapped AUTOSAR software component has been created and open AUTOSAR Dictionary.

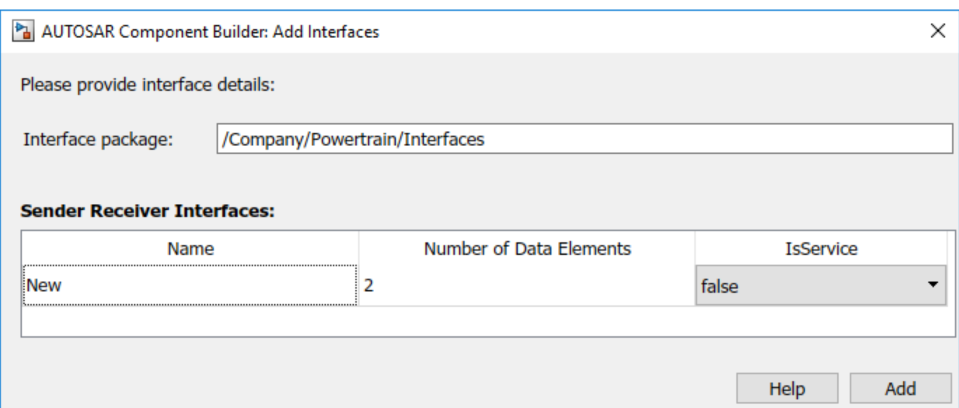
1. In the left-hand pane of AUTOSAR Dictionary, select **S-R Interfaces**.

   The S-R interfaces view in 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 clicking its name and then editing the name text.
   - Click the **Add** button to open an Add Interfaces dialog box to add one or more S-R interfaces.
   - Select an S-R interface and then click the **Delete** button to remove it.



The Add Interfaces dialog box lets you specify the name of the new interface, the number of associated data elements it contains, whether the interface is a service, and the path of the Interface package. Click **Add** to add the interface and return to the S-R interfaces view.



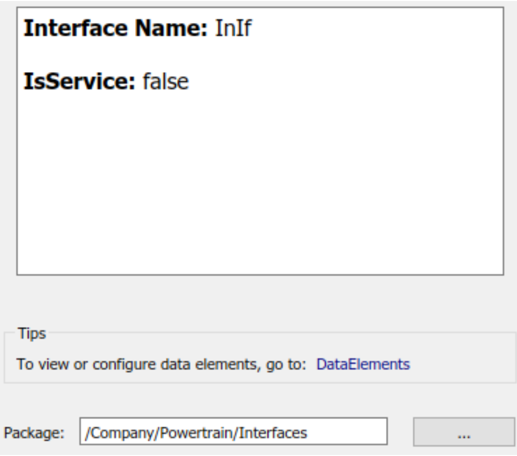
2 In the left-hand pane of AUTOSAR Dictionary, expand **S-R Interfaces** and select an S-R interface from the list.

The S-R interface view in 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 4-150.

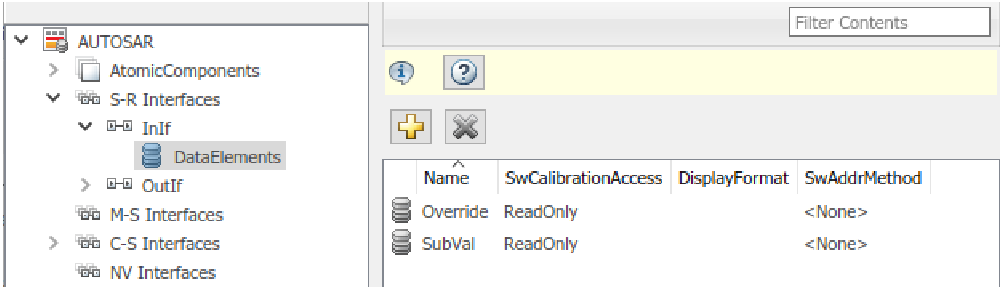


3 In the left-hand pane of AUTOSAR Dictionary, expand the selected interface and select **DataElements**.

The data elements view in AUTOSAR Dictionary lists AUTOSAR sender-receiver interface data elements and their properties. You can:

- Select an S-R interface data element and edit the name value.
- Specify the level of measurement and calibration 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 measurement and calibration 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 4-266.

- 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 measurement and calibration tools. For more information, see “Configure AUTOSAR SwAddrMethods” on page 4-60.
- Click the **Add** button to add a data element.
- Select a data element and then click the **Delete** button to remove it.



Mode-Switch Interfaces

The **M-S Interfaces** view in AUTOSAR Dictionary supports modeling AUTOSAR mode-switch (M-S) communication in Simulink. You use 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 4-225.

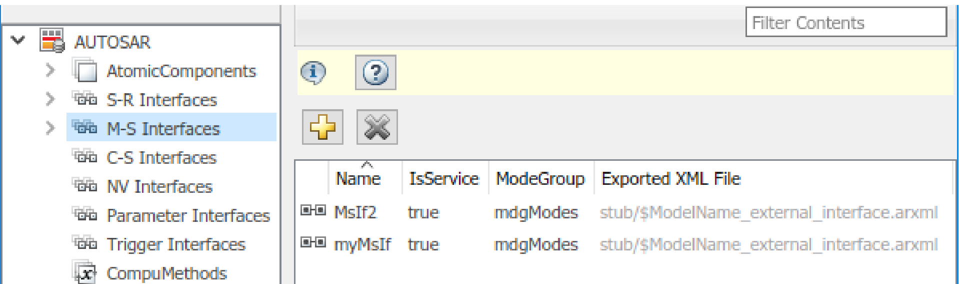
To configure AUTOSAR M-S interface elements and properties, open a model for which a mapped AUTOSAR software component has been created and open AUTOSAR Dictionary.

1. In the left-hand pane of AUTOSAR Dictionary, select **M-S Interfaces**.

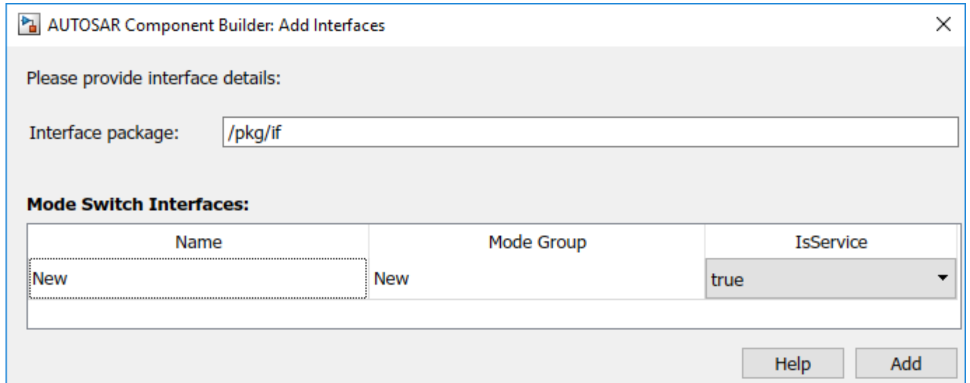
   The M-S interfaces view in AUTOSAR Dictionary lists AUTOSAR mode-switch interfaces and their properties. You can:

   - 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 4-225.
- Rename an M-S interface by clicking its name and then editing the name text.
- Click the **Add** button to open an Add Interfaces dialog box to add one or more M-S interfaces.
- Select an M-S interface and then click the **Delete** button to remove it.



The Add Interfaces dialog box lets you specify the name of the new interface, the name of a mode group, whether the interface is a service, and the path of the Interface package. Click **Add** to add the interface and return to the M-S interfaces view.

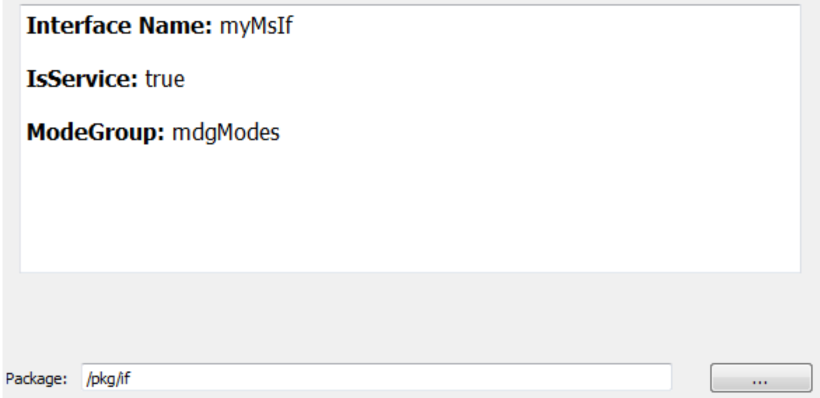


2 In the left-hand pane of AUTOSAR Dictionary, expand **M-S Interfaces** and select an M-S interface from the list.

The M-S interface view in 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 4-150.



Client-Server Interfaces

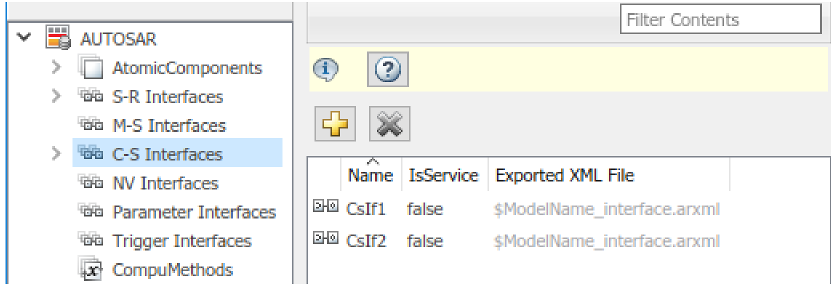
The **C-S Interfaces** view in AUTOSAR Dictionary supports modeling AUTOSAR client-server (C-S) communication in Simulink. You use AUTOSAR Dictionary to configure AUTOSAR C-S ports, C-S interfaces, and C-S operations in your model. For more information, see “Configure AUTOSAR Client-Server Communication” on page 4-197.

To configure AUTOSAR C-S interface elements and properties, open a model for which a mapped AUTOSAR software component has been created and open AUTOSAR Dictionary.

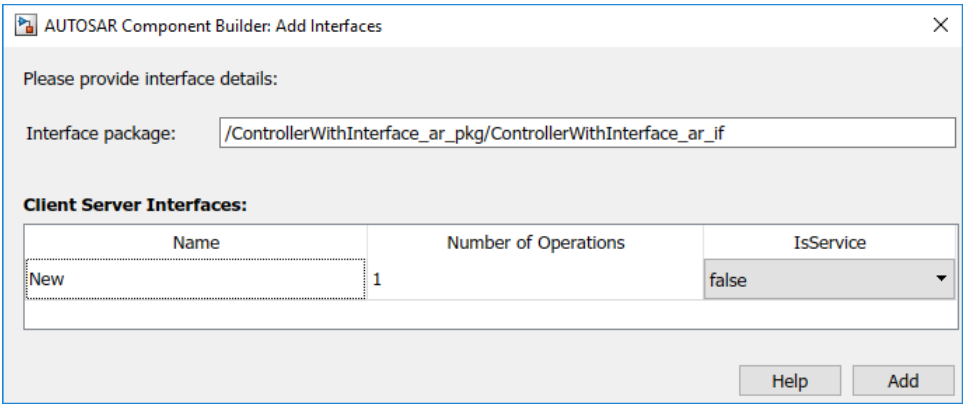
1 In the left-hand pane of AUTOSAR Dictionary, select **C-S Interfaces**.

   The C-S interfaces view in 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 clicking its name and then editing the name text.
- Click the **Add** button to open an Add Interfaces dialog box to add one or more C-S interfaces.
- Select a C-S interface and then click the **Delete** button to remove it.



The Add Interfaces dialog box lets you specify the name of the new interface, the number of associated operations it contains, whether the interface is a service, and the path of the Interface package. Click **Add** to add the interface and return to the C-S interfaces view.



2 In the left-hand pane of AUTOSAR Dictionary, expand **C-S Interfaces** and select a C-S interface from the list.

The C-S interface view in 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.

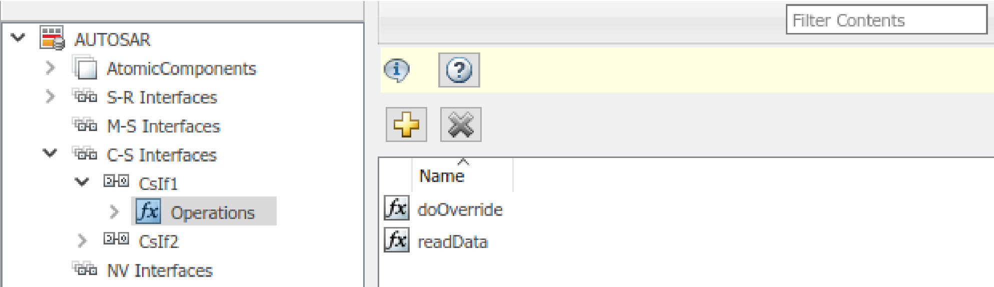
To modify the AUTOSAR package for the interface, you can do either of the following:

- 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 4-150.

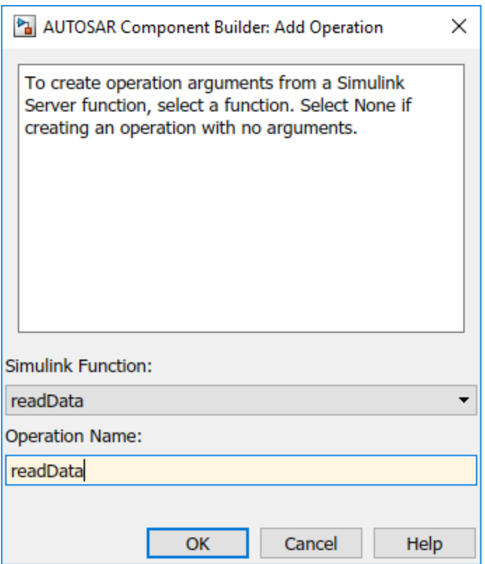
3 In the left-hand pane of AUTOSAR Dictionary, expand the selected interface and select **Operations**.

The operations view in AUTOSAR Dictionary lists AUTOSAR client-server interface operations. You can:

- Select a C-S interface operation and edit the name value.
- Click the **Add** button to open an Add Operation dialog box to add a C-S interface operation.
- Select an operation and then click the **Delete** button to remove it.

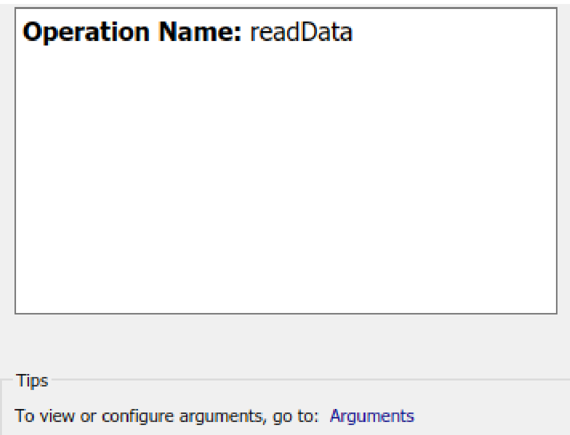


The Add Operation dialog box lets you specify the name of a new C-S interface operation. To create operation arguments from a Simulink function, select the associated Simulink function among those present in the configuration. Select `None` if you are creating an operation without arguments.



4 In the left-hand pane of AUTOSAR Dictionary, expand **Operations** and select an operation from the list.

The operations view in AUTOSAR Dictionary displays the name of the selected C-S operation.

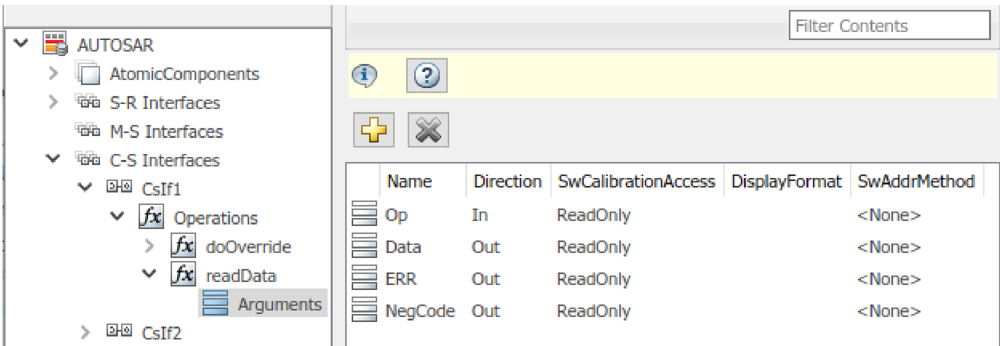


5 In the left-hand pane of AUTOSAR Dictionary, expand the selected operation and select **Arguments**.

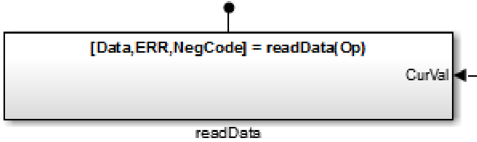
The arguments view in AUTOSAR Dictionary lists AUTOSAR client-server operation arguments and their properties. You can:

- Select a C-S operation argument and edit the 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 4-216.
- Specify the level of measurement and calibration 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 measurement and calibration 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 4-266.
- 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 measurement and calibration tools. For more information, see "Configure AUTOSAR SwAddrMethods" on page 4-60.

- Click the **Add** button to add an argument.
- Select an argument and then click the **Delete** button to remove it.



The displayed server operation arguments were created from the following Simulink Function block.



Nonvolatile Data Interfaces

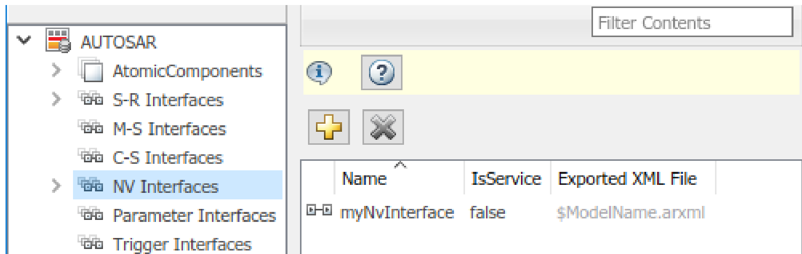
The **NV Interfaces** view in AUTOSAR Dictionary supports modeling AUTOSAR nonvolatile (NV) data communication in Simulink. You use AUTOSAR Dictionary to configure AUTOSAR NV ports, NV interfaces, and NV data elements in your model. For more information, see “Configure AUTOSAR Nonvolatile Data Communication” on page 4-234.

To configure AUTOSAR NV interface elements and properties, open a model for which a mapped AUTOSAR software component has been created and open AUTOSAR Dictionary.

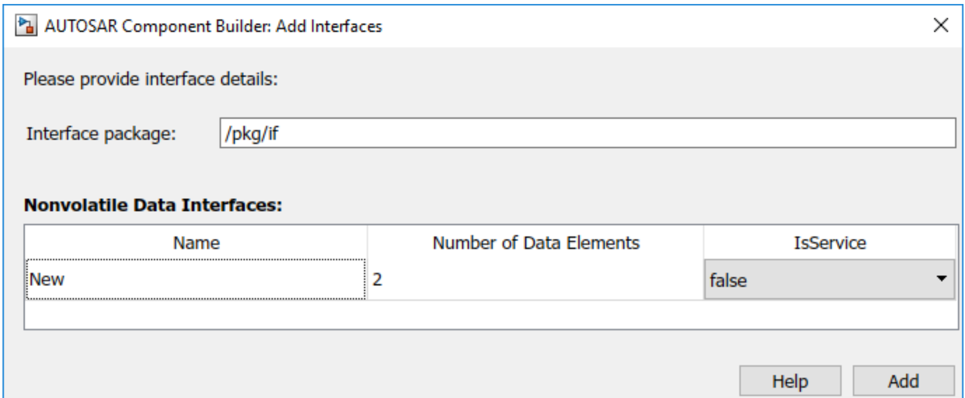
1 In the left-hand pane of AUTOSAR Dictionary, select **NV Interfaces**.

   The NV interfaces view in 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 clicking its name and then editing the name text.
- Click the **Add** button to open an Add Interfaces dialog box to add one or more NV interfaces.
- Select an NV interface and then click the **Delete** button to remove it.



The Add Interfaces dialog box lets you specify the name of the new interface, the number of associated data elements it contains, whether the interface is a service, and the path of the Interface package. Click **Add** to add the interface and return to the NV interfaces view.

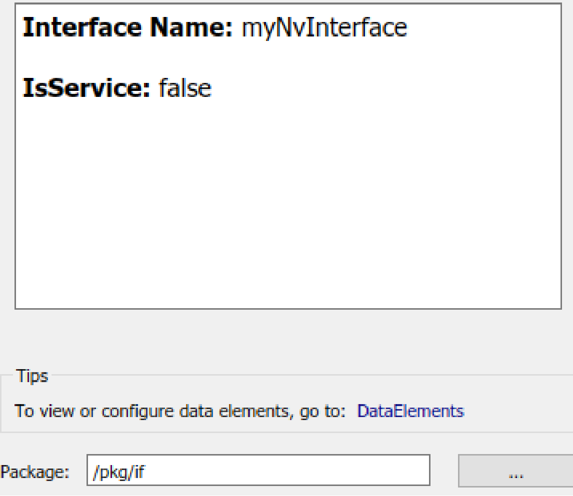


2 In the left-hand pane of AUTOSAR Dictionary, expand **NV Interfaces** and select an NV interface from the list.

The NV interface view in 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 4-150.

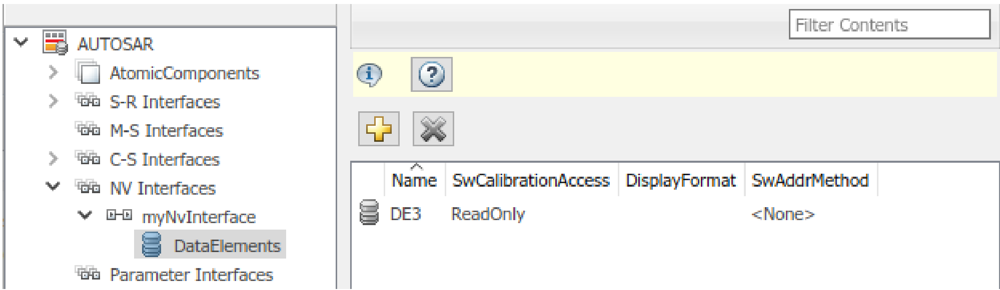


3 In the left-hand pane of AUTOSAR Dictionary, expand the selected interface and select **DataElements**.

   The data elements view in AUTOSAR Dictionary lists AUTOSAR NV interface data elements and their properties. You can:

   - Select an NV interface data element and edit the name value.
   - Specify the level of measurement and calibration 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 measurement and calibration 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 4-266.
- 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 measurement and calibration tools. For more information, see “Configure AUTOSAR SwAddrMethods” on page 4-60.
- Click the **Add** button to add a data element.
- Select a data element and then click the **Delete** button to remove it.



Parameter Interfaces

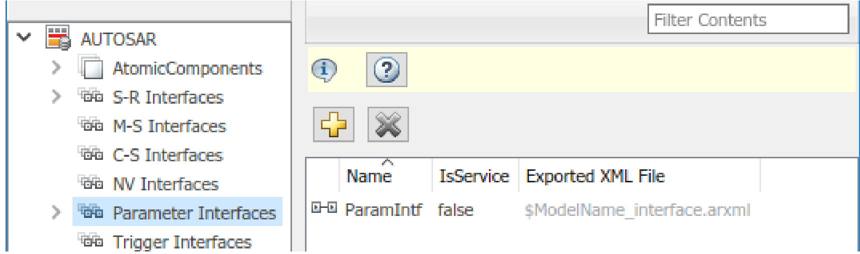
The **Parameter Interfaces** view in AUTOSAR Dictionary supports modeling the receiver side of AUTOSAR parameter communication in Simulink. You use AUTOSAR Dictionary to configure AUTOSAR parameter receiver ports, parameter interfaces, and parameter data elements in your model. For more information, see “Configure Receiver for AUTOSAR Parameter Communication” on page 4-237.

To configure AUTOSAR parameter interface elements and properties, open a model for which a mapped AUTOSAR software component has been created and open AUTOSAR Dictionary.

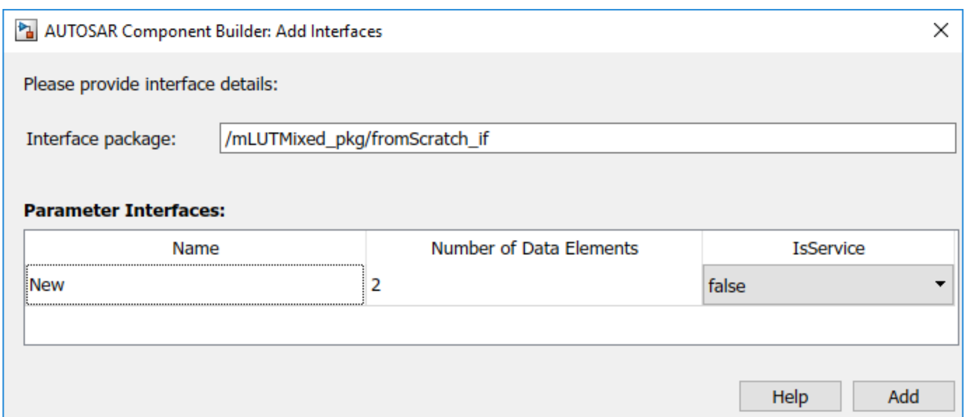
1. In the left-hand pane of AUTOSAR Dictionary, select **Parameter Interfaces**.

The parameter interfaces view in 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 clicking its name and then editing the name text.
- Click the **Add** button to open an Add Interfaces dialog box to add one or more parameter interfaces.
- Select a parameter interface and then click the **Delete** button to remove it.



The Add Interfaces dialog box lets you specify the name of the new interface, the number of associated data elements it contains, whether the interface is a service, and the path of the Interface package. Click **Add** to add the interface and return to the parameter interfaces view.

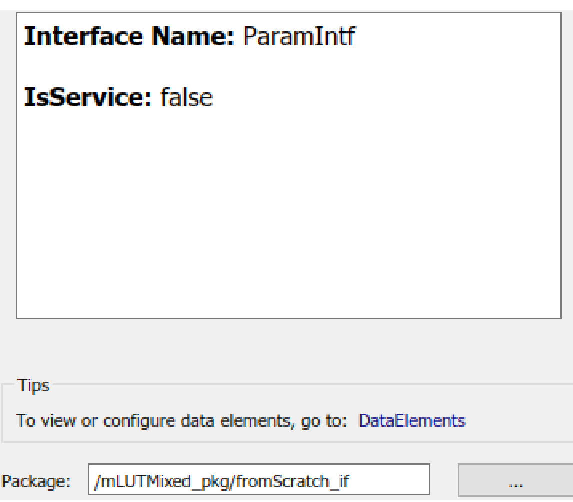


2 In the left-hand pane of AUTOSAR Dictionary, expand **Parameter Interfaces** and select a parameter interface from the list.

The parameter interface view in 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 4-150.

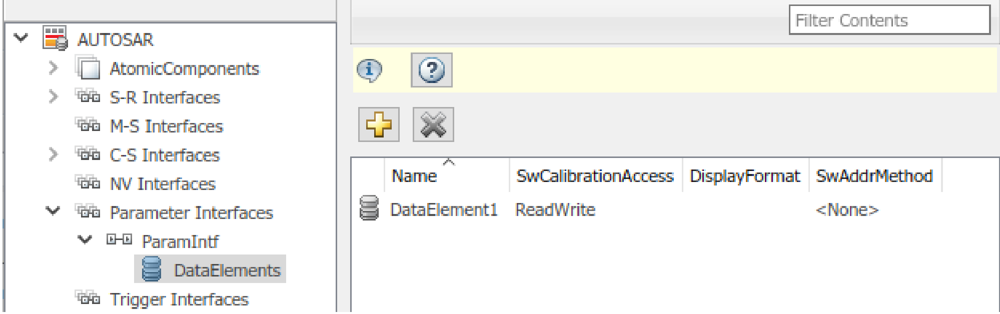


3 In the left-hand pane of AUTOSAR Dictionary, expand the selected interface and select **DataElements**.

The data elements view in AUTOSAR Dictionary lists AUTOSAR parameter interface data elements and their properties. You can:

- Select a parameter interface data element and edit the name value.

- Specify the level of measurement and calibration 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 measurement and calibration 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 4-266.
- 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 measurement and calibration tools. For more information, see “Configure AUTOSAR SwAddrMethods” on page 4-60.
- Click the **Add** button to add a data element.
- Select a data element and then click the **Delete** button to remove it.



Trigger Interfaces

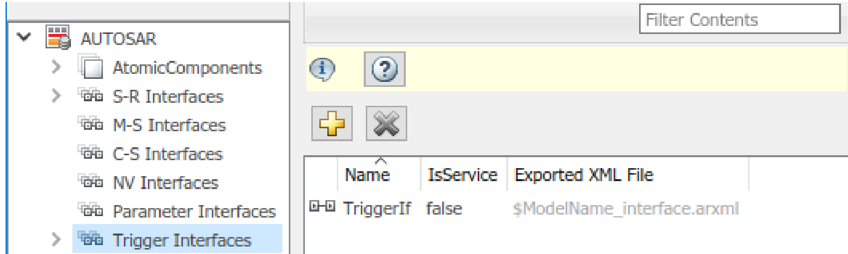
The **Trigger Interfaces** view in AUTOSAR Dictionary supports modeling the receiver side of AUTOSAR trigger communication in Simulink. You use AUTOSAR Dictionary to configure AUTOSAR trigger receiver ports, trigger interfaces, and triggers in your model. For more information, see “Configure Receiver for AUTOSAR External Trigger Event Communication” on page 4-240.

To configure AUTOSAR trigger interface elements and properties, open a model for which a mapped AUTOSAR software component has been created and open AUTOSAR Dictionary.

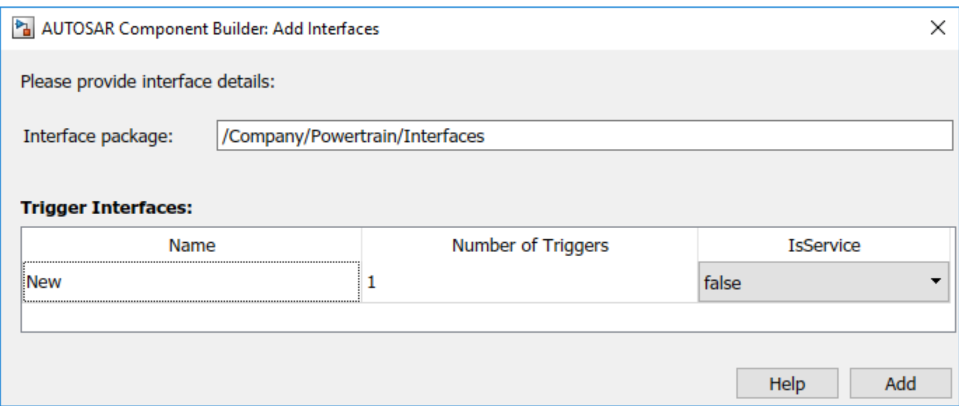
1 In the left-hand pane of AUTOSAR Dictionary, select **Trigger Interfaces**.

   The trigger interfaces view in 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 clicking its name and then editing the name text.
   - Click the **Add** button to open an Add Interfaces dialog box to add one or more trigger interfaces.
   - Select a trigger interface and then click the **Delete** button to remove it.



   The Add Interfaces dialog box lets you specify the name of the new interface, the number of associated triggers it contains, whether the interface is a service, and the path of the Interface package. Click **Add** to add the interface and return to the trigger interfaces view.

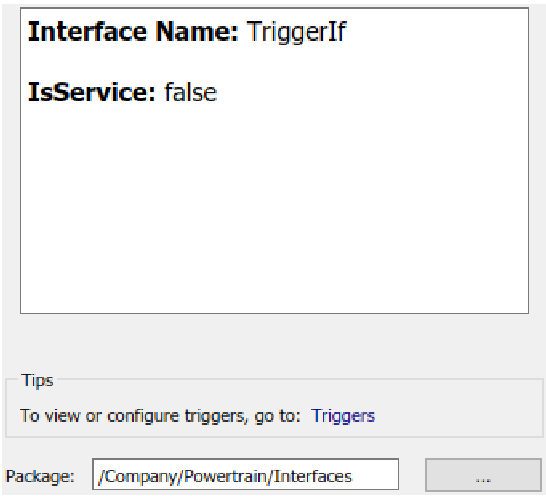


2 In the left-hand pane of AUTOSAR Dictionary, expand **Trigger Interfaces** and select a trigger interface from the list.

The trigger interface view in 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 4-150.



3 In the left-hand pane of AUTOSAR Dictionary, expand the selected interface and select **Triggers**.

The triggers view in AUTOSAR Dictionary lists AUTOSAR triggers and their properties. You can:

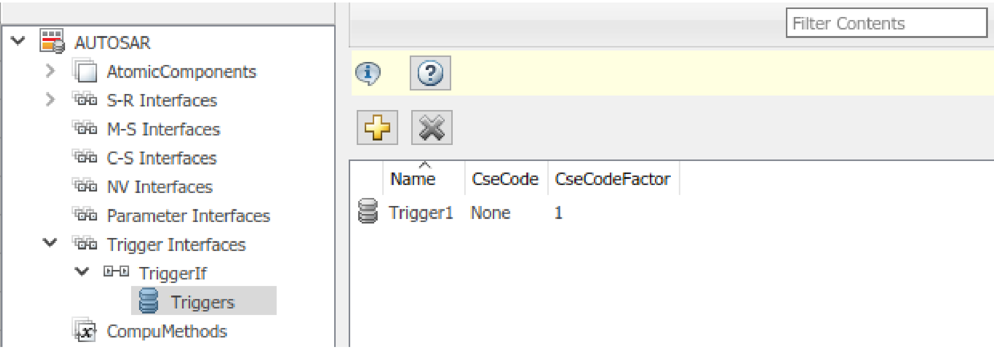
- Select a trigger and edit the name value.
- If the trigger is periodic, you can use **CseCode** and **CseCodeFactor** to specify a period for the trigger. (Otherwise, leave the period unspecified.)
  - To specify the time base of the period, select a value from the **CseCode** menu. The values are 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 µsec (microsecond)
`CSE1`	10 µsec

CseCode	Time Base
CSE2	100 µsec
CSE3	1 msec (millisecond)
CSE4	10 msec
CSE5	100 msec
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)

- Click the **Add** button to add a trigger.
- Select a trigger and then click the **Delete** button to remove it.



Configure AUTOSAR Computation Methods

The **CompuMethods** view in AUTOSAR Dictionary supports modeling AUTOSAR computation methods (CompuMethods), which specify conversions between internal values and physical representation of AUTOSAR data, in Simulink. You use AUTOSAR Dictionary to create and configure AUTOSAR CompuMethods. For more information, see “Configure AUTOSAR CompuMethods” on page 4-320.

To configure AUTOSAR CompuMethod elements and properties, open a model for which a mapped AUTOSAR software component has been created and open AUTOSAR Dictionary. Select **CompuMethods**.

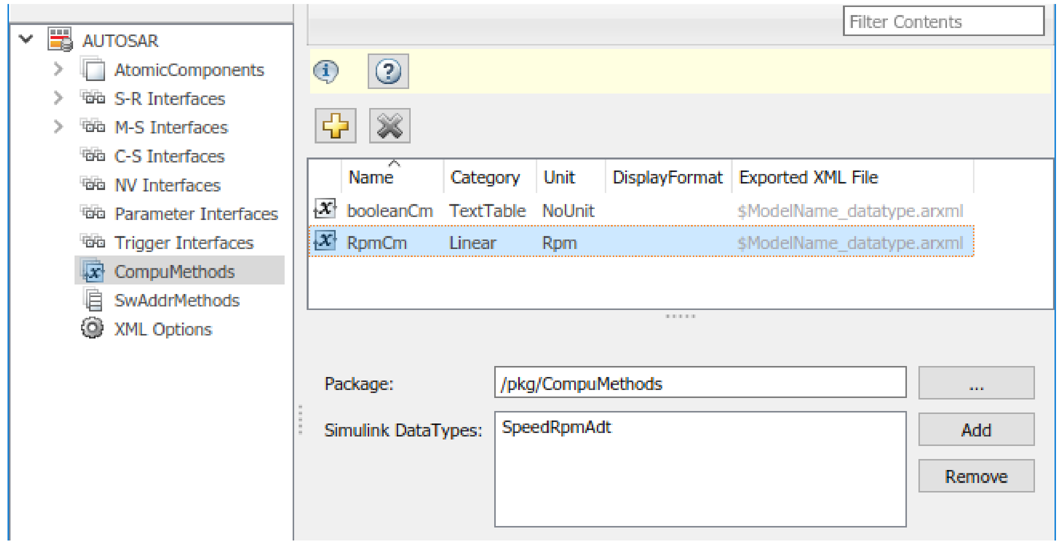
The CompuMethods view in AUTOSAR Dictionary displays CompuMethods and their properties. You can:

- Select a CompuMethod and modify properties, such as name, category, unit, display format for measurement and calibration, 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 4-320.
- Click the **Add** button to open an Add CompuMethod dialog box to add a CompuMethod.
- Select a CompuMethod and then click the **Delete** button to remove it.

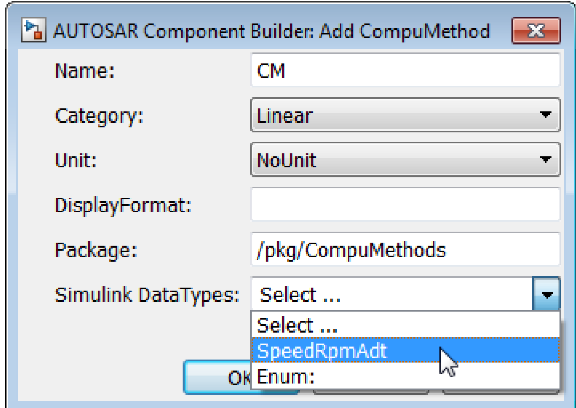
To modify the AUTOSAR package for a CompuMethod, you can do either of the following:

- 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 4-150.

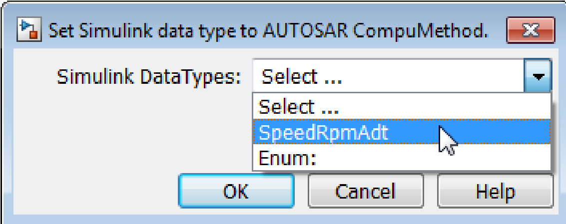
To associate a CompuMethod with a Simulink data type used in the model, select a CompuMethod and click the **Add** button to the right of **Simulink DataTypes**. This action opens a dialog box with a list of available data types. Select a data type and click OK to add it to the **Simulink DataTypes** list. 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 new CompuMethod and specify its initial properties, such as name, category, unit, display format for measurement and calibration, 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 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 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 4-270.

To configure AUTOSAR SwAddrMethod elements and properties, open a model for which a mapped AUTOSAR software component has been created and open AUTOSAR Dictionary. Select **SwAddrMethods**.

The SwAddrMethods view in 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.

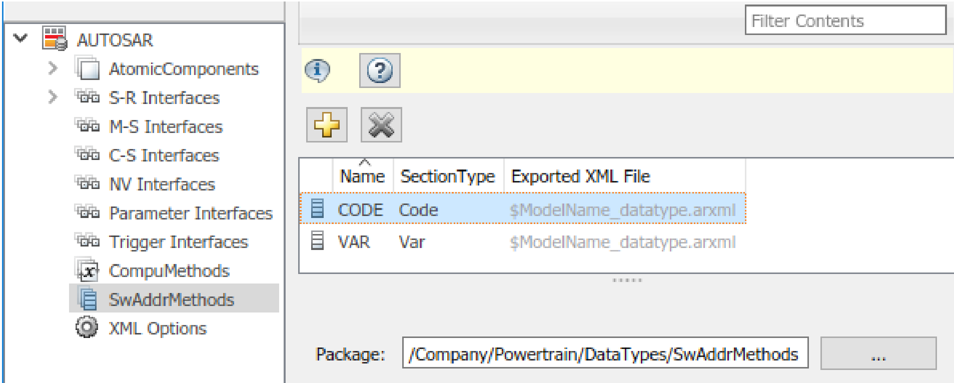
  To modify the section type, select a value from the **SectionType** drop-down list. The listed values correspond to SwAddrMethod section types listed in the AUTOSAR standard.

SectionType Value	SwAddrMethod Section Type
`CalibrationVariables`	CALIBRATION-VARIABLES
`Calprm`	CALPRM
`Code`	CODE
`ConfigData`	CONFIG-DATA
`Const`	CONST
`ExcludeFromFlash`	EXCLUDE-FROM-FLASH
`Var`	VAR

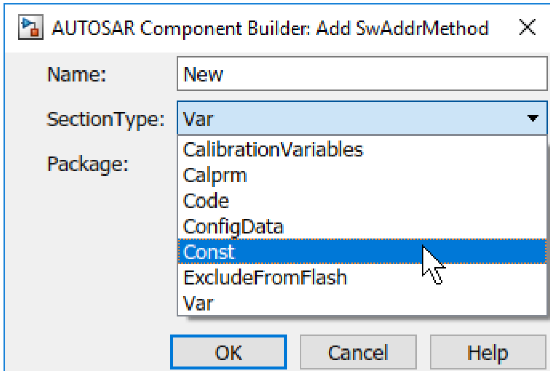
- Click the **Add** button to open an Add SwAddrMethod dialog box to add a SwAddrMethod.
- Select a SwAddrMethod and then click the **Delete** button to remove it.

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 4-150.



The Add SwAddrMethod dialog box lets you create a new SwAddrMethod and specify its initial properties, such as name, section type, and AUTOSAR package to be generated for the SwAddrMethod.



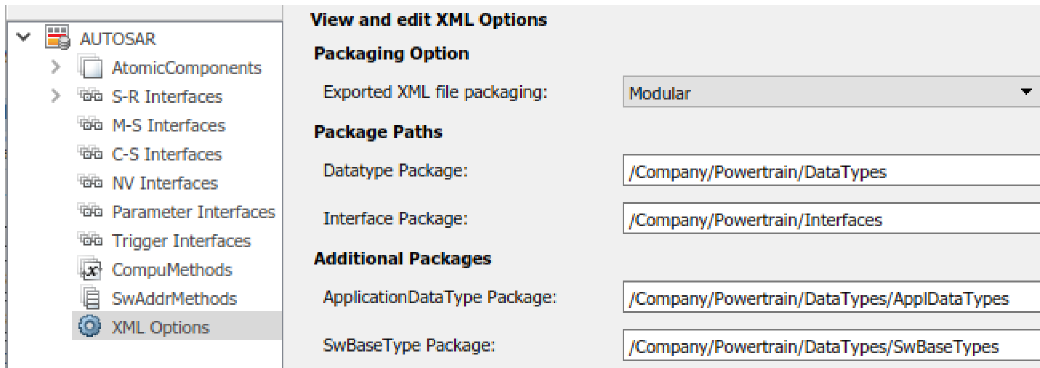
Configure AUTOSAR XML Options

To configure AUTOSAR XML options for `arxml` file export, open a model for which a mapped AUTOSAR software component has been created and open AUTOSAR Dictionary. Select **XML Options**.

The XML options view in 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 4-63
- "AUTOSAR Package Paths" on page 4-64
- "Additional XML Options" on page 4-65

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 3-46.) 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...
*modelname*_component.arxml	Software components, including calibration components. This is the main arxml file exported for the Simulink model. In addition to AUTOSAR software components, the file includes elements for which AUTOSAR packages (AR-PACKAGEs) are not configured, and AR-PACKAGEs that do not align with the package paths in the other exported arxml files. For more information on AR-PACKAGEs and their location in modular exported arxml files, see "AR-PACKAGE Location in Exported ARXML Files" on page 4-146.
*modelname*_datatype.arxml	Data types and related elements.
modelname-_implementation.arxml	Software component implementation.
*modelname*_interface.arxml	Communication interfaces, including S-R, C-S, M-S, NV, parameter, and trigger interfaces.
*modelname*_behavior.arxml	Software component internal behavior (generated only for schema 3.x or earlier).

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 5-16.

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 4-138.

Inspect and modify the AUTOSAR package paths grouped under the headings **Package Paths** and **Additional Packages**.

Package Paths

Datatype Package:	/Company/Powertrain/DataTypes
Interface Package:	/Company/Powertrain/Interfaces

Additional Packages

ApplicationDataType Package:	/Company/Powertrain/DataTypes/ApplDataTypes
SwBaseType Package:	/Company/Powertrain/DataTypes/SwBaseTypes
DataTypeMappingSet Package:	
ConstantSpecification Package:	/Company/Powertrain/DataTypes/Ground
Physical DataConstraints Package:	y/Powertrain/DataTypes/ApplDataTypes/DataConstrs
SystemConstant Package:	/Company/Powertrain/DataTypes/SystemConstants
SwAddressMethod Package:	/Company/Powertrain/DataTypes/SwAddrMethods
ModeDeclarationGroup Package:	
CompuMethod Package:	/Company/Powertrain/DataTypes/CompuMethods
Unit Package:	/Company/Powertrain/DataTypes/Units
SwRecordLayout Package:	
Internal DataConstraints Package:	/Company/Powertrain/DataTypes/DataConstrs

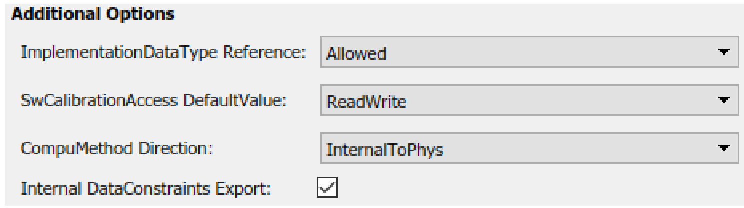
Alternatively, you can programmatically configure an AUTOSAR package path by calling the AUTOSAR `set` function. Specify a package property name and a package path. For example:

```
arProps = autosar.api.getAUTOSARProperties(hModel);
set(arProps,'XmlOptions','ApplicationDataTypePackage','/Company/Powertrain/DataTypes/ApplDataTypes');
```

Fore more information about AUTOSAR package property names and defaults, see “Configure AUTOSAR Packages and Paths” on page 4-140.

Additional XML Options

In the XML options view, under the heading **Additional Options**, you can configure aspects of exported AUTOSAR XML content.



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 4-314.
- 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 measurement and calibration tools.

  For more information, see “Configure SwCalibrationAccess” on page 4-264.
- 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 4-323.
- 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 4-333.

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 `ImplementationTypeReference`. 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 4-68
- “Configure and Map AUTOSAR Component Programmatically” on page 4-347
- “Configure AUTOSAR Code Generation” on page 5-12

More About

- “AUTOSAR Component Configuration” on page 4-3

Map AUTOSAR Elements for Code Generation

In Simulink, you can use Code Mappings editor and 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 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 **Inports**, **Outports**, and **Entry-Point Functions**. 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 4-68
"Map Inports and Outports to AUTOSAR Sender-Receiver Ports" on page 4-70
"Map Entry-Point Functions to AUTOSAR Runnables" on page 4-72
"Map Data Transfers to AUTOSAR Inter-Runnable Variables" on page 4-74
"Map Function Callers to AUTOSAR Client-Server Ports and Operations" on page 4-74
"Map Model Workspace Parameters to AUTOSAR Component Internal Parameters" on page 4-75
"Map Block Signals and States to AUTOSAR Variables" on page 4-79
"Map Data Stores to AUTOSAR Variables" on page 4-81
"Specify C Type Qualifiers for AUTOSAR Static and Constant Memory" on page 4-84

Simulink to AUTOSAR Mapping Workflow

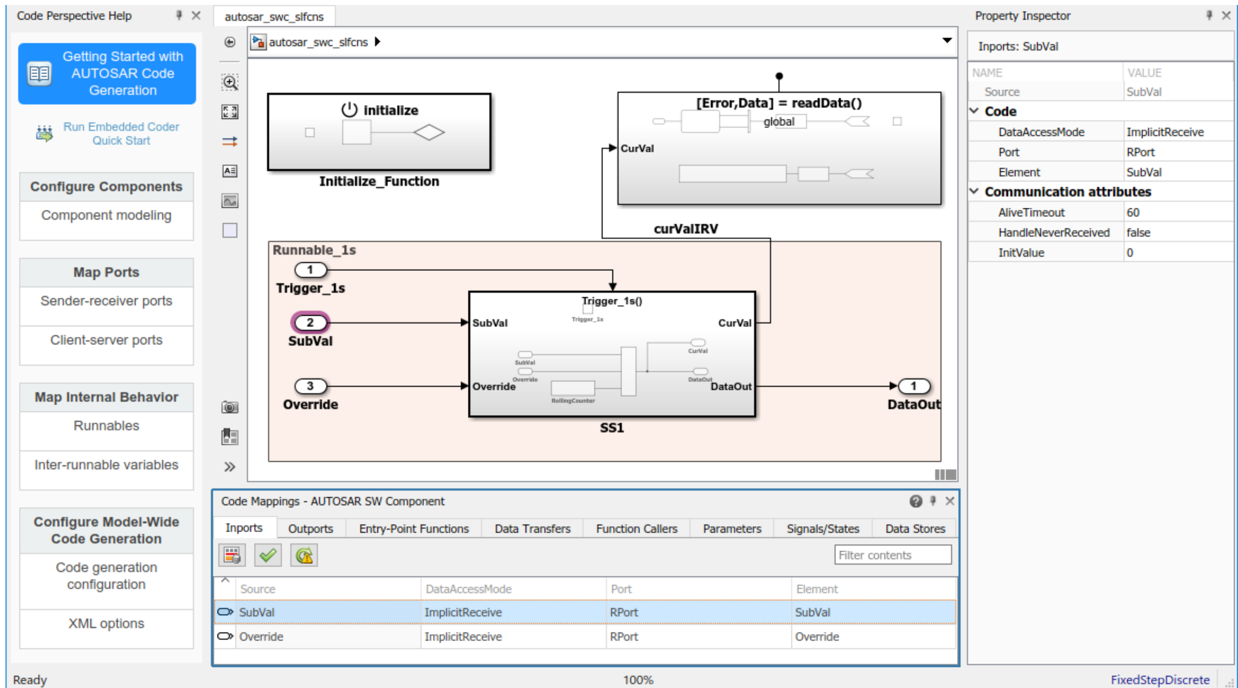
To map Simulink model elements to AUTOSAR software component elements:

1. Open a model for which the AUTOSAR system target file (`autosar.tlc`) has been selected.
2. Create or open a mapped view of the AUTOSAR model. In the model window, do one of the following:

   - Select **Code > C/C++ Code > Configure Model in Code Perspective**

- Click the perspective control in the lower-right corner and select **Code**.

The model opens in the AUTOSAR code perspective. This perspective displays a help panel, a Property Inspector dialog box, and, directly under the model, Code Mappings editor.



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 Code Mappings editor tabs to perform these actions:

- 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 entry-point function to an AUTOSAR runnable.
- 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.

- Map a Simulink lookup table to an AUTOSAR parameter.
- Map a Simulink model workspace parameter to an AUTOSAR component internal parameter.
- Map a Simulink block signal or state to an AUTOSAR variable.
- Map a Simulink data store to an AUTOSAR variable.

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.

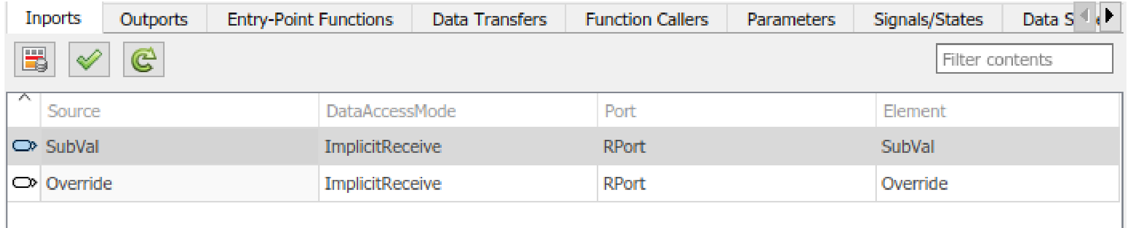
Map Inports and Outports to AUTOSAR Sender-Receiver Ports

The **Inports** and **Outports** tabs of Code Mappings editor support modeling AUTOSAR sender-receiver (S-R) communication in Simulink. After using AUTOSAR Dictionary to create AUTOSAR S-R ports, S-R interfaces, and S-R data elements in your model, open 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.

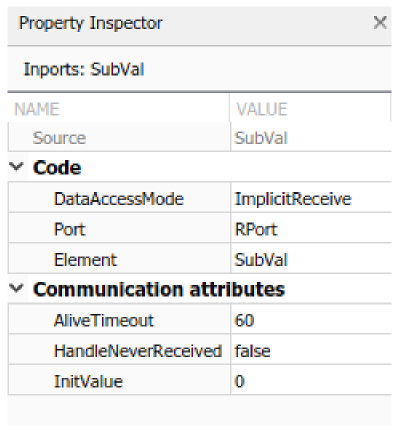
For more information, see “Configure AUTOSAR Sender-Receiver Communication” on page 4-153 and “Configure AUTOSAR Queued Sender-Receiver Communication” on page 4-172.

The **Inports** tab of 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`.

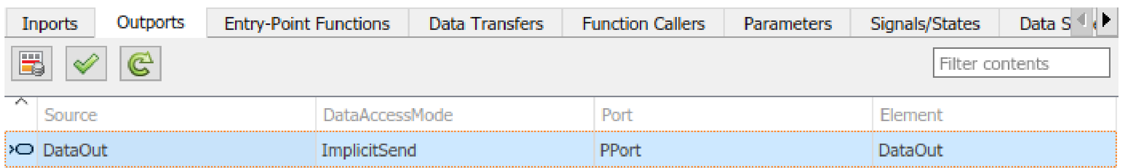


When you select an inport that maps to an AUTOSAR nonqueued receiver port, the Property Inspector displays additional port communication specification (ComSpec) attributes. For AUTOSAR receiver ports, you can modify ComSpec attributes `AliveTimeout`, `HandleNeverReceived`, and `InitValue`. For more information, see “Configure AUTOSAR Sender-Receiver Port ComSpecs” on page 4-168.

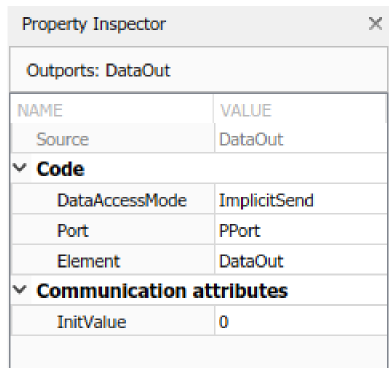


The **Outports** tab of Code Mappings editor maps each Simulink root outport to an AUTOSAR sender port and an S-R interface data element. In the **Outports** tab, you can:

- 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`.



When you select an outport that maps to an AUTOSAR nonqueued sender port, the Property Inspector displays additional port communication specification (ComSpec) attributes. For AUTOSAR sender ports, you can modify the ComSpec attribute `InitValue`. For more information, see “Configure AUTOSAR Sender-Receiver Port ComSpecs” on page 4-168.



Map Entry-Point Functions to AUTOSAR Runnables

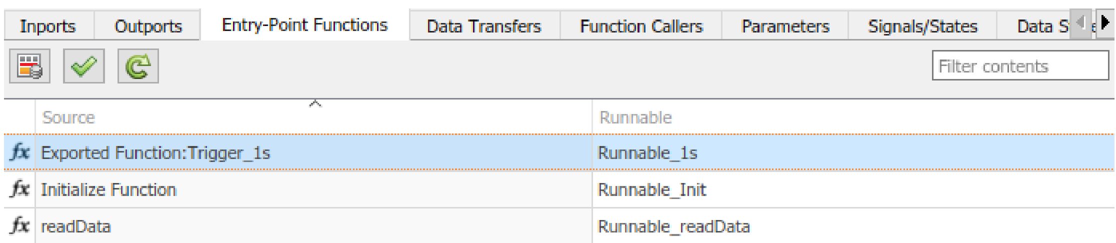
The **Entry-Point Functions** tab of Code Mappings editor supports modeling AUTOSAR runnable entities (runnables) in Simulink. After using AUTOSAR Dictionary to create AUTOSAR runnables and AUTOSAR events, which implement aspects of internal behavior in an AUTOSAR component, open Code Mappings editor. Use the **Entry-Point Functions** tab to map Simulink entry-point functions to AUTOSAR runnables.

For more information, see “Configure AUTOSAR Runnables and Events” on page 4-277.

The **Entry-Point Functions** tab of 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.

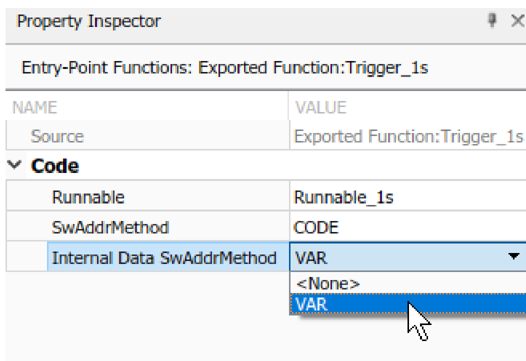
In the **Entry-Point Functions** tab, you can:

- 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.



- 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 4-270.

  To specify SwAddrMethods for a runnable, select the corresponding entry-point function. Property Inspector displays the code attributes **SwAddrMethod** and **Internal Data SwAddrMethod** for the selected function. In the Property Inspector, 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 4-60.

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`.

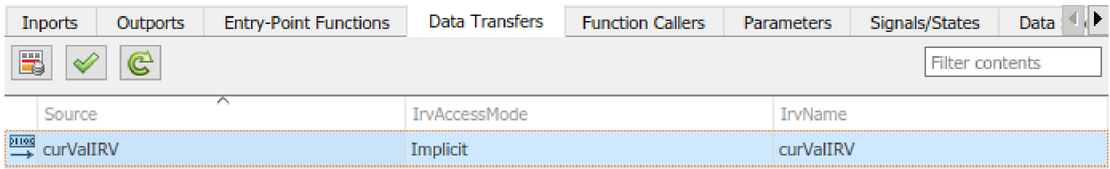
Map Data Transfers to AUTOSAR Inter-Runnable Variables

The **Data Transfers** tab of Code Mappings editor supports modeling AUTOSAR inter-runnable variables (IRVs) in Simulink. After using AUTOSAR Dictionary to create AUTOSAR IRVs, which connect runnables and implement aspects of internal behavior in an AUTOSAR component, open 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 2-38. 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 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.



Map Function Callers to AUTOSAR Client-Server Ports and Operations

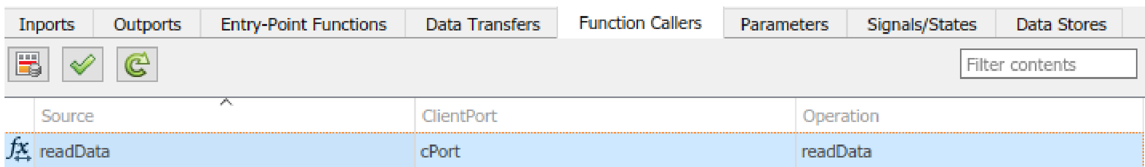
The **Function Callers** tab of Code Mappings editor supports modeling the client side of AUTOSAR client-server (C-S) communication in Simulink. After using AUTOSAR Dictionary to create AUTOSAR client ports, C-S interfaces, and C-S operations in your

model, open 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 4-197.

The **Function Callers** tab of 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.



Map Model Workspace Parameters to AUTOSAR Component Internal Parameters

The **Parameters** tab of Code Mappings editor supports mapping 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 internal calibration parameters, you can model AUTOSAR parameters for integrated and distributed lookups. For more information, see “Configure Lookup Tables for AUTOSAR Measurement and Calibration” on page 4-250.

After creating model workspace parameters, for example, using Model Explorer, open Code Mappings editor. Use the **Parameters** tab to map Simulink model workspace

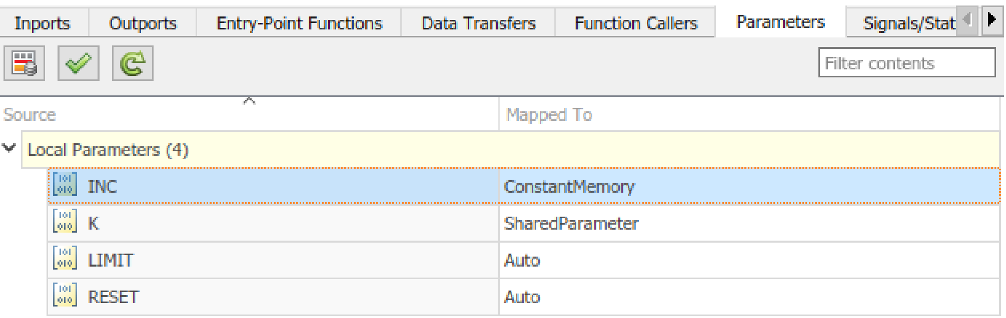
parameters to AUTOSAR component internal parameters, such as constant memory, shared parameters, or per-instance parameters.

For more information, see “Configure AUTOSAR Constant Memory” on page 4-305 and “Configure AUTOSAR Shared or Per-Instance Parameters” on page 4-308.

The **Parameters** tab of Code Mappings editor maps each Simulink model workspace parameter to an AUTOSAR parameter. In the **Parameters** tab:

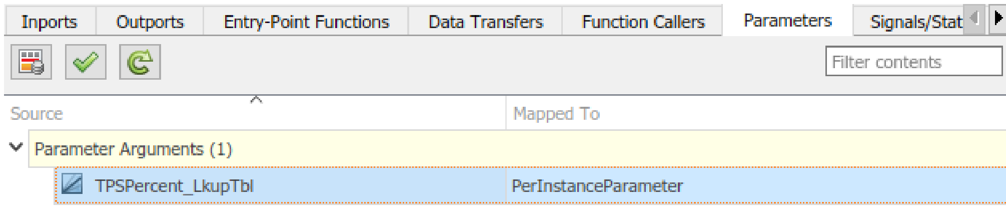
- If a Simulink model workspace parameter is not configured as a model argument (that is, not unique to each instance of a multi-instance model), you can map the parameter by selecting it and then selecting a menu value for an AUTOSAR parameter type. For this workflow, the valid parameter types are `ConstantMemory`, `SharedParameter`, or `Auto`. To accept software mapping defaults, specify `Auto`.

  For example, here is the **Parameters** tab for example model `autosar_swc_counter`.

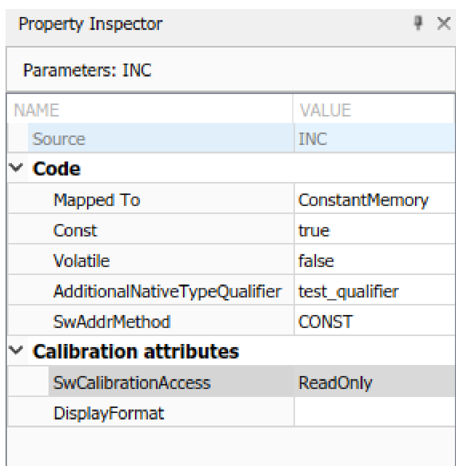
  

- If a Simulink model workspace parameter is configured as a model argument (that is, unique to each instance of a multi-instance model), map the parameter by selecting it and then selecting a menu value for an AUTOSAR parameter type. For this workflow, the valid parameter types are `PerInstanceParameter` or `Auto`. To accept software mapping defaults, specify `Auto`.

  For example, here is the **Parameters** tab for example model `autosar_swc_throttle_sensor`. Example model `autosar_composition` contains two instances of `autosar_swc_throttle_sensor`.



- If you select a parameter type other than `Auto`, use Property Inspector to view or modify other code and calibration attributes for the parameter.



Attribute	Purpose
Const (`ConstantMemory` only)	Specify whether to include C type qualifier `const` in generated code for the AUTOSAR parameter. For more information, see “Specify C Type Qualifiers for AUTOSAR Static and Constant Memory” on page 4-84.

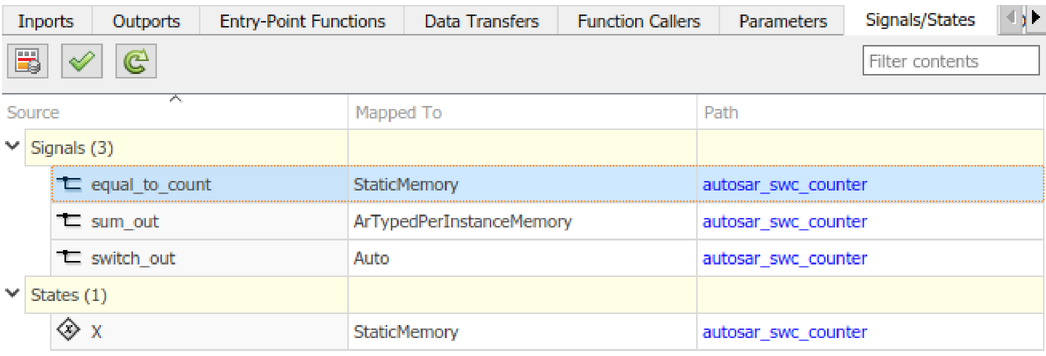
Attribute	Purpose
Volatile (`ConstantMemory` only)	Specify whether to include C type qualifier `volatile` in generated code for the AUTOSAR parameter. For more information, see “Specify C Type Qualifiers for AUTOSAR Static and Constant Memory” on page 4-84.
AdditionalNativeTypeQualifier (`ConstantMemory` only)	Specify an AUTOSAR additional native type qualifier to include in generated code for the AUTOSAR parameter. For more information, see “Specify C Type Qualifiers for AUTOSAR Static and Constant Memory” on page 4-84.
SwAddrMethod	Select a `SwAddrMethod` name among those listed as valid for the AUTOSAR parameter. Code generation uses the `SwAddrMethod` name to group AUTOSAR parameters in a memory section for access by measurement and calibration tools. For more information, see “Configure SwAddrMethod” on page 4-270.
SwCalibrationAccess	Specify how measurement and calibration tools can access the AUTOSAR parameter. Valid access values include `ReadOnly`, `ReadWrite`, and `NotAccessible`. For more information, see “Configure SwCalibrationAccess” on page 4-264.
DisplayFormat	Specify display format for the AUTOSAR parameter. AUTOSAR display format specifications control the width and precision display for measurement and calibration data. For more information, see “Configure DisplayFormat” on page 4-266.

Map Block Signals and States to AUTOSAR Variables

The **Signals/States** tab of Code Mappings editor support mapping Simulink block signals and states to AUTOSAR variables for AUTOSAR run-time calibration. After creating named or test-pointed Simulink block signals or Simulink state owner blocks in your model, open Code Mappings editor. Use the **Signals/States** tab to map the block signals and states to AUTOSAR variables, such as AUTOSAR-typed per-instance memory or AUTOSAR static memory.

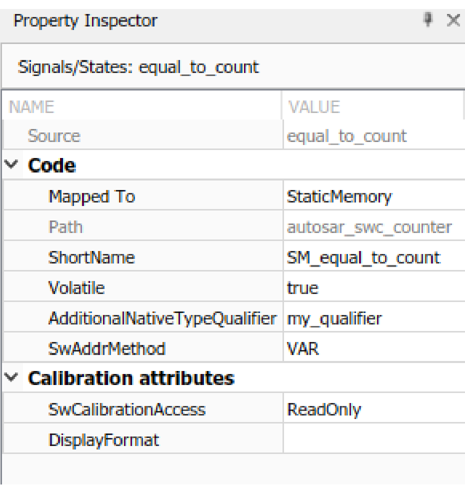
For more information, see “Configure AUTOSAR Per-Instance Memory” on page 4-295 and “Configure AUTOSAR Static Memory” on page 4-300.

The **Signals/States** tab, **Signals** node, of Code Mappings editor maps each named or test-pointed Simulink block signal to an AUTOSAR variable. In the **Signals** node, 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`.



The **Signals/States** tab, **States** node, of Code Mappings editor maps each configurable Simulink block state to an AUTOSAR variable. In the **States** node, 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`.

In either tab, if you map a signal or state to a variable type other than `Auto`, you can use Property Inspector to view or modify other code and calibration attributes for the variable.



Attribute	Purpose
ShortName	Specify short name for the AUTOSAR variable. If unspecified, `arxml` export automatically generates a short name. • 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)	Specify whether to include C type qualifier `volatile` in generated code for the AUTOSAR variable. For more information, see "Specify C Type Qualifiers for AUTOSAR Static and Constant Memory" on page 4-84.

Attribute	Purpose
AdditionalNativeTypeQualifier (`StaticMemory` only)	Specify an AUTOSAR additional native type qualifier to include in generated code for the AUTOSAR variable. For more information, see “Specify C Type Qualifiers for AUTOSAR Static and Constant Memory” on page 4-84.
SwAddrMethod	Select a `SwAddrMethod` name among those listed as valid for the AUTOSAR variable. Code generation uses the `SwAddrMethod` name to group AUTOSAR variables in a memory section for access by measurement and calibration tools. For more information, see “Configure SwAddrMethod” on page 4-270.
SwCalibrationAccess	Specify how measurement and calibration tools can access the AUTOSAR variable. Valid access values include `ReadOnly`, `ReadWrite`, and `NotAccessible`. For more information, see “Configure SwCalibrationAccess” on page 4-264.
DisplayFormat	Specify display format for the AUTOSAR variable. AUTOSAR display format specifications control the width and precision display for measurement and calibration data. For more information, see “Configure DisplayFormat” on page 4-266.

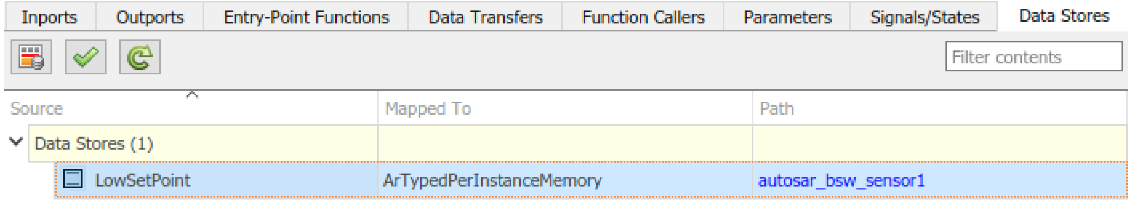
Map Data Stores to AUTOSAR Variables

The **Data Stores** tab of Code Mappings editor supports mapping Simulink data store memory blocks to AUTOSAR variables for AUTOSAR run-time calibration. After creating data store memory blocks in your model, open Code Mappings editor. Use the **Data Stores** tab to map the data stores to AUTOSAR variables, such as AUTOSAR-typed per-instance memory or AUTOSAR static memory.

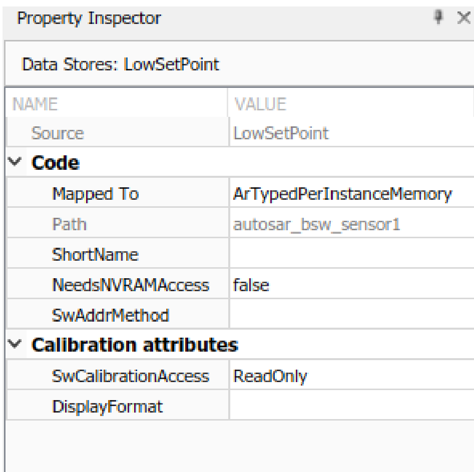
For more information, see “Configure AUTOSAR Per-Instance Memory” on page 4-295 and “Configure AUTOSAR Static Memory” on page 4-300.

The **Data Stores** tab of Code Mappings editor maps each data store to an AUTOSAR variable. In the **Data Stores** tab, 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`.



- If you select a variable type other than `Auto`, use Property Inspector to view or modify other code and calibration attributes for the variable.



Attribute	Purpose
ShortName	Specify short name for the AUTOSAR variable. If unspecified, `arxml` export automatically generates a short name.

Attribute	Purpose
Volatile (`StaticMemory` only)	Specify whether to include C type qualifier `volatile` in generated code for the AUTOSAR variable. For more information, see "Specify C Type Qualifiers for AUTOSAR Static and Constant Memory" on page 4-84.
AdditionalNativeTypeQualifier (`StaticMemory` only)	Specify an AUTOSAR additional native type qualifier to include in generated code for the AUTOSAR variable. For more information, see "Specify C Type Qualifiers for AUTOSAR Static and Constant Memory" on page 4-84.
NeedsNVRAMAccess (`ArTypedPerInstanceMemory` only)	Specify whether the AUTOSAR variable needs access to nonvolatile RAM on a processor. Select `true` to configure the per-instance memory to be a mirror block for a specific NVRAM block.
SwAddrMethod	Select a `SwAddrMethod` name among those listed as valid for the AUTOSAR variable. Code generation uses the `SwAddrMethod` name to group AUTOSAR variables in a memory section for access by measurement and calibration tools. For more information, see "Configure SwAddrMethod" on page 4-270.
SwCalibrationAccess	Specify how measurement and calibration tools can access the AUTOSAR variable. Valid access values include `ReadOnly`, `ReadWrite`, and `NotAccessible`. For more information, see "Configure SwCalibrationAccess" on page 4-264.

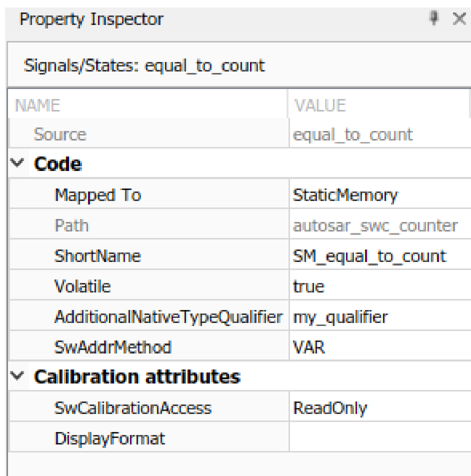
Attribute	Purpose
DisplayFormat	Specify display format for the AUTOSAR variable. AUTOSAR display format specifications control the width and precision display for measurement and calibration data. For more information, see “Configure DisplayFormat” on page 4-266.

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 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 Code Mappings editor, **Signals/States** tab, suppose that you map a signal to `StaticMemory`. Select the signal to display code attributes in Property Inspector.



If you set the `Volatile` attribute to `true` 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 4-79, “Map Data Stores to AUTOSAR Variables” on page 4-81, and “Map Model Workspace Parameters to AUTOSAR Component Internal Parameters” on page 4-75.

See Also

Related Examples

- “Configure AUTOSAR Elements and Properties” on page 4-8
- “Configure and Map AUTOSAR Component Programmatically” on page 4-347

More About

- "AUTOSAR Component Configuration" on page 4-3

Configure AUTOSAR Adaptive Elements and Properties

In Simulink, you can use AUTOSAR Dictionary and 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.

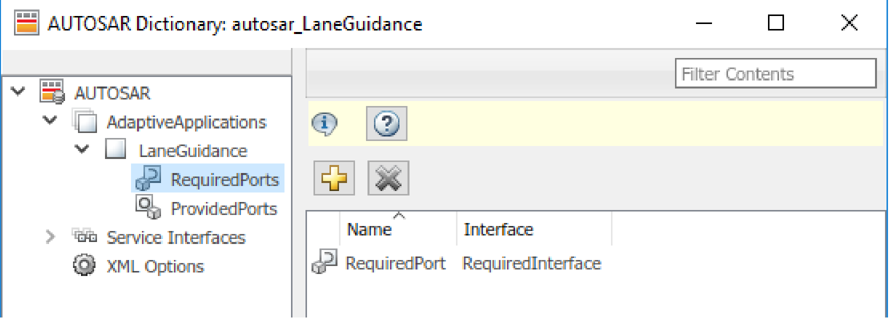
Use AUTOSAR Dictionary to configure AUTOSAR elements from an AUTOSAR perspective. Using a tree format, AUTOSAR Dictionary displays a mapped AUTOSAR adaptive component and its elements, communication interfaces, and XML options. Use the tree to select AUTOSAR elements and configure their properties. The 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 4-87
“Configure AUTOSAR Adaptive Software Components” on page 4-88
“Configure AUTOSAR Required and Provided Ports” on page 4-91
“Configure AUTOSAR Service Communication Interfaces” on page 4-94
“Configure AUTOSAR Adaptive XML Options” on page 4-98

AUTOSAR Elements Configuration Workflow

To configure AUTOSAR component elements for the Adaptive Platform in Simulink:

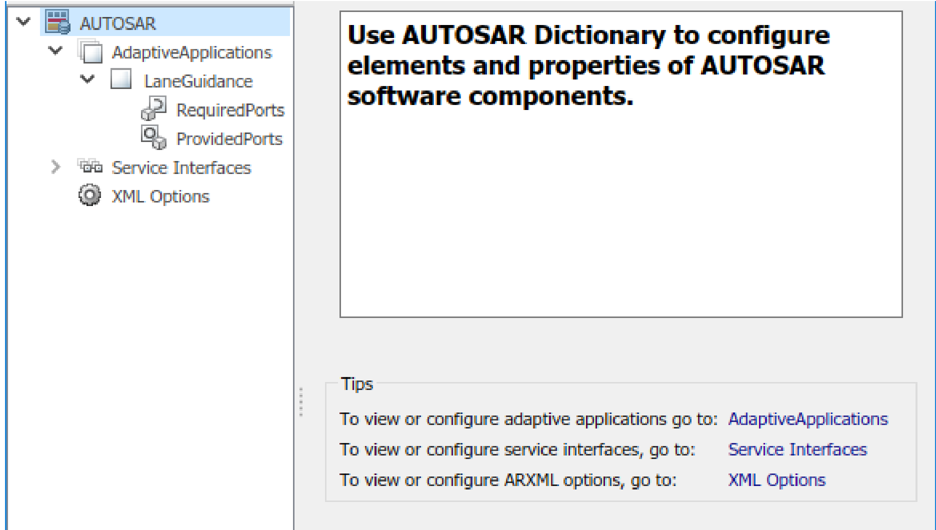
1. Open a model for which AUTOSAR system target file `autosar_adaptive.tlc` has been selected.
2. If the model does not yet have a mapped AUTOSAR software component, create one. In the model window, select **Code > C/C++ Code > Configure Model in Code Perspective** or click the perspective control in the lower-right corner and select **Code**. The AUTOSAR Component Quick Start opens. Work through the quick-start procedure. When you click **Finish**, the model opens in the AUTOSAR code perspective. For more information, see “Create AUTOSAR Software Component in Simulink” on page 3-2.
3. Open AUTOSAR Dictionary. Either click the **AUTOSAR Dictionary** button in Code Mappings editor or select **Code > C/C++ Code > Configure AUTOSAR Dictionary**.



4 Navigate the AUTOSAR Dictionary tree to configure AUTOSAR elements and properties. 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 Code Mappings editor. Use 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 Adaptive Software Components

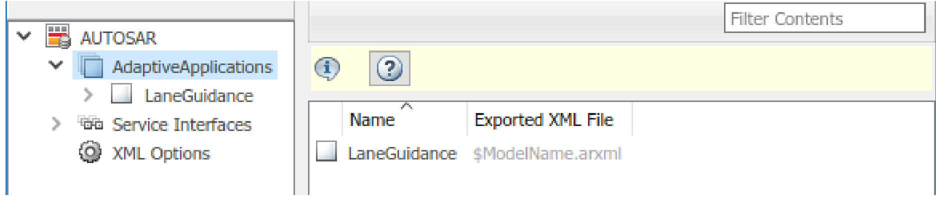
AUTOSAR adaptive software components contain AUTOSAR elements defined in the AUTOSAR standard, such as ports, service interfaces, and events. In AUTOSAR Dictionary, component elements appear in a tree format under the component that owns them. To access component elements and their properties, you expand the component name.



To configure AUTOSAR atomic software component elements and properties:

1. Open a model for which a mapped AUTOSAR adaptive software component has been created. For more information, see “Component Creation”.
2. Open AUTOSAR Dictionary. Either click the **AUTOSAR Dictionary** button in Code Mappings editor or select **Code > C/C++ Code > Configure AUTOSAR Dictionary**.
3. In the left-hand pane of AUTOSAR Dictionary, under **AUTOSAR**, select **AdaptiveApplications**.

   The adaptive applilcations view in AUTOSAR Dictionary displays adaptive software components. You can rename an AUTOSAR adaptive component by clicking its name and then editing the name text.



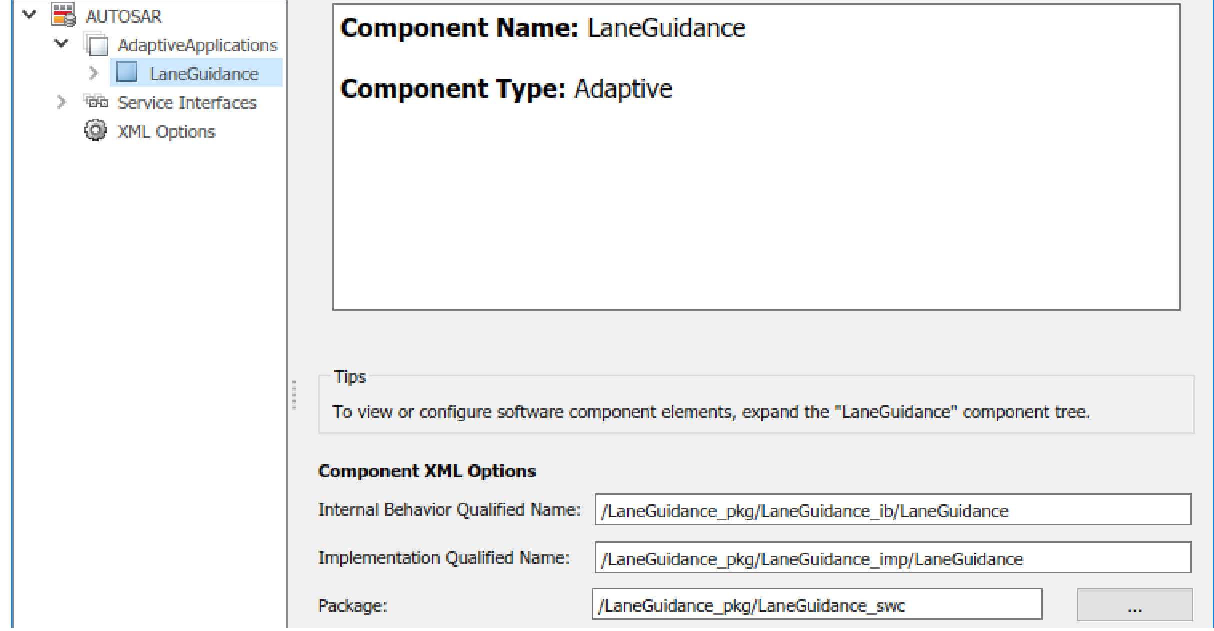
4 In the left-hand pane of AUTOSAR Dictionary, expand **AdaptiveApplications** and select an AUTOSAR adaptive component.

   The component view in 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 4-150.

   For more information about component XML options, see "Configure AUTOSAR Packages" on page 4-138.



Configure AUTOSAR Required and Provided Ports

An AUTOSAR adaptive software component contains service communication ports defined in the AUTOSAR standard, including required and provided ports. In 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.

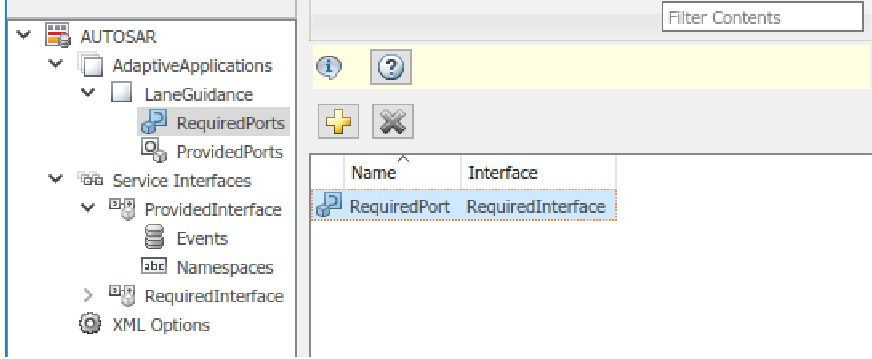
The AUTOSAR Dictionary views of required and provided ports support modeling AUTOSAR service interface communication in Simulink. You use AUTOSAR Dictionary to configure AUTOSAR required and provided ports, service interfaces, service interface events, and C++ namepaces in your model. For more information, see “Configure AUTOSAR Adaptive Service Communication” on page 4-245.

To configure AUTOSAR required and provided port elements and properties, open a model for which a mapped AUTOSAR adaptive software component has been created and open AUTOSAR Dictionary.

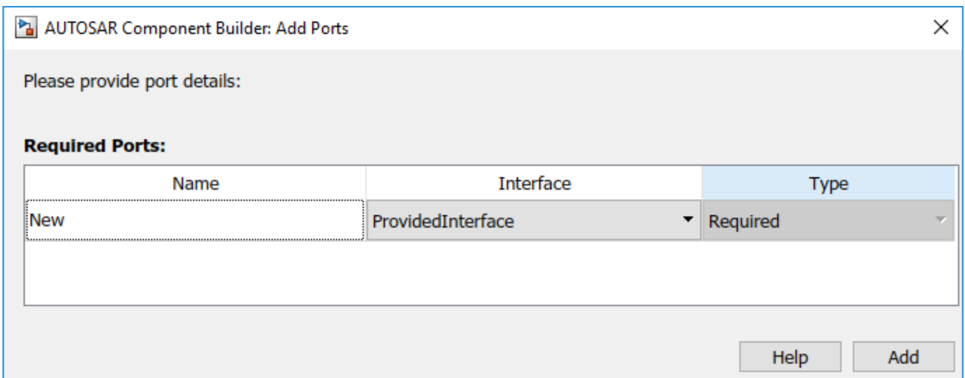
1 In the left-hand pane of AUTOSAR Dictionary, expand the component name and select **RequiredPorts**.

   The required ports view in 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 an AUTOSAR required port by clicking its name and then editing the name text.
   - Click the **Add** button to open an Add Ports dialog box to add a port.
   - Select a port and then click the **Delete** button to remove it.



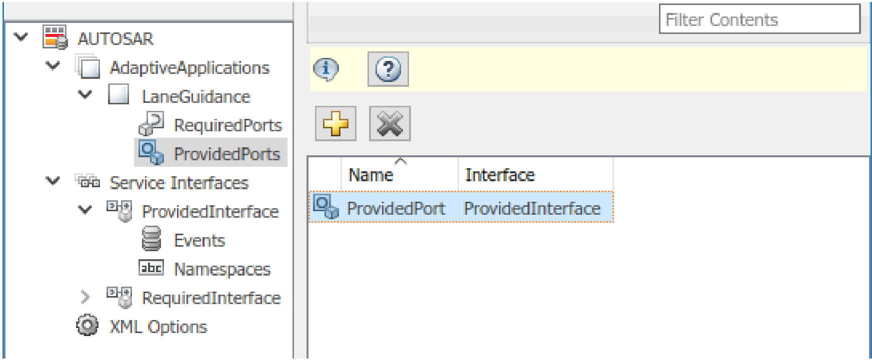
   The Add Ports dialog box lets you add a required port and associate it with an existing service interface. To add the port and return to the required ports view, click **Add**.



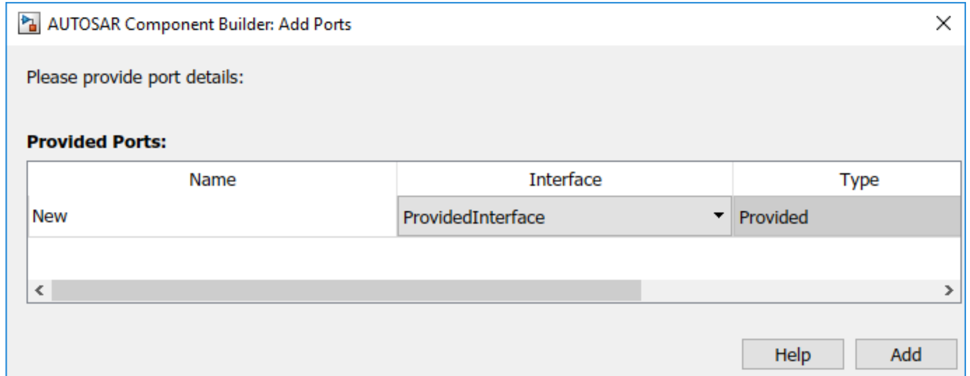
2 In the left-hand pane of AUTOSAR Dictionary, select **ProvidedPorts**.

The provided ports view in 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 an AUTOSAR provided port by clicking its name and then editing the name text.
- Click the **Add** button to open an Add Ports dialog box to add a port.
- Select a port and then click the **Delete** button to remove it.



The Add Ports dialog box lets you add a provided port and associate it with an existing service interface. Click **Add** to add the port and return to the provided ports view.



Configure AUTOSAR Service Communication Interfaces

An AUTOSAR software component uses communication interfaces defined in the AUTOSAR standard, including adaptive service interfaces. In 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.

The **Service Interfaces** view in AUTOSAR Dictionary supports modeling AUTOSAR adaptive service communication in Simulink. You use AUTOSAR Dictionary to configure AUTOSAR required and provided ports, service interfaces, service interface events, and C ++ namepaces in your model. For more information, see “Configure AUTOSAR Adaptive Service Communication” on page 4-245.

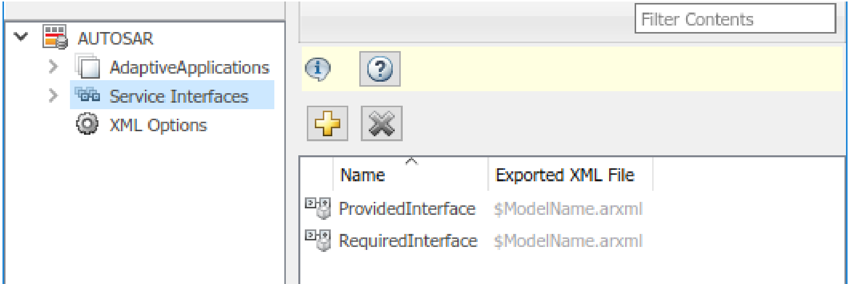
To configure AUTOSAR service interface elements and properties, open a model for which a mapped AUTOSAR adaptive software component has been created and open AUTOSAR Dictionary.

1 In the left-hand pane of AUTOSAR Dictionary, select **Service Interfaces**.

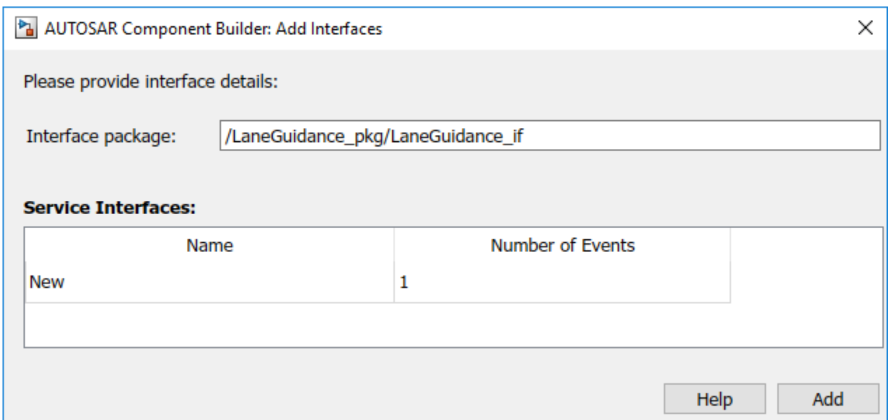
   The service interfaces view in AUTOSAR Dictionary lists AUTOSAR service interfaces and their properties. You can:

   - Select a service interface and rename it by editing its name text.

- Click the **Add** button to open an Add Interfaces dialog box to add one or more service interfaces.
- Select a service interface and then click the **Delete** button to remove it.



The Add Interfaces dialog box lets you specify the name of the new interface, the number of associated events it contains, and the path of the Interface package. Click **Add** to add the interface and return to the service interfaces view.

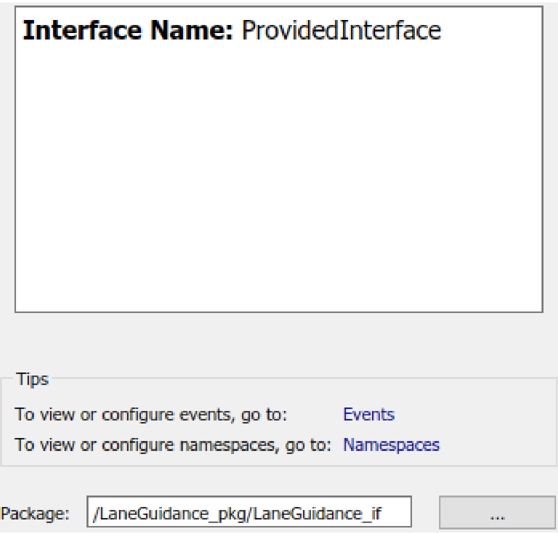


2 In the left-hand pane of AUTOSAR Dictionary, expand **Service Interfaces** and select a service interface from the list.

The service interface view in 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.
- 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 4-150.



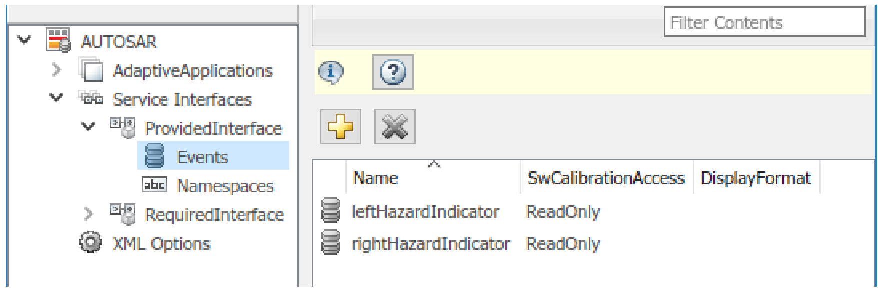
3 In the left-hand pane of AUTOSAR Dictionary, expand the selected interface and select **Events**.

The events view in 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 measurement and calibration 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 measurement and calibration 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, 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 4-266.

- Click the **Add** button to add an event.
- Select an event and then click the **Delete** button to remove it.

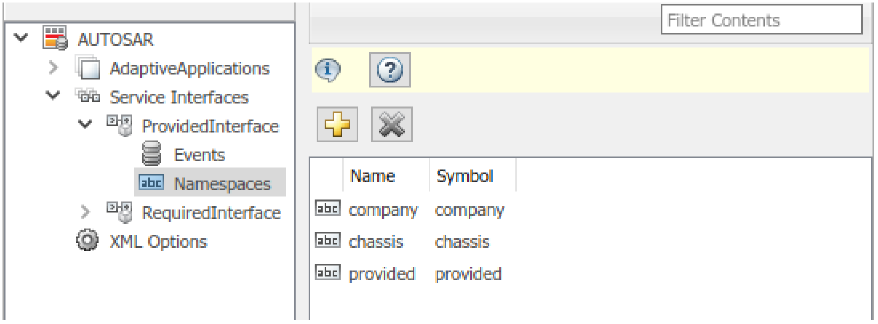


4 In the left-hand pane of AUTOSAR Dictionary, below **Events**, select **Namespaces**.

The namespaces view in AUTOSAR Dictionary 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, you can:

- Select a namespace element and edit the name value.
- Click the **Add** button to add a namespace element to the namespace specification.
- Select a namespace element and then click the **Delete** button to remove it.

For example, the namespaces view below defines namespace `company::chassis::provided` for service interface `ProvidedInterface`.

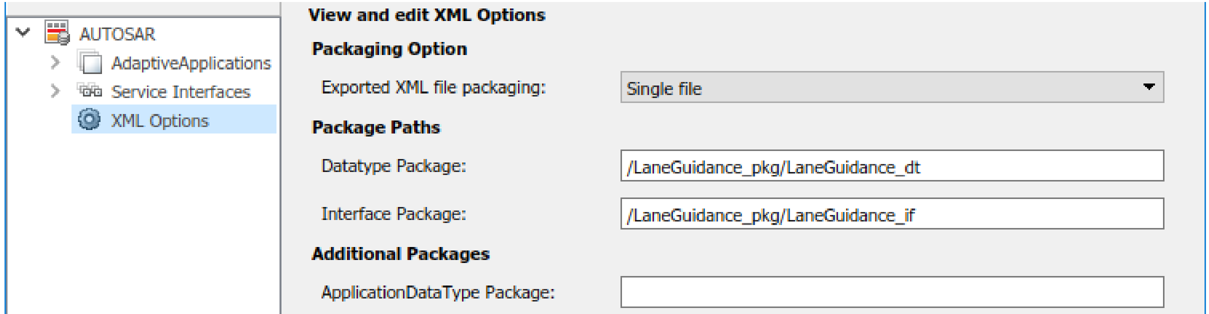


Configure AUTOSAR Adaptive XML Options

To configure AUTOSAR adaptive XML options for `arxml` file export, open a model for which a mapped AUTOSAR adaptive software component has been created and open AUTOSAR Dictionary. Select **XML Options**.

The XML options view in 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 4-99
- “AUTOSAR Package Paths” on page 4-100
- “Additional XML Options” on page 4-100

Exported XML File Packaging

In the XML options view, you can specify the granularity of XML file packaging for AUTOSAR elements created in Simulink. 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...
`modelname``_component.arxml`	Software components. This is the main `arxml` file exported for the Simulink model. In addition to AUTOSAR software components, the file includes elements for which AUTOSAR packages (AR-PACKAGEs) are not configured, and AR-PACKAGEs that do not align with the package paths in the other exported `arxml` files. For more information on AR-PACKAGEs and their location in modular exported `arxml` files, see "AR-PACKAGE Location in Exported ARXML Files" on page 4-146.
`modelname``_datatype.arxml`	Data types and related elements.
`modelname``_implementation.arxml`	Software component implementation.
`modelname``_interface.arxml`	Communication interfaces, including service interfaces.
`modelname``_behavior.arxml`	Software component internal behavior (generated only for schema 3.x or earlier).

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 Adaptive C++ and XML Files" on page 5-18.

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 4-138.

Inspect and modify the AUTOSAR package paths grouped under the headings **Package Paths** and **Additional Packages**.

Package Paths

Datatype Package:	/LaneGuidance_pkg/LaneGuidance_dt
Interface Package:	/LaneGuidance_pkg/LaneGuidance_if

Additional Packages

ApplicationDataType Package:	
SwBaseType Package:	
ConstantSpecification Package:	/LaneGuidance_pkg/LaneGuidance_dt/Ground
Physical DataConstraints Package:	/LaneGuidance_pkg/LaneGuidance_dt/ApplDataTypes/DataConstrs
Unit Package:	/LaneGuidance_pkg/LaneGuidance_dt
SwRecordLayout Package:	
Internal DataConstraints Package:	

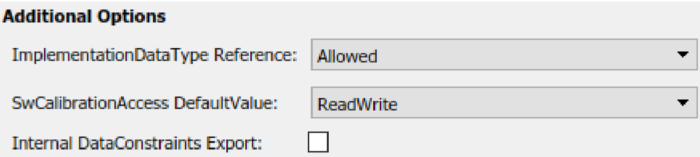
Alternatively, you can programmatically configure an AUTOSAR package path by calling the AUTOSAR `set` function. Specify a package property name and a package path. For example:

```
arProps = autosar.api.getAUTOSARProperties(hModel);
set(arProps,'XmlOptions','ApplicationDataTypePackage','/Company/Powertrain/DataTypes/ApplDataTypes');
```

Fore more information about AUTOSAR package property names and defaults, see "Configure AUTOSAR Packages and Paths" on page 4-140.

Additional XML Options

In the XML options view, under the heading **Additional Options**, you can configure aspects of exported AUTOSAR XML content.



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 4-314.
- 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 measurement and calibration tools.

  For more information, see “Configure SwCalibrationAccess” on page 4-264.
- 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 4-333.

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`, and `ImplementationTypeReference`. For example:

```
arProps = autosar.api.getAUTOSARProperties(hModel);
set(arProps,'XmlOptions','ImplementationTypeReference','NotAllowed');
set(arProps,'XmlOptions','SwCalibrationAccessDefault','ReadOnly');
set(arProps,'XmlOptions','InternalDataConstraintExport',true);
```

See Also

Related Examples

- "Map AUTOSAR Adaptive Elements for Code Generation" on page 4-103
- "Configure and Map AUTOSAR Component Programmatically" on page 4-347
- "Configure AUTOSAR Code Generation" on page 5-12

More About

- "AUTOSAR Component Configuration" on page 4-3

Map AUTOSAR Adaptive Elements for Code Generation

In Simulink, you can use Code Mappings editor and 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 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 4-103
"Map Inports and Outports to AUTOSAR Required and Provided Service Ports" on page 4-105

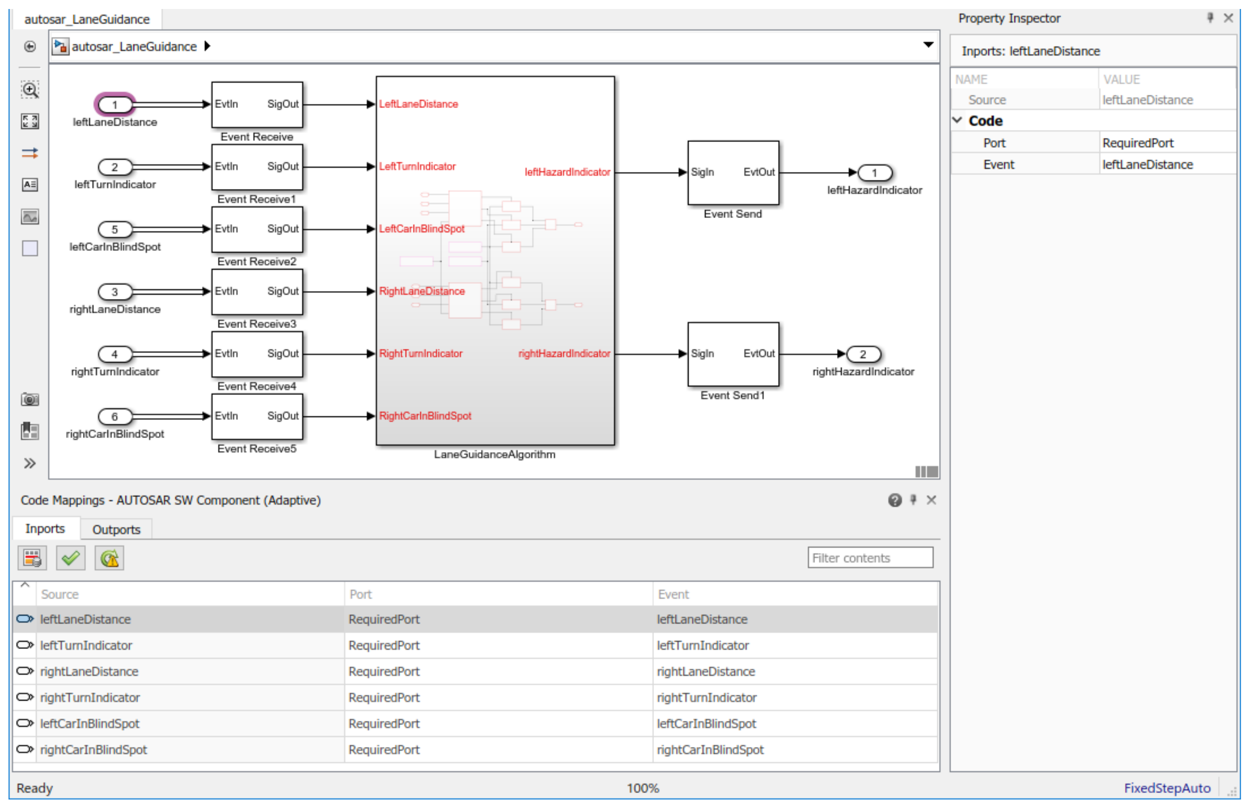
Simulink to AUTOSAR Mapping Workflow

To map Simulink model elements to AUTOSAR adaptive software component elements:

1. Open a model for which the AUTOSAR system target file (`autosar_adaptive.tlc`) has been selected.
2. Create or open a mapped view of the AUTOSAR model. In the model window, do one of the following:

   - Select **Code** > **C/C++ Code** > **Configure Model in Code Perspective**
   - Click the perspective control in the lower-right corner and select **Code**.

   The model opens in the AUTOSAR code perspective. This perspective displays a help panel, a Property Inspector dialog box, and, directly under the model, Code Mappings editor.



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 Code Mappings editor tabs to map each Simulink inport or outport to an AUTOSAR required or provided port and a service interface event.

   Use the **Filter contents** field (where available) to selectively display some elements, while omitting others, in the current view.

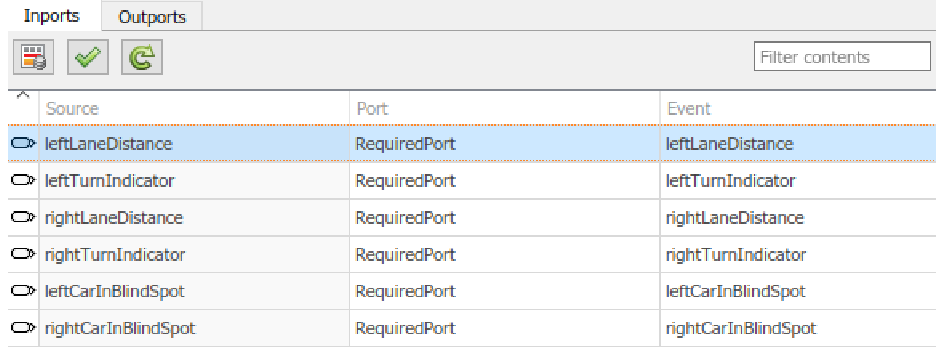
4. After mapping model elements, click the **Validate** button to validate the AUTOSAR component configuration. If errors are reported, address them and then retry validation.

Map Inports and Outports to AUTOSAR Required and Provided Service Ports

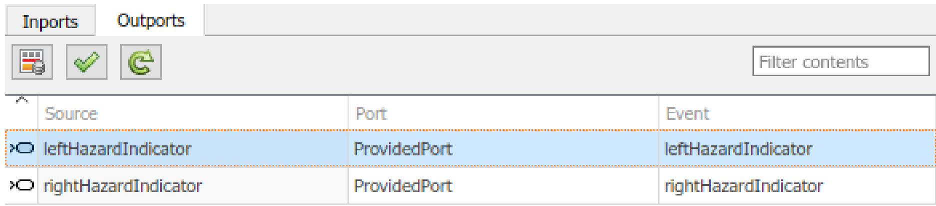
The **Inports** and **Outports** tabs of Code Mappings editor support modeling AUTOSAR service interface communication in Simulink. After using AUTOSAR Dictionary to create AUTOSAR required and provided service ports, service interfaces, and service interface events in your model, open Code Mappings editor. Use the **Inports** and **Outports** tabs to map Simulink root inports and outports to AUTOSAR required and provided service ports and AUTOSAR service interface events.

For more information, see "Configure AUTOSAR Adaptive Service Communication" on page 4-245.

The **Inports** tab of Code Mappings editor maps each Simulink root inport to an AUTOSAR required port and a service interface event. 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 event, among those listed for the AUTOSAR component.



The **Outports** tab of 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 those listed for the AUTOSAR component.



See Also

Related Examples

- “Configure AUTOSAR Adaptive Elements and Properties” on page 4-87
- “Configure and Map AUTOSAR Component Programmatically” on page 4-347

More About

- “AUTOSAR Component Configuration” on page 4-3

Incrementally Update AUTOSAR Mapping after Model Changes

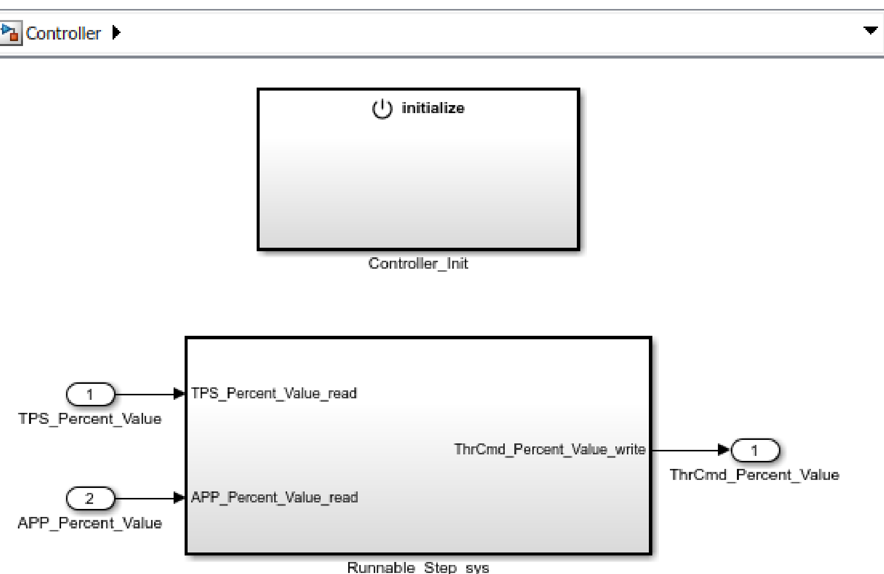
While developing an AUTOSAR software component model, you can use function `autosar.api.create` to incrementally configure and map Simulink elements as you add them to your model. When used with a mapped AUTOSAR model, `autosar.api.create` does not recreate or replace the current Simulink to AUTOSAR mapping, but instead 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.

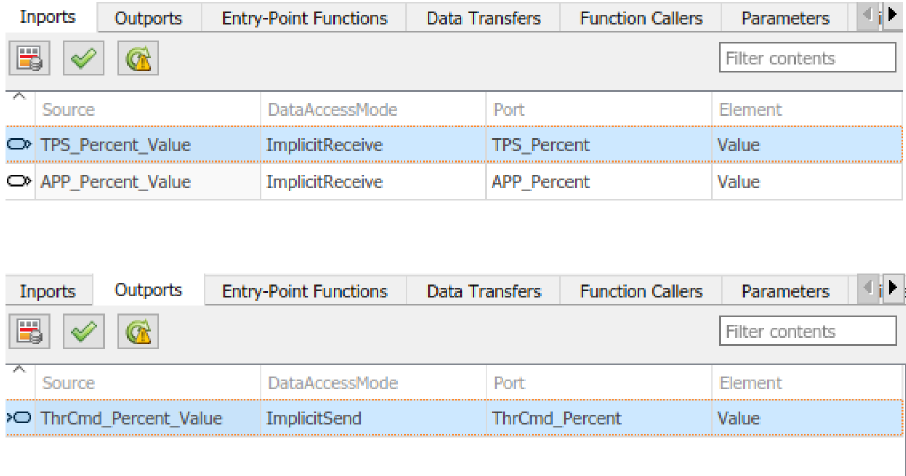
In this example, you add inports and outports to a mapped AUTOSAR software component model. Then you use `autosar.api.create` to create and map corresponding AUTOSAR elements with default naming and properties. After the incremental update, you can edit the default naming and properties as you require.

1 Open a mapped AUTOSAR software component model. To create a model named `Controller` for this example, use these commands.

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

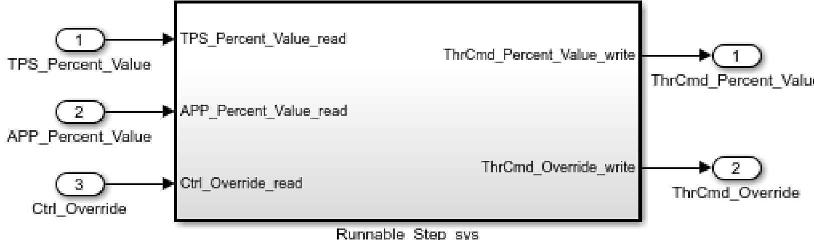


Here is the initial Simulink to AUTOSAR mapping of Simulink inports and outports in the model.



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.

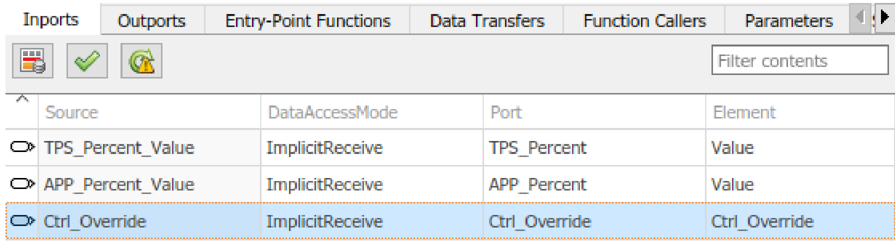


3 To configure and map the added inports and outports, call the `autosar.api.create` function. Use either of these forms.

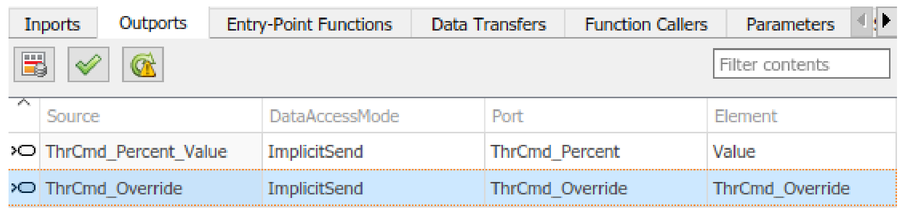
```
autosar.api.create('Controller','incremental');
autosar.api.create('Controller');
```

For more information about function syntax and behavior, see `autosar.api.create`.

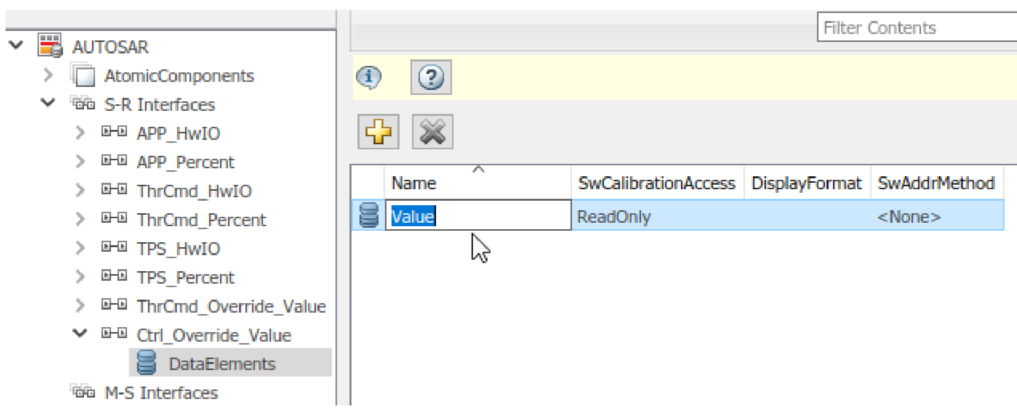
4 Here is the updated Simulink to AUTOSAR mapping of Simulink inports and outports in the model. Notice that the added inport and outport are each mapped to an AUTOSAR port and data element, which the function created in AUTOSAR Dictionary. The function also created the S-R interfaces that own each data 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



5 The function provided default naming and properties for the AUTOSAR ports, S-R interfaces, and data elements created in 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

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.

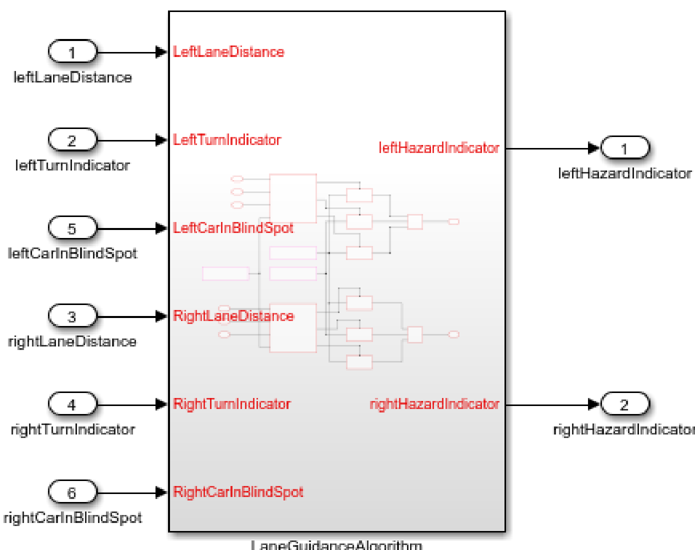
An AUTOSAR adaptive software component provides and consumes services. Each software component contains:

- 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.

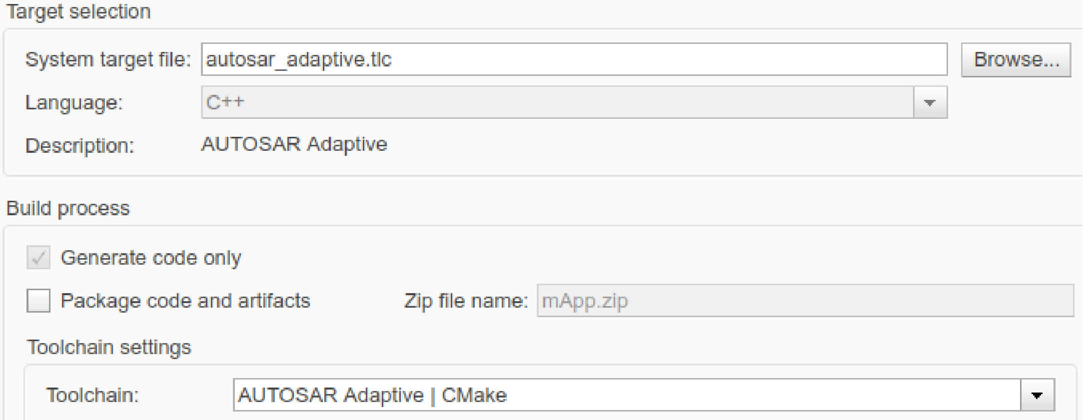
For more information, see “Model AUTOSAR Adaptive Software Components” on page 2-62.

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 new setting affects other model settings. For example, the target file selection:

- Sets **Language** to `C++`.

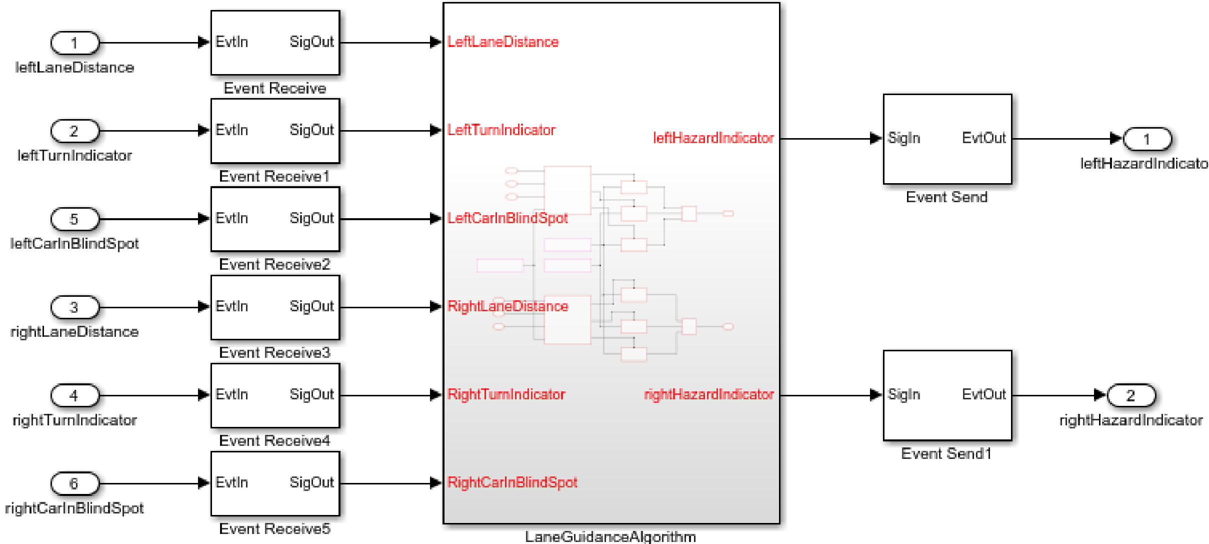
- Selects **Generate code only**.
- Sets **Toolchain** to `AUTOSAR Adaptive | CMake`.
- Sets **Interface > Code interface packaging** to `C++ class`.

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 algorithmic model to an AUTOSAR adaptive software component.

a To map the algorithm model, in the model window, select **Code** > **C/C++ Code** > **Configure Model in Code Perspective**. (Alternatively, click the perspective control in the lower-right corner and select **Code**, or call MATLAB function `autosar.api.create(modelName)`.) An AUTOSAR Component Quick Start dialog box opens.

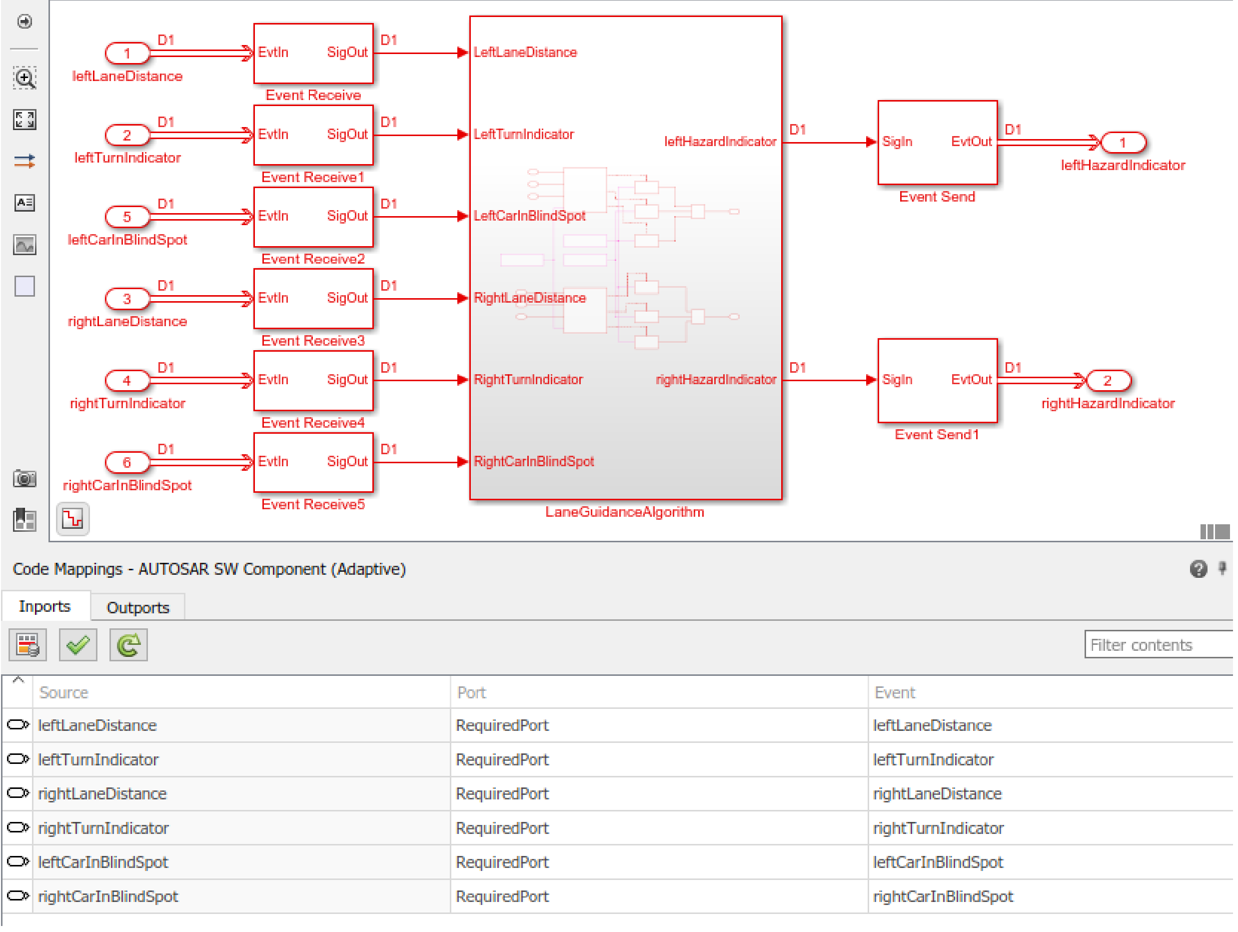
b To set up your AUTOSAR model, click through the AUTOSAR Component Quick Start panes. When a Quick Start pane offers default or delayed component creation, select **Create default mapping**.

Simulink elements must be mapped to AUTOSAR component elements. Create a default mapping or defer the mapping to later?

◉ Create default mapping

◯ Create mapping later

c Click the Quick Start **Finish** button. An AUTOSAR code perspective opens. 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.

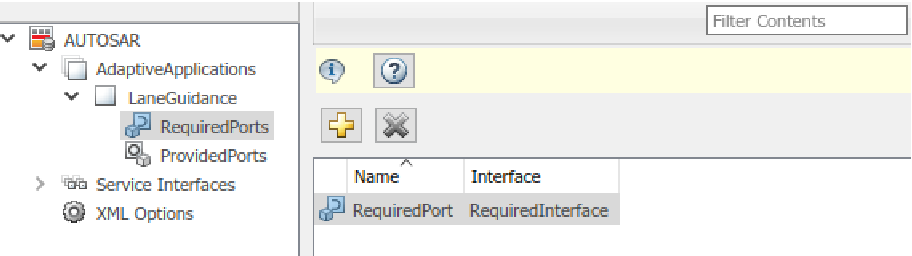


6 If you completed the adaptive configuration steps, save the AUTOSAR adaptive software component model with a unique name.

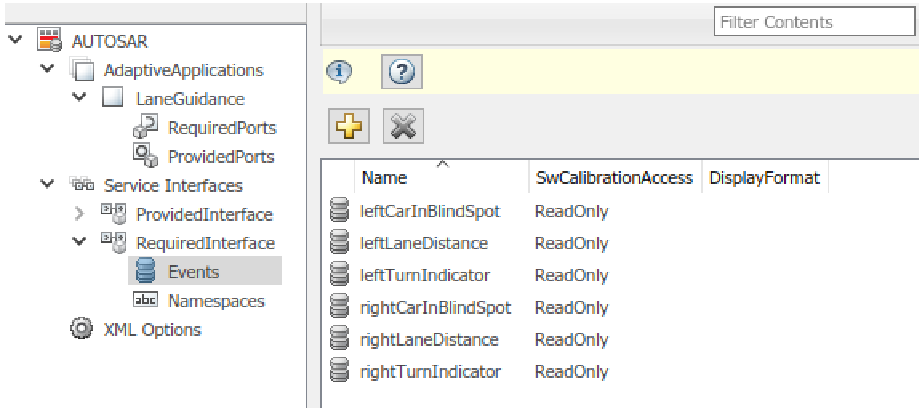
If you skipped any steps, open an example of a finished mapped AUTOSAR adaptive software component,example model `autosar_LaneGuidance`.

7 Using AUTOSAR code perspective and the AUTOSAR Dictionary (or equivalent AUTOSAR map and property functions), further refine the AUTOSAR adaptive model configuration.

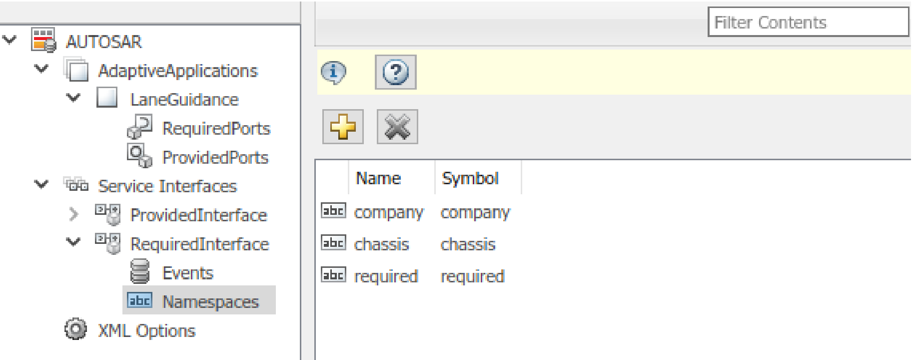
- In the model window, check model data to see if you need to make post-mapping adjustments to types or other attributes. For example, verify that event data is configured correctly for your design.
- In the AUTOSAR code perspective, examine the mapping of Simulink inports and outports to AUTOSAR required 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.



- 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 a model class header file.

  [-] **Interface files**

  autosar_LaneGuidance.arxml

  rtmodel.h

- C++ files that implement a main program module.

[-] **Other files**

MainUtils.hpp

main.cpp

- `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 AUTOSAR Dictionary.

```
// Model initialize function
void mObjectDetectionModelClass::initialize()
{
  {
    ara::com::ServiceHandleContainer< company::chassis::required::proxy::
      RequiredInterfaceProxy::HandleType > handles;
    ...
    handles = company::chassis::required::proxy::RequiredInterfaceProxy::
      FindService();
    if (handles.size() > 0U) {
      RequiredPort = std::make_shared<company::chassis::required::proxy::
        RequiredInterfaceProxy>(*handles.begin());
    }

    // Subscribe event
    RequiredPort->rightCarInBlindSpot.Subscribe(ara::com::EventCacheUpdatePolicy::
      kNewestN, 1U);
    ...
  }
}
```

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 message pop function to fetch and handle new `rightCarInBlindSpot` event data received by ARA middleware.

```
// Model step function
void mObjectDetectionModelClass::step()
{
...
  // Chart: '<S6>/Event Receive'
  autosar_LaneGuidance_DW.EvtIn_isValid = false;
  if (autosar_LaneGuidance_sf_msg_pop_EvtIn_n()) {
    autosar_LaneGuidance_B.SigOut = *(real_T *)
      autosar_LaneGuidance_DW.EvtIn_msgDataPtr;
  }

  autosar_LaneGuidance_sf_msg_discard_EvtIn_kl();
```

```
  // End of Chart: '<S6>/Event Receive'
...
}
```

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 2-62
- “Create and Configure AUTOSAR Adaptive Software Component” on page 3-17
- “Map AUTOSAR Adaptive Elements for Code Generation” on page 4-103
- “Configure AUTOSAR Adaptive Elements and Properties” on page 4-87
- “Configure AUTOSAR Adaptive Service Communication” on page 4-245
- “Configure AUTOSAR Adaptive Code Generation” on page 5-18

More About

- “Comparison of AUTOSAR Classic and Adaptive Platforms” on page 1-4

Design AUTOSAR Components, Simulate, 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

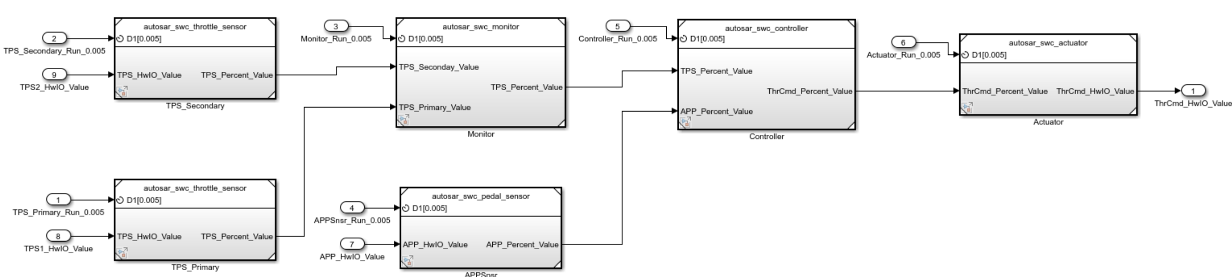
To develop AUTOSAR components in Simulink®, you first create a Simulink representation of an AUTOSAR software component. AUTOSAR component creation can start from an `arxml` component description or an existing Simulink design.

- 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 3-32 or example “Import AUTOSAR Composition to Simulink” on page 6-2.
- 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.

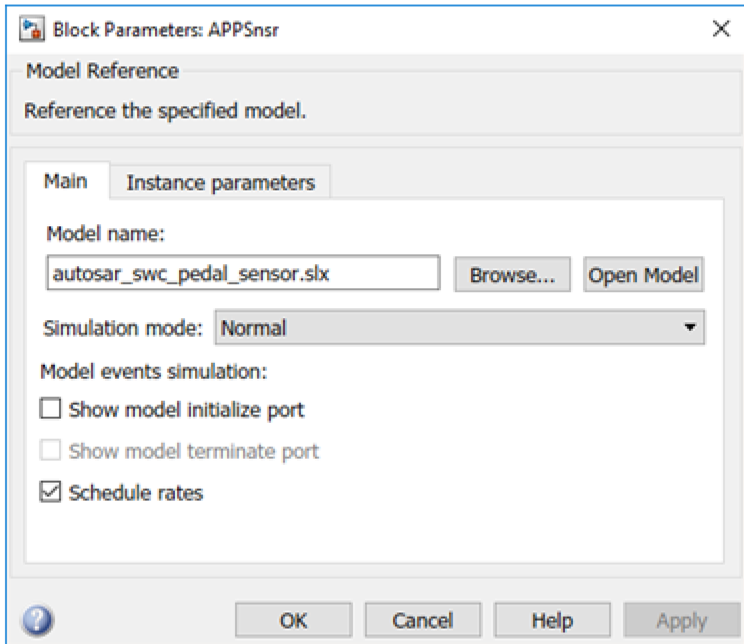
Open the composition model `autosar_composition`.

```
open_system('autosar_composition');
```



Signal lines between component models represent AUTOSAR assembly connectors, and 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) and 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 expects 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.

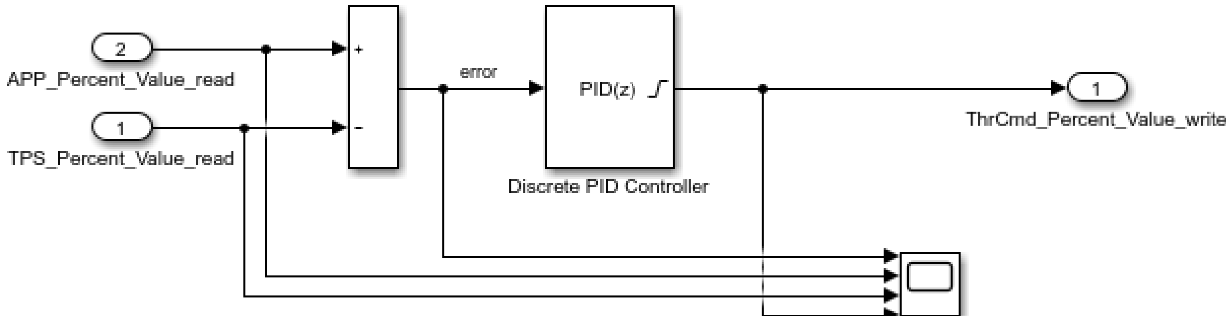
Develop AUTOSAR Component Algorithms

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.

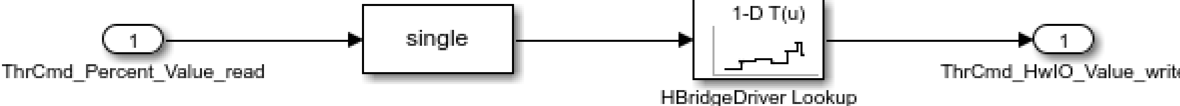
Here is the implementation of the throttle position controller behavior provided by the application component model `autosar_swc_controller`. The component takes as inputs an APP sensor percent value from a pedal position sensor and a TPS percent value

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.



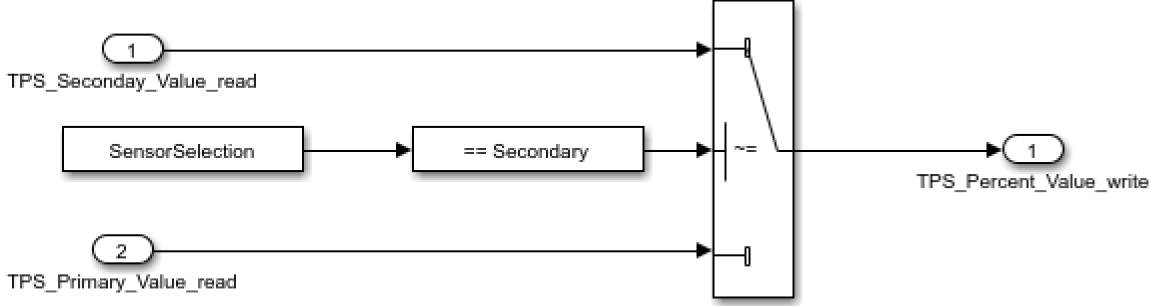
The sensor and actuator component models in the `autosar_composition` model use lookup tables to implement their value conversions. For example, consider the `autosar_swc_actuator` component model. When first imported or created in Simulink, the initial representation of the `autosar_swc_actuator` component likely contained an initial stub implementation of the actuator behavior.

Here is the implementation of the throttle position actuator behavior provided by the actuator component model `autosar_swc_actuator`. 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.

Here is the implementation of the throttle position monitor behavior provided by the application component model `autosar_swc_monitor`. 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 together 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 are licensed for Simulink Coder and Embedded Coder, 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 `rtwbuild('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

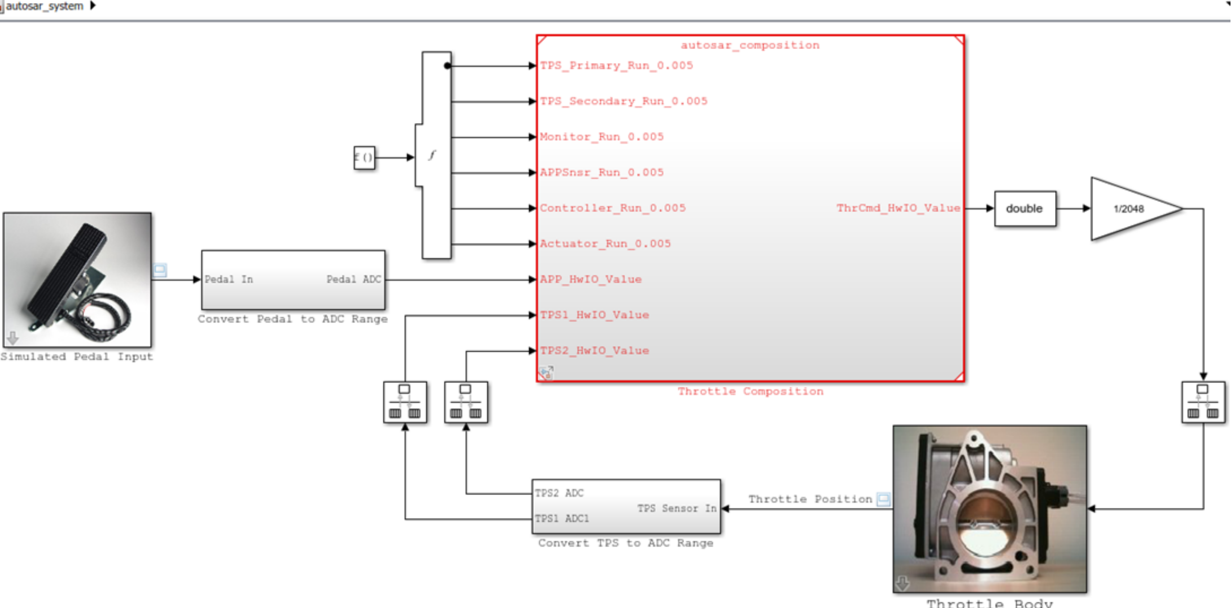
After you develop AUTOSAR components and compositions, you can test groups of components that belong together in a system-level simulation. Alternatives include:

- 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 using Simulink Test, see "Testing AUTOSAR Compositions". To open the Simulink Test example, enter the MATLAB command `showdemo('sltestAUTOSARCompositionDemo')`.

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`.

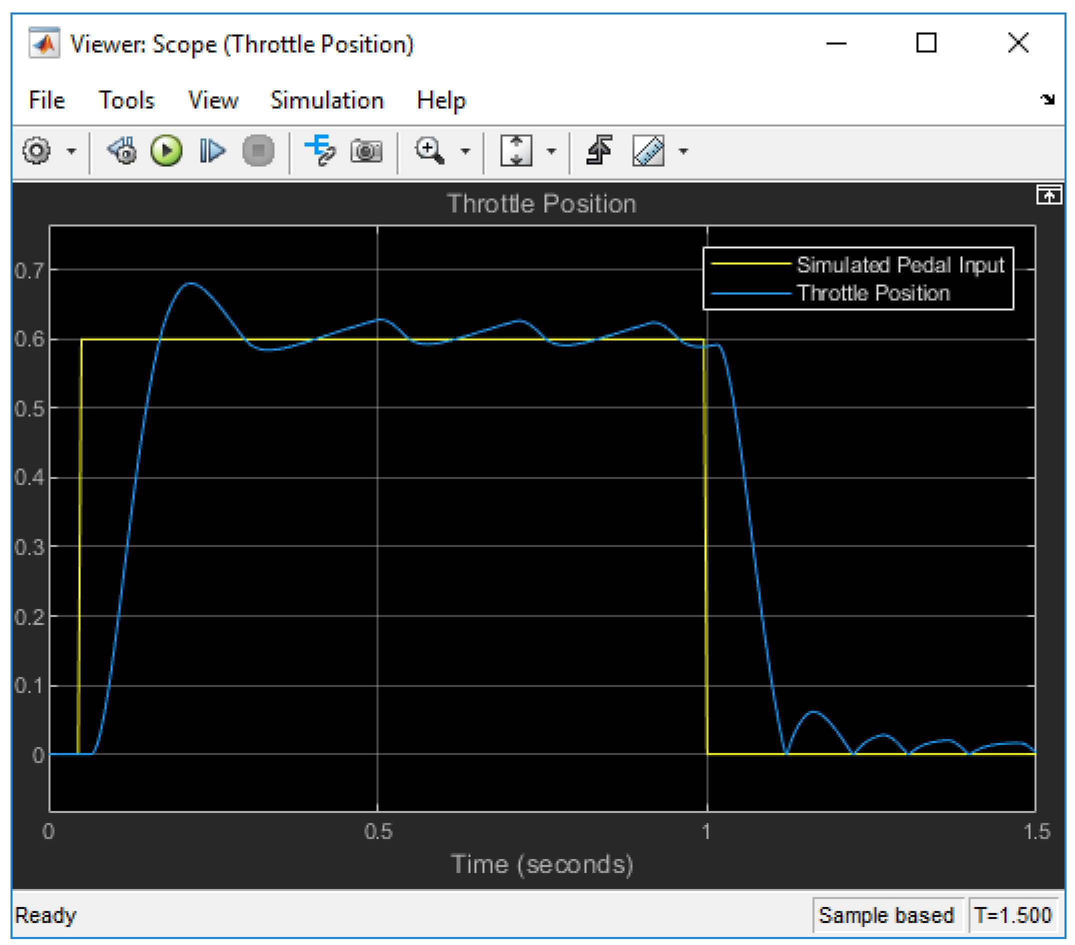
```
open_system('autosar_system');
```



The objective of the system-level model `autosar_system` is to allow 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 change a sensor source.

```
simOutSystem = sim('autosar_system');
```



Related Links

- "Import AUTOSAR Component to Simulink" on page 3-32
- "Import AUTOSAR Composition to Simulink" on page 6-2
- "Create AUTOSAR Software Component in Simulink" on page 3-2
- "Component Development"
- Testing AUTOSAR Compositions (Simulink Test featured example)

Configure Parameters and Signals for AUTOSAR Calibration and Measurement

Configure Simulink® model workspace parameters and signals for AUTOSAR run-time calibration and measurement.

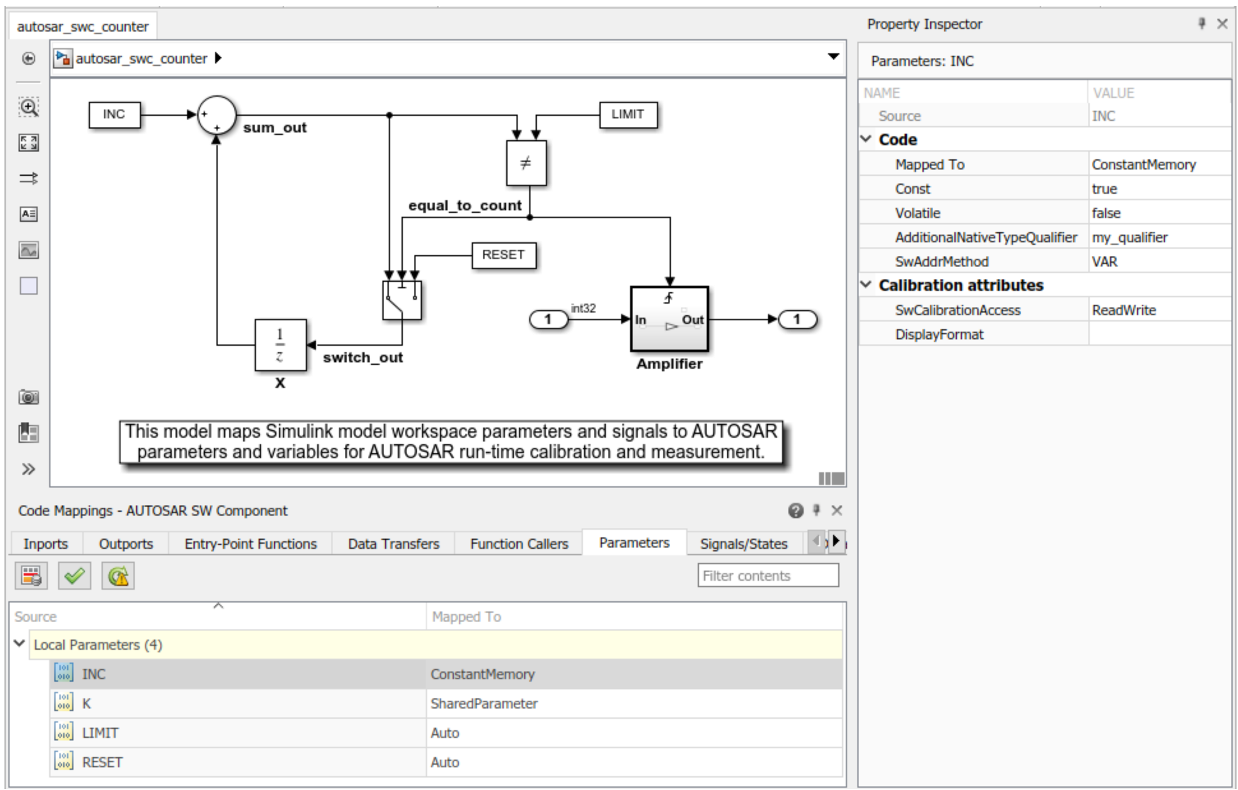
Map Model Workspace Parameters to AUTOSAR Parameters

Open the example model `autosar_swc_counter.slx`.

```
open_system('autosar_swc_counter')
```

To open the AUTOSAR code perspective, click the perspective control in the lower-right corner and select **Code**. In Code Mappings editor, select the **Parameters** tab. Expand the list of available local parameters and select `INC`. In the **Mapped To** drop-down list, select `ConstantMemory`.

Selecting a parameter displays the parameter attributes in Property Inspector. Use Property Inspector to modify the parameter attributes. For more information about parameter code and calibration attributes, see “Map Model Workspace Parameters to AUTOSAR Component Internal Parameters” on page 4-75.



If you are licensed for Simulink Coder and Embedded Coder, you can generate algorithmic C code and `arxml` component descriptions for testing in Simulink or integration into an AUTOSAR run-time environment. When you generate code:

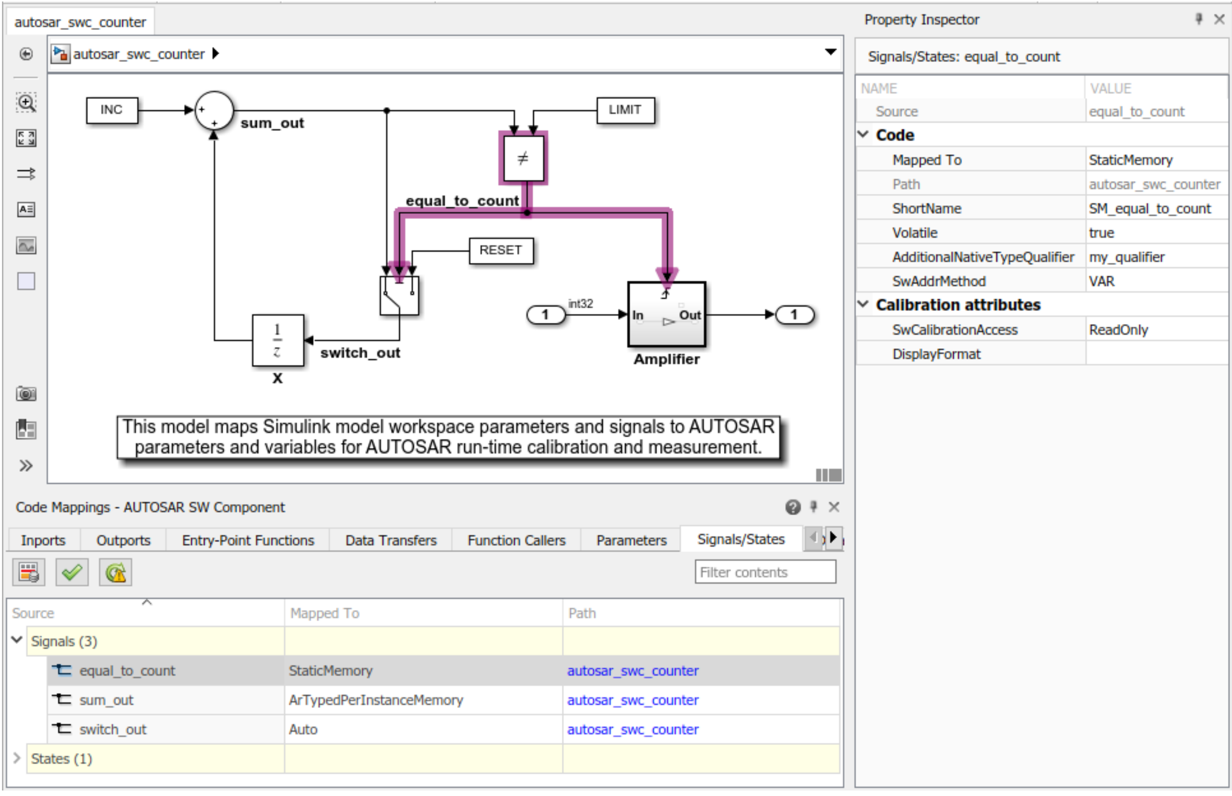
- Exported `arxml` files contain `CONSTANT-MEMORYS` descriptions for parameters that you configured as `ConstantMemory`. In the AUTOSAR Runtime Environment (RTE), calibration tools can access AUTOSAR `ConstantMemory` blocks for measurement and calibration.
- Generated C code declares and references the constant memory parameters.

Map Simulink Block Signals and States to AUTOSAR Variables

Open the example model `autosar_swc_counter.slx`, if it is not already open.

To open the AUTOSAR code perspective, click the perspective control in the lower-right corner and select **Code**. In Code Mappings editor, select the **Signals** tab. Expand the list of available signals and select `equal_to_count`. In the **Mapped To** drop-down list, select `StaticMemory`.

Selecting a signal highlights the signal in the model diagram and displays the signal attributes in Property Inspector. Use Property Inspector to modify the signal attributes. For more information about signal code and calibration attributes, see “Map Block Signals and States to AUTOSAR Variables” on page 4-79.



If you are licensed for Simulink Coder and Embedded Coder, you can generate algorithmic C code and `arxml` component descriptions for testing in Simulink or integration 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 Runtime Environment (RTE), calibration tools can access AUTOSAR `StaticMemory` blocks for measurement and calibration.
- Generated C code declares and references the static memory variables.

Related Links

- “Map Model Workspace Parameters to AUTOSAR Component Internal Parameters” on page 4-75
- “Configure AUTOSAR Constant Memory” on page 4-305
- “Map Block Signals and States to AUTOSAR Variables” on page 4-79
- “Configure AUTOSAR Static Memory” on page 4-300
- “Configure AUTOSAR Per-Instance Memory” on page 4-295

Model AUTOSAR Runnables Using Exported Functions

Use Simulink® exported functions to model AUTOSAR runnables.

Multiple Periodic Runnables Configured for Function Export

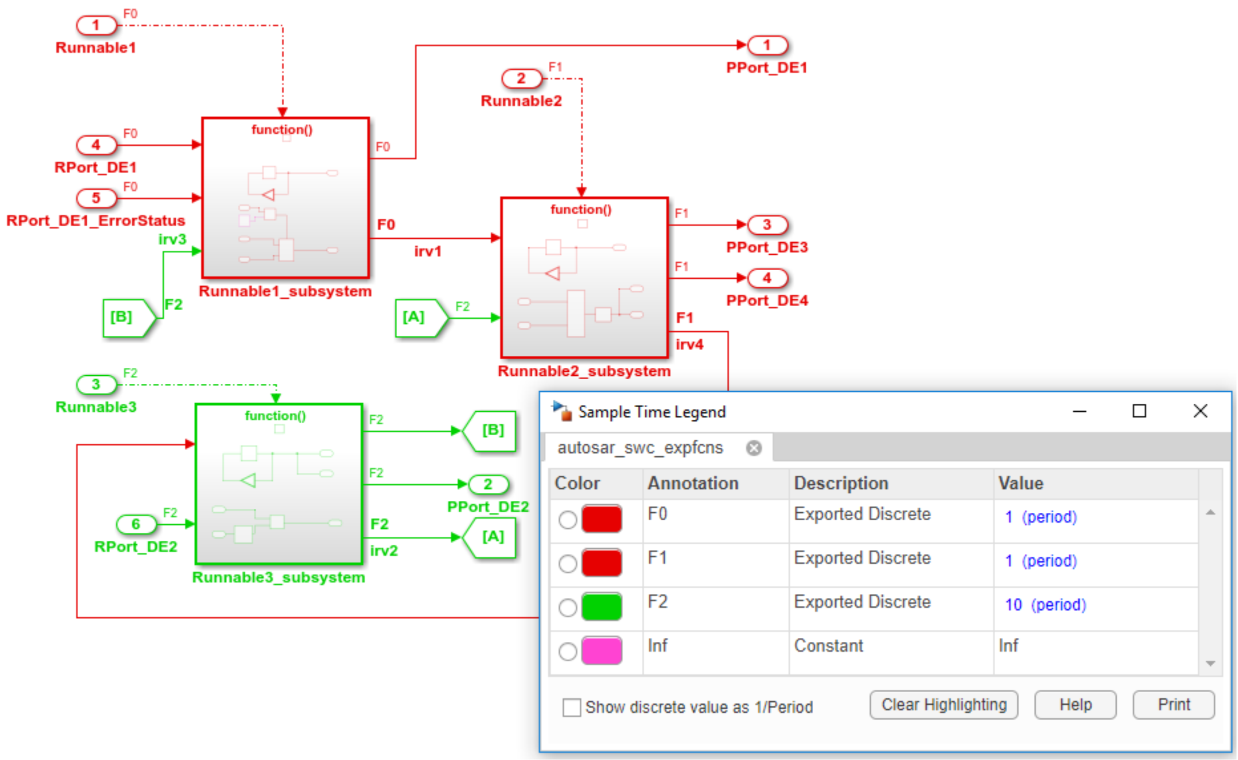
Open the example model `autosar_swc_expfcns.slx`.

```
open_system('autosar_swc_expfcns')
```

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. The function code can be integrated 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, select **Display > Sample Time > 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 are licensed for Simulink Coder and Embedded Coder, you can generate algorithmic C code and `arxml` component descriptions for testing in Simulink or integration 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 `rtwbuild('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    /* Outport: '<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 ConversionAtIn2Outport1'
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>/OutportBufferForAdder'
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" (Simulink)

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.

In this section...

"AR-PACKAGE Structure" on page 4-138
"Configure AUTOSAR Packages and Paths" on page 4-140
"Control AUTOSAR Elements Affected by Package Path Modifications" on page 4-143
"Export AUTOSAR Packages" on page 4-144
"AR-PACKAGE Location in Exported ARXML Files" on page 4-146

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.

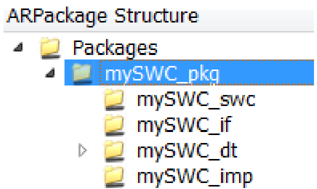
The `arxml` importer imports AR-PACKAGEs, their elements, and their paths into Simulink. The model configuration preserves the packages for subsequent export to `arxml` code. In general, the software preserves AUTOSAR packages across round-trips between an AAT and Simulink.

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 (including calibration components)
- Data types
- Port interfaces
- Internal behavior (schema 3.x or earlier)

- 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 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 AUTOSAR Dictionary to specify additional AR-PACKAGEs. (See “Configure AUTOSAR XML Options” on page 4-62.) 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 (schema 4.x)
- Software base types (schema 4.x)
- Data type mapping sets (schema 4.x)
- Constants and values
- Physical data constraints (referenced by application data types or data prototypes)
- System constants (schema 4.x)
- Software address methods
- Mode declaration groups
- Computation methods
- Units and unit groups (schema 4.x)
- Software record layouts (for application data types of category CURVE, MAP, CUBOID, or COM_AXIS)

- Internal data constraints (referenced by implementation data types)

Note

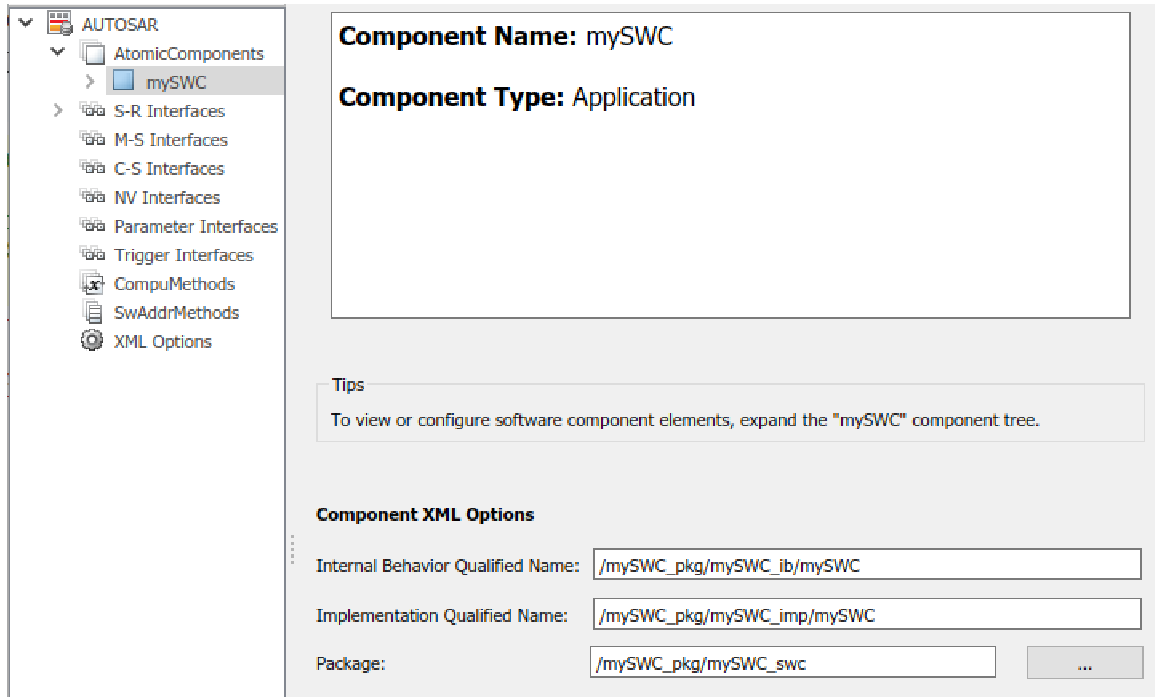
- For packages that you define in XML options, the `arxml` exporter generates a package only if the model contains an element of the package category. For example, the exporter generates a software address method package only if the model contains a software address method element.
- If your component uses schema 4.x, you can specify separate packages for the listed schema 4.x elements, for example, application data types. Schema 4.x 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 4-146.

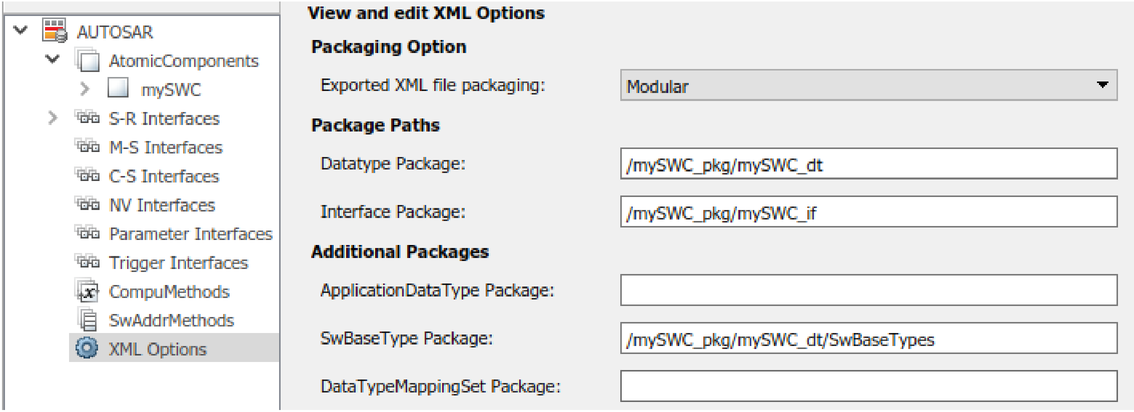
Configure AUTOSAR Packages and Paths

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 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 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 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.



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	*modelname*_pkg/*modelname*_ib/*modelname*
ImplementationQualifiedName	*modelname*_pkg/*modelname*_imp/*modelname*
ComponentQualifiedName	*modelname*_pkg/*modelname*_swc/*modelname* (dialog box displays the component path without the short name)
DataTypePackage	*modelname*_pkg/*modelname*_dt
InterfacePackage	*modelname*_pkg/*modelname*_if
ApplicationDataTypePackage	*DataTypePackage*/ApplDataTypes
SwBaseTypePackage	*DataTypePackage*/SwBaseTypes
DataTypeMappingPackage	*DataTypePackage*/DataTypeMappings
ConstantSpecificationPackage	*DataTypePackage*/Ground
DataConstraintPackage	*ApplicationDataTypePackage*/DataConstrs
SystemConstantPackage	*DataTypePackage*/SystemConstants
SwAddressMethodPackage	*DataTypePackage*/SwAddrMethods
ModeDeclarationGroupPackage	*DataTypePackage*
CompuMethodPackage	*DataTypePackage*
UnitPackage	*DataTypePackage*
SwRecordLayoutPackage	*ApplicationDataTypePackage*/RecordLayouts
InternalDataConstraintPackage	*DataTypePackage*/DataConstrs

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_swc_counter';
open_system(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 4-376.

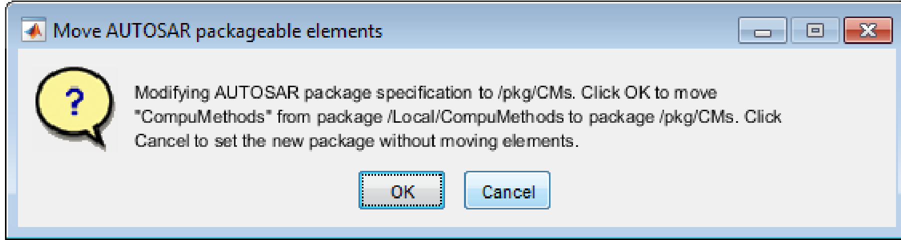
For an example of configuring and exporting AUTOSAR packages, see “Export AUTOSAR Packages” on page 4-144.

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 AUTOSAR Dictionary, and if packageable elements of that category are affected, a dialog box opens. For example, if you modify the XML option **CompuMethod Package** from path value `/Local/CompuMethods` to `/pkg/CMs`, the software opens the following dialog box.



To move CompuMethod 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.

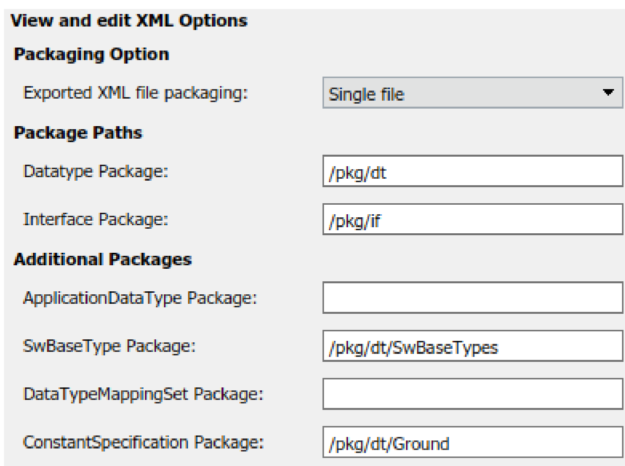
For example, the following code sets a new CompuMethod package path without moving existing CompuMethod elements to the new package.

```
arProps = autosar.api.getAUTOSARProperties('cmSpeed');
set(arProps,'XmlOptions','CompuMethodPackage','/pkg/CMs','MoveElements','None');
```

Export AUTOSAR Packages

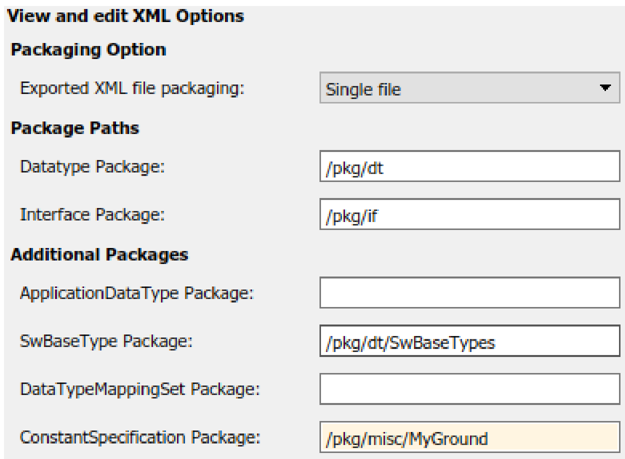
This example shows how to configure and export AUTOSAR packages for an AUTOSAR software component that originated in Simulink.

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_expfcns`.
2. Open AUTOSAR Dictionary and select **XML Options**. Here are initial settings for some of the AUTOSAR package parameters.

   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 sets the **ConstantSpecification Package** parameter to `/pkg/misc/MyGround`. (This value overrides the rule-based default, `/pkg/dt/Ground`.)



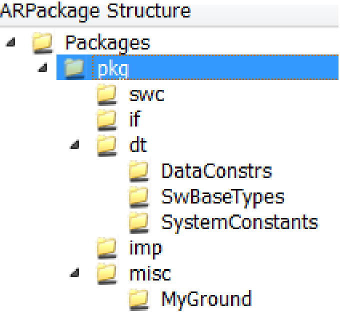
4. Generate code for the model.
5. Open the generated file `*modelname*.arxml`. (If you set **Exported XML file packaging** to `Modular`, open the generated file `*modelname*_component.arxml`.)
6. Search the XML code for the packages that you configured, for example, using the text AR-PACKAGE or an element name. For the example model, searching `autosar_swc_expfcns.arxml` for the text `MyGround` finds the constant specification package and many references to it. Here is a sample code excerpt.

```
<AR-PACKAGE UUID="95f4dcf1-66ea-5f72-cd27-34f4eb36ad65">
    <SHORT-NAME>MyGround</SHORT-NAME>
    <ELEMENTS>
        <CONSTANT-SPECIFICATION UUID="be03b363-5267-5a11-531f-22a443475036">
            <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="f1220406-2ece-568f-f709-693cc6b12b8a">
            <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>
```

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_multirunnables` with the following AR-PACKAGE structure. (See the steps in “Export AUTOSAR Packages” on page 4-144). In this configuration, the specified path of the constant specification package, `/pkg/misc/myGround`, overrides the rule-based default, `/pkg/dt/Ground`.



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_expfcns.arxml:`

```
<AR-PACKAGES>
    <AR-PACKAGE UUID="...">
        <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 packages to `arxml` files based on package path, not on package content or element category. For example, the exporter maps the data-type-oriented package, `/pkg/misc/myGround`, to the component file, `autosar_swc_exps_component.arxml`, based on the package path. To group the

package with other data-type-oriented packages, both in the AR-PACKAGE structure and in `arxml` output, specify a package path beginning with `/pkg/dt/`.

autosar_swc_expfcns_component.arxml:

```
<AR-PACKAGES>
    <AR-PACKAGE UUID="...">
        <SHORT-NAME>pkg</SHORT-NAME>
        ...
                <SHORT-NAME>swc</SHORT-NAME>
                ...
                <SHORT-NAME>misc</SHORT-NAME>
                ...
                        <SHORT-NAME>MyGround</SHORT-NAME>
                        ...
    </AR-PACKAGE>
</AR-PACKAGES>
```

autosar_swc_expfcns_interface.arxml:

```
<AR-PACKAGES>
    <AR-PACKAGE UUID="...">
        <SHORT-NAME>pkg</SHORT-NAME>
        ...
                <SHORT-NAME>if</SHORT-NAME>
                ...
    </AR-PACKAGE>
</AR-PACKAGES>
```

autosar_swc_expfcns_datatype.arxml:

```
<AR-PACKAGES>
    <AR-PACKAGE UUID="...">
        <SHORT-NAME>pkg</SHORT-NAME>
        ...
                <SHORT-NAME>dt</SHORT-NAME>
                ...
                        <SHORT-NAME>SwBaseTypes</SHORT-NAME>
                        ...
    </AR-PACKAGE>
</AR-PACKAGES>
```

autosar_swc_expfcns_implementation.arxml:

```
<AR-PACKAGES>
    <AR-PACKAGE UUID="...">
        <SHORT-NAME>pkg</SHORT-NAME>
        ...
                <SHORT-NAME>imp</SHORT-NAME>
                ...
    </AR-PACKAGE>
</AR-PACKAGES>
```

See Also

Related Examples

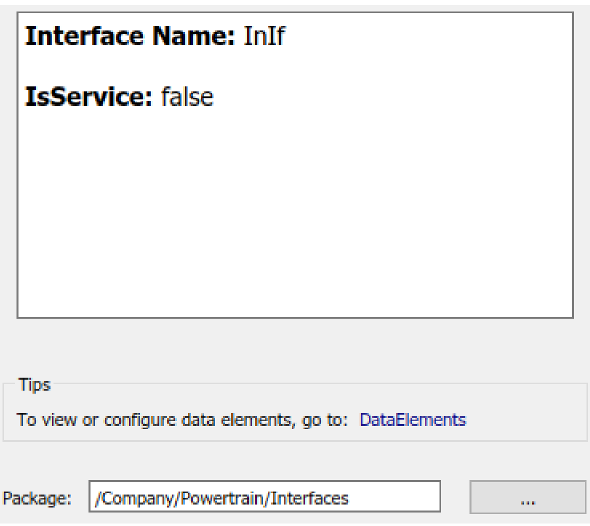
- “Import AUTOSAR Software Component” on page 3-27
- “Configure AUTOSAR Package for Component, Interface, CompuMethod, or SwAddrMethod” on page 4-150
- “Configure AUTOSAR XML Options” on page 4-62
- “Configure AUTOSAR XML Export” on page 4-376
- “Configure AUTOSAR Code Generation” on page 5-12

More About

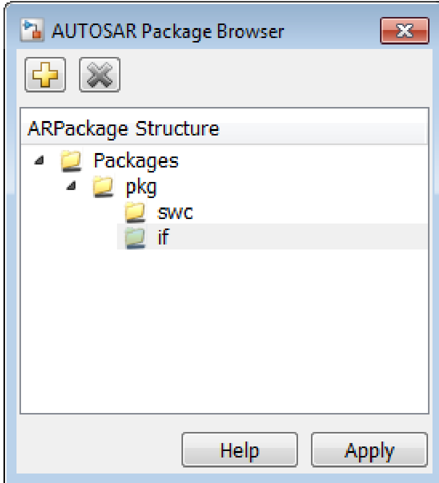
- “AUTOSAR Component Configuration” on page 4-3

Configure AUTOSAR Package for Component, Interface, CompuMethod, or SwAddrMethod

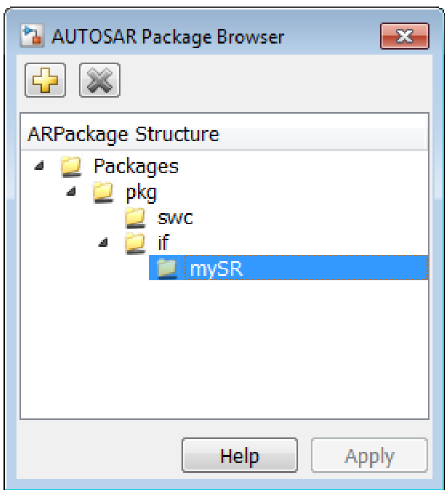
As part of configuring an AUTOSAR component, interface, CompuMethod, or SwAddrMethod, you specify the AUTOSAR package (AR-PACKAGE) to be generated for individual components, interfaces, CompuMethods, or SwAddrMethods in your configuration. For example, here is the AUTOSAR Dictionary view for an individual 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: /pkg/if/mySR

For more information about AR-PACKAGEs, see “Configure AUTOSAR Packages” on page 4-138.

See Also

Related Examples

- “Configure AUTOSAR Elements and Properties” on page 4-8
- “Configure and Map AUTOSAR Component Programmatically” on page 4-347

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 provide and require ports that send and receive data.

In Simulink, you can create AUTOSAR S-R interfaces and ports, and map Simulink inports and outports to AUTOSAR ports. You model AUTOSAR provide and require ports with Simulink root-level outports and inports, as described in “Sender-Receiver Interface” on page 2-27.

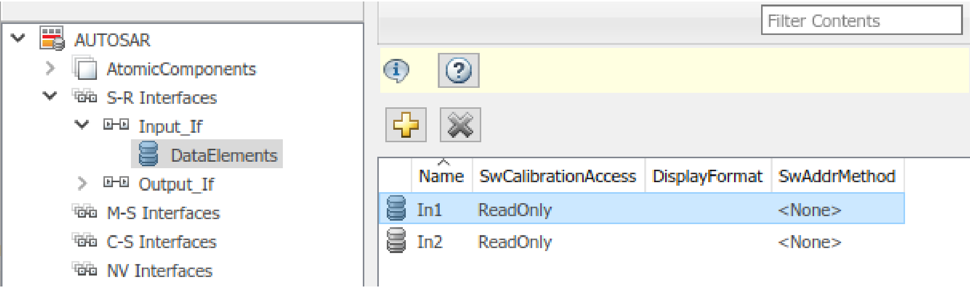
For queued sender-receiver communication, see “Configure AUTOSAR Queued Sender-Receiver Communication” on page 4-172.

In this section...
“Configure AUTOSAR Sender-Receiver Interface” on page 4-153
“Configure AUTOSAR Provide-Require Port” on page 4-155
“Configure AUTOSAR Receiver Port for IsUpdated Service” on page 4-157
“Configure AUTOSAR Sender-Receiver Data Invalidation” on page 4-159
“Configure AUTOSAR S-R Interface Port for End-To-End Protection” on page 4-163
“Configure AUTOSAR Receiver Port for DataReceiveErrorEvent” on page 4-165
“Configure AUTOSAR Sender-Receiver Port ComSpecs” on page 4-168

Configure AUTOSAR Sender-Receiver Interface

To create an S-R interface and ports in Simulink:

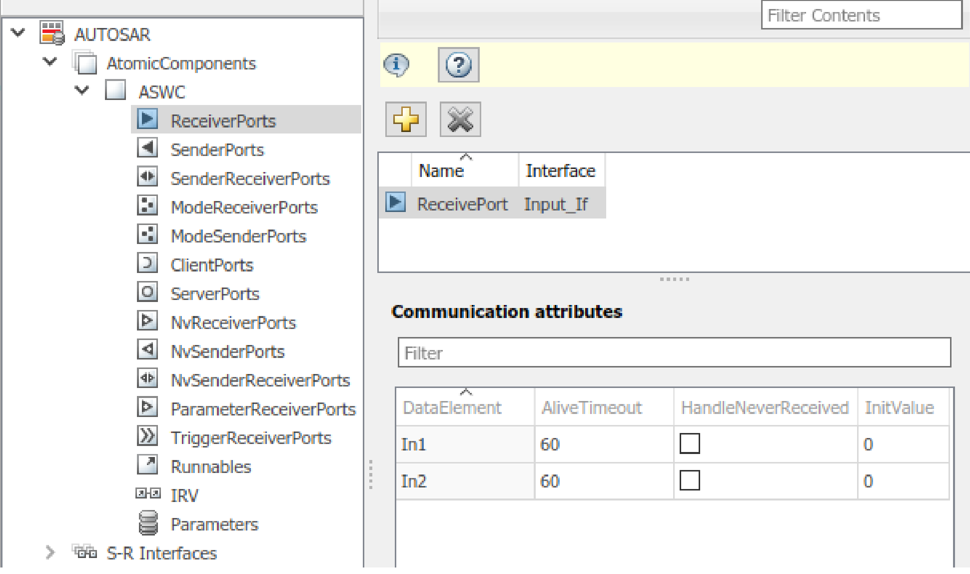
1. Open AUTOSAR Dictionary and select **S-R Interfaces**. Click the **Add** button to create a new AUTOSAR S-R data interface. Specify its name and the number of associated S-R data elements.
2. Select and expand the new S-R interface. Select **DataElements**, and modify the AUTOSAR data element attributes.



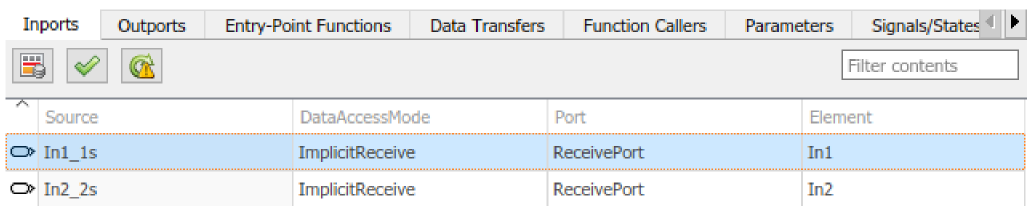
3 In AUTOSAR Dictionary, expand the **AtomicComponents** node and select an AUTOSAR component. Expand the component.

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 4-168.



5. Open 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.

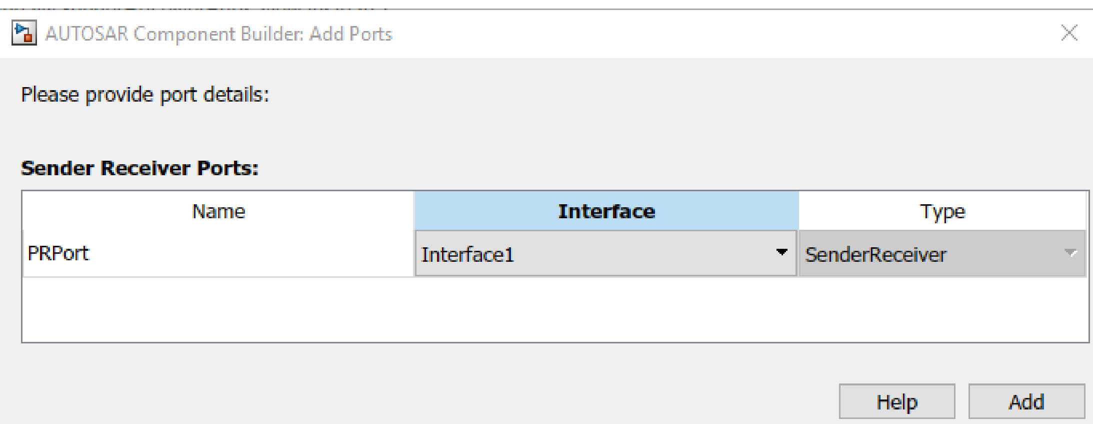


Configure AUTOSAR Provide-Require Port

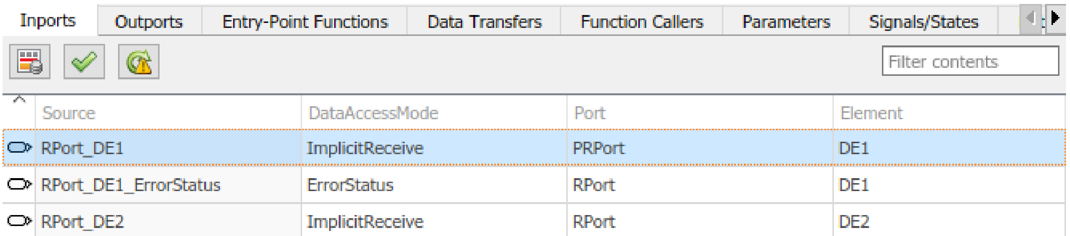
AUTOSAR Release 4.1 introduced the AUTOSAR provide-require port (`PRPort`). Modeling an AUTOSAR `PRPort` involves using a Simulink inport and outport pair with matching data type, dimension, and signal type. You can associate a `PRPort` with a sender-receiver (S-R) interface or a nonvolatile (NV) data interface.

To configure an AUTOSAR `PRPort` for S-R communication in Simulink:

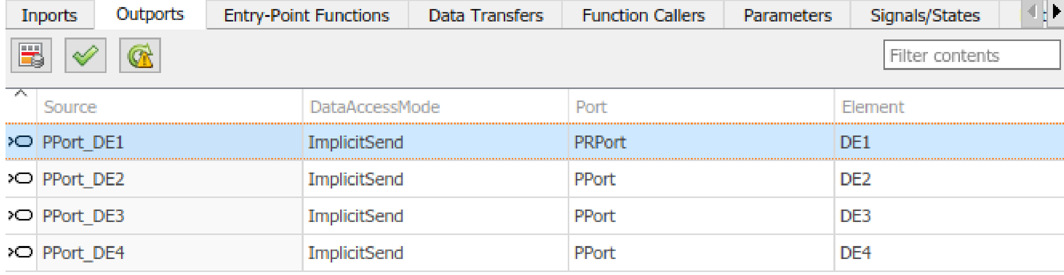
1. Open a model that is configured for AUTOSAR, and in which a runnable has an inport and an outport suitable for pairing into an AUTOSAR `PRPort`. In this example, the `RPort_DE1` inport and `PPort_DE1` outport both use data type `int8`, port dimension 1, and signal type `real`.
2. Open AUTOSAR Dictionary 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 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 outport will share.



5 Select the **Outports** tab. To map a Simulink outport to the AUTOSAR sender-receiver port you created, select the outport, set **Port** to the value `PRPort`, and set **Element** to the same data element selected in the previous step.



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 outport 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 outport pair.

```
hModel = 'my_autosar_expfcns';
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.

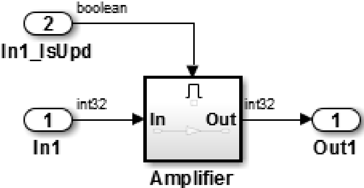
For the sender, the AUTOSAR Runtime Environment (RTE) sets the status of an update flag, indicating whether the data element has been written. The receiver calls the `Rte_IsUpdated_`*`Port_Element`* API, which reads the update flag and returns a value indicating whether the data element has been updated 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:

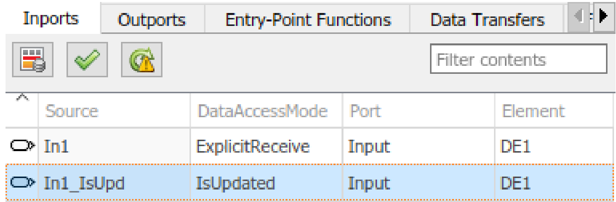
1. Open a model for which an AUTOSAR sender-receiver interface is configured.
2. Identify the inport that corresponds to the AUTOSAR receiver port for which `IsUpdated` service is required. Create a second inport, set its data type to `boolean`, and connect it to the same block. For example:



3. Open 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.



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

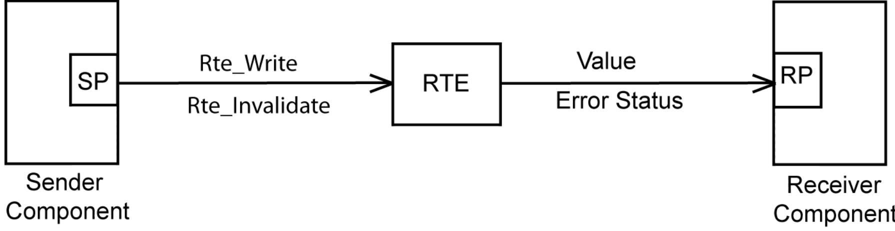
The AUTOSAR standard defines an invalidation mechanism for AUTOSAR data elements used in sender-receiver (S-R) communication. A sender component can notify a downstream receiver component that data in a sender port is invalid. Each S-R data element can have an invalidation policy. In Simulink, you can:

- 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**:

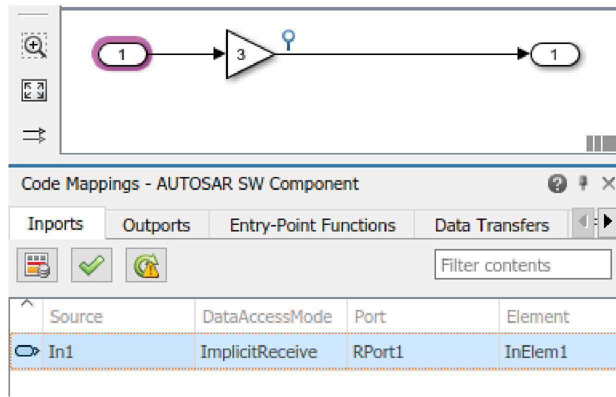
- `Keep` - Replace the input data value with the last valid signal value.
- `Replace` - Replace the input data value with an **Initial value** parameter.
- `DontInvalidate` - Do not replace the input data value.



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 outport named `Out` is mapped to AUTOSAR sender port `PPort` and data element `OutElem`. In 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 AUTOSAR Dictionary, AUTOSAR receiver port `RPort1` selects S-R interface `In1`, which contains the data element `InElem1`.

2 Add a Signal Invalidation block to the model.

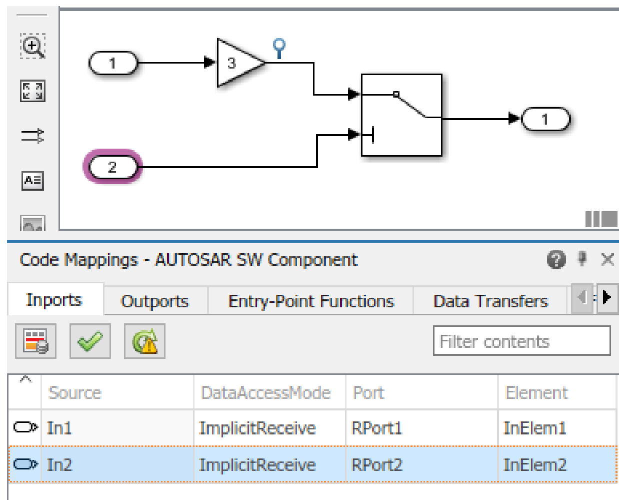
   a The block must be connected directly to a root outport block. Connect the block to root outport `Out`.

   b Connect the first block input, a data value, to the data path from root inport `In1`.

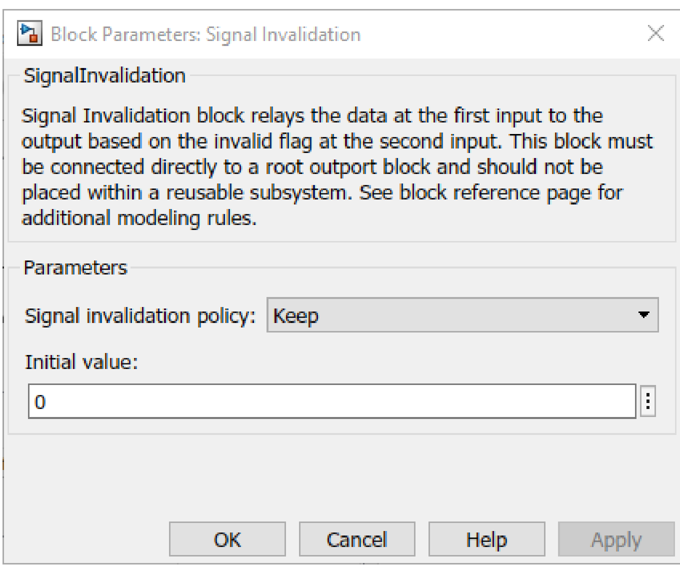
   c For the second block input, an invalidation control flag, add a root inport named `In2` to the model. Set its data type to scalar `boolean`. Map the new inport to a second AUTOSAR sender port. If a second AUTOSAR sender port does not exist, use AUTOSAR Dictionary to create the AUTOSAR port, S-R interface, and data element.

   In this example, Simulink inport `In2` is mapped to AUTOSAR receiver port `RPort2` and data element `InElem2`. In AUTOSAR Dictionary, AUTOSAR receiver port `RPort2` selects S-R interface `In2`, which contains the data element `InElem2`.

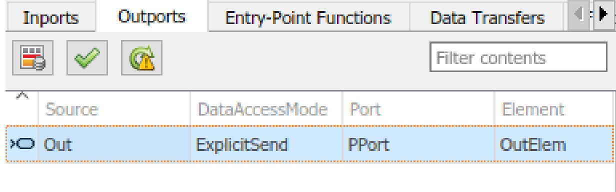
   Connect the second block input to root inport `In2`.



3 View the Signal Invalidation block parameters, for example, in Property Inspector or the block parameters dialog box. Examine the **Signal invalidation policy** and **Initial value** attributes. For more information, see the Signal Invalidation block reference page.



4 Open Code Mappings editor and select the **Outports** tab. For the root outport `Out`, verify that the AUTOSAR data access mode is set to `ExplicitSend` or `EndToEndWrite`.



5 To validate the AUTOSAR component configuration, open Code Mappings editor and click the **Validate** button.

6 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.

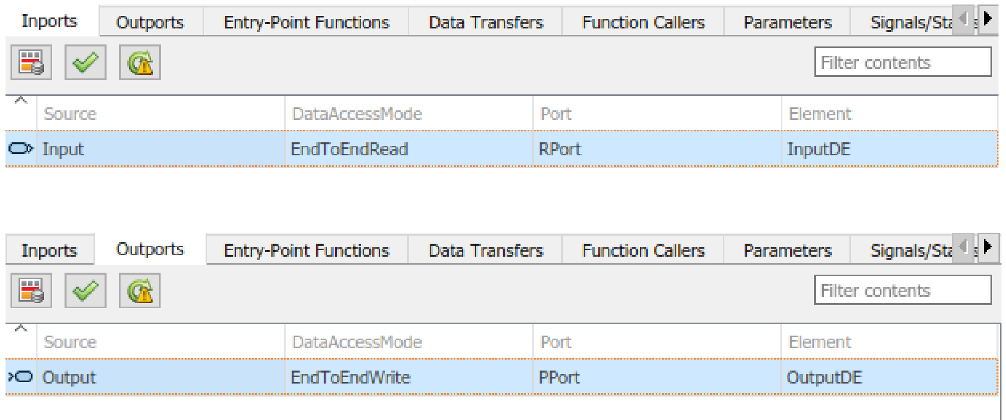
You should configure E2E protection for AUTOSAR sender and receiver ports that use explicit write and read data access modes. When you change the data access mode of an AUTOSAR port from explicit write to end-to-end write, or from explicit read to end-to-end read:

- Simulation behavior is unaffected.
- Code generation is similar to explicit write and read, with these differences:
  - E2E initialization wrapper API calls appear in C initialization code.
  - E2E protection wrapper API calls appear in C step code.
  - When combined with an error status inport, end-to-end read returns `uint32` rather than `uint8`.

- For receiver and sender COM-SPECs, the `arxml` exporter generates property USES-END-TO-END-PROTECTION with value `true`.

To configure an AUTOSAR sender or receiver port for E2E protection:

1. Open a model for which an AUTOSAR sender-receiver interface is configured.
2. Open Code Mappings editor. Navigate to the Simulink inport or outport that models the AUTOSAR receiver or sender port for which you want to configure E2E protection. Select the port.
3. Set the AUTOSAR data access mode to `EndToEndRead` (inport) or `EndToEndWrite` (outport).



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 `E2E` API calls.

```
void Runnable_Step(void)
{
  …
  /* Inport: '<Root>/Input' */
  E2EPW_Read_RPort_InputDE(…);
  …
  /* Outport: '<Root>/Output'... */
  (void) E2EPW_Write_PPort_OutputDE(…);
  …
}
…
void Runnable_Init(void)
{
  …
```

```
  /* End-to-End (E2E) initialization */
  E2EPW_ReadInit_RPort_InputDE();
  E2EPW_WriteInit_PPort_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 `DataReceiveErrorEvent`s in AUTOSAR receiver components. In Simulink, you can:

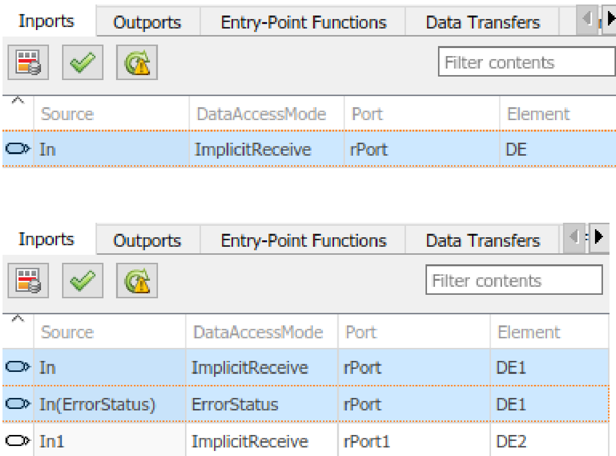
- Import an AUTOSAR `DataReceiveErrorEvent` definition.
- Define a `DataReceiveErrorEvent`.
- Generate `arxml` code for AUTOSAR receiver ports for which a `DataReceiveErrorEvent` is configured.

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 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

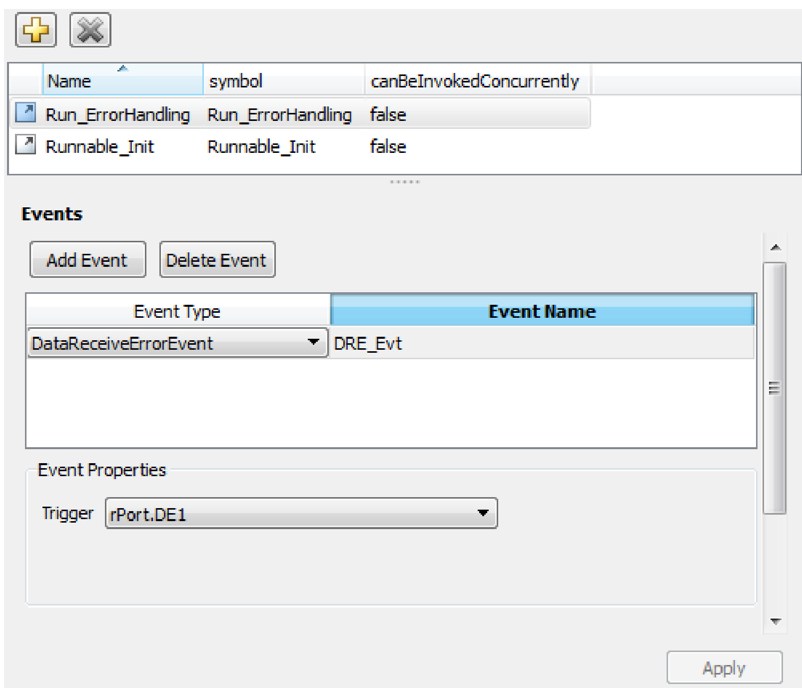
`ImplicitReceive`, `ExplicitReceive`, or `EndToEndRead`. Here are two examples, without and with a coupled ErrorStatus port.



3 Open 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_swc/ReceivingASWC/IB/Run_ErrorHandling</START-ON-EVENT-REF>
        <DATA-IREF>
            <CONTEXT-R-PORT-REF DEST="R-PORT-PROTOTYPE">
              /Root/mDemoModel_swc/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>
```

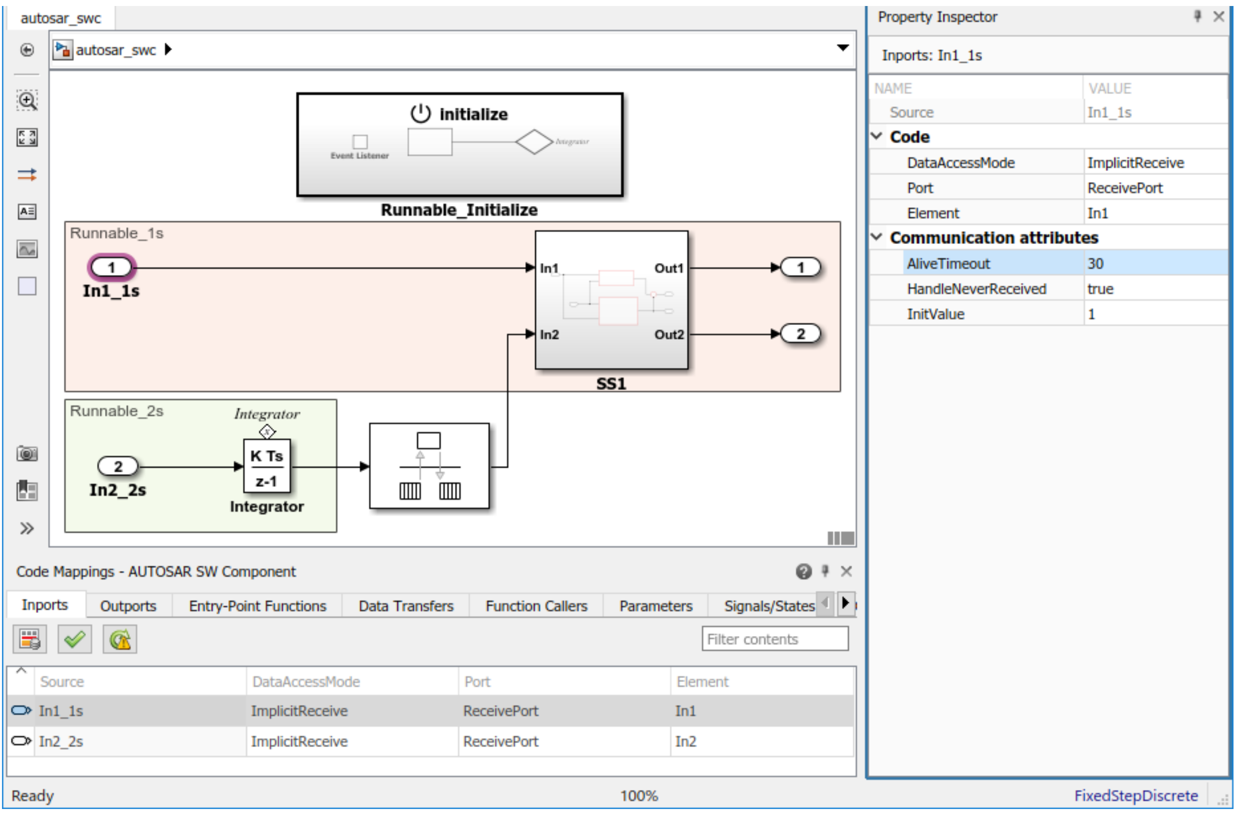
Configure AUTOSAR Sender-Receiver Port ComSpecs

In AUTOSAR software components, a sender or receiver port optionally can specify a communication specification (ComSpec). ComSpecs describe additional communication requirements for port data.

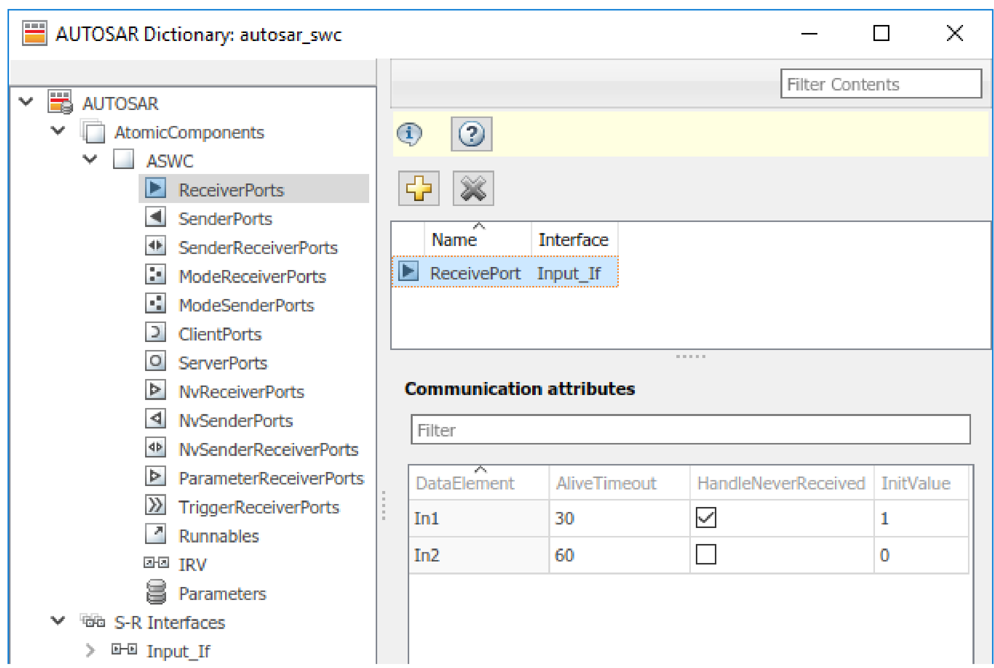
To model AUTOSAR sender and receiver ComSpecs in Simulink, you can:

- Import sender and receiver ComSpecs from `arxml` files
- Create sender and receiver ComSpecs in Simulink
- For nonqueued sender and receiver ports, modify ComSpec attribute `InitValue`
- For nonqueued receiver ports, modify ComSpec attributes `AliveTimeout` and `HandleNeverReceived`
- Export ComSpecs to `arxml` files

For example, if you create an AUTOSAR receiver port in Simulink, you use 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.



If you import or create an AUTOSAR receiver port, you can use 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 nonqueued sender or receiver port, use the AUTOSAR property function `set`. For example:

```
hModel = 'autosar_swc';
open_system(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')
```

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 2-26
- “Import AUTOSAR Software Component” on page 3-27
- “Configure AUTOSAR Code Generation” on page 5-12

More About

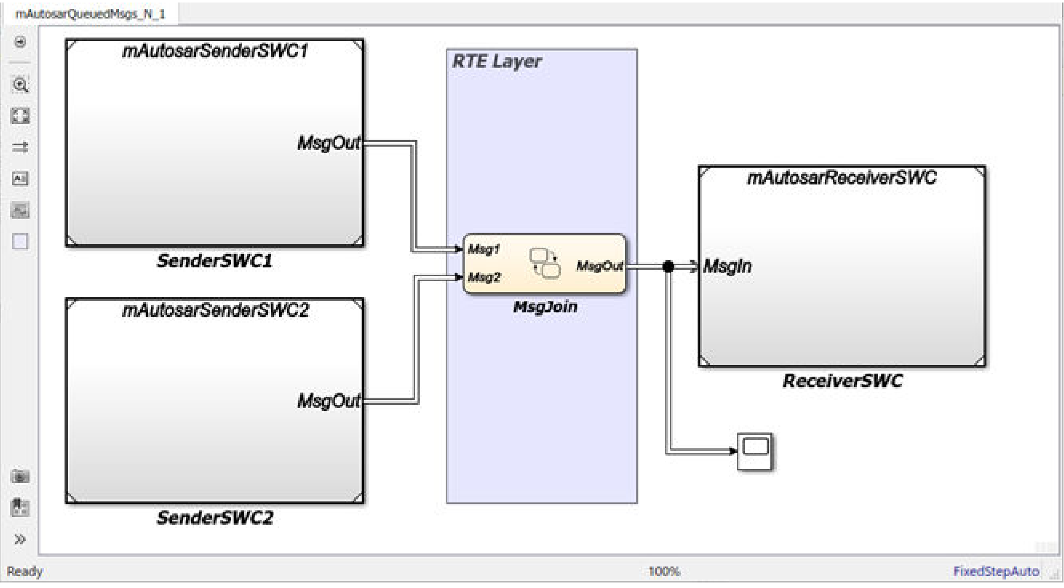
- “AUTOSAR Component Configuration” on page 4-3

Configure AUTOSAR Queued Sender-Receiver Communication

In AUTOSAR queued communication, 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.

In Simulink, you can use Stateflow® messages to model sending and receiving AUTOSAR data using a queue. You can handle errors that occur when the queue is empty or full. You can specify the size of the queue.

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 sender and receiver ports from `arxml` files into Simulink, or use Simulink to create queued sender and receiver ports.

In this section...

"Simulink Workflow for Modeling AUTOSAR Queued Send and Receive" on page 4-173
"Configure AUTOSAR Sender and Receiver Components for Queued Communication" on page 4-174
"Implement AUTOSAR Queued Send and Receive Messaging" on page 4-177
"Configure Simulation of AUTOSAR Queued Sender-Receiver Communication" on page 4-183
"Simulate N-to-1 AUTOSAR Queued Sender-Receiver Communication" on page 4-184
"Simulate Event-Driven AUTOSAR Queued Sender-Receiver Communication" on page 4-188

Simulink Workflow for Modeling AUTOSAR Queued Send and Receive

Here is 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 AUTOSAR Dictionary and 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 outport 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 4-174.
2. To implement AUTOSAR queued sender or receiver component behavior, use Stateflow messages. Follow the general procedure described in "Model an Assembly Line Feeder" (Stateflow). Create a chart, add message states, implement state entry actions, specify state transition conditions or events, and define data to store state variables. Finally, connect the chart message-line inputs and outputs to Simulink root inports and outports.

For more information, see “Implement AUTOSAR Queued Send and Receive Messaging” on page 4-177.

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 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`.

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.

   - If you have one sender component and one receiver component, you potentially can connect the models directly. For example, see the 1-to-1 composition model used in “Configure Simulation of AUTOSAR Queued Sender-Receiver Communication” on page 4-183.
   - 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 4-184 and “Simulate Event-Driven AUTOSAR Queued Sender-Receiver Communication” on page 4-188.

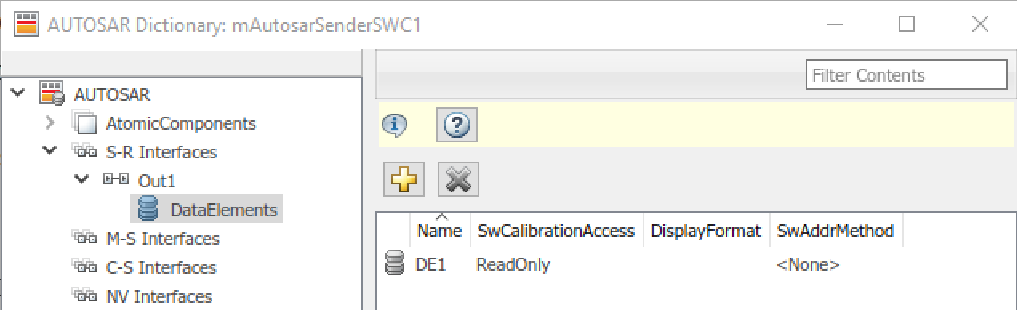
Configure AUTOSAR Sender and Receiver Components for Queued Communication

This example configures AUTOSAR queued sender and receiver components in Simulink. The example uses two models in the folder *`matlabroot`*`/help/toolbox/autosar/examples` (open). If you copy the files to a working folder, collocate the models. To see these models connected for simulation, see “Configure Simulation of AUTOSAR Queued Sender-Receiver Communication” on page 4-183.

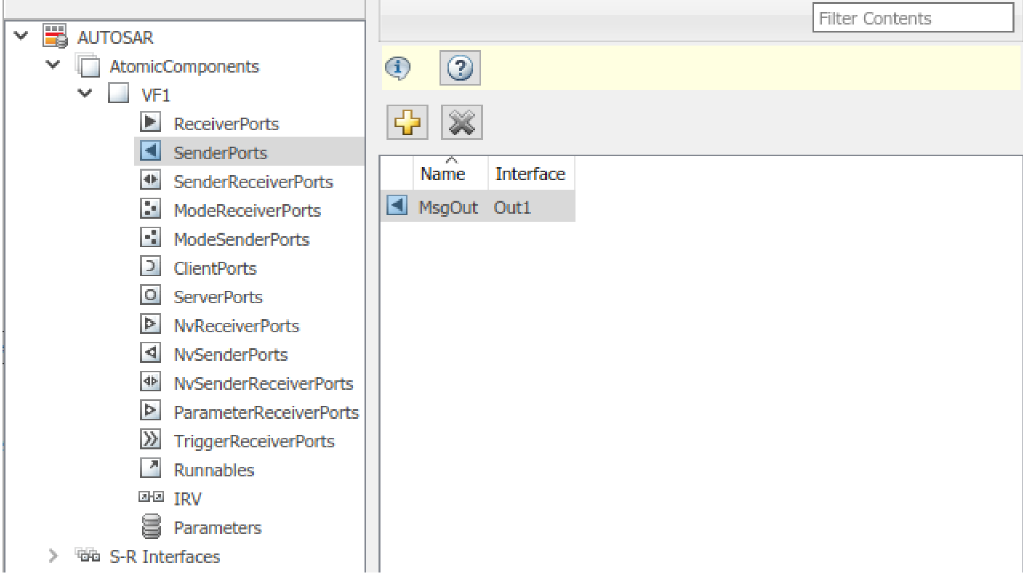
- `mAutosarSenderSWC1.slx`
- `mAutosarReceiverSWC.slx`

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:

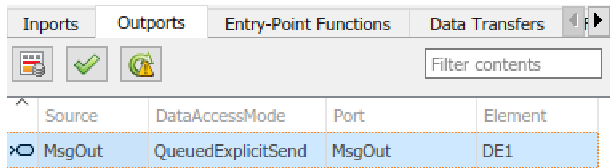
1. Open AUTOSAR Dictionary.
2. Select **S-R Interfaces**. To create an S-R data interface, click the **Add** button. Specify its name and the number of associated S-R data elements. This example uses one data element in both the sender and receiver components.
3. Select and expand the new S-R interface. Select **DataElements** and modify the data element attributes. Here is 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, here is sender port `MsgOut`, which uses S-R interface `Out1`.



6 Open 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, here is Simulink outport `MsgOut`, which is mapped to AUTOSAR sender port `MsgOut` and data element `DE1`, with data access mode `QueuedExplicitSend`.



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 Stateflow messages. See “Implement AUTOSAR Queued Send and Receive Messaging” on page 4-177.

Implement AUTOSAR Queued Send and Receive Messaging

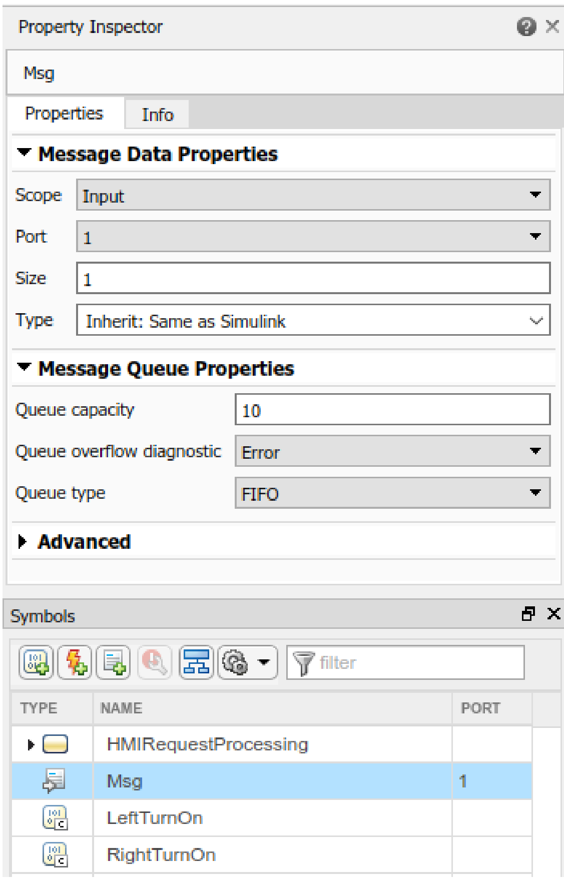
To implement AUTOSAR queued sender or receiver component behavior, use Stateflow messages. To create a Stateflow chart, follow the general procedure described in “Model an Assembly Line Feeder” (Stateflow).

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 Action Types” (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 Property Inspector, a Message properties dialog box, or Model Explorer. To view or modify message properties with Property Inspector:

1 Open a chart that uses messages.
2 In the model window, select **View** > **Property Inspector** and **View** > **Symbols**.
3 In the Symbols view, select a message. Property Inspector displays panes for **Message Data Properties** and **Advanced** properties. If the chart is in a receiver component, Property Inspector also displays **Message Queue Properties**.

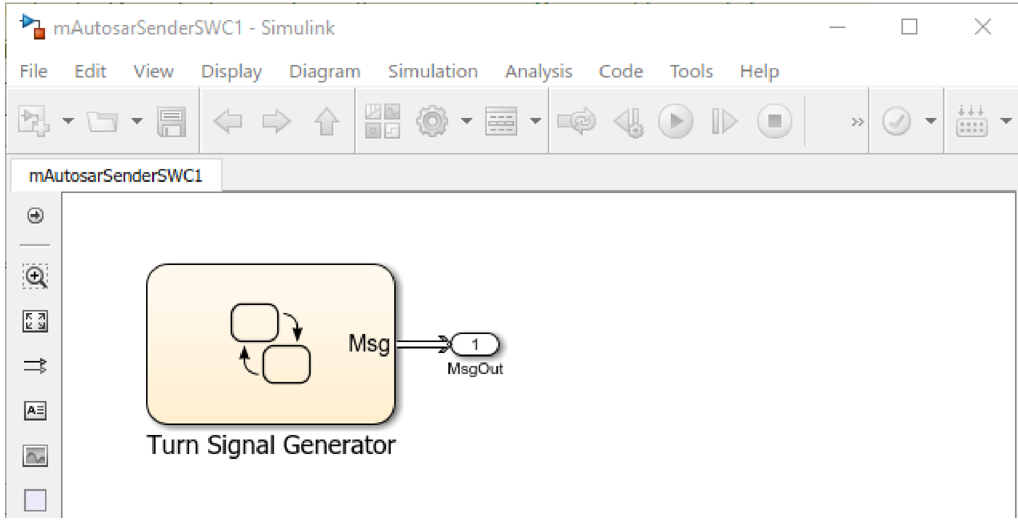


By default, message data type and queue capacity values are inherited from the Stateflow message to which a Simulink root port is attached. Message data can use Simulink parameter data types, such as `int` types, `uint` types, floating-point types, fixed-point types, `boolean`, `Enum`, or `Bus` (`struct`).

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.

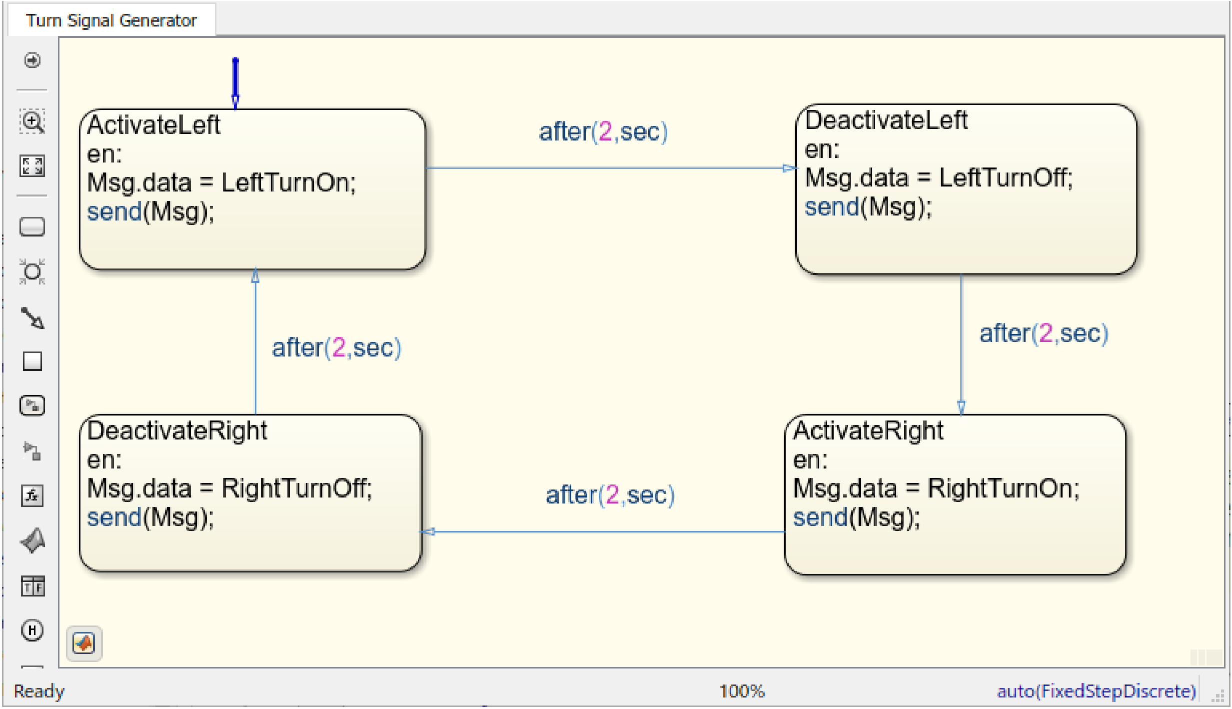
For sample implementations of queued sender and receiver components in a 1-to-1 configuration, see the example component models used in both “Configure AUTOSAR Sender and Receiver Components for Queued Communication” on page 4-174 and “Configure Simulation of AUTOSAR Queued Sender-Receiver Communication” on page 4-183. Models `mAutosarSenderSWC1.slx` and `mAutosarReceiverSWC.slx` are located in the folder *`matlabroot`*`/help/toolbox/autosar/examples` (open).

Here is 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 outport `MsgOut`.

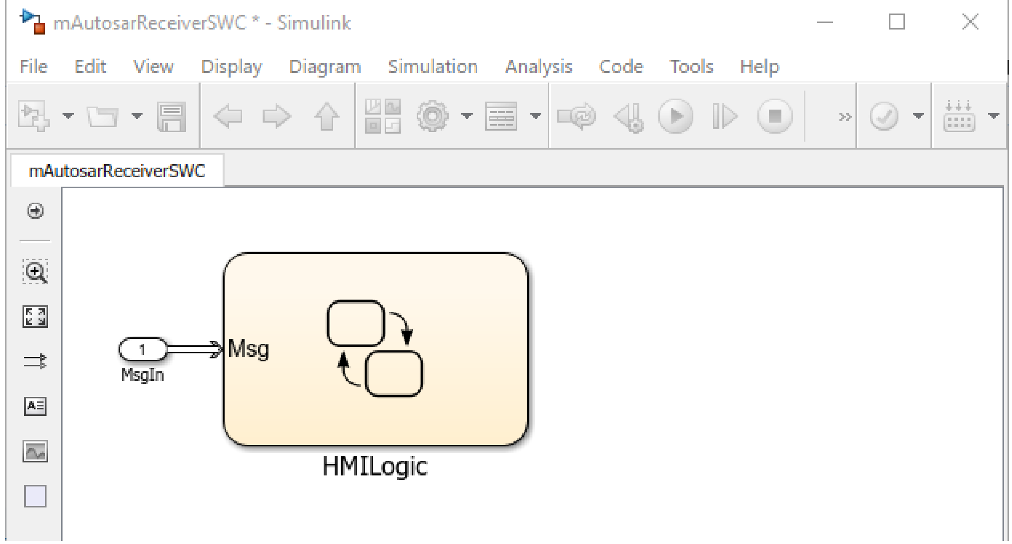


Here is the logic implemented in the `Turn Signal Generator` chart. The chart has four states – `ActivateLeft`, `DeactivateLeft`, `ActivateRight`, and `DeactivateRight`.

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.



Here is 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)`:

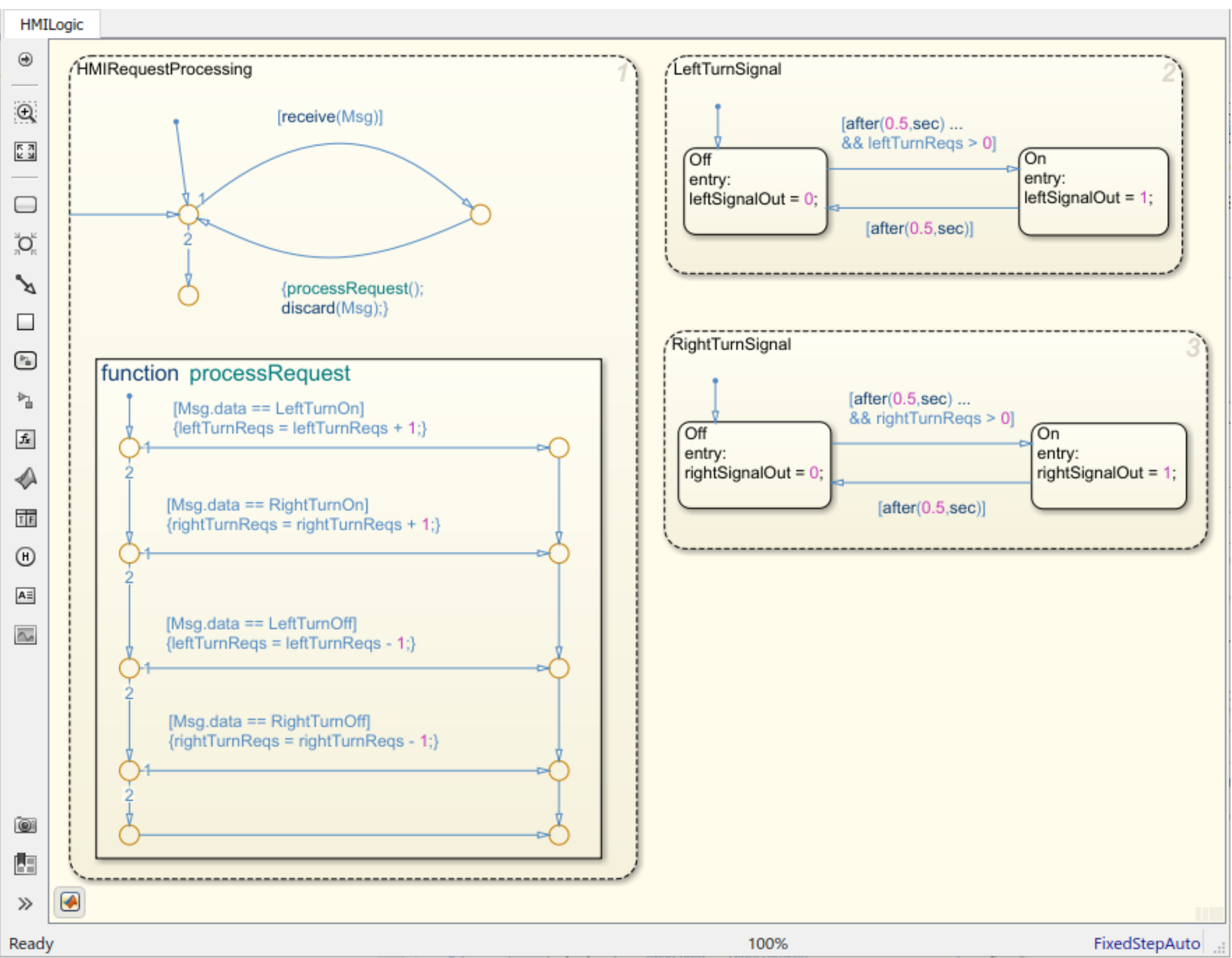
- If a valid message M exists, `receive(M)` returns true.
- If a valid message does not exist, the chart removes a message from its associated queue, and `receive(M)` returns true. If `receive(M)` removes a message from the queue, the length of the queue drops by one.
- If message M is invalid, and another 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).

Here is 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 4-184.

For sample implementations of event-driven queued messaging, see the example models used in “Simulate Event-Driven AUTOSAR Queued Sender-Receiver Communication” on page 4-188.

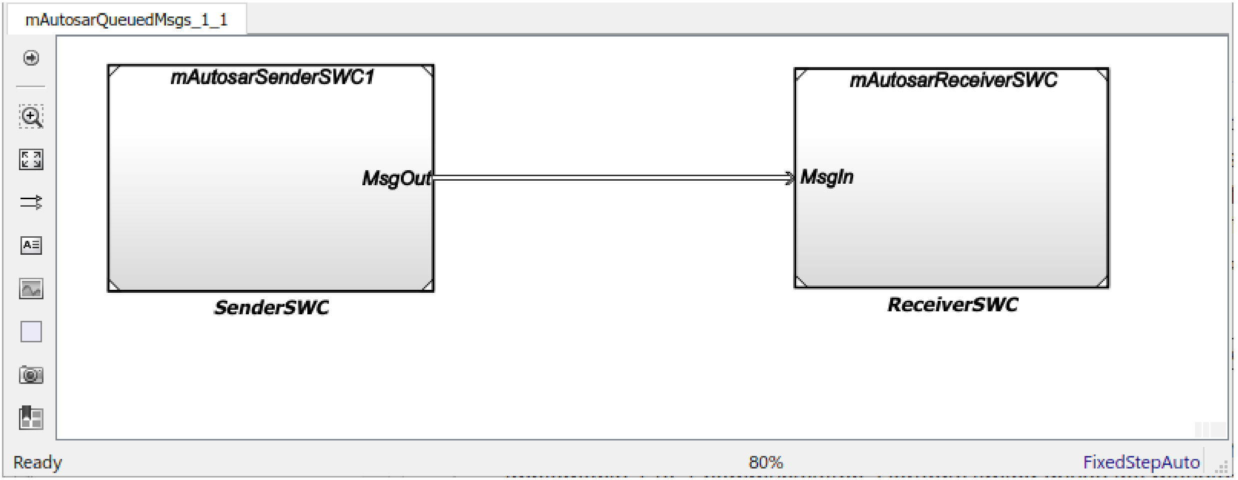
Configure Simulation of AUTOSAR Queued Sender-Receiver Communication

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 4-184 and “Simulate Event-Driven AUTOSAR Queued Sender-Receiver Communication” on page 4-188.

Here is a composition-level model that contains queued sender and receiver component models and implements 1-to-1 communication. Periodic timing drives the messaging. This example uses three models in the folder `matlabroot/help/toolbox/autosar/examples` (open). If you copy the files to a working folder, collocate the models.

- `mAutosarQueuedMsgs_1_1.slx` (top model)
- `mAutosarSenderSWC1.slx`
- `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 4-174 and implemented in “Implement AUTOSAR Queued Send and Receive Messaging” on page 4-177. Composition-level model mAutosarQueuedMsgs_1_1 includes them as referenced models and connects sender component port MsgOut to receiver component port MsgIn.

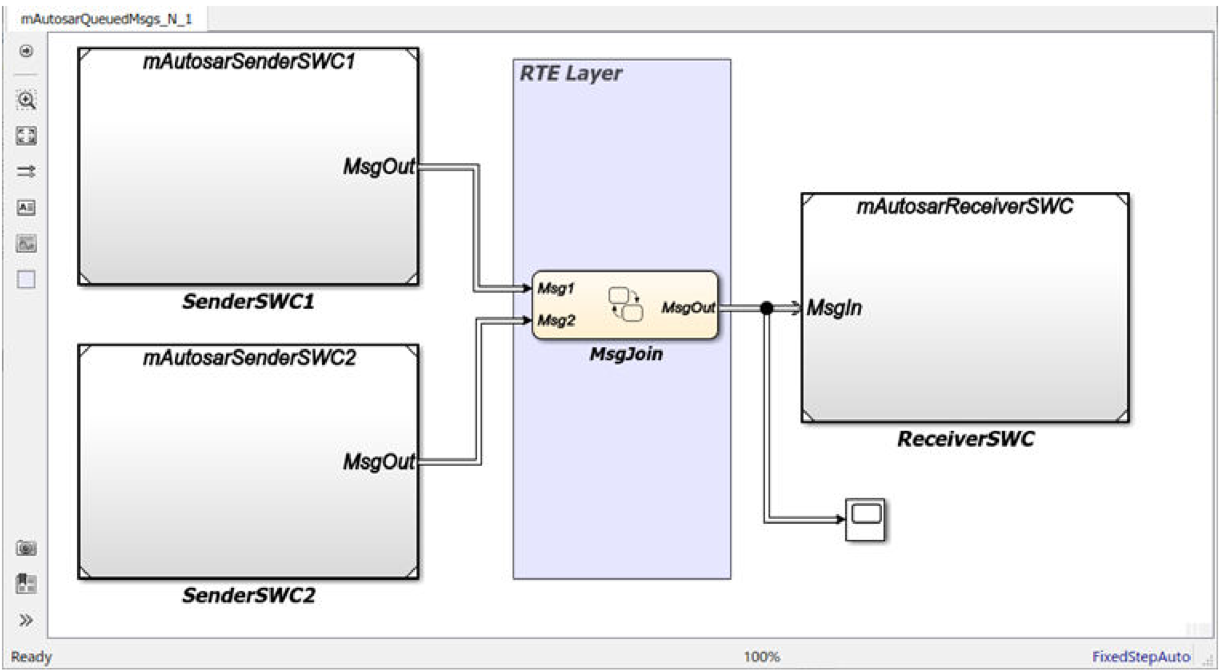
The top model mAutosarQueuedMsgs_1_1 is for simulation only. You can generate AUTOSAR C code and arxml files for the sender and receiver component models, but not for the containing composition-level model.

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

Here is a composition-level model that contains two sender and one receiver component models, and implements N-to-1 communication. Periodic timing drives the messaging. This 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 in the folder *matlabroot*/help/toolbox/autosar/examples (open). If you copy the files to a working folder, collocate the models.

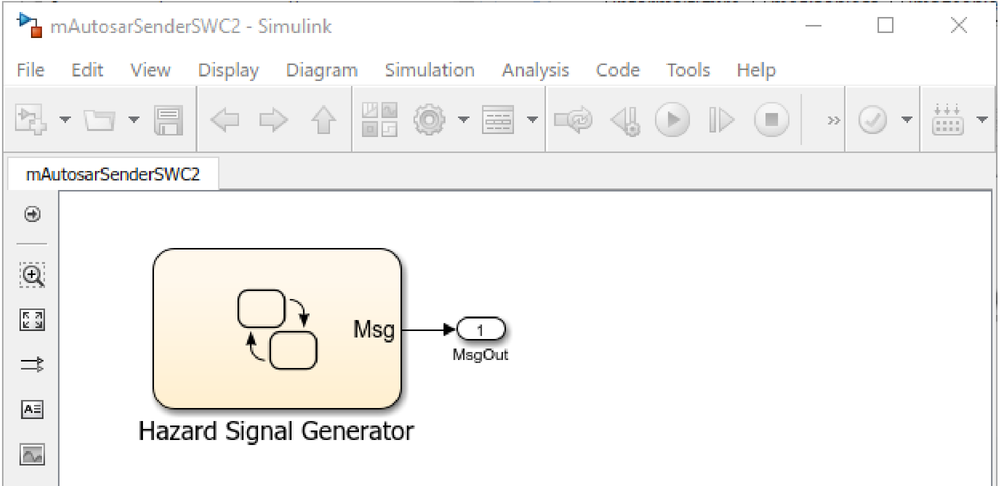
- `mAutosarQueuedMsgs_N_1.slx` (top model)
- `mAutosarSenderSWC1.slx`
- `mAutosarSenderSWC2.slx`
- `mAutosarReceiverSWC.slx`



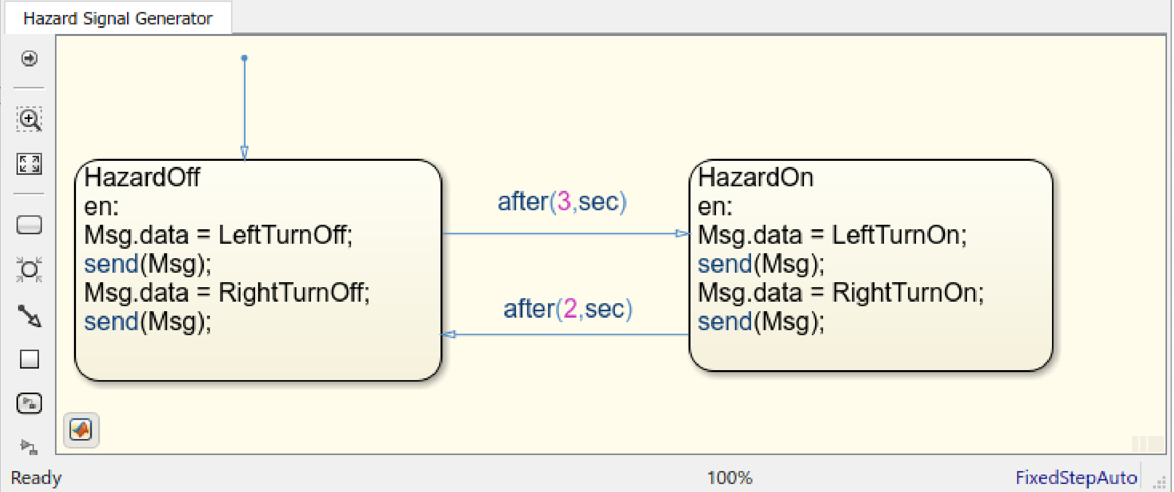
Composition-level model `mAutosarQueuedMsgs_N_1` includes two sender components and a receiver component as referenced models. It connects the sender component `MsgOut` ports to intermediate `MsgJoin` processing logic, which in turn connects to a receiver component `MsgIn` port.

Models `mAutosarSenderSWC1` and `mAutosarReceiverSWC` are the same sender and receiver components configured in “Configure AUTOSAR Sender and Receiver Components for Queued Communication” on page 4-174 and implemented in “Implement AUTOSAR Queued Send and Receive Messaging” on page 4-177. The second sender component, `mAutosarSenderSWC2`, is similar to `mAutosarSenderSWC1`, but implements a second type of message input for the receiver to process.

Here is 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 outport `MsgOut`.

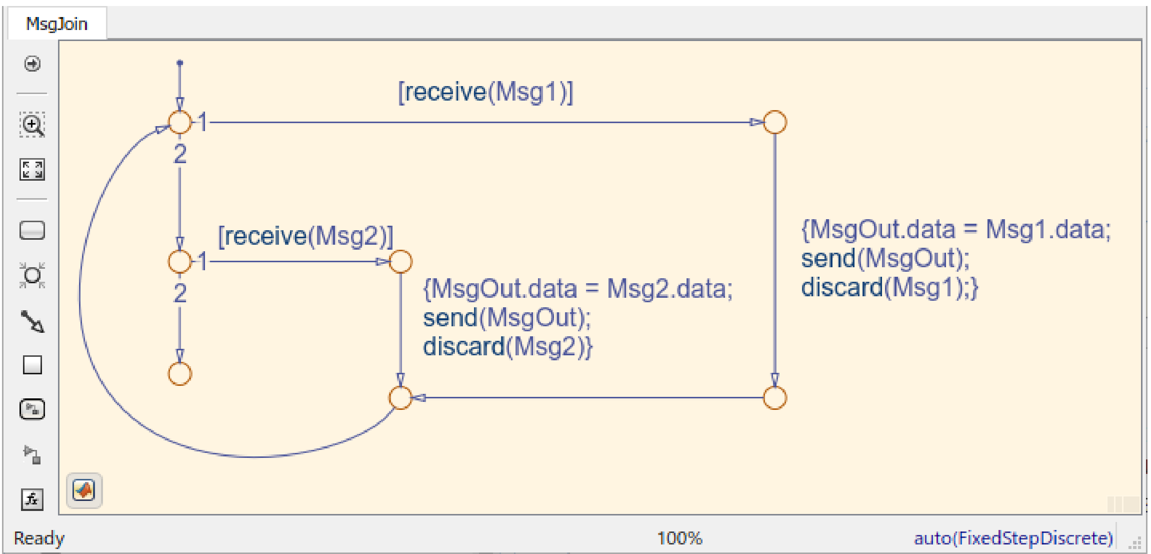


Here is the logic implemented in the `Hazard Signal Generator` chart. The chart has two states – `HazardOff` and `HazardOn`. Each state contains entry actions that assign values to message data and send messages. (See “Communicate with Stateflow Charts by Sending Messages” (Stateflow).) Periodic timing drives the message output.



Here is the `MsgJoin` chart located between the sender and receiver components.

Here is 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).)



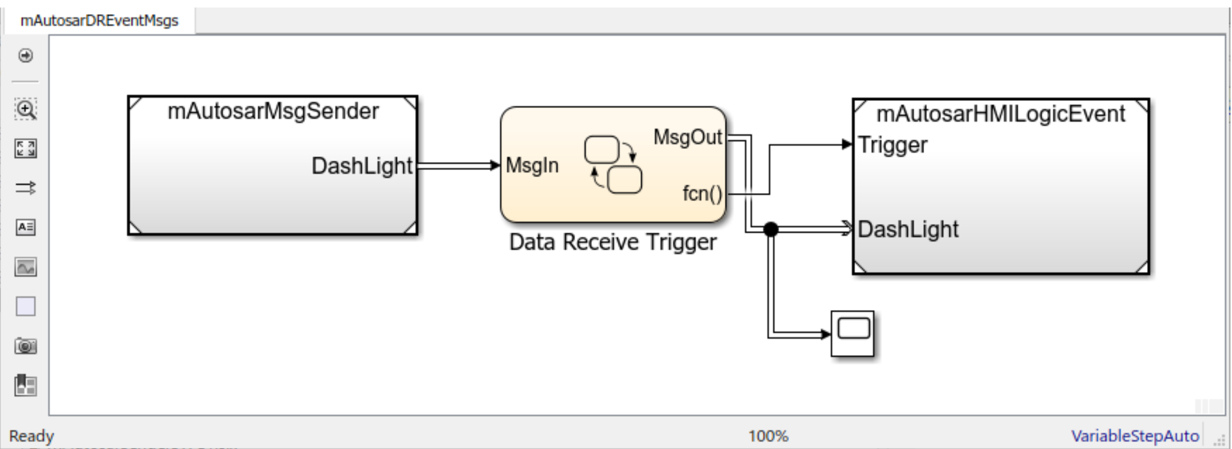
The top model `mAutosarQueuedMsgs_N_1` is for simulation only. You can generate AUTOSAR C code and arxml files for the referenced sender and receiver component models, but not for the containing composition-level model.

Similarly, you can run software-in-the-loop (SIL) simulation for the sender and receiver component models, but not for the composition-level model.

Simulate Event-Driven AUTOSAR Queued Sender-Receiver Communication

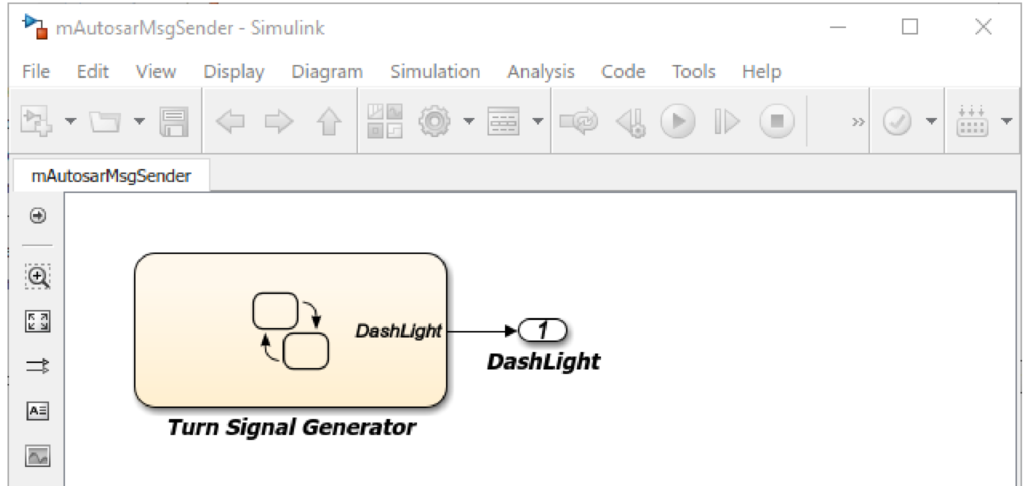
Here is a composition-level model in which a Simulink function-call input event activates receiver component processing of a queued message. This example uses three models in the folder *`matlabroot`*`/help/toolbox/autosar/examples` (open). If you copy the files to a working folder, collocate the models.

- `mAutosarDREventMsgs.slx` (top model)
- `mAutosarMsgSender.slx`
- `mAutosarHMILogicEvent.slx`

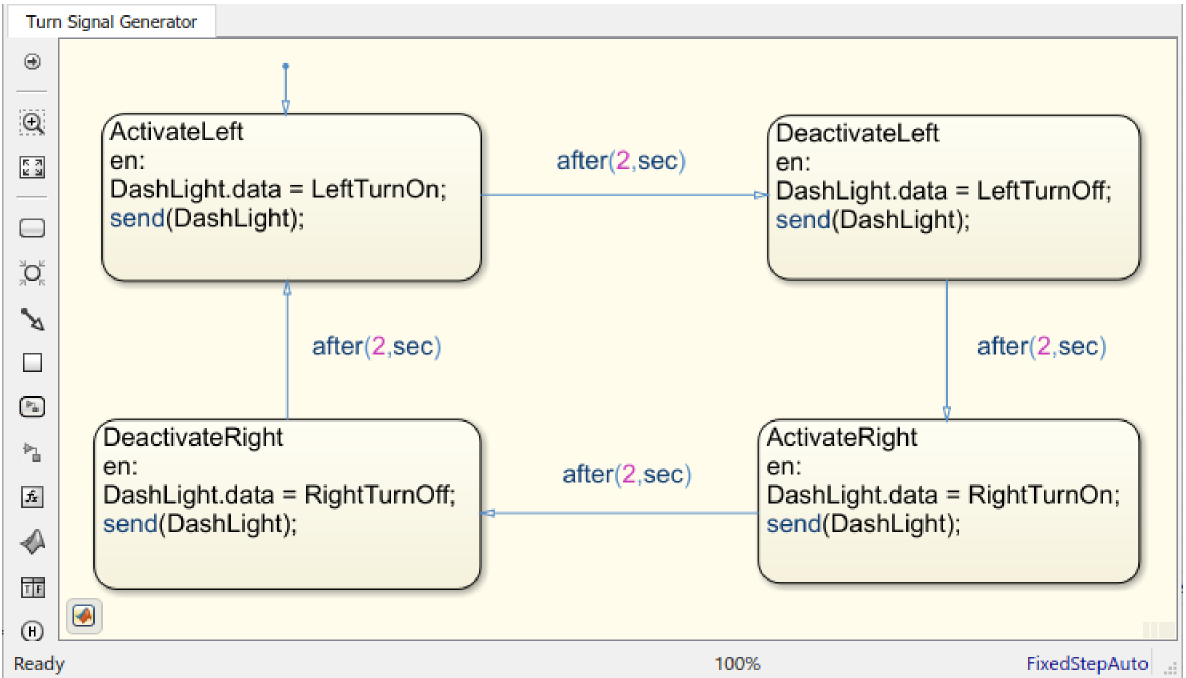


Composition-level model `mAutosarDREEventMsgs` includes a sender component and a receiver component as referenced models. It connects the sender message port `DashLight` to intermediate `Data Receive Trigger` logic, which in turn connects to receiver message port `MsgIn` and function trigger port `Trigger`.

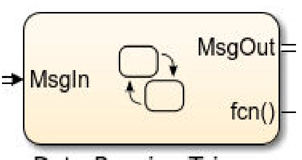
Here is the top level of AUTOSAR queued sender component `mAutosarMsgSender`, which contains Stateflow chart `Turn Signal Generator`. The chart message-line output is connected to Simulink root outport `DashLight`. (This sender component is similar to component `mAutosarSenderSWC1` in the 1-to-1 and N-to-1 simulation examples.)



Here is 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.



Here is the `Data Receiver Trigger` chart located between the sender and receiver components.

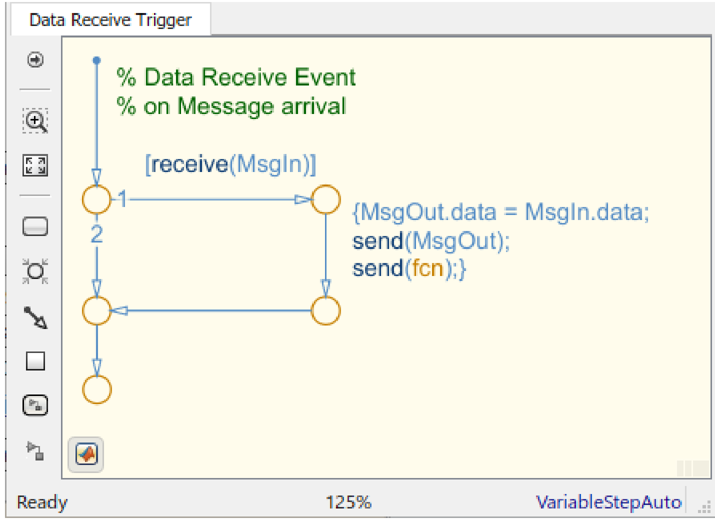


To receive a message, queued receiver logic uses `receive(M)`:

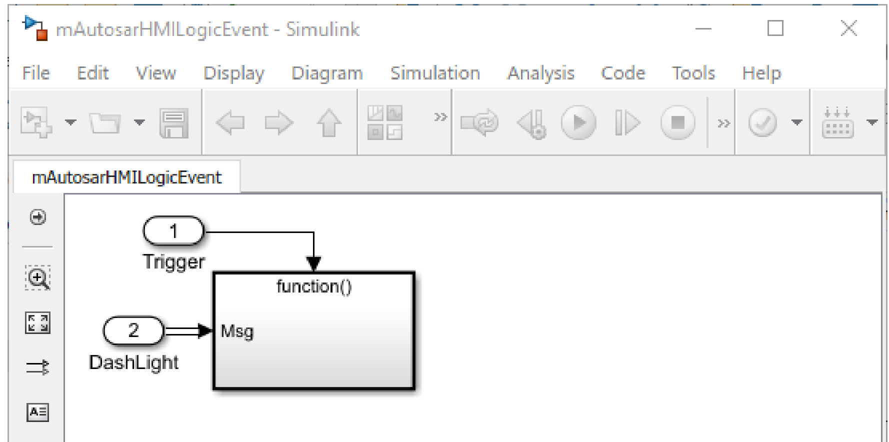
- If a valid message M exists, `receive(M)` returns true.
- If a valid message does not exist, the chart removes a message from its associated queue, and `receive(M)` returns true. If `receive(M)` removes a message from the queue, the length of the queue drops by one.
- If message M is invalid, and another 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).

Here is the logic implemented in the `Data Receiver Trigger` chart. The chart receives queued messages from the sender component. For each message received, the chart copies the received message data to the outbound message, sends the data, and sends a function-call event. (See “Communicate with Stateflow Charts by Sending Messages” (Stateflow).)

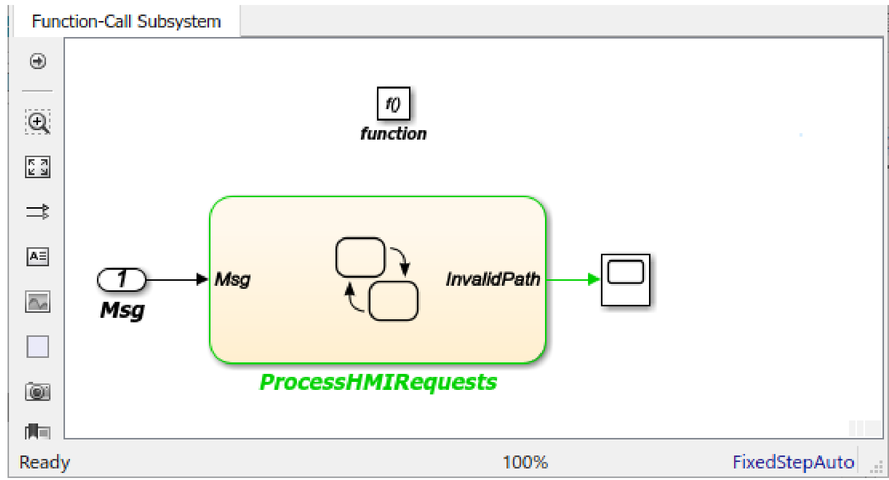


Here is the top level of AUTOSAR queued receiver component `mAutosarHMILogicEvent`, which contains a Simulink function-call subsystem. The subsystem inports are a function-call trigger and message receiver port `DashLight`, which is configured for AUTOSAR data access mode `QueuedExplicitReceive`.



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.

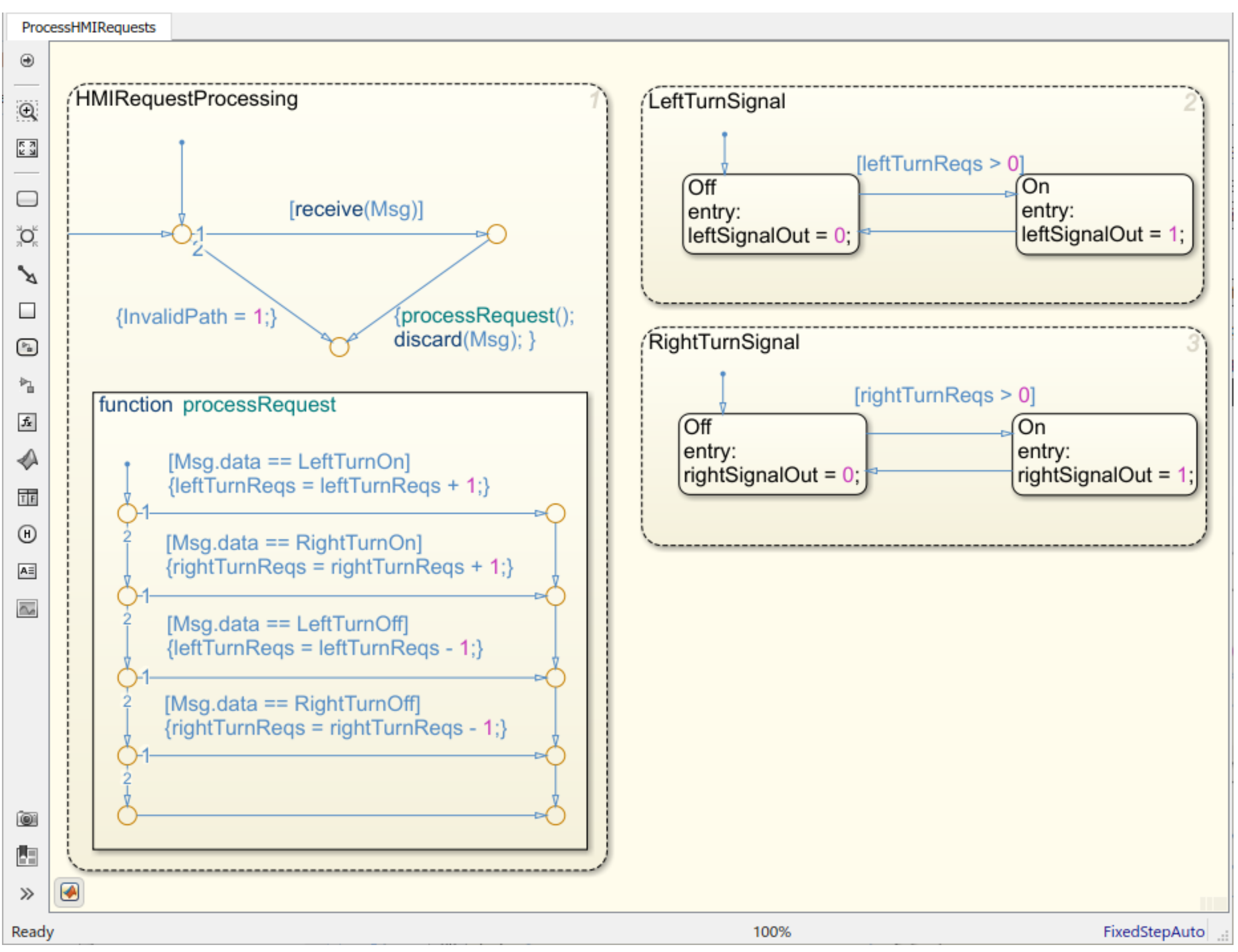
The Trigger Port block is configured for a function-call trigger and triggered sample time. Function-call input events sent from the `Data Receiver Trigger` chart in the top model activate the chart.



Here is 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.

See Also

Related Examples

- “Model an Assembly Line Feeder” (Stateflow)

More About

- “Messages” (Stateflow)
- “AUTOSAR Communication”
- “AUTOSAR Component Configuration” on page 4-3

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 2-28.

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.

In this section...

"Configure AUTOSAR Server" on page 4-197
"Configure AUTOSAR Client" on page 4-207
"Configure AUTOSAR Client-Server Error Handling" on page 4-216
"Concurrency Constraints for AUTOSAR Server Runnables" on page 4-221
"Configure and Map AUTOSAR Server and Client Programmatically" on page 4-223

Configure AUTOSAR Server

This example shows how to configure a Simulink Function block as an AUTOSAR server. The example uses these files in the folder *`matlabroot`*`/help/toolbox/autosar/examples` (open):

- `mControllerWithInterface_server.slx`
- `ExampleApplicationErrorType.m`

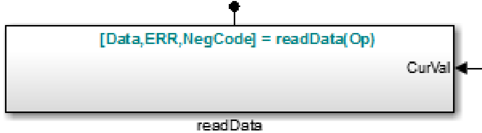
If you copy the files to a working folder, collocate the MATLAB file with the model file.

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. In the Simulink Library Browser, the Simulink Function block is in **User-Defined Functions**.

The example model provides two Simulink Function blocks, `doOverride` and `readData`.

3 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`.

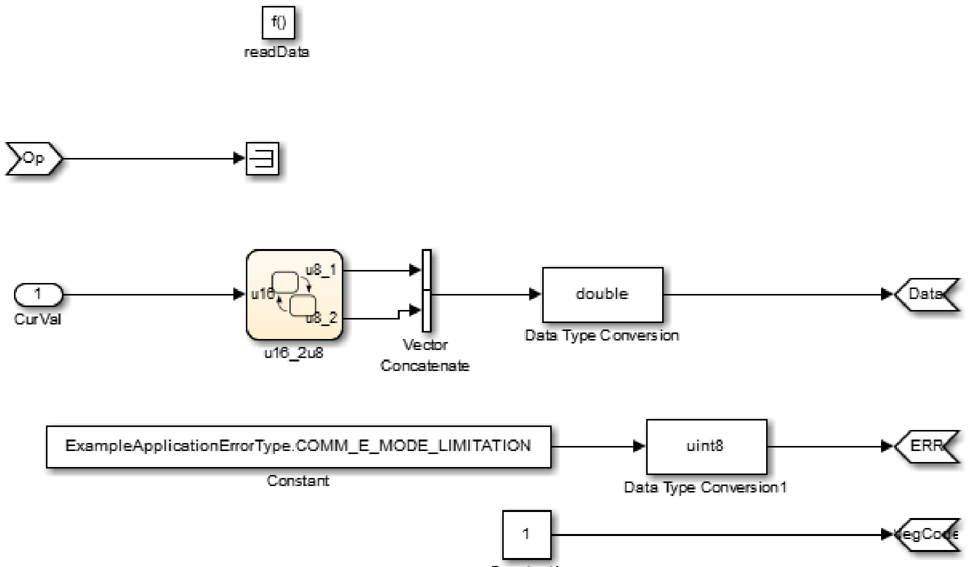


   The contents include:

- 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)`.

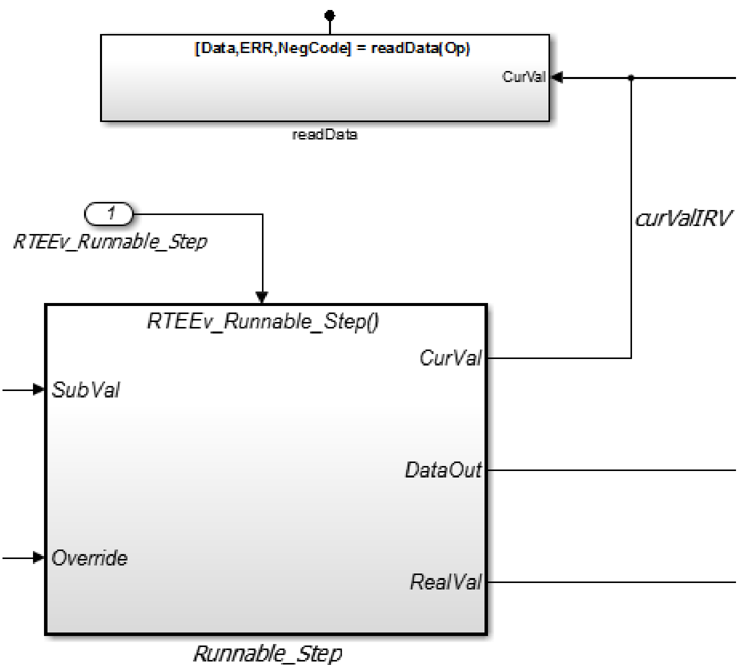
Note When configuring server function arguments, you must specify signal data type, port dimensions, and signal type on the **Signal Attributes** tab of the inport and outport blocks. The AUTOSAR configuration fails validation if signal attributes are absent for server function arguments.

- 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.

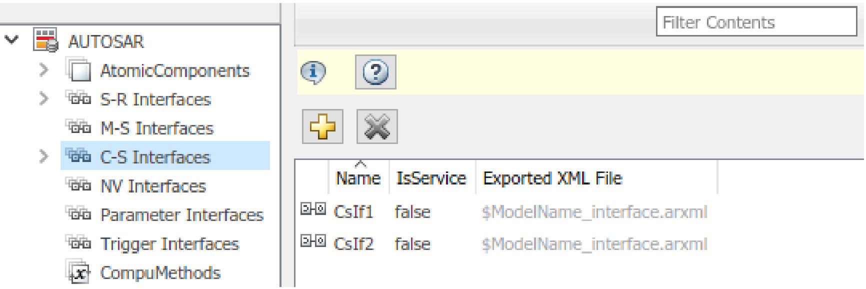
The example model is an AUTOSAR model, into which the Simulink Function block `readData` has been copied. In place of a meaningful `Op` input value for the `readData` function, Simulink data transfer line `CurVal` provides an input value that is used in the function algorithm.



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

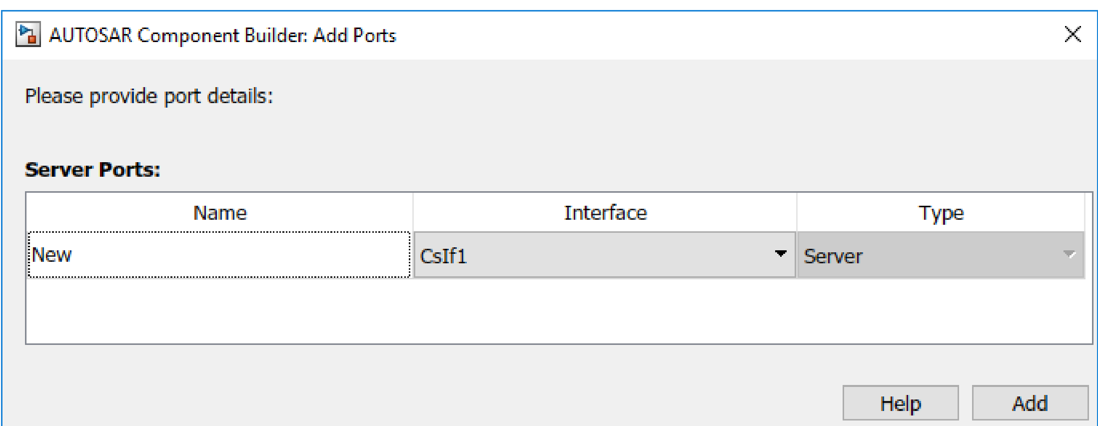
Open AUTOSAR Dictionary. To view AUTOSAR C-S interfaces in the model, go to the **C-S Interfaces** view. The example model already contains client-server interfaces.



If a C-S interface does not yet exist in your model, create one.

a In the C-S interfaces view, click the **Add** button. This action opens the Add Interfaces dialog box.

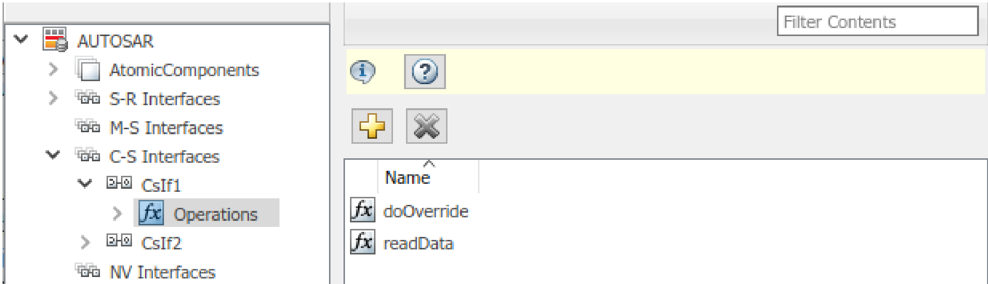
b In the dialog box, name the new C-S Interface, and specify the number of operations you intend to associate with the interface. Leave other parameters at their defaults. Click **Add**. The new interface appears in the C-S interfaces view.



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`.

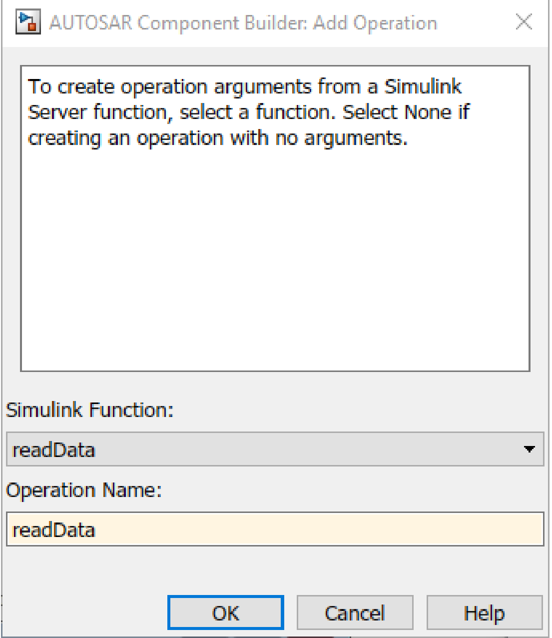


If a server operation does not yet exist in your model, create one. (If your C-S interface contains a placeholder operation named `Operation1`, you can safely delete it.)

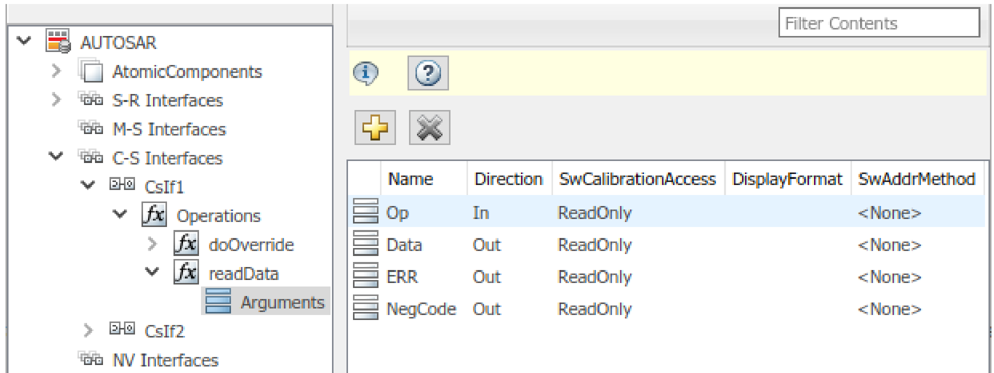
a In the operations view, click the **Add** button . This action opens the Add Operation dialog box.

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.

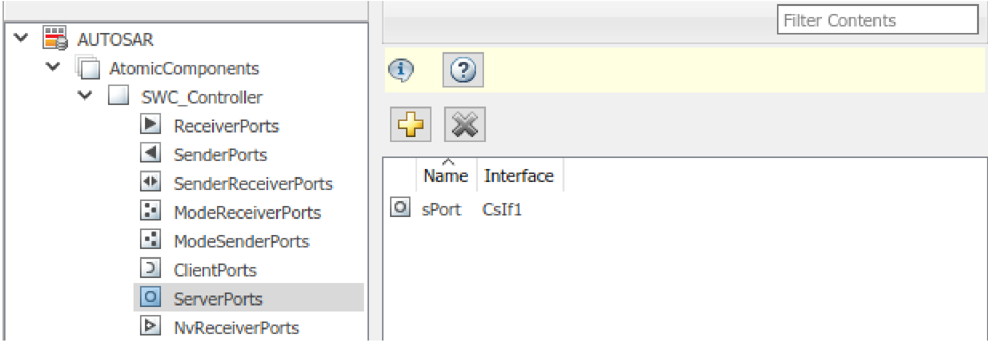


7 Examine the arguments listed for the AUTOSAR server operation. Expand **Operations**, expand the individual operation (for example, `readData`), and select **Arguments**. The listed arguments correspond to the Simulink server function prototype.



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

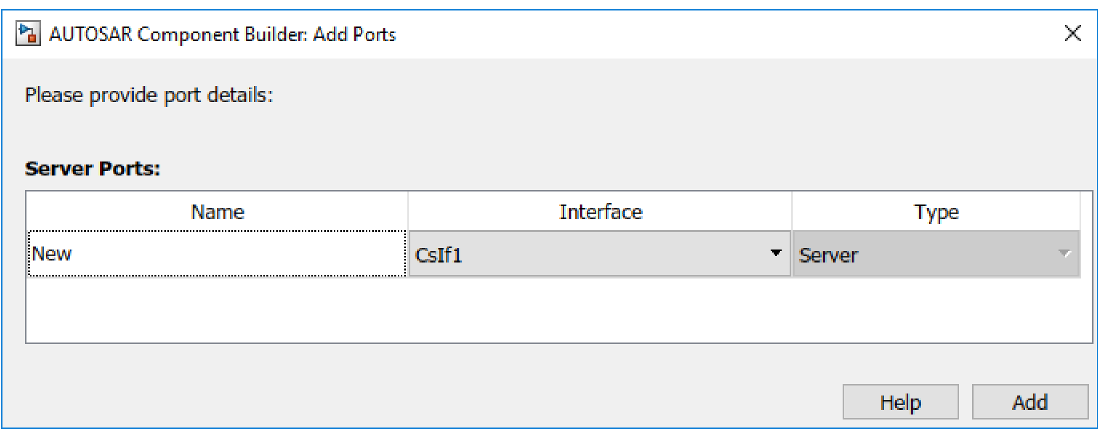
select **ServerPorts**. The example model already contains an AUTOSAR server port named `sPort`.



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. The **symbol** name specified for the runnable must match the Simulink server function name. (In the example model, the **symbol** name for `Runnable_readData` is the function name `readData`.)

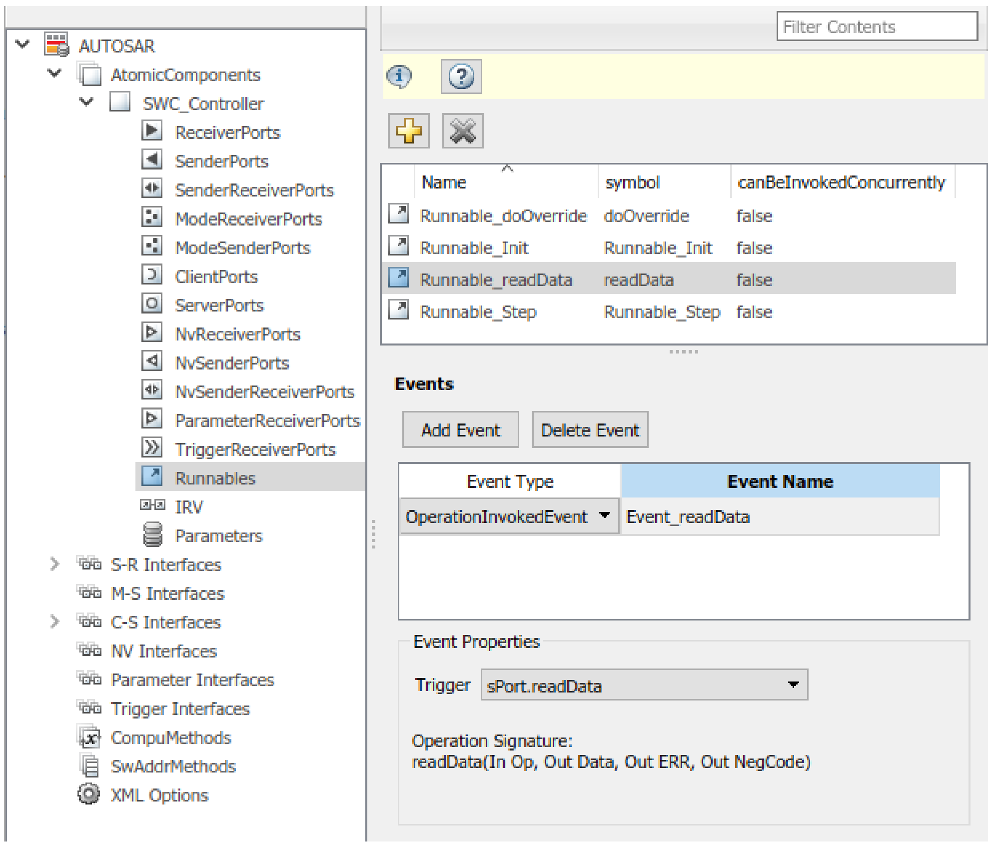
c Create an operation-invoked event to trigger the server runnable. (The example model defines event `event_readData` for server runnable `Runnable_readData`.)

  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**.

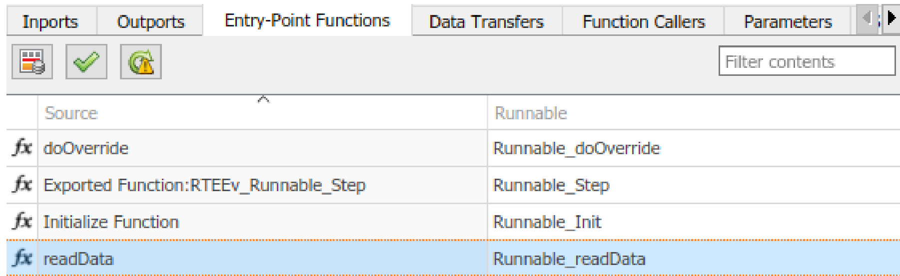


This step completes the configuration of an AUTOSAR server in the AUTOSAR Dictionary view of the configuration.

10 Switch to the Code Mappings editor view of the configuration and map the Simulink server function to the AUTOSAR server runnable.

a Open Code Mappings editor. Select the **Entry-Point 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`.



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 4-207.

Configure AUTOSAR Client

After you configure an AUTOSAR server, as described in "Configure AUTOSAR Server" on page 4-197, 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 file *`matlabroot`*`/help/toolbox/autosar/examples/mControllerWithInterface_client.slx`.

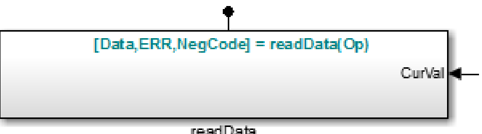
1 Open a model in which you want to create and configure an AUTOSAR client, or open the example model `mControllerWithInterface_client.slx`.

2 Add a Function Caller block to the model. In the Simulink Library Browser, the Function Caller block is in **User-Defined Functions**.

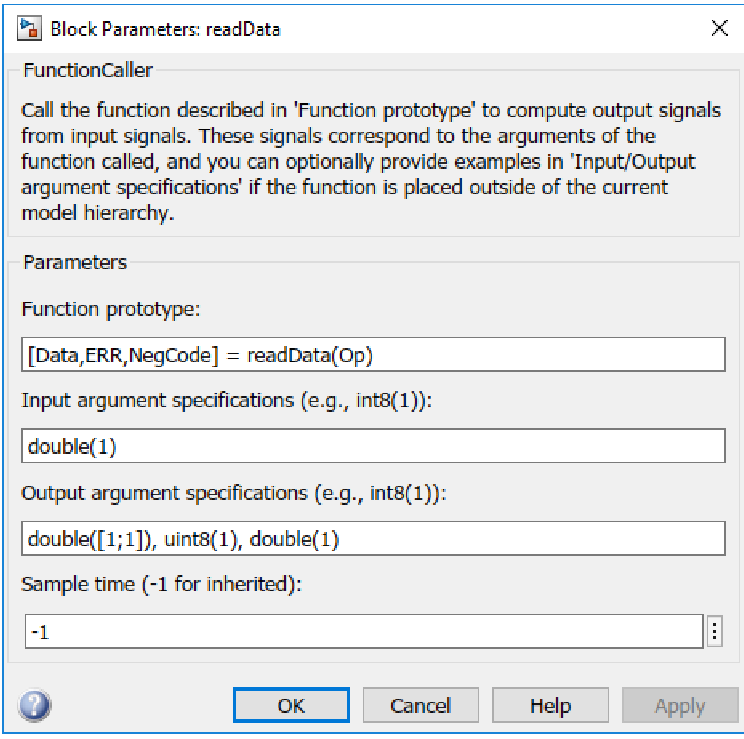
   The example model provides a Simulink Function block named `readData`, which is located inside `Runnable3_Subsystem`.

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 4-197. 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 4-197. 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.

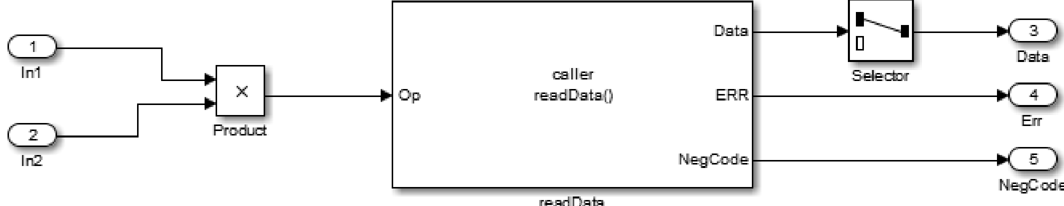
---

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, select **Create Default Component** rather than **Create Component Interactively**. 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.



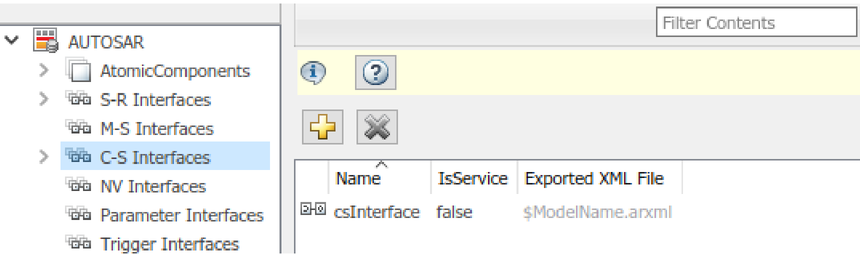
Note Whenever you add or change a Function Caller block in an AUTOSAR model, update function callers in the AUTOSAR configuration. Open Code Mappings editor (**Code** > **C/C++ Code** > **Configure Model in Code Perspective**). In the dialog box, click the **Update** button . This action loads or updates 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 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

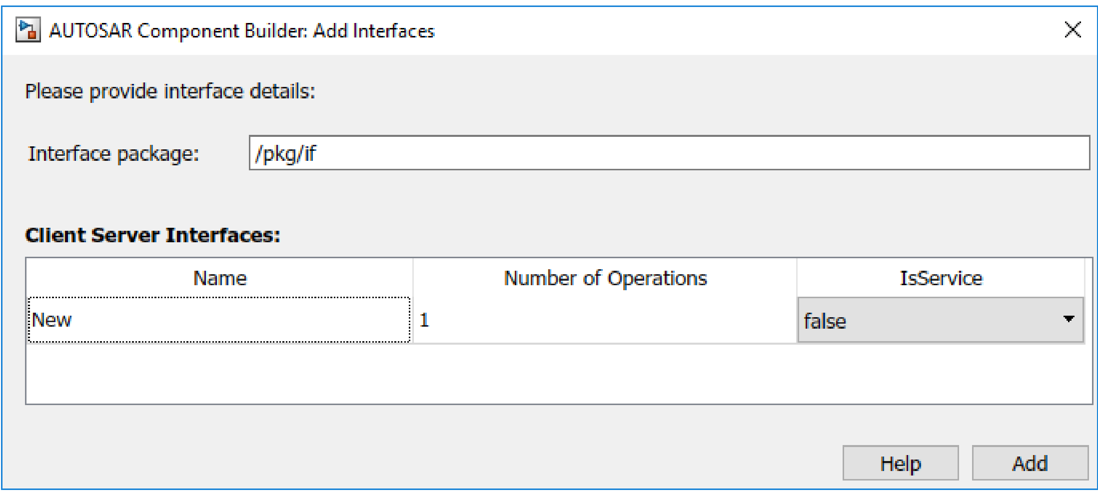
   Open AUTOSAR Dictionary. To view AUTOSAR C-S interfaces in the model, go to the **C-S Interfaces** view. The example model already contains a client-server interface named `csInterface`.



If a C-S interface does not yet exist in the AUTOSAR configuration, create one.

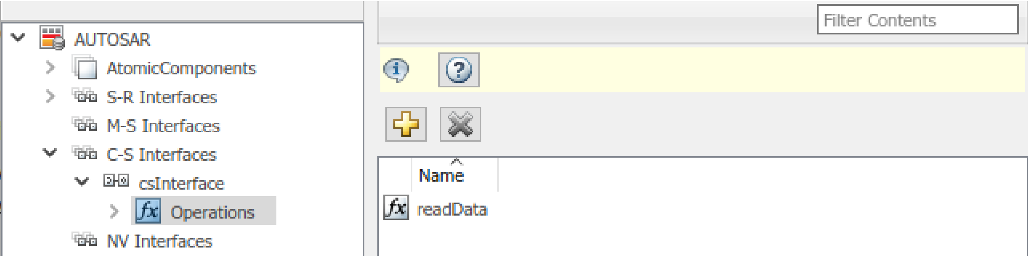
a In the C-S interfaces view, click the **Add** button. This action opens the Add Interfaces dialog box.

b 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.

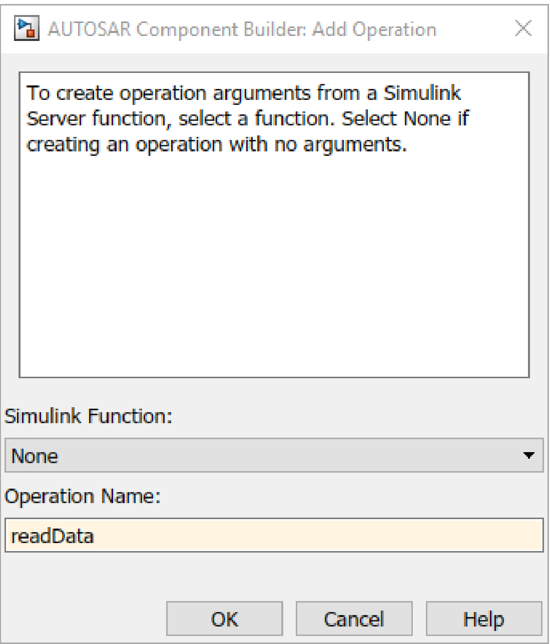
Expand **C-S Interfaces** and expand the individual C-S interface to which you want to add an AUTOSAR operation. (In the example model, expand `CsInterface`.) To view operations for the interface, select **Operations**. The example model already contains an AUTOSAR operation named `readData`.



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.

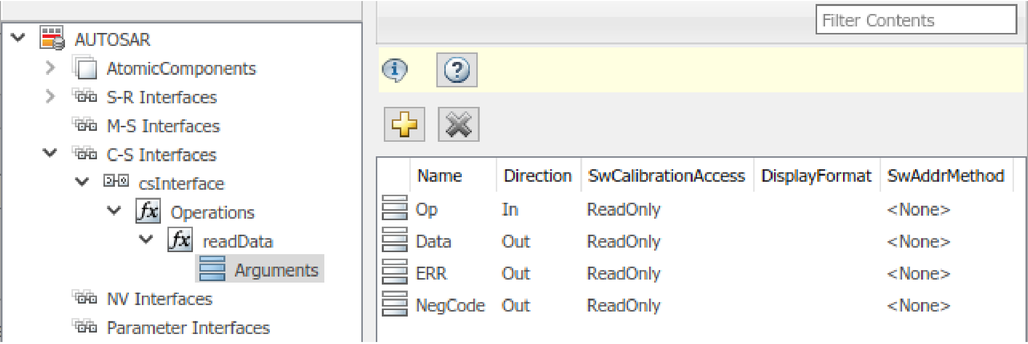


7 Add the AUTOSAR operation arguments.

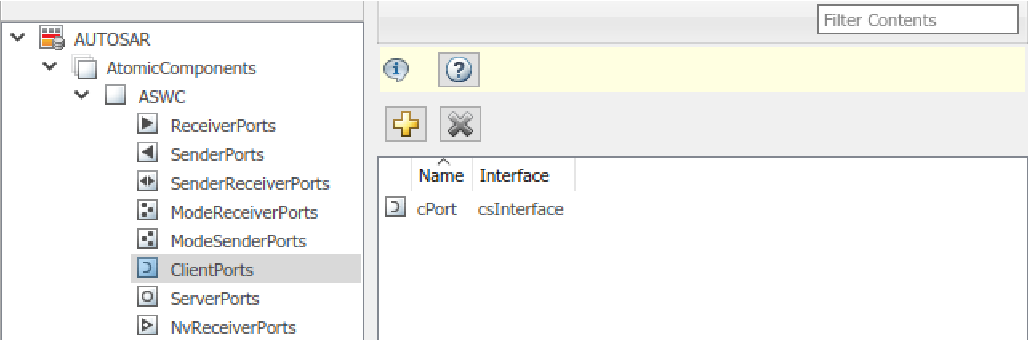
   a Expand **Operations**, expand the individual operation (for example, `readData`), and select **Arguments**.

   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.



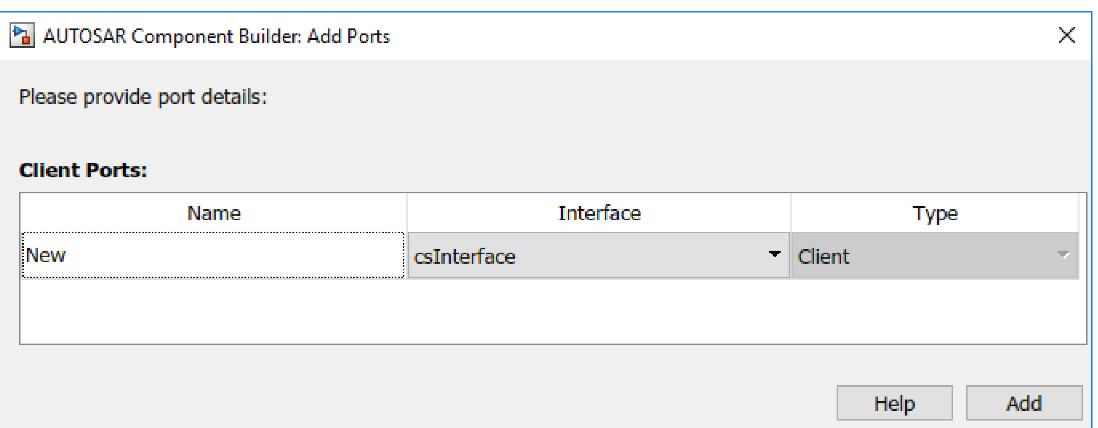
8 To view AUTOSAR client ports in the model, go to the client ports view. Expand **AtomicComponents**, expand the individual component that you are configuring, and select **ClientPorts**. The example model already contains an AUTOSAR client port named `cPort`.



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.

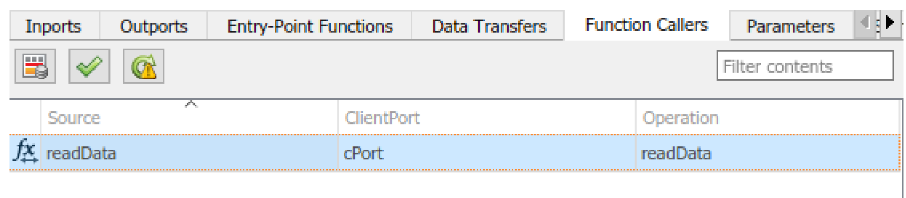


This step completes the configuration of an AUTOSAR client in the AUTOSAR Dictionary view of the configuration.

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 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:

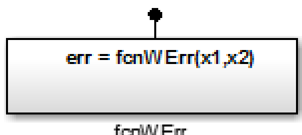
1. Define the possible error status values that the AUTOSAR server returns in a Simulink data type. Define one or more error codes in the range 0-63, inclusive. The underlying storage of the data type must be an unsigned 8-bit integer. The data scope must be `Exported`. For example, define an enumeration 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()
```

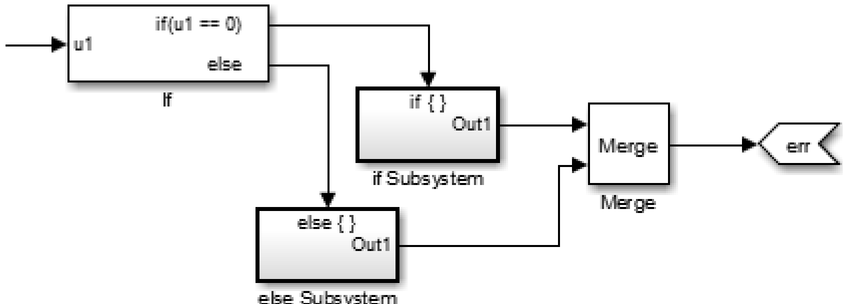
```
            retVal = false;
        end

        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.

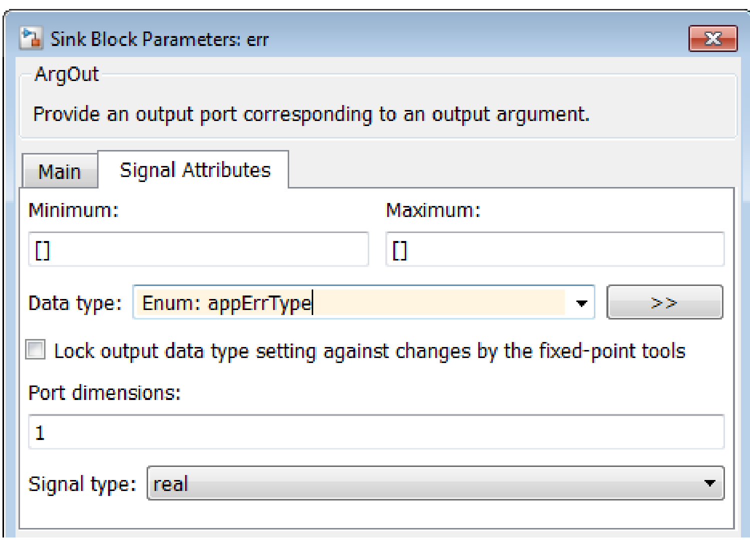
2 Define an error status output argument for the Simulink Function block that models the AUTOSAR server. Configure the error status argument as the only function output or add it to other outputs. For example, here is a Simulink Function block that returns an error status value in output `err`.



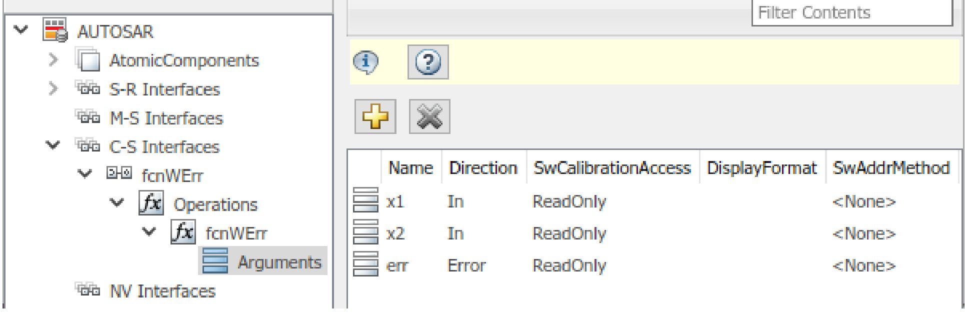
The Simulink Function block implements an algorithm to return error status.



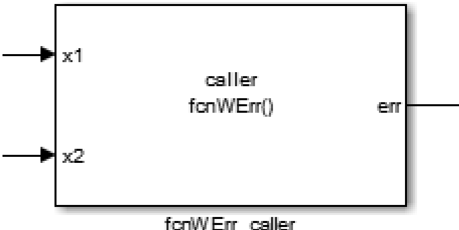
3 Reference the possible error values type in the model. In the Argument Outport block parameters for the error outport, specify the error status data type, in this case, `appErrType`. Set **Port dimensions** to 1 and **Signal type** to `real`.



4 Configure the AUTOSAR properties of the error argument in the client-server interface. Open 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.

Parameters

Function prototype:

err = fcnWErr(x1,x2)

Input argument specifications (e.g., int8(1)):

double(1), double(1)

Output argument specifications (e.g., int8(1)):

appErrType(1)

Configure the AUTOSAR properties of the error argument to match the information in 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`.

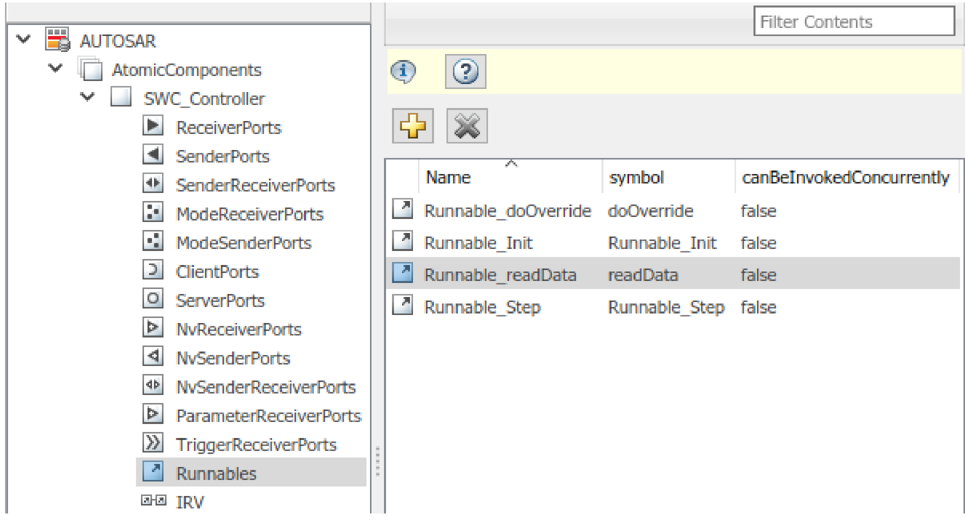
```
>> arProps = autosar.api.getAUTOSARProperties(bdroot)
>> set(arProps,'fcnWErr/fcnWErr/err','Direction','Error')
>> get(arProps,'fcnWErr/fcnWErr/err','Direction')
ans =
    Error
```

Concurrency Constraints for AUTOSAR Server Runnables

The following blocks and modeling patterns are incompatible with concurrent execution of an AUTOSAR server runnable.

- Blocks 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 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 the property `canBeInvokedConcurrently` to `true` only for an AUTOSAR server runnable — that is, a runnable with an `OperationInvokedEvent`. Runnables with other event triggers, such as timing events, cannot be concurrently invoked. 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 =
    'SWC_Controller/ControllerWithInterface_ar/Runnable_readData'
```

```
invConc = get(arProps,'SWC_Controller/ControllerWithInterface_ar/Runnable_readData',...
  'canBeInvokedConcurrently')
```

```
invConc =
     0

set(arProps,'SWC_Controller/ControllerWithInterface_ar/Runnable_readData',...
  'canBeInvokedConcurrently',true)
invConc = get(arProps,'SWC_Controller/ControllerWithInterface_ar/Runnable_readData',...
  'canBeInvokedConcurrently')

invConc =
     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 4-371.

See Also

Argument Inport | Argument Outport | Function Caller | Simulink Function | Trigger

Related Examples

- “Client-Server Interface” on page 2-28
- “Configure AUTOSAR Client-Server Interfaces” on page 4-371
- “Import AUTOSAR Software Component” on page 3-27
- “Configure AUTOSAR Code Generation” on page 5-12

More About

- “Model AUTOSAR Communication” on page 2-26
- “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.

In this section...
"Configure Mode Receiver Port and Mode-Switch Event for Mode User" on page 4-225
"Configure Mode Sender Port and Mode Switch Point for Application Mode Manager" on page 4-230

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.

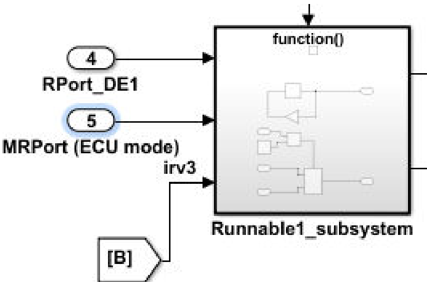
---

1. Open the example model `autosar_swc_expfcns`. Save a copy to a writable work folder.
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:

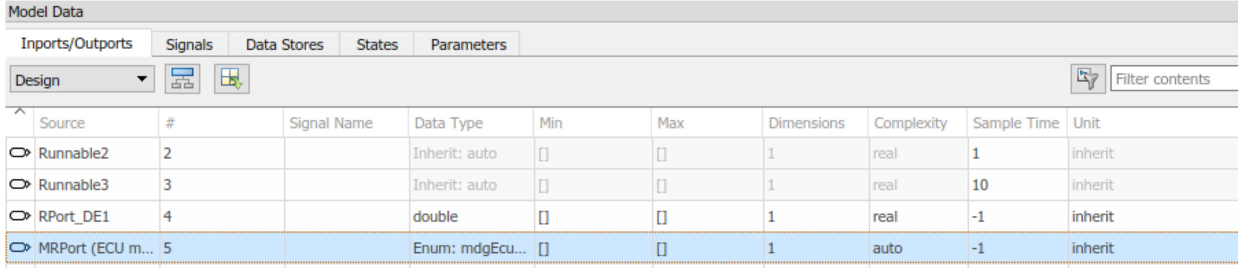
```
Simulink.defineIntEnumType('mdgEcuModes', ...
 {'Run', 'Sleep'}, [0;1], ...
 'Description', 'Mode declaration group for ECU modes', ...
 'DefaultValue', 'Run', ...
```

```
'HeaderFile', 'Rte_Type.h', ...
'AddClassNameToEnumNames', false,...
'StorageType', 'uint16');
```

3. Rename the Simulink inport `RPort_DE1 (ErrorStatus)` to `MRPort (ECU mode)`. For example, in the model window, open the Model Data Editor (**View** > **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`.

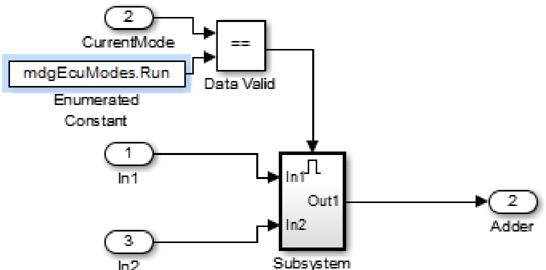


Source	#	Signal Name	Data Type	Min	Max	Dimensions	Complexity	Sample Time	Unit
Runnable2	2		Inherit: auto	[]	[]	1	real	1	inherit
Runnable3	3		Inherit: auto	[]	[]	1	real	10	inherit
RPort_DE1	4		double	[]	[]	1	real	-1	inherit
MRPort (ECU m...	5		Enum: mdgEcu...	[]	[]	1	auto	-1	inherit

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**.

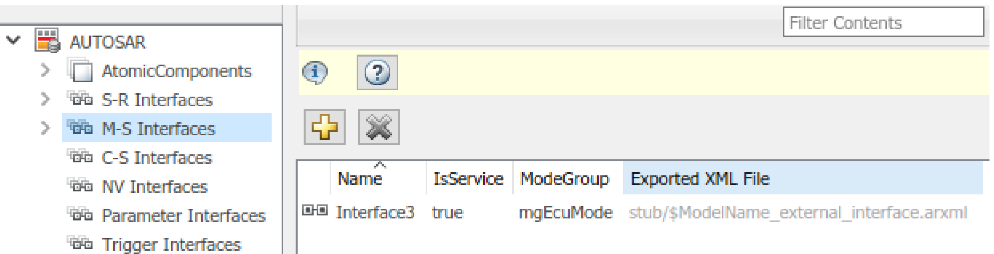


6 Add an AUTOSAR mode-switch interface to the model. Open AUTOSAR Dictionary.

Select **M-S Interfaces**. Click the **Add** button. In the Add Interfaces dialog box, specify **Name** as `Interface3` and specify **ModeGroup** as `mgEcuMode`.

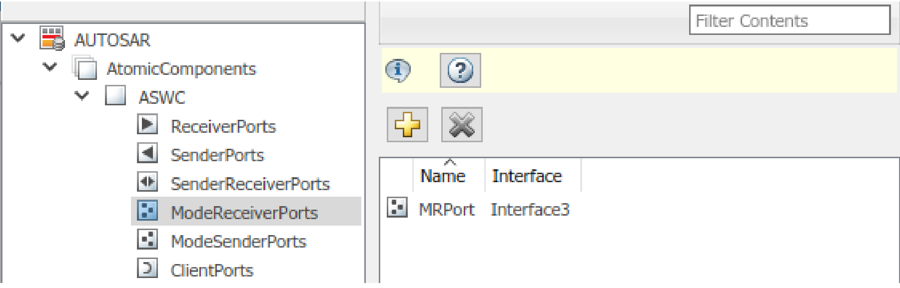
The **IsService** property of an M-S interface defaults to `true`. For the purposes of this example, you can leave **IsService** at its default setting, unless you have a reason to change it.

Click **Add**.

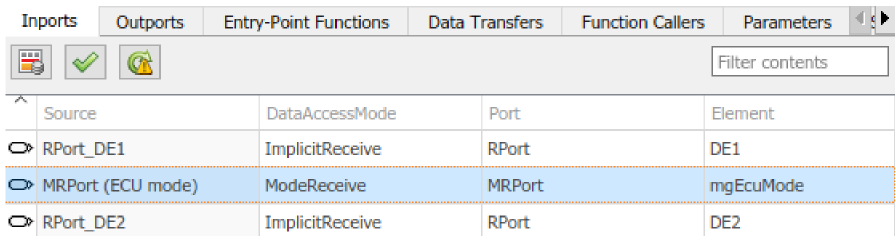


The value you specify for the AUTOSAR mode group is used in a later step, when you map a Simulink inport to an AUTOSAR mode-receiver port and element.

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 Code Mappings editor, map the Simulink inport `MRPort (ECU mode)` to the AUTOSAR mode-receiver port and element. Open 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`.)



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.

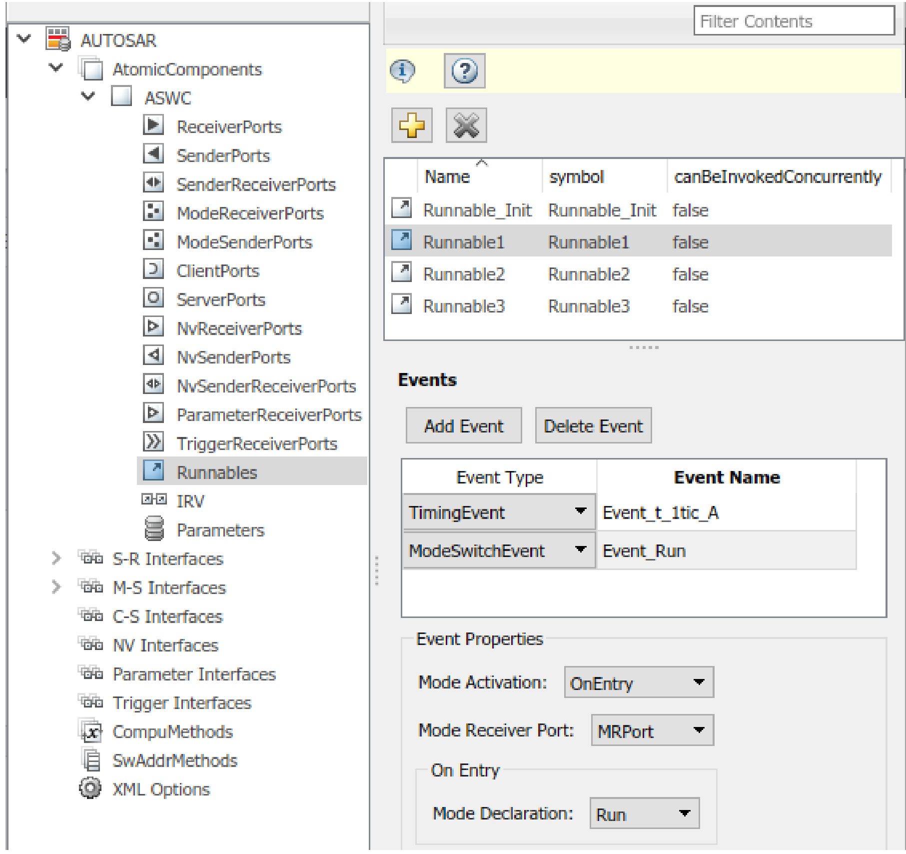
---

9 To add an AUTOSAR mode-switch event for a runnable:

a Open 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.

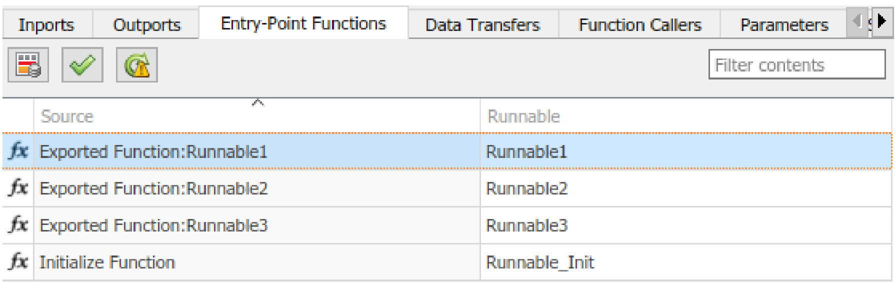
b To add an event to the list of events for `Runnable1`, click **Add Event**. For the new event, set **Event Type** to `ModeSwitchEvent`. (This activates an **Event Properties** subpane.) Specify **Event Name** as `Event_Run`.

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 Code Mappings editor and select the **Entry-Point 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.



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.

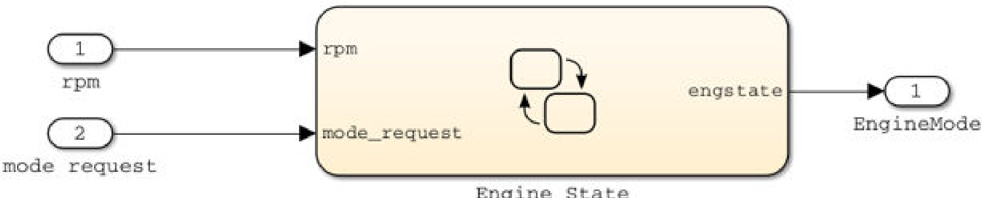
Configure Mode Sender Port and Mode Switch Point for Application Mode Manager

To model an application mode manager software component, use an AUTOSAR mode sender port (as defined in AUTOSAR Release 4). Mode sender ports use a mode-switch (M-S) interface to output a mode switch to connected mode user components.

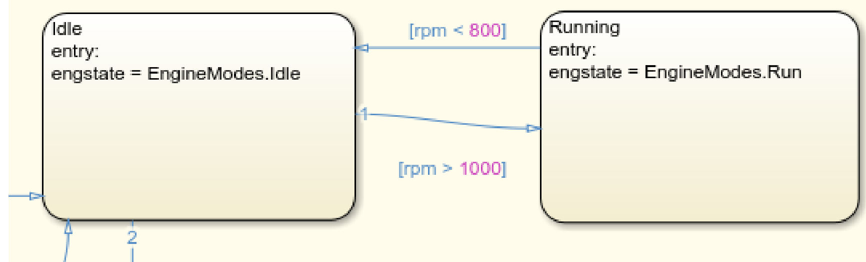
You model the mode sender port as a model root outport, which is mapped to an AUTOSAR mode sender port and a mode-switch (M-S) interface. The outport data type is an enumeration class with an unsigned integer storage type, representing an AUTOSAR mode declaration group.

This example shows how to configure a mode sender port and related elements for an application mode manager.

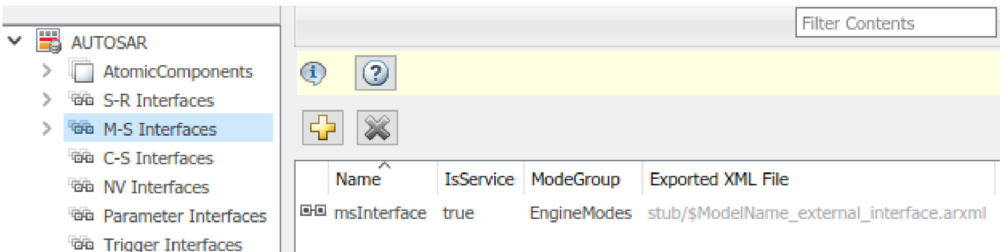
1 Open a model configured for AUTOSAR code generation. This example uses a model that contains Stateflow logic for maintaining engine state. The model outputs the current engine mode value.



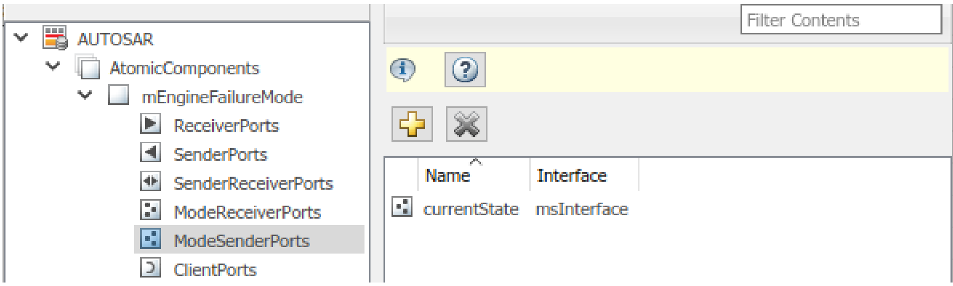
2 Declare a mode declaration group — a group of mode values. You can declare mode values with Simulink enumeration. In this example, the Stateflow logic defines `EngineModes` values `Off`, `Crank`, `Stall`, `Idle`, and `Run`. For example:



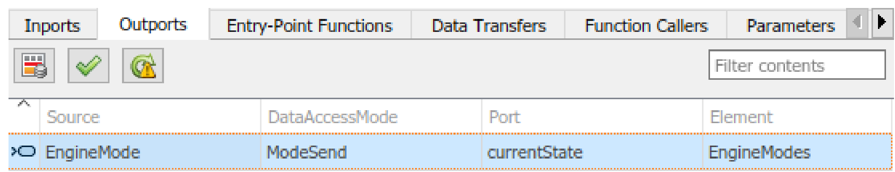
3 Add an AUTOSAR M-S interface to the model. Open AUTOSAR Dictionary and select **M-S Interfaces**. Click the **Add** button. In the Add Interfaces dialog box, set **isService** to `true` and enter a **ModeGroup** name. In this example, the mode declaration group is `EngineModes`.



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.



5 Map the Simulink outport that outputs the mode value to the AUTOSAR mode sender port you created. Open Code Mappings editor and select the **Outports** tab. To map the outport 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.



6 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

- “Mode-Switch Interface” on page 2-30
- “Configure AUTOSAR Mode-Switch Interfaces” on page 4-374
- “Import AUTOSAR Software Component” on page 3-27
- “Configure AUTOSAR Code Generation” on page 5-12

More About

- “Model AUTOSAR Communication” on page 2-26
- “AUTOSAR Component Configuration” on page 4-3

Configure AUTOSAR Nonvolatile Data Communication

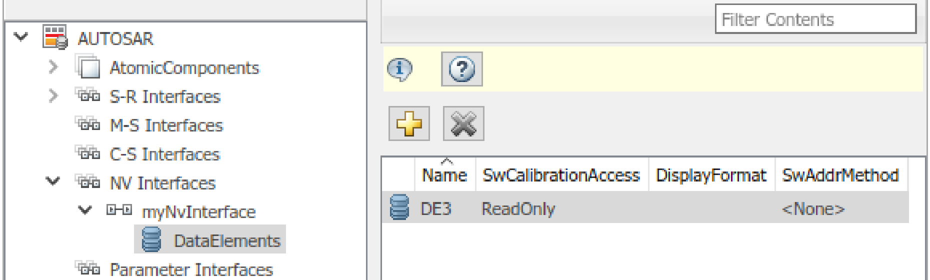
AUTOSAR Release 4.0 introduced port-based nonvolatile (NV) data communication, in which an AUTOSAR software component reads and writes data to AUTOSAR nonvolatile components. To implement NV data communication, AUTOSAR software components define provide and require ports that send and receive NV data.

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 2-27.

To create an NV data interface and ports in Simulink:

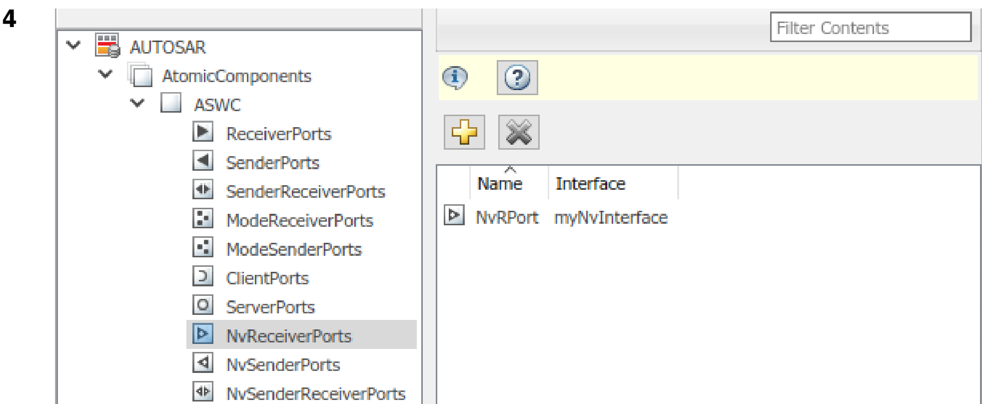
1 Add an AUTOSAR NV interface to the model. Open AUTOSAR Dictionary and select **NV Interfaces**. Click the **Add** button . In the Add Interfaces dialog box, specify the interface name and the number of associated NV data elements.

2 Select and expand the new NV interface. Select **DataElements** and modify the data element attributes.

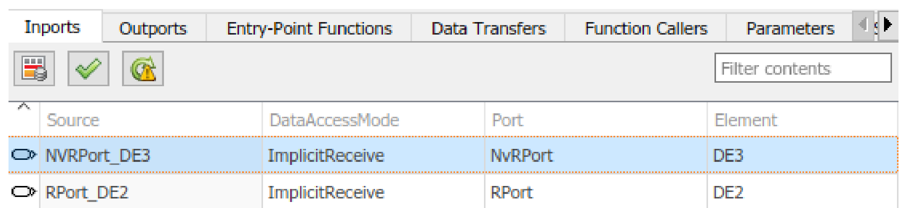


3 Add AUTOSAR NV ports to the model. Expand **AtomicComponents** and expand the component. Select and use the **NvReceiverPorts**, **NvSenderPorts**, and **NvSenderReceiverPorts** views to add the NV ports you require. For each NV port, select the NV interface you created.

4



5 Map Simulink inports and outports to the AUTOSAR NV ports you created. Open Code Mappings editor. Select and use the **Inports** and **Outports** tabs to map the ports. For each inport or outport, select an AUTOSAR port, data element, and data access mode.



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 2-35
- “Configure AUTOSAR Sender-Receiver Interfaces” on page 4-369
- “Import AUTOSAR Software Component” on page 3-27
- “Configure AUTOSAR Code Generation” on page 5-12

More About

- “Model AUTOSAR Communication” on page 2-26
- “AUTOSAR Component Configuration” on page 4-3

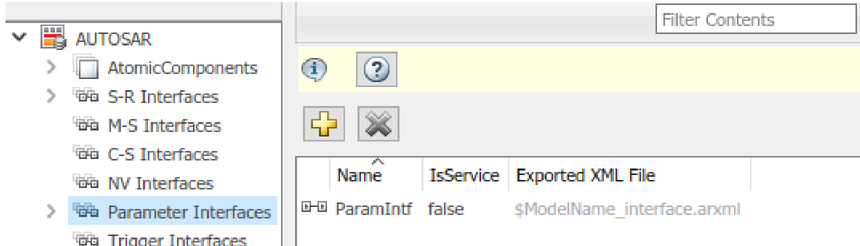
Configure Receiver for AUTOSAR Parameter Communication

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 parameter software component represents memory containing AUTOSAR parameters and provides parameter data to connected atomic software components.

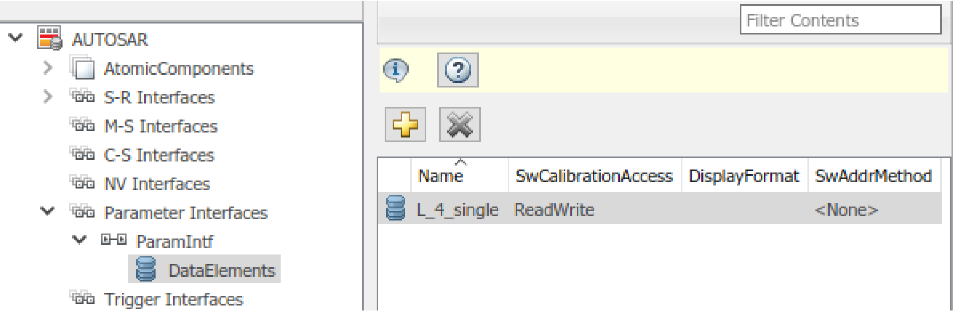
In Simulink, you can model the receiver portion of AUTOSAR port-based parameter communication. In an AUTOSAR atomic software component, you create a parameter interface with data elements and a parameter receiver port.

This example shows how to configure an AUTOSAR software component as a receiver for parameter communication.

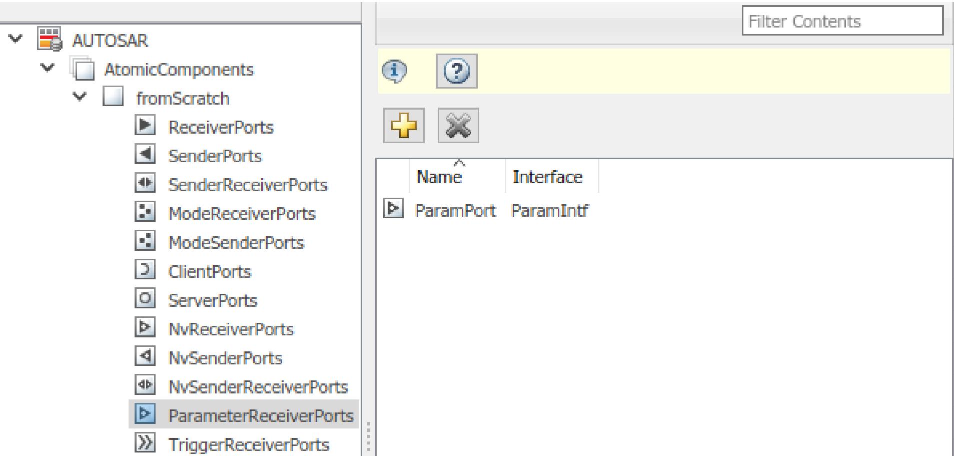
1. Open a model configured for AUTOSAR code generation in which the software component requires port-based access to parameter data.
2. Open AUTOSAR Dictionary. Select the **Parameter Interfaces** view and use the **Add** button to add a parameter interface to the model. In the Add Interfaces dialog box, specify the name of the new interface and set **Number of Data Elements** to 1.



3. Expand **Parameter Interfaces** and select the **DataElements** view. Examine and modify the properties of the associated data element that you created.



4 Expand **AtomicComponents** and expand the component. Go to the **ParameterReceiverPorts** view and use the **Add** button to add a parameter 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 parameter interface you created.



5 The AUTOSAR parameter interface data elements that you create then are available for lookup table mapping, using AUTOSAR function `mapLookupTable`. This code maps Simulink lookup table `L_4_single` to an AUTOSAR parameter receiver port and a parameter interface data element, which is accessed as a port-based parameter.

```
hModel = 'mySWC';
open_system(hModel)
slMap=autosar.api.getSimulinkMapping(hModel);
mapLookupTable(slMap,'L_4_single','PortParameter',...
  'ParamPort','L_4_single');
```

```
[arParameterAccessMode,arPortName,arParameterData] = ...
  getLookupTable(slMap,'L_4_single')

arParameterAccessmode =
  PortParameter

arPortName =
  ParamPort

arParameterData =
  L_4_single
```

See Also

`getLookupTable` | `mapLookupTable`

Related Examples

- "Create AUTOSAR Calibration Parameters By Using AUTOSAR Dictionary"

More About

- "Model AUTOSAR Communication" on page 2-26

Configure Receiver for AUTOSAR External Trigger Event Communication

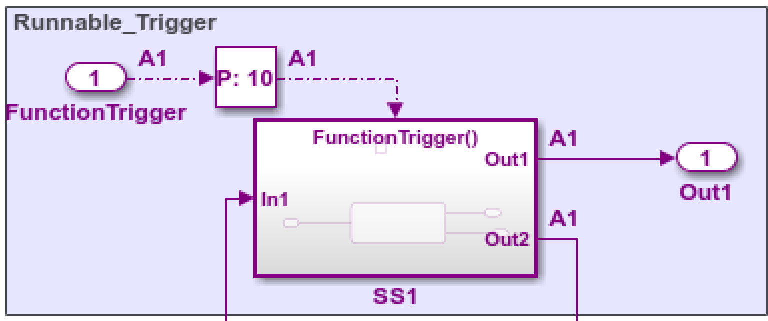
AUTOSAR Release 4.0 introduced external trigger event communication, in which an AUTOSAR software component or service signals an external trigger occurred event (`ExternalTriggerOccurredEvent`) to another component. The receiving component activates a runnable in response to the event.

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.

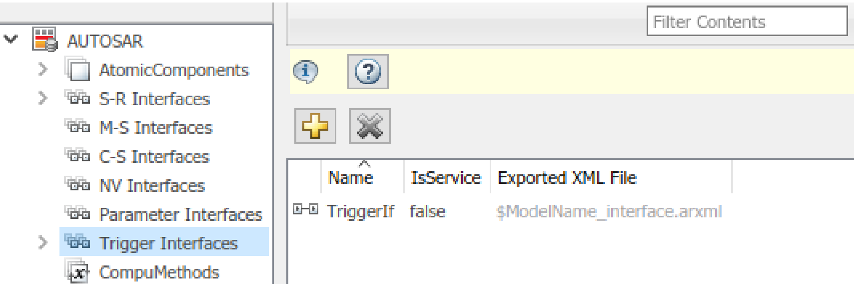
This example shows how to configure an AUTOSAR software component as a receiver for external trigger event communication.

1 Open a model configured for AUTOSAR code generation, in which you want to activate a runnable based on receiving an AUTOSAR `ExternalTriggerOccurredEvent`.

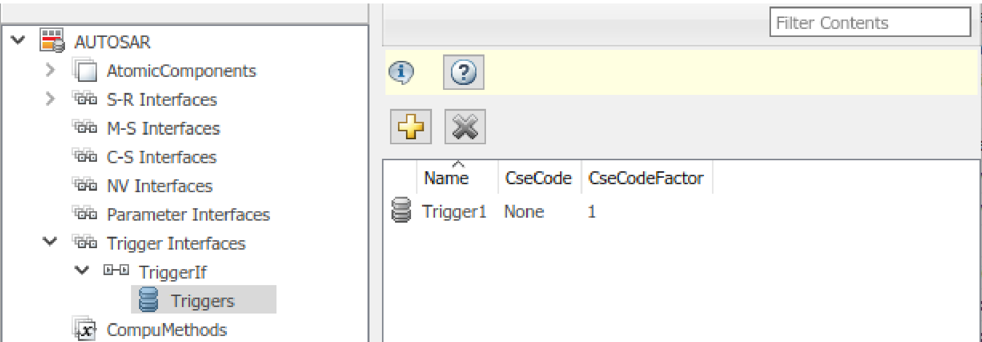
   For a sample model that uses external trigger event communication, see `autosar_swc_fcncalls`. In `autosar_swc_fcncalls`, asynchronous function-call subsystem SS1 models an AUTOSAR runnable. An `ExternalTriggerOccurredEvent` activates the runnable.



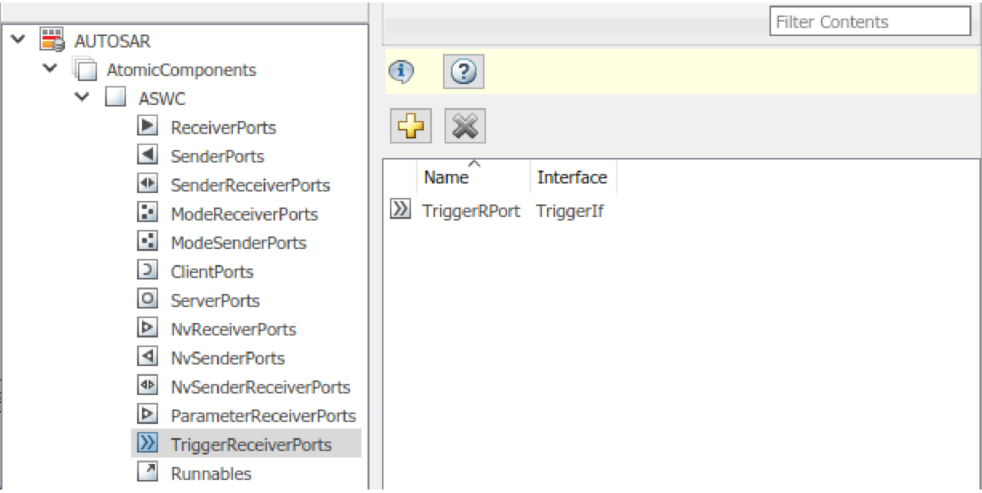
2 Open AUTOSAR Dictionary. Select the **Trigger Interfaces** view and use the **Add** button to add a trigger interface to the model. In the Add Interfaces dialog box, specify the name of the new interface and set **Number of Triggers** to 1.



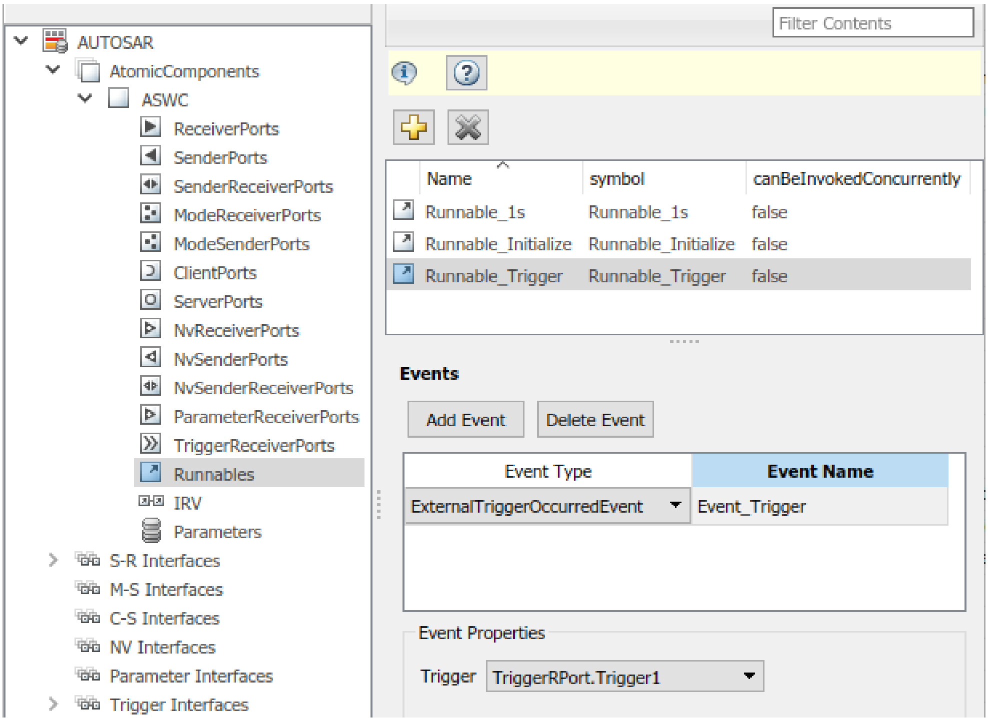
3 Expand **Trigger Interfaces** and select the **Triggers** view. Examine the properties of the associated trigger. For an asynchronous (nonperiodic) trigger, set **CseCode** to `None`, indicating an unspecified trigger period. For more information about specifying trigger periods, click the help button in the triggers view.



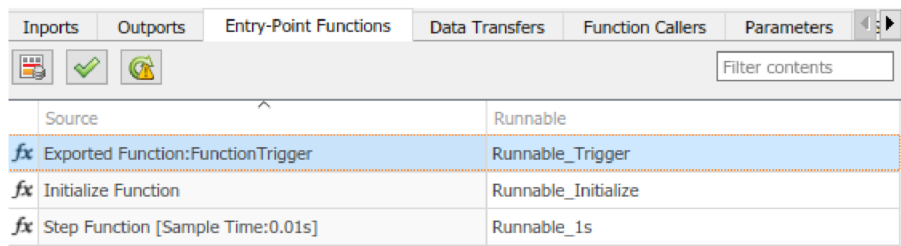
4 Expand **AtomicComponents** and expand the component. Select the **TriggerReceiverPorts** view and use the **Add** button to add a trigger receiver port to the model. In the Add Ports dialog box, specify the name of the new port and set **Interface** to the name of the trigger interface you created.



5 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.



6 To complete the trigger receiver configuration, open Code Mappings editor and select the **Entry-Point 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 4-288
- "Import AUTOSAR Software Component" on page 3-27
- "Configure AUTOSAR Code Generation" on page 5-12

More About

- "Model AUTOSAR Communication" on page 2-26
- "AUTOSAR Component Configuration" on page 4-3

Configure 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.

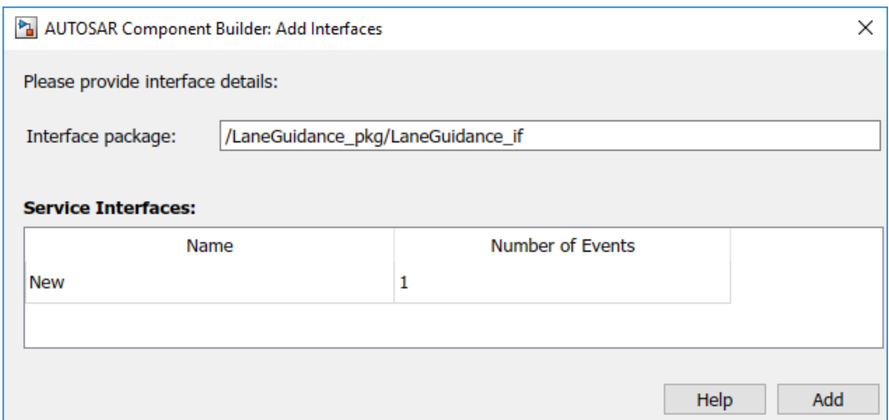
To model adaptive service communication in Simulink, you can:

- Create AUTOSAR required and provided ports, service interfaces, service interface events, and C++ namepaces.
- Create root-level inports and outports and map them to AUTOSAR required and provided ports and service interface events.

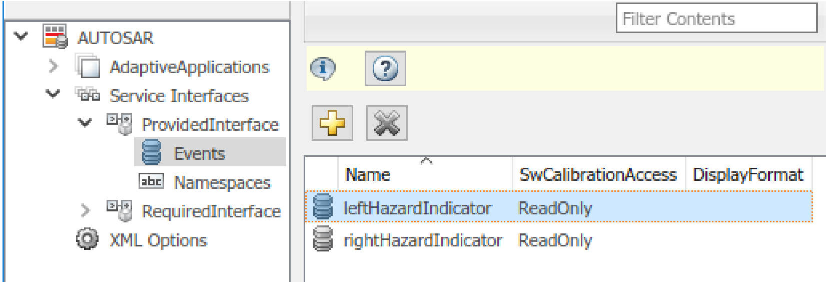
If you are licensed for Simulink Coder and Embedded Coder, you can generate C++ code and `arxml` descriptions for AUTOSAR service communication.

To implement adaptive service communication in Simulink:

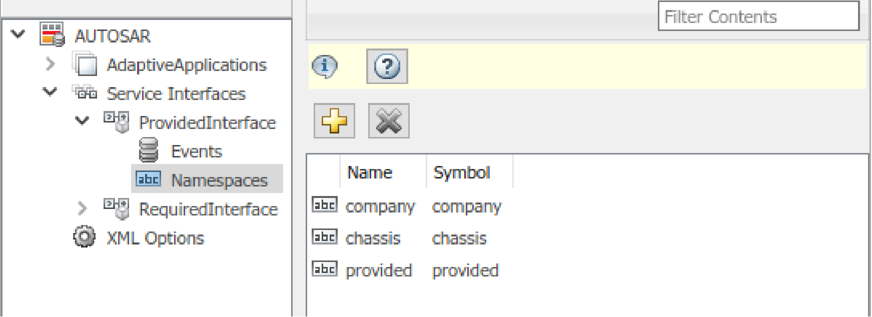
1. Open a model configured for the AUTOSAR Adaptive Platform. Displays in this example use model `autosar_LaneGuidance`.
2. Open AUTOSAR Dictionary and select **Service Interfaces**. To create an AUTOSAR service interface, click the **Add** button . In the Add Interfaces dialog box, specify the interface name and the number of associated events.



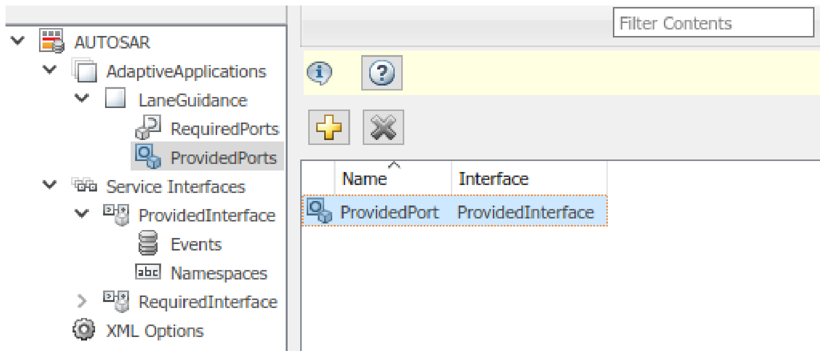
3 Expand the **Service Interfaces** node. Expand the new service interface and select **Events**. In the events view, select each service event and configure its attributes.



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, the namespaces view below defines namespace `company::chassis::provided` for service interface `ProvidedInterface`.

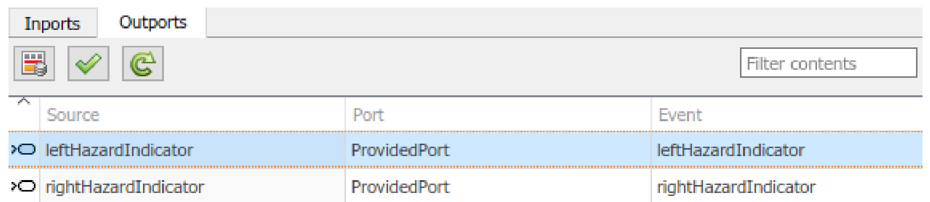


5 At the top level of AUTOSAR Dictionary, expand **AdaptiveApplications** and expand the adaptive software component. Use the **RequiredPorts** and **ProvidedPorts** views to add AUTOSAR required and provided ports that you want to associate with the new service interface. For each new service port, select the service interface you created.



6 In the model widow, to model AUTOSAR adaptive service ports, create root-level inports and outports.

7 Open 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 a service interface event.



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';
open_system(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');
```

See Also

Related Examples

- “Service Interface (Adaptive Platform)” on page 2-36
- “Configure AUTOSAR Adaptive Elements and Properties” on page 4-87
- “Map AUTOSAR Adaptive Elements for Code Generation” on page 4-103

More About

- “Model AUTOSAR Communication” on page 2-26
- “AUTOSAR Component Configuration” on page 4-3

Configure Lookup Tables for AUTOSAR Measurement and Calibration

In Simulink, you can implement standard axis (STD_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 measurement and calibration 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.

In this section...

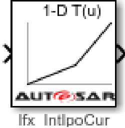
"Configure STD_AXIS Lookup Tables By Using Lookup Table Objects" on page 4-250
"Configure COM_AXIS Lookup Tables By Using Lookup Table and Breakpoint Objects" on page 4-255

Configure STD_AXIS Lookup Tables By Using Lookup Table Objects

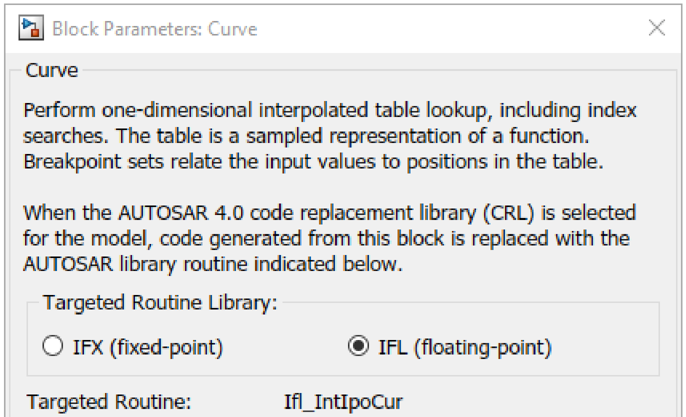
This example shows how to create STD_AXIS lookup tables in Simulink, using `Simulink.LookupTable` objects, and configure the lookup tables for AUTOSAR code generation. The example uses the model *`matlabroot`*`/help/toolbox/autosar/examples/mAutosarLutObjs.slx`.

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.

Simulink.LookupTable: L_4_single

Number of table dimensions: 1

Table

Value	Data type	Dimensions	Min	Max	Unit	Field name	Description
[10 20 30 40]	single	[1 4]	[ ]	[ ]		Table	

Breakpoints

Specification: Explicit values ☑ Support tunable size

	Value	Data type	Dimensions	Min	Max	Unit	Field name	Tunable size name	Description
1	[1 2 3 4]	single	[1 4]	[ ]	[ ]		Bp1	Nx	

☐ Argument

Code generation options

Data definition

Storage class: Auto

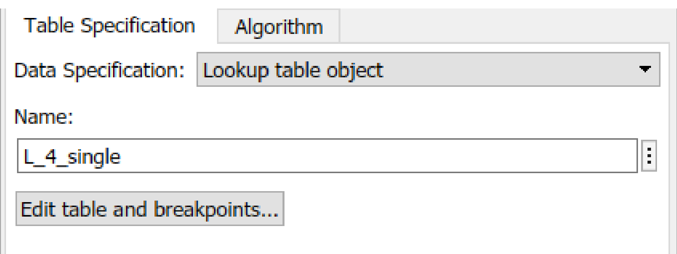
Struct type definition

Name: LUT_4_single

Data scope: Auto

Header file:

d In the Curve block dialog box, **Table Specification** tab, enter the `Simulink.LookupTable` object name in the **Name** field.

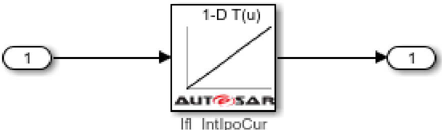


e In the block dialog box, **Algorithm** tab, set **Integer Rounding Method** to `Zero`. Leave **Interpolation Method** set to `Linear point-slope` and **Index Search Method** set to `Linear search`.

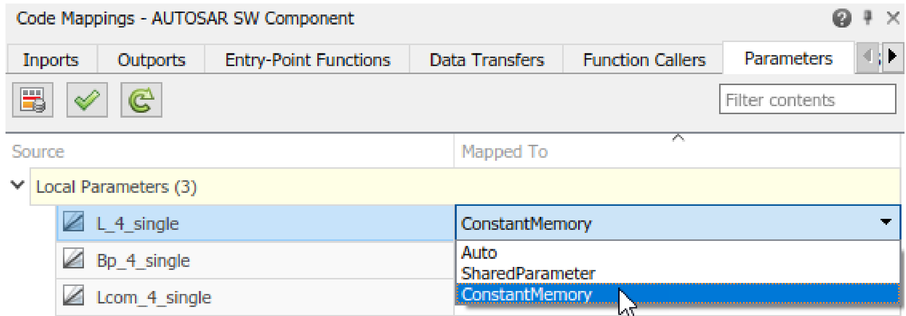
Table data appears in generated AUTOSAR C code as fields of a single structure. To control the characteristics of the structure type, such as its name, use the properties of the object.

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 outport to the Curve or Map block.

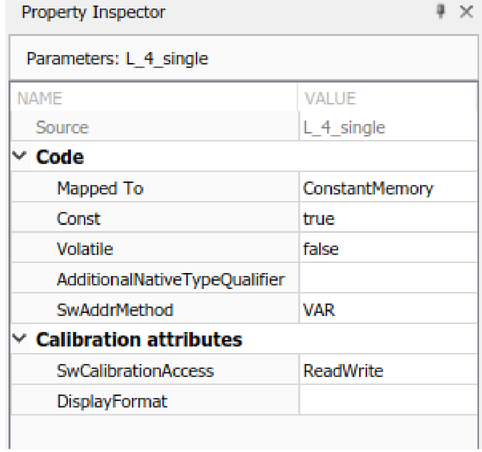


3 In the AUTOSAR code perspective, use Code Mappings editor to map `Simulink.LookupTable` objects to AUTOSAR internal calibration parameters. In the **Parameters** tab, select each `Simulink.LookupTable` object that you created. Map each object to AUTOSAR parameter type `ConstantMemory`, `SharedParameter`, 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`, use Property Inspector to view or modify other code and calibration attributes. For more information on parameter properties, see “Map Model Workspace Parameters to AUTOSAR Component Internal Parameters” on page 4-75.



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 5-20.

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(Ifl_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.

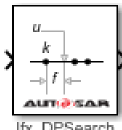
Configure COM_AXIS Lookup Tables By Using Lookup Table and Breakpoint Objects

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 `*matlabroot*/help/toolbox/autosar/examples/mAutosarLutObjs.slx`.

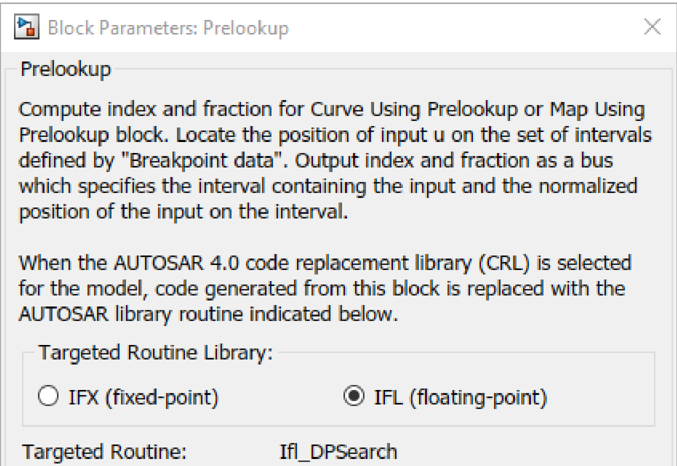
In this example, to model an AUTOSAR lookup table in a COM_AXIS configuration, you pair AUTOSAR Blockset Prelookup blocks with Curve Using Prelookup or Map Using Prelookup blocks.

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.

Simulink.Breakpoint: Bp_4_single

Breakpoints

☑ Support tunable size

Value	Data type	Dimensions	Min	Max	Unit	Field name	Tunable size name	Description
[1 2 3 4]	single	[1 4]	[ ]	[ ]		Bp1	Nx	

☐ Argument

Code generation options

Data definition

Storage class: Auto

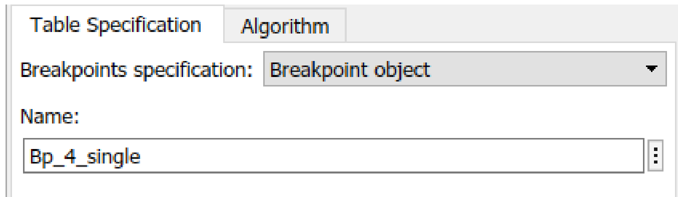
Struct type definition

Name: BP_4_single_type

Data scope: Auto

Header file:

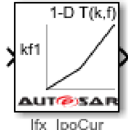
d In the Prelookup block dialog box, **Table Specification** tab, enter the `Simulink.Breakpoint` object name in the **Name** field. You can reduce memory consumption by sharing breakpoint data between lookup tables.



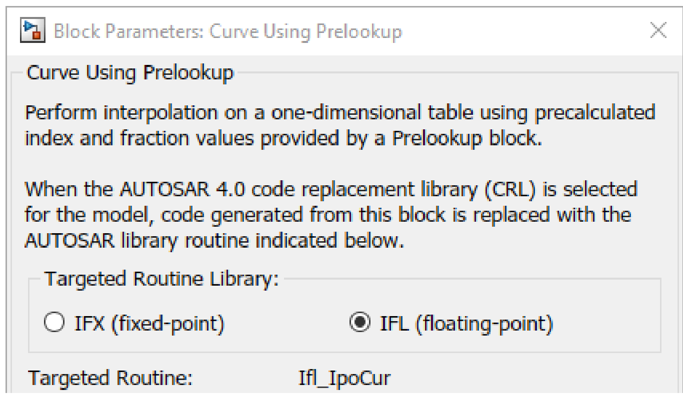
e In the block dialog box, **Algorithm** tab, set **Integer Rounding Method** to `Zero`. Leave **Index Search Method** set to `Linear search`.

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.



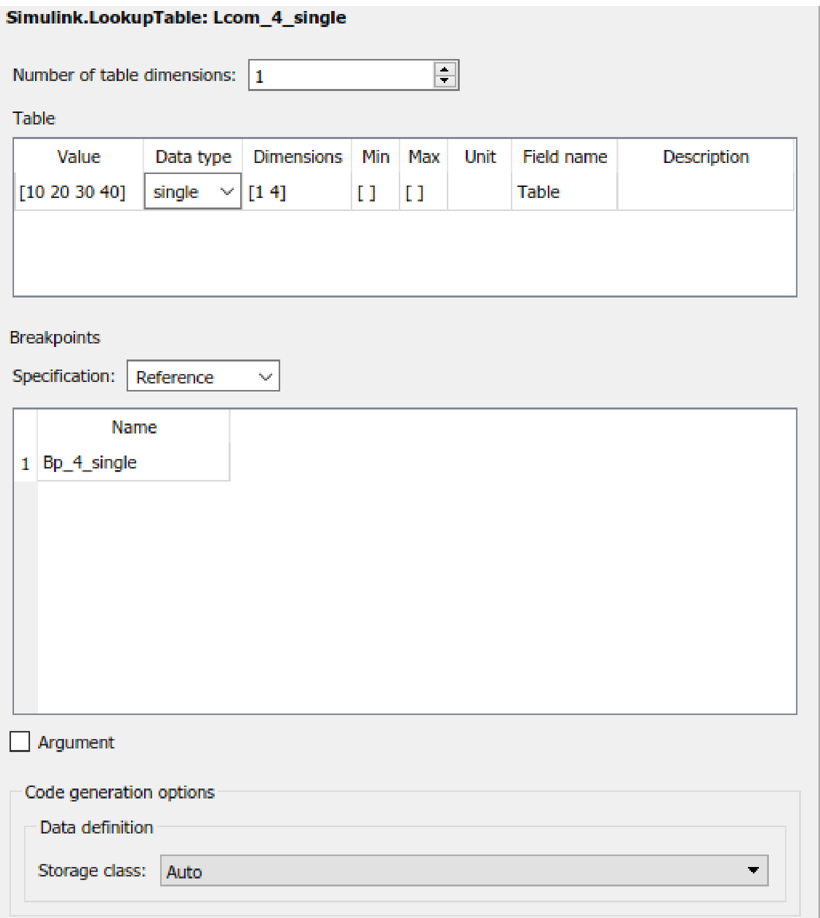
b Open each Curve Using Prelookup or Map Using Prelookup block and configure it to generate a routine from the AUTOSAR 4.0 code replacement library (CRL). As you modify block settings, the block dialog box updates the name of the targeted AUTOSAR routine.



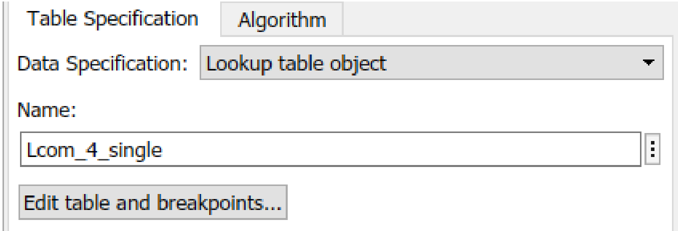
In the block dialog box, make these selections:

- 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.

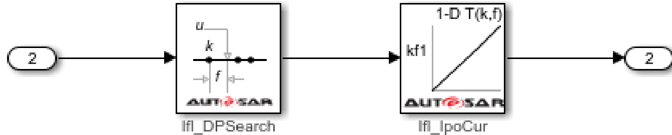


e In the block dialog box, **Algorithm** tab, set **Integer Rounding Method** to `Zero`. Leave **Interpolation Method** set to `Linear point-slope`.

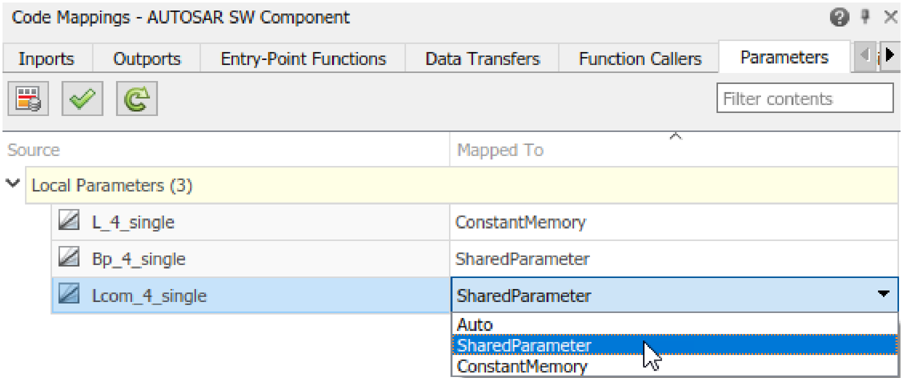
Each set of table data appears in the generated C code as a separate array variable. If the table size is tunable, each breakpoint vector appears as a structure. The structure contains a field to store the breakpoint data and, optionally, a field to store the length of the vector. The second field enables 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 outports 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.

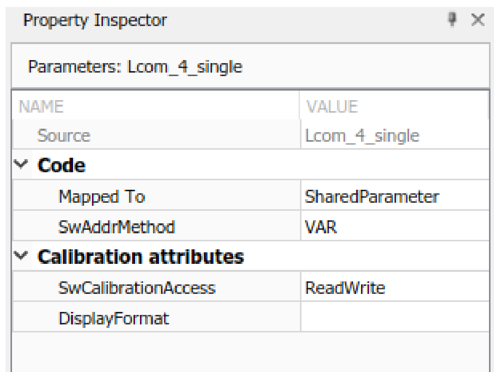


4 In the AUTOSAR code perspective, use Code Mappings editor to map `Simulink.LookupTable` and `Simulink.Breakpoint` objects to AUTOSAR internal calibration parameters. In the **Parameters** tab, select each `Simulink.LookupTable` and `Simulink.Breakpoint` object that you created. Map each object to AUTOSAR 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`, use Property Inspector to view or modify other code and calibration attributes. For more information on parameter properties, see "Map Model Workspace Parameters to AUTOSAR Component Internal Parameters" on page 4-75.



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 5-20.

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(Ifl_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.

See Also

Curve | Curve Using Prelookup | Map | Map Using Prelookup | Prelookup | `Simulink.Breakpoint` | `Simulink.LookupTable` | `getParameter` | `mapParameter`

Related Examples

- “Map Model Workspace Parameters to AUTOSAR Component Internal Parameters” on page 4-75

More About

- “Code Generation with AUTOSAR Code Replacement Library” on page 5-20
- “Model AUTOSAR Calibration Parameters and Lookup Tables” on page 2-48

Configure AUTOSAR Data for Measurement and Calibration

In Simulink, you can import and export AUTOSAR software data definition properties and modify the properties for some forms of AUTOSAR data.

In this section...

"About Software Data Definition Properties (SwDataDefProps)" on page 4-263
"Configure SwCalibrationAccess" on page 4-264
"Configure DisplayFormat" on page 4-266
"Configure SwAddrMethod" on page 4-270
"Configure SwAlignment" on page 4-274
"Export SwImplPolicy" on page 4-274
"Export SwRecordLayout for Lookup Table Data" on page 4-275

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 measurement and calibration tool access to a data object.
- Display format (`DisplayFormat`) — Specifies measurement and calibration 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 measurement and calibration 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 4-264, “Configure DisplayFormat” on page 4-266, “Configure SwAddrMethod” on page 4-270, “Configure SwAlignment” on page 4-274, “Export SwImplPolicy” on page 4-274, and “Export SwRecordLayout for Lookup Table Data” on page 4-275.

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 does not appear in the generated description file and is not accessible with measurement and calibration 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 4-264
- “Specify Default SwCalibrationAccess for Application Data Types” on page 4-266

Specify SwCalibrationAccess for AUTOSAR Data Elements

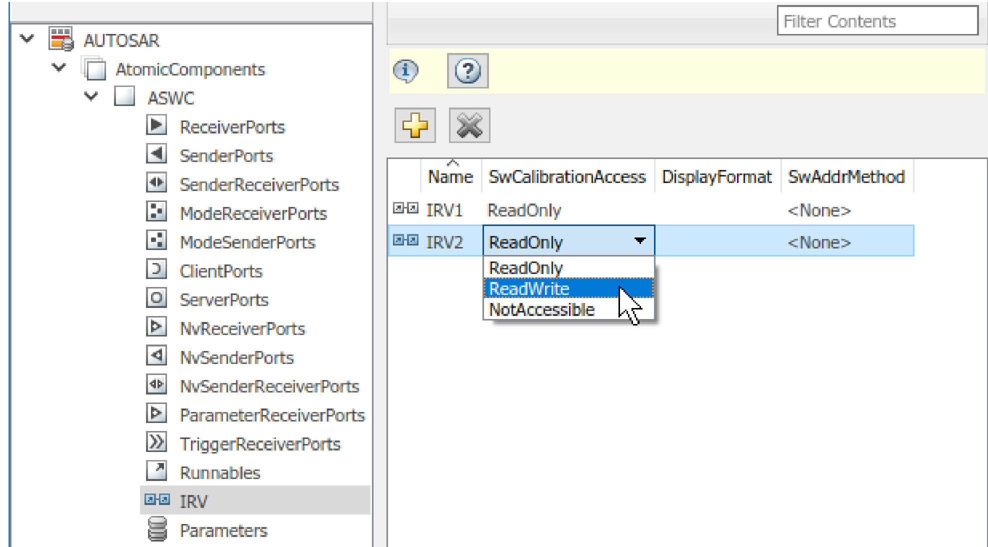
You can use either AUTOSAR Dictionary or MATLAB function calls to specify the `SwCalibrationAccess` property for the following AUTOSAR data elements:

- 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 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 measurement and calibration 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_fcncalls` example model and sets measurement and calibration access to inter-runnable variable IRV2 to `ReadWrite`.

```
hModel = 'autosar_swc_fcncalls';
open_system(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 measurement and calibration access for AUTOSAR application data types from `ReadOnly` to `ReadWrite`.

```
hModel = 'autosar_swc_fcncalls';
open_system(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 measurement and calibration 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:
	• `('')`: Insert a space before the value.
	• `-`: Left-justify.
	• `+`: Display plus or minus sign, even for positive numbers.
	• `#`:
	  • 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: • 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.

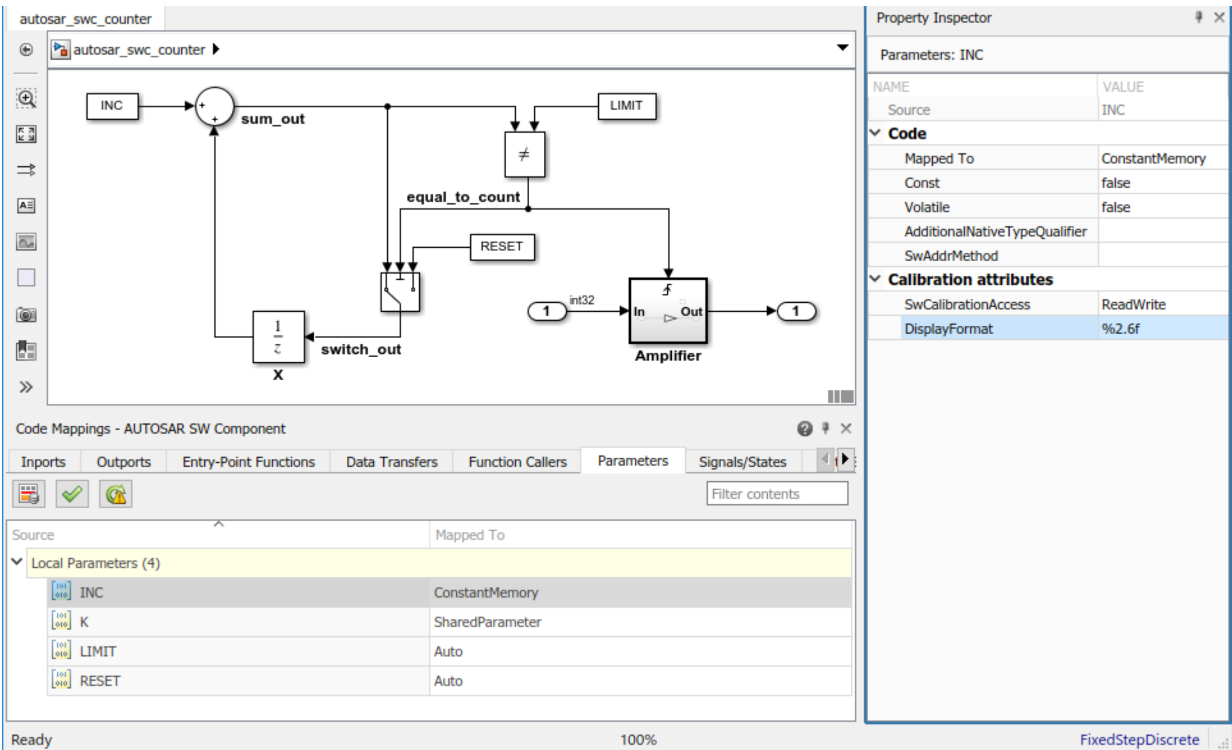
Field	Description

`type`	Characters specifying a numeric conversion type supported by AUTOSAR schemas:
	• `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`.
	  • `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`.
	  • `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_swc_counter';
open_system(hModel);
slMap = autosar.api.getSimulinkMapping(hModel);
mapParameter(slMap,'INC','ConstantMemory','DisplayFormat','%2.6f')
```



If you specify a display format, exporting `arxml` code generates a corresponding DISPLAY-FORMAT specification.

```
<PARAMETER-DATA-PROTOTYPE UUID="...">
    <SHORT-NAME>INC</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>
                <DISPLAY-FORMAT>%2.6f</DISPLAY-FORMAT>
```

```
            <SW-IMPL-POLICY>STANDARD</SW-IMPL-POLICY>
        </SW-DATA-DEF-PROPS-CONDITIONAL>
    </SW-DATA-DEF-PROPS-VARIANTS>
</SW-DATA-DEF-PROPS>
...
</PARAMETER-DATA-PROTOTYPE>
```

Configure SwAddrMethod

AUTOSAR software components use software address methods (`SwAddrMethods`) to group data and functions into memory sections for access by run-time measurement and calibration tools. For a Simulink model mapped to an AUTOSAR software component, you can associate `SwAddrMethods` with these elements:

- Model parameter mapped to AUTOSAR constant memory
- Model parameter mapped to AUTOSAR internal calibration parameter
- Signal, state, or data store mapped to AUTOSAR static memory
- Signal, state, or data store mapped to 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 4-271.

2. Associate `SwAddrMethods` with model data and functions. Open Code Mappings editor and select the **Parameters**, **Signals/States**, **Data Stores**, or **Entry-Point Functions** tab. Select an element within that tab and use Property Inspector to set the **SwAddrMethod** attribute. Alternatively, use the equivalent AUTOSAR map function. For more information, see “Associate SwAddrMethod with Model Data or Function” on page 4-273.

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>
```

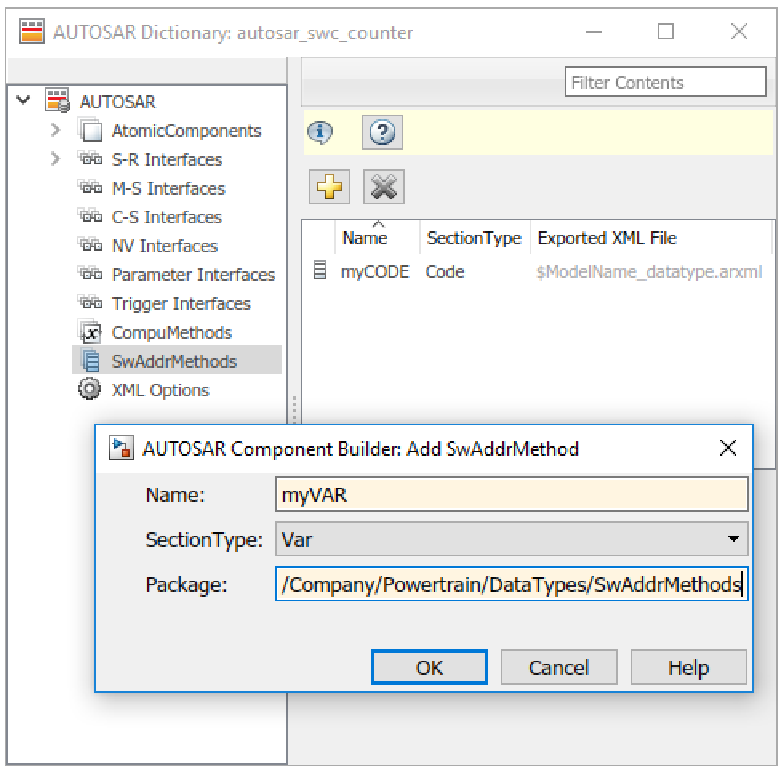
- AUTOSAR-compliant C code contains comments and `#define` and `#include` statements that provide a wrapper around data or function definitions belonging to each `SwAddrMethod` memory section.

```
/* Static Memory for Internal Data */

/* SwAddrMethod VAR for Internal Data */
#define autosar_swc_counter_START_SEC_VAR
#include "autosar_swc_counter_MemMap.h"

volatile my_qualifier boolean SM_equal_to_count;

#define autosar_swc_conter_STOP_SEC_VAR
#include "autosar_swc_counter_MemMap.h"
```

- “Create SwAddrMethods in Simulink” on page 4-271
- “Associate SwAddrMethod with Model Data or Function” on page 4-273

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.

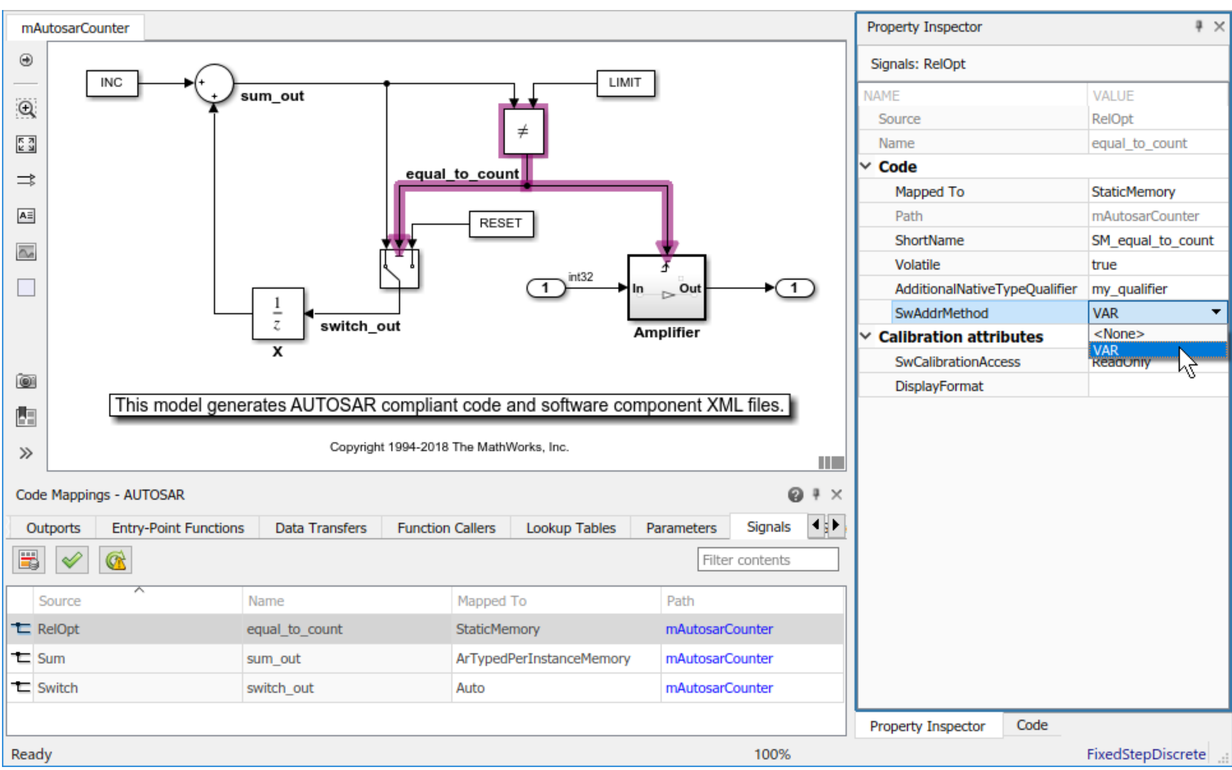
```
hModel = 'autosar_swc_counter';
open_system(hModel)
arProps = autosar.api.getAUTOSARProperties(hModel);
addPackageableElement(arProps,'SwAddrMethod',...
    '/Company/Powertrain/DataTypes/SwAddrMethods','myCODE',...
    'SectionType','Code')
swAddrPaths = find(arProps,[],'SwAddrMethod','PathType','FullyQualified',...
    '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 Code Mappings editor and select the **Parameters**, **Signals/States**, **Data Stores**, or **Entry-Point Functions** tab. Select an element within that tab and use Property Inspector to set the **SwAddrMethod** attribute. In this example, SwAddrMethod VAR is selected for signal equal_to_count, which is mapped to AUTOSAR static memory.



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 = 'El1';
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>Appl_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 “Reuse AUTOSAR Element Descriptions” on page 3-49.

See Also

Related Examples

- "Import AUTOSAR Software Component" on page 3-27
- "Model AUTOSAR Calibration Parameters and Lookup Tables" on page 2-48
- "Configure AUTOSAR CompuMethods" on page 4-320
- "Configure AUTOSAR Code Generation" on page 5-12

More About

- "AUTOSAR Component Configuration" on page 4-3

Configure AUTOSAR Runnables and Events

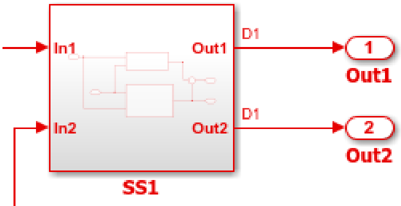
The internal behavior of an AUTOSAR software component is implemented by a set of runnable entities (runnables). A runnable is a sequence of operations provided by the component that can be started by the Runtime Environment (RTE). The component configures an event to activate each runnable – for example, 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 4-364
`DataReceiveErrorEvent`	Sender-receiver (S-R) communication	"Configure AUTOSAR Receiver Port for DataReceiveErrorEvent" on page 4-165
`ExternalTrigger-OccurredEvent`	External trigger event communication	"Configure Receiver for AUTOSAR External Trigger Event Communication" on page 4-240
`InitEvent`	R4.1 activation of initialization runnable	"Configure AUTOSAR Initialization Runnable (R4.1)" on page 4-291
`ModeSwitchEvent`	Mode-switch (M-S) communication	"Configure AUTOSAR Mode-Switch Communication" on page 4-225
`OperationInvokedEvent`	Client-server (C-S) communication	"Configure AUTOSAR Client-Server Communication" on page 4-197
`TimingEvent`	Periodic activation of runnable	"Configure AUTOSAR TimingEvent for Periodic Runnable" on page 4-361

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_swc`.
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_swc`, the subsystem SS1 generates rate-based model step function `Runnable_1s`.



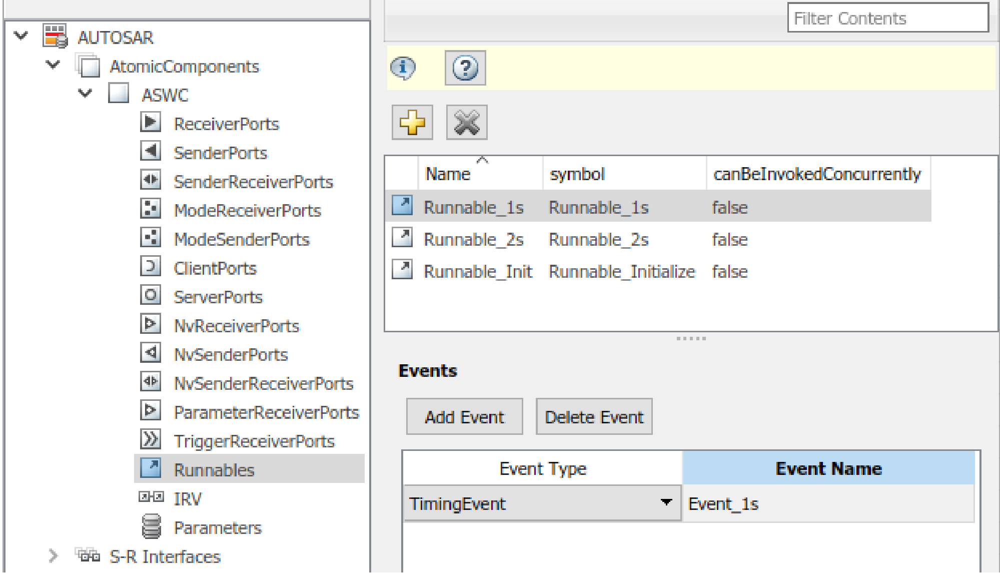
3 Create or identify an AUTOSAR runnable to which to map the Simulink entry point function. Open AUTOSAR Dictionary. Expand **AtomicComponents**, expand the component, and select the **Runnables** view. If you need to create a new AUTOSAR runnable, click the plus sign. The model `autosar_swc` contains 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 nonserver runnables, leave `canBeInvokedConcurrently` set to `false`. For more information, see "Concurrency Constraints for AUTOSAR Server Runnables" on page 4-221.

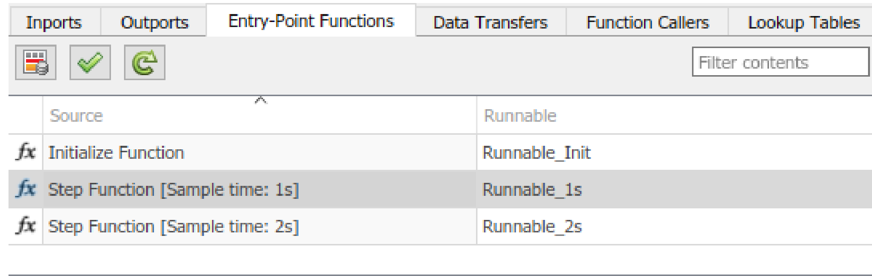
5 Configure an event to activate the runnable. Go to the **Events** pane for the selected runnable. If you need to create an event, click **Add Event**. Enter an event name and set the event type.

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_swc`, 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 Code Mappings editor and select the **Entry-Point Functions** tab. For model `autosar_swc`, select the model step 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.

See Also

Related Examples

- “Import AUTOSAR Software Component” on page 3-27
- “Modeling Patterns for AUTOSAR Runnables” on page 2-13
- “Model AUTOSAR Software Components” on page 2-3
- “Configure AUTOSAR Code Generation” on page 5-12

More About

- “AUTOSAR Component Configuration” on page 4-3

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.

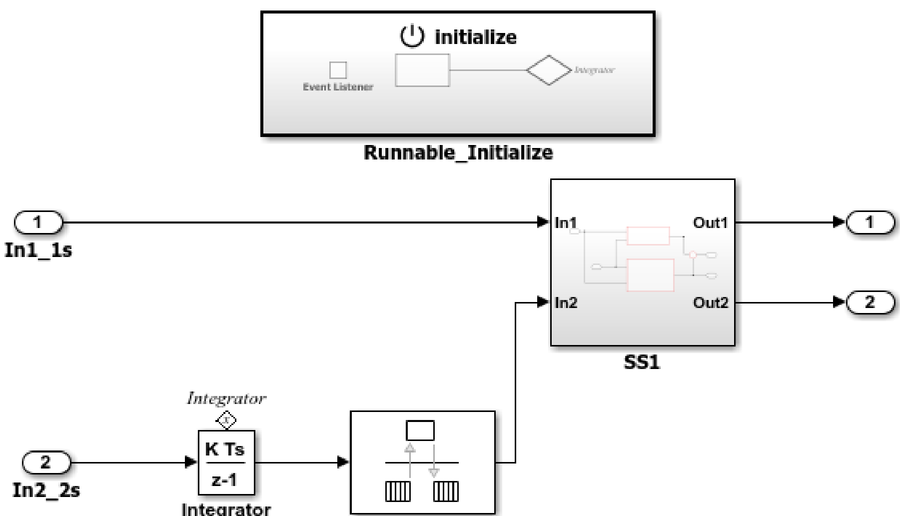
The Initialize Function and Terminate Function blocks can control execution of a component in response to initialize, reset, or terminate events. For more information, see “Using Initialize, Reset, and Terminate Functions” (Simulink), “Generate Code That Responds to Initialize, Reset, and Terminate Events” (Simulink Coder), and AUTOSAR topic “Startup, Reset, and Shutdown” on page 2-11.

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.)

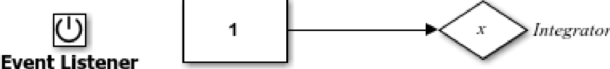
This example shows how to configure an AUTOSAR software component for simple startup and termination processing, using the Initialize Function and Terminate Function blocks.

1. Open a model that is configured for AUTOSAR code generation. This example uses a writable copy of the example model `autosar_swc`.

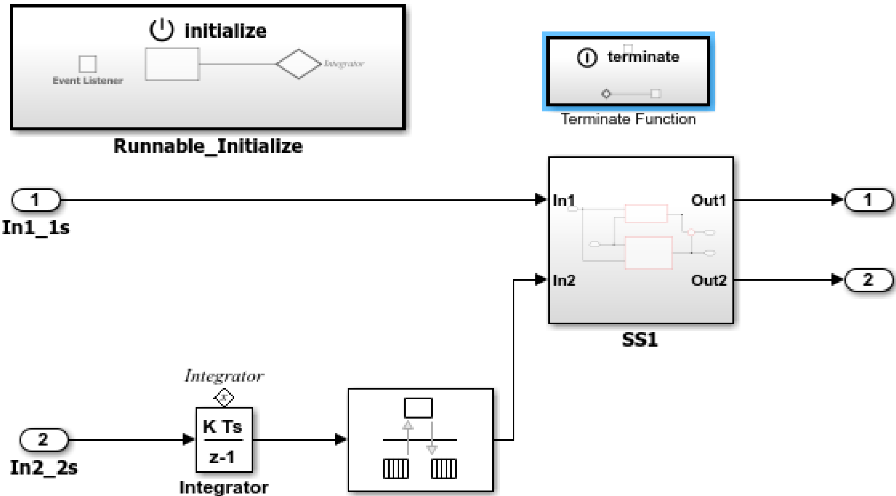
   Add an Initialize Function block to the model.



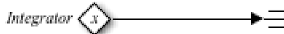
2 In the Initialize Function block, develop the logic that is required to execute during component initialization, using the techniques described in “Using Initialize, Reset, and Terminate Functions” (Simulink).



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, Reset, and Terminate Functions” (Simulink).



In this example, the Terminator block is a placeholder for saving the state value.

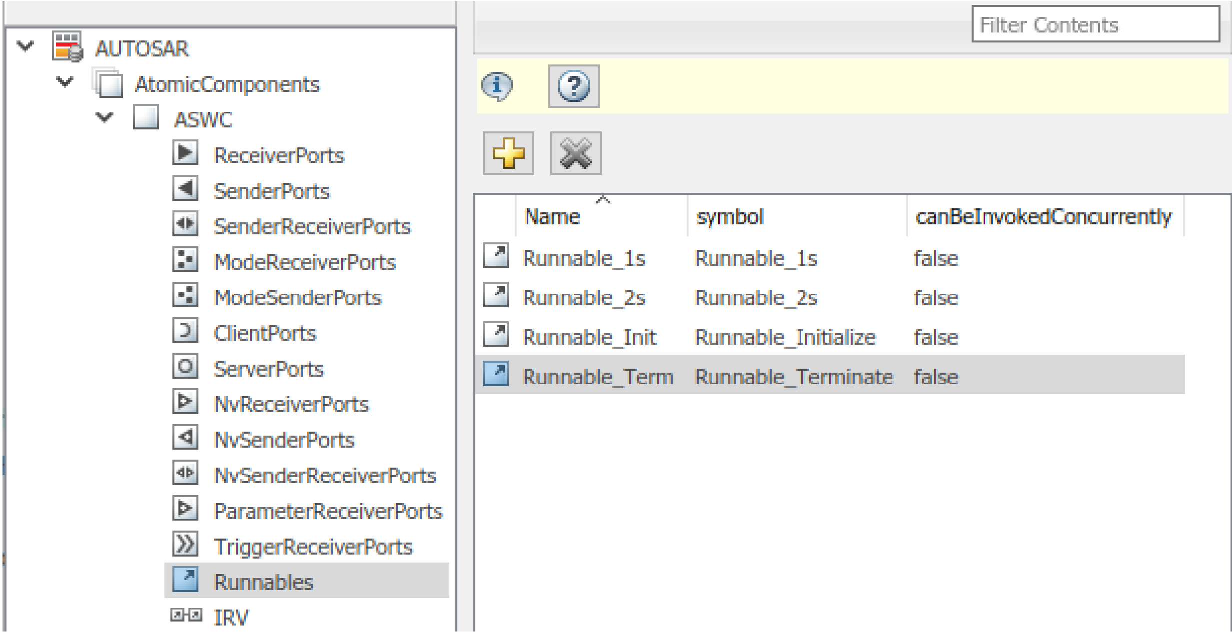
5 Add a terminate entry-point function to the model. In the Configuration Parameters dialog box, in the **Code Generation > Interface** pane, under **Advanced parameters**, select the option **Terminate function required**. Click **Apply**.

6 Open 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 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.

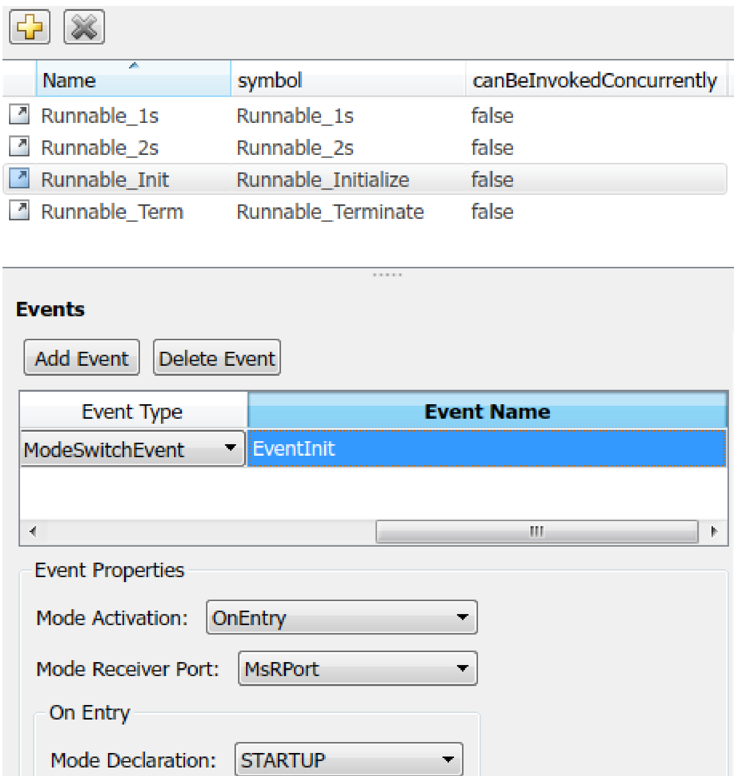
The runnable **symbol** value shown in the runnables view becomes the runnable function name. The runnable **Name** value is used in the names of RTE access methods generated for the runnable.



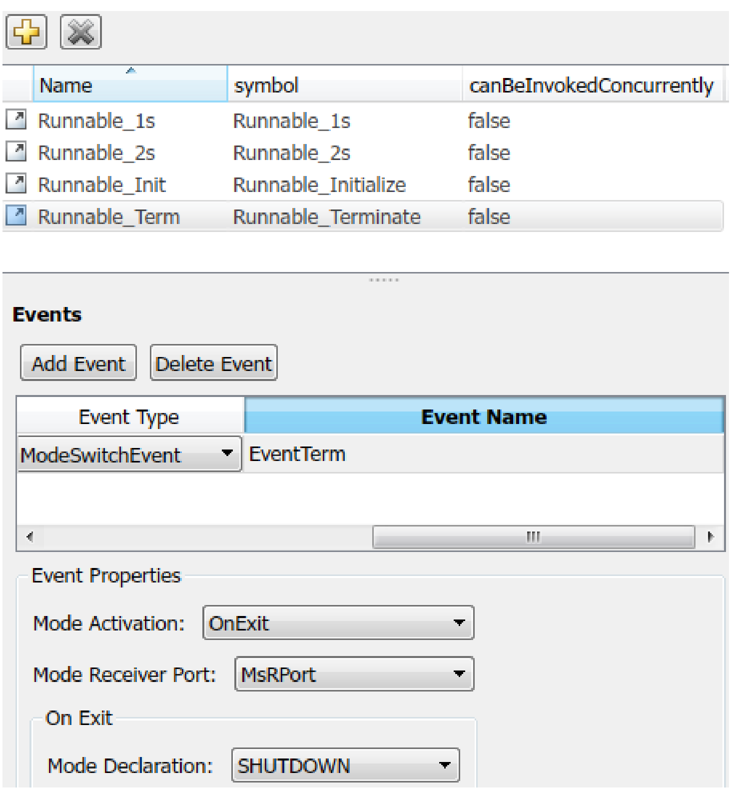
8 For both the initialize and terminate runnables, configure an AUTOSAR event that activates the runnable.

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 4-225.

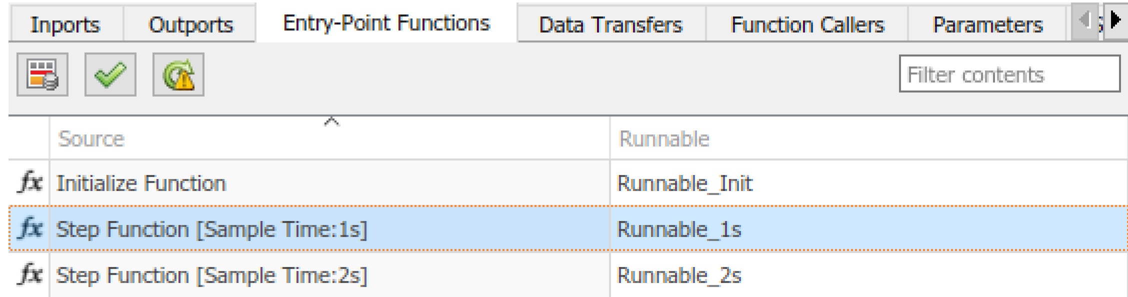
In the runnables view, click the initialize runnable name to display and modify its associated event properties. Add and configure an event.



In the runnables view, click the terminate runnable name to display and modify its associated event properties. Add and configure an event.



9 Open Code Mappings editor and select the **Entry-Point 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

Event Listener | Initialize Function | State Reader | State Writer | Terminate Function

Related Examples

- “Using Initialize, Reset, and Terminate Functions” (Simulink)
- “Generate Code That Responds to Initialize, Reset, and Terminate Events” (Simulink Coder)
- “Create Test Harness to Generate Function Calls” (Simulink)
- “Configure AUTOSAR Mode-Switch Communication” on page 4-225

More About

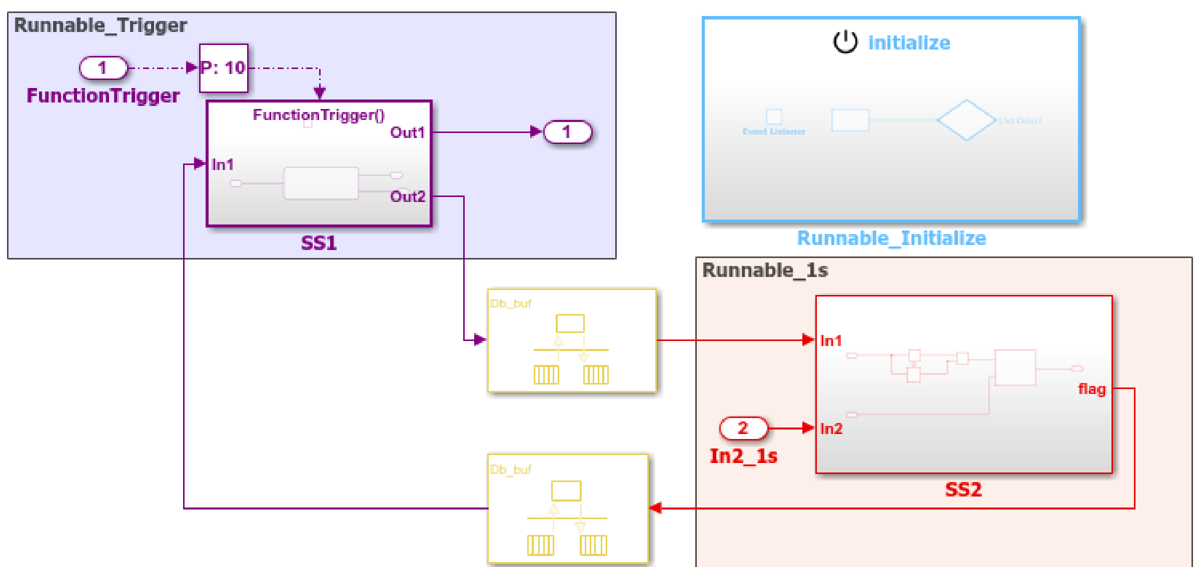
- “Startup, Reset, and Shutdown” on page 2-11
- “Configure Code Generation for Model Entry-Point Functions” (Simulink Coder)
- “AUTOSAR Component Configuration” on page 4-3

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 Japan MBD Automotive Advisory Board (JMAAB) document *Control Algorithm Modeling Guidelines Using MATLAB, Simulink, and Stateflow - Version 4.01*. The document is available from the MAAB web page at `https://www.mathworks.com/solutions/automotive/standards/maab.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_swc_fcncalls`. In this model, an asynchronous function-call runnable at the top level of the model interacts with a periodic rate-based runnable.

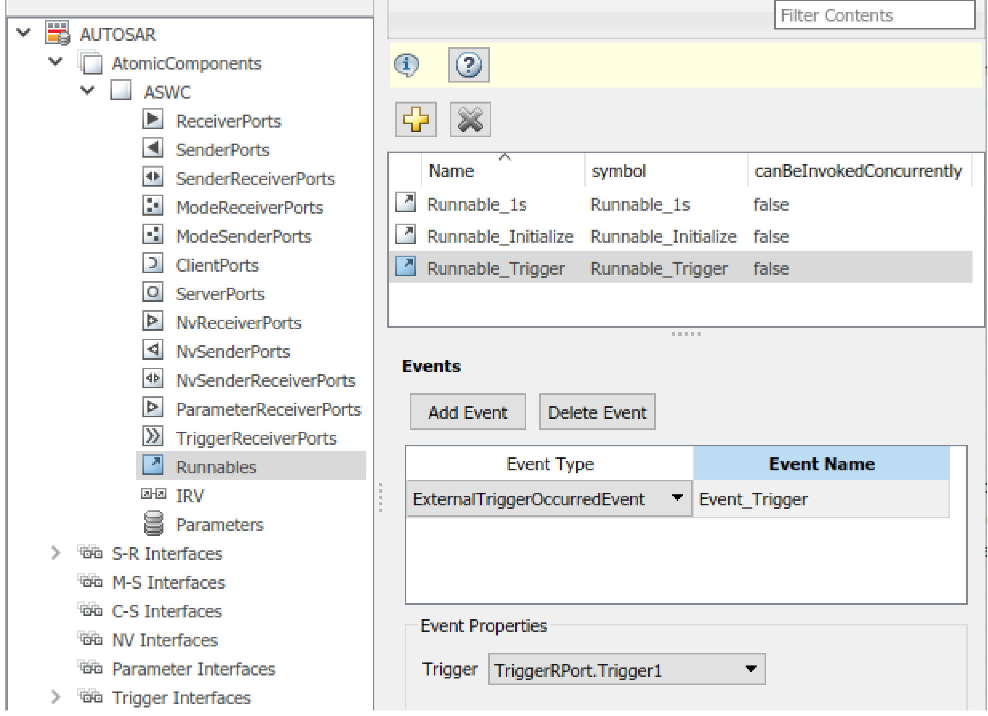


Some guidelines apply to AUTOSAR modeling of the JMAAB type beta controller layout:

- IRVs must be modeled with Rate Transition blocks.

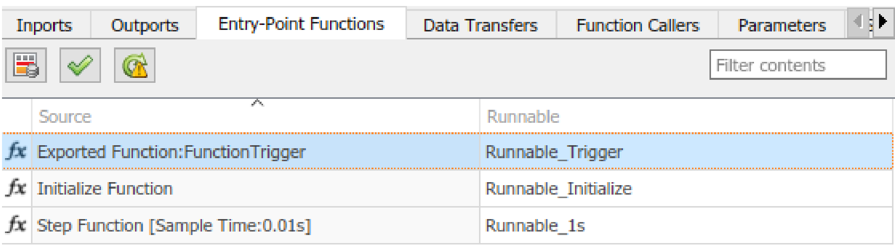
- Function-call subsystems must have asynchronous rates. (In the function-call subsystem Trigger block, **Sample time type** must be triggered, not periodic.)
- For each asynchronous function-call subsystem, you must insert an Asynchronous Task Specification task block between the function-call root 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 4-240.

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.



See Also

Asynchronous Task Specification | Rate Transition

Related Examples

- “Configure Receiver for AUTOSAR External Trigger Event Communication” on page 4-240
- “Modeling Patterns for AUTOSAR Runnables” on page 2-13
- “Configure AUTOSAR Code Generation” on page 5-12

More About

- “Model AUTOSAR Software Components” on page 2-3
- “AUTOSAR Component Configuration” on page 4-3

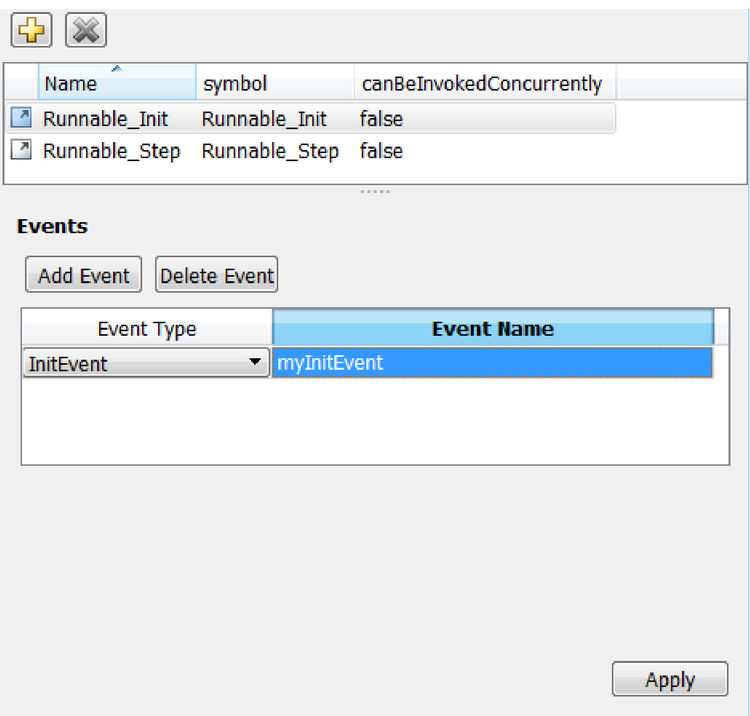
Configure AUTOSAR Initialization Runnable (R4.1)

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 4-225.)

If you import `arxml` code that describes a runnable with an `InitEvent`, the `arxml` importer configures the runnable in Simulink as an initialization runnable.

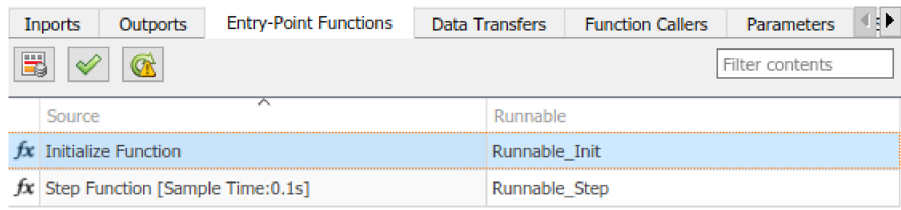
Alternatively, you can configure a runnable to be the initialization runnable in Simulink. For example,

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 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 Code Mappings editor and select the **Entry-Point 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 Function` is mapped to AUTOSAR runnable `Runnable_Init`.



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>
```

Configure Disabled Mode for AUTOSAR Runnable Event

AUTOSAR Release 4.0 introduced the ability to 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.

```
addpath (fullfile(matlabroot,'/help/toolbox/autosar/examples'));
hModel = 'mAutosarMsConfigAfter';
open_system(hModel)
arProps = autosar.api.getAUTOSARProperties(hModel);
eventPaths = find(arProps,[],'TimingEvent')

eventPaths =
    {'ASWC/Behavior/Event_t_1tic_B'}    {'ASWC/Behavior/Event_t_10tic'}

dsblModes = get(arProps,eventPaths{1},'DisabledMode')

dsblModes =
  1×0 empty cell array

set(arProps,eventPaths{1},'DisabledMode',{'myMRPort.STARTUP','myMRPort.SHUTDOWN'});
dsblModes = get(arProps,eventPaths{1},'DisabledMode')

dsblModes =
    {'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.

Configure AUTOSAR Per-Instance Memory

You can model AUTOSAR per-instance memory (PIM) for AUTOSAR applications. To model AUTOSAR per-instance memory, 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 2-41.

AUTOSAR typed per-instance memory (`ArTypedPerInstanceMemory`), introduced in AUTOSAR schema version 4.0, defines an AUTOSAR typed memory block that is available for each instance of an AUTOSAR software component. In the AUTOSAR Runtime Environment (RTE), calibration tools can access `arTypedPerInstanceMemory` blocks for measurement and calibration.

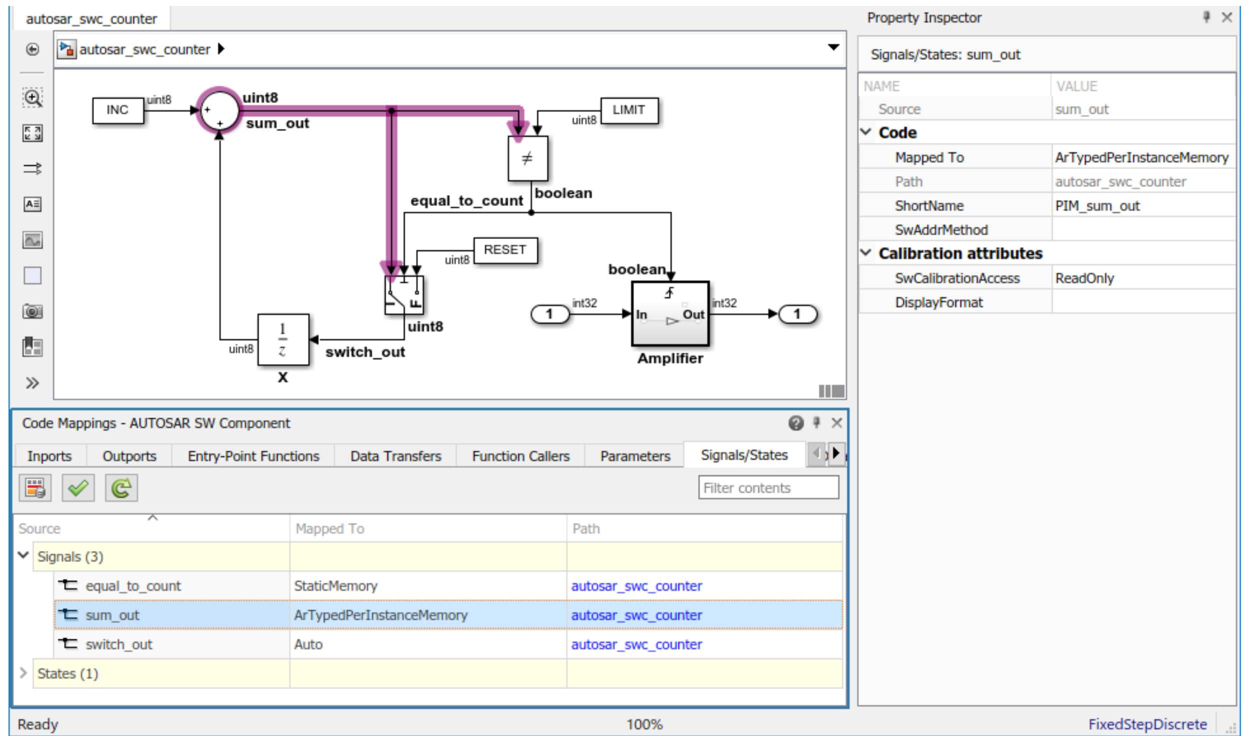
To model AUTOSAR PIM, you can use Simulink block signals, discrete states, or data stores in your model.

In this section...
“Configure Block Signals and States as AUTOSAR Typed Per-Instance Memory” on page 4-295
“Configure Data Stores as AUTOSAR Typed Per-Instance Memory” on page 4-297

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 Code Mappings editor. Use the **Signals/States** tab to map signals and states 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 AUTOSAR code perspective, open 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 and displays the signal attributes in Property Inspector. Use Property Inspector to modify the signal attributes. In the **Mapped To** drop-down list, select `ArTypedPerInstanceMemory`. For more information about signal code and calibration attributes, see “Map Block Signals and States to AUTOSAR Variables” on page 4-79.



3 In the **Signals/States** tab, in the list of available states, select state X. Use Property Inspector to modify the state attributes. In the **Mapped To** drop-down list, select `ArTypedPerInstanceMemory`.

When you generate code:

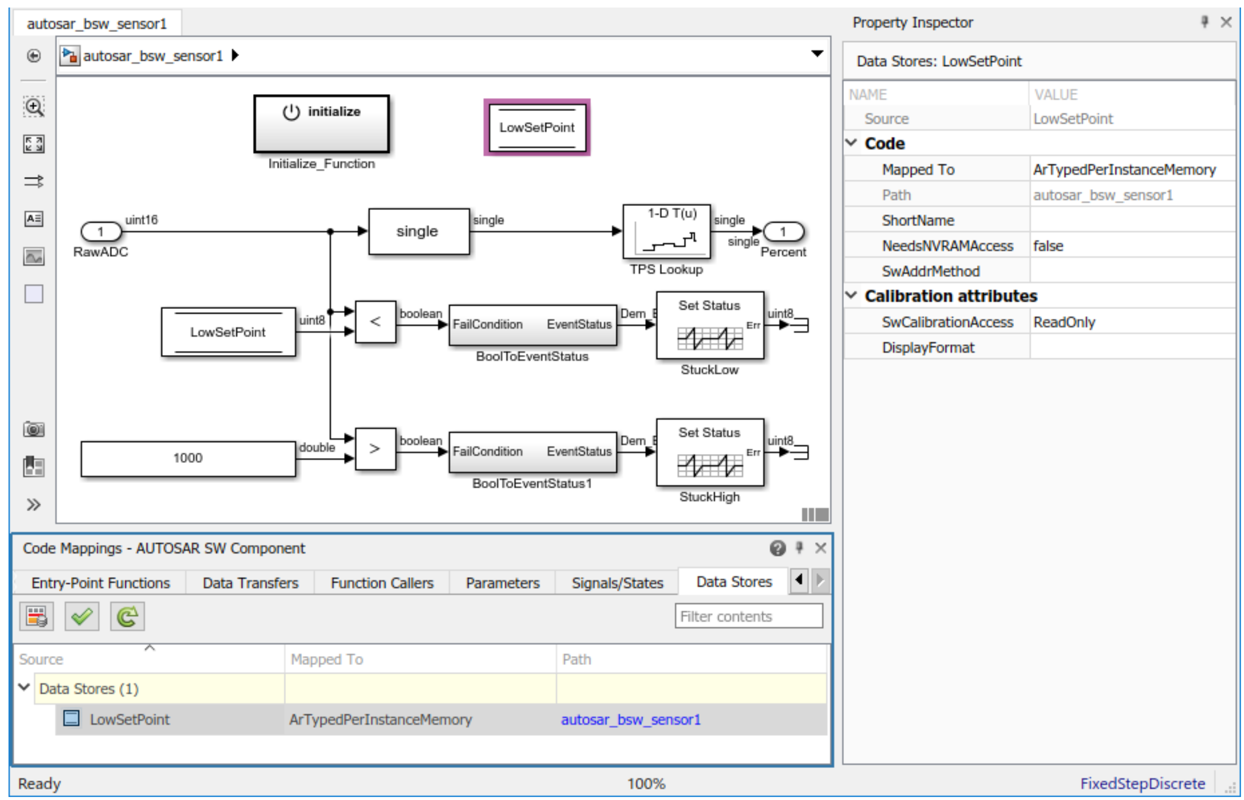
- Exported `arxml` files contain `AR-TYPED-PER-INSTANCE-MEMORYS` descriptions for signals and states that you configured as `ArTypedPerInstanceMemory`.
- Generated C code contains `Rte_Pim_*` API calls for signal and state variables.

For referenced models within an AUTOSAR component model, Embedded Coder automatically maps internal signal and states for model reference code generation. Internal signals and states automatically map to AUTOSAR `ArTypedPerInstanceMemory` for multi-instance model reference, or 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 Code Mappings editor. Use the **Data Stores** tabs to map data stores to `arTypedPerInstanceMemory`. For example:

1. Open an AUTOSAR model that contains data stores for which you want to generate `arTypedPerInstanceMemory` blocks. This example uses model `autosar_bsw_sensor1`.
2. In AUTOSAR code perspective, open 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 and displays the data store attributes in Property Inspector. Use Property Inspector to modify the data store attributes. In the **Mapped To** drop-down list, select `ArTypedPerInstanceMemory`. For more information about data store code and calibration attributes, see “Map Data Stores to AUTOSAR Variables” on page 4-81.



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.

Note The software does not support per-instance memory code generation for data stores in referenced models.

If you select **needsNVRAMAccess**, a `SERVICE-NEEDS` entry (schema version 3.0 or later) or `NVRAM-MAPPINGS` entry (schema version 2.1) 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.

See Also

Data Store Memory | `getDataStore` | `getSignal` | `getState` | `mapDataStore` | `mapSignal` | `mapState`

Related Examples

- “Map Block Signals and States to AUTOSAR Variables” on page 4-79
- “Map Data Stores to AUTOSAR Variables” on page 4-81
- “Model AUTOSAR Component Behavior” on page 2-38

More About

- “Per-Instance Memory” on page 2-41

Configure AUTOSAR Static Memory

You can model AUTOSAR static memory for AUTOSAR applications. To model AUTOSAR static memory, 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 2-42.

AUTOSAR static memory (`StaticMemory`), introduced in AUTOSAR schema version 4.0, corresponds to Simulink internal global signals. In the AUTOSAR Runtime Environment (RTE), calibration tools can access `StaticMemory` blocks for measurement and calibration.

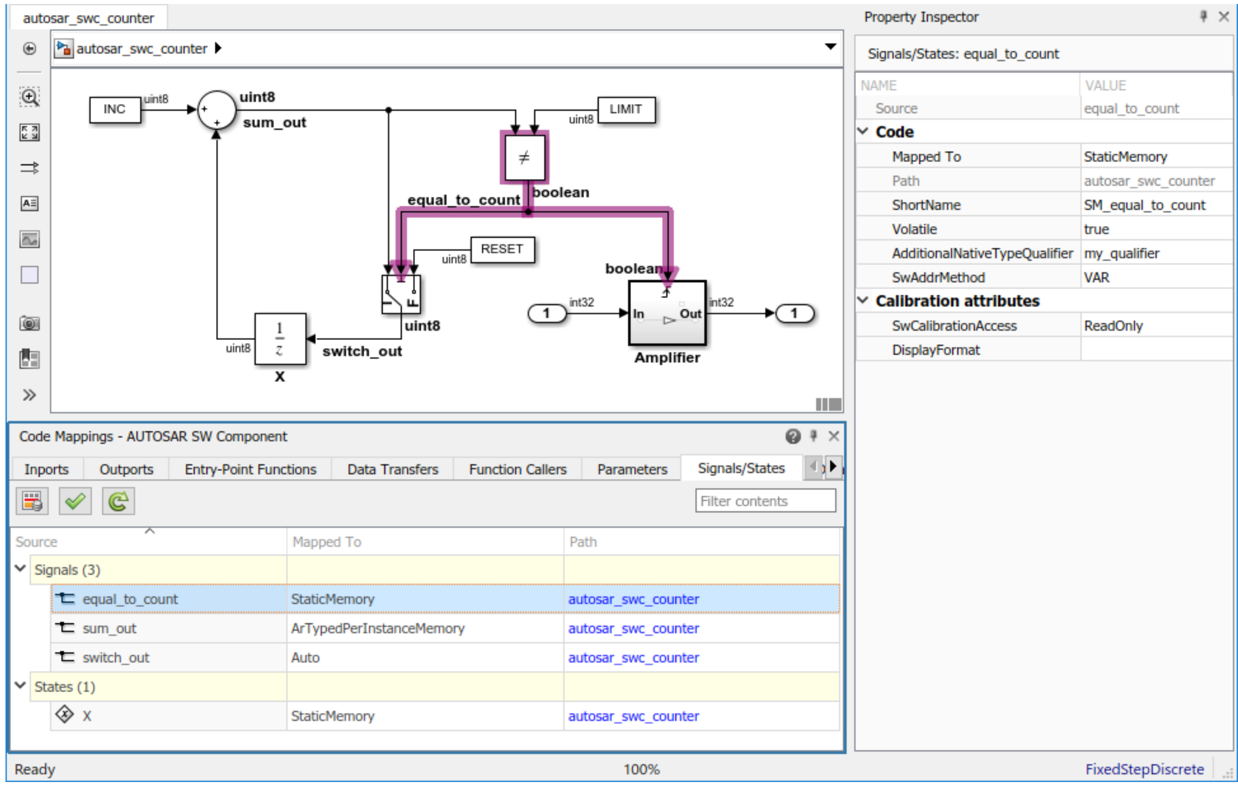
To model AUTOSAR static memory, you can use Simulink block signals, discrete states, or data stores in your model.

In this section...
“Configure Block Signals and States as AUTOSAR Static Memory” on page 4-300
“Configure Data Stores as AUTOSAR Static Memory” on page 4-302

Configure Block Signals and States as AUTOSAR Static Memory

To generate `StaticMemory` blocks for block signal and discrete state data in your AUTOSAR model, open Code Mappings editor. Use the **Signals/States** tab to map signals and states to `StaticMemory`. For example:

1. Open an AUTOSAR model that contains signals or states for which you want to generate `StaticMemory` blocks. This example uses model `autosar_swc_counter`.
2. In AUTOSAR code perspective, open 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 and displays the signal attributes in Property Inspector. Use Property Inspector to modify the signal attributes. In the **Mapped To** drop-down list, select `StaticMemory`. For more information about signal code and calibration attributes, see “Map Block Signals and States to AUTOSAR Variables” on page 4-79.



3 Select the **Signals/States** tab and select state X. Use Property Inspector to modify the state attributes. In the **Mapped To** drop-down list, select `StaticMemory`.

When you generate code:

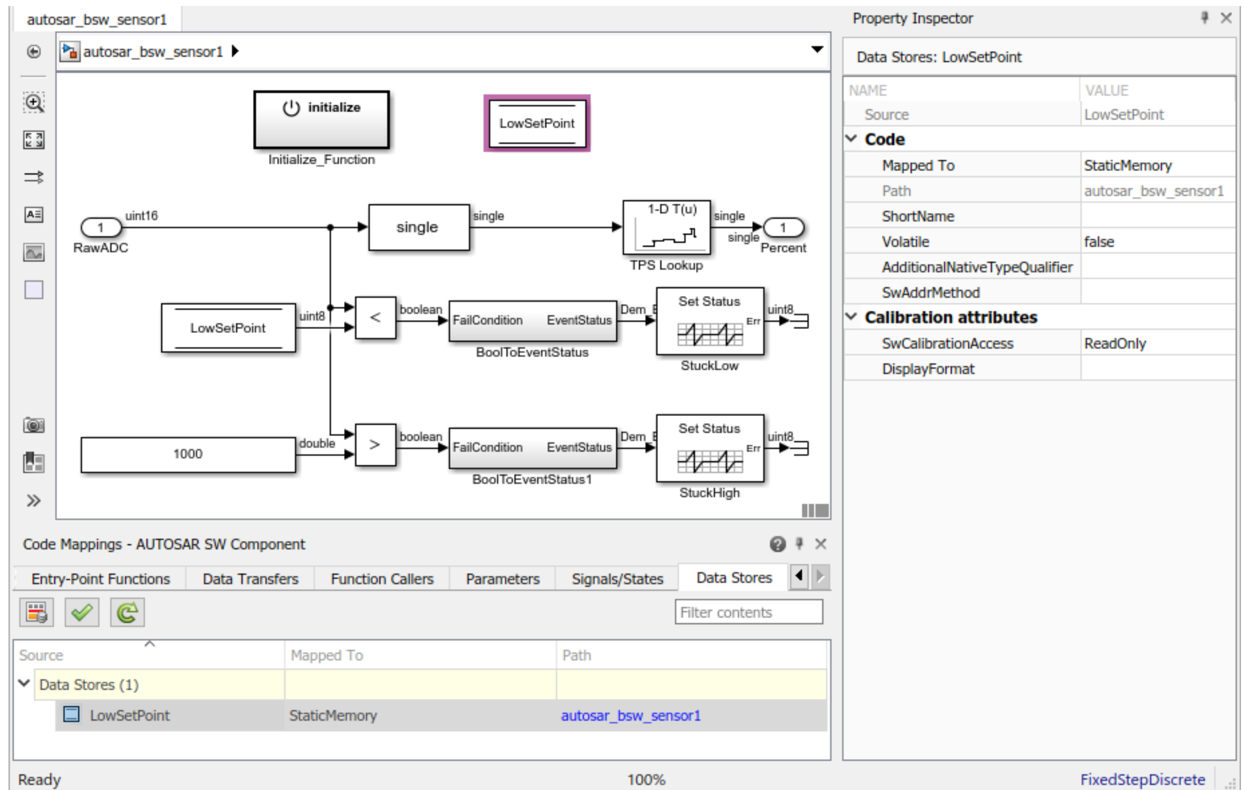
- Exported `arxml` files contain `STATIC-MEMORYS` descriptions for signals and states that you configured as `StaticMemory`.
- Generated C code declares and references the static memory variables.

For referenced models within an AUTOSAR component model, Embedded Coder automatically maps internal signal and states for model reference code generation. Internal signals and states automatically map to AUTOSAR `ArTypedPerInstanceMemory` for multi-instance model reference, or AUTOSAR `StaticMemory` for single-instance model reference.

Configure Data Stores as AUTOSAR Static Memory

To generate `StaticMemory` blocks for Simulink data store memory data in your AUTOSAR model, open Code Mappings editor. Use the **Data Stores** tabs to map data stores to `StaticMemory`. For example:

1. Open an AUTOSAR model that contains data stores for which you want to generate `StaticMemory` blocks. This example uses model `autosar_bsw_sensor1`.
2. In AUTOSAR code perspective, open 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 and displays the data store attributes in Property Inspector. Use Property Inspector to modify the data store attributes. In the **Mapped To** drop-down list, select `StaticMemory`. For more information about data store code and calibration attributes, see “Map Data Stores to AUTOSAR Variables” on page 4-81.



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 The software does not support static memory code generation for data stores in referenced models.

See Also

Data Store Memory | `getDataStore` | `getSignal` | `getState` | `mapDataStore` | `mapSignal` | `mapState`

Related Examples

- “Map Block Signals and States to AUTOSAR Variables” on page 4-79
- “Map Data Stores to AUTOSAR Variables” on page 4-81
- “Model AUTOSAR Component Behavior” on page 2-38

More About

- “Static and Constant Memory” on page 2-42

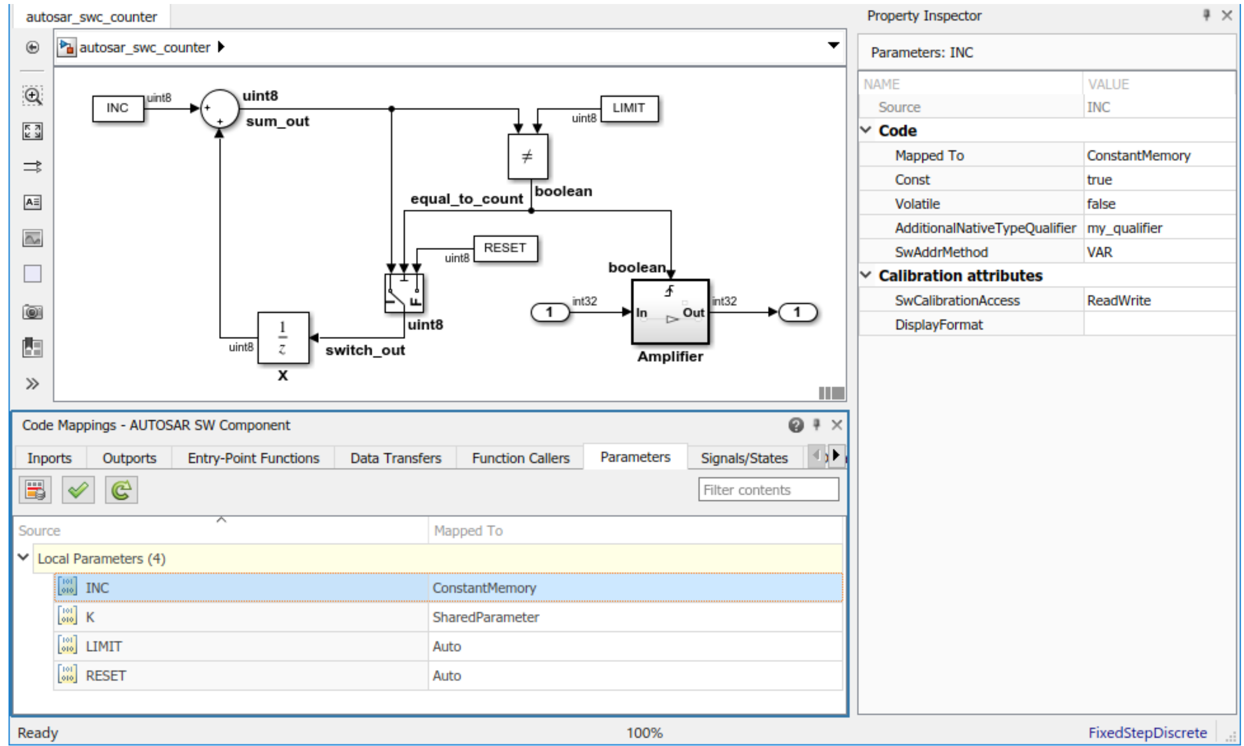
Configure AUTOSAR Constant Memory

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 2-42.

AUTOSAR constant memory (`ConstantMemory`), introduced in AUTOSAR schema version 4.0, corresponds to Simulink internal global parameters. In the AUTOSAR Runtime Environment (RTE), calibration tools can access `ConstantMemory` blocks for measurement and calibration.

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 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 AUTOSAR code perspective, open Code Mappings editor and select the **Parameters** tab. In the list of available parameters, select `INC`. Selecting a parameter displays the parameter attributes in Property Inspector. Use Property Inspector to modify the parameter attributes. In the **Mapped To** drop-down list, select `ConstantMemory`. For more information about parameter code and calibration attributes, see “Map Model Workspace Parameters to AUTOSAR Component Internal Parameters” on page 4-75.



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 Internal Parameters” on page 4-75
- “Model AUTOSAR Component Behavior” on page 2-38

More About

- "Static and Constant Memory" on page 2-42

Configure AUTOSAR Shared or Per-Instance Parameters

In this section...
"Configure Model Workspace Parameters as AUTOSAR Shared Parameters" on page 4-308
"Configure Model Workspace Parameters as AUTOSAR Per-Instance Parameters" on page 4-310

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 measurement and calibration.

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 2-42 .

To model AUTOSAR parameters in Simulink, you use model workspace parameters.

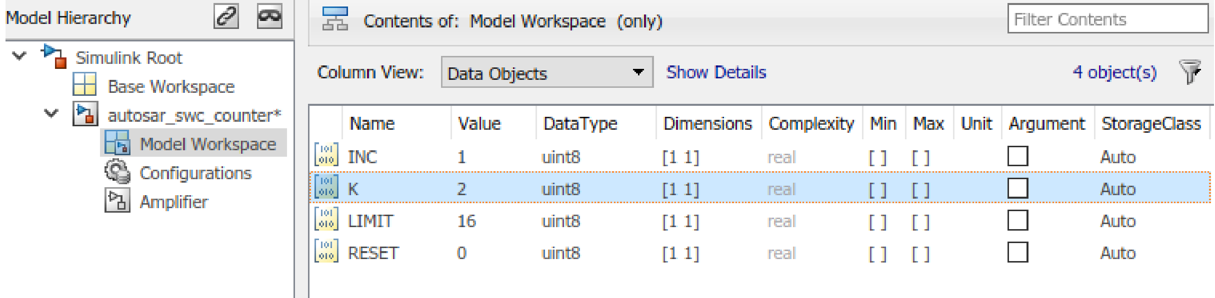
In this section...
"Configure Model Workspace Parameters as AUTOSAR Shared Parameters" on page 4-308
"Configure Model Workspace Parameters as AUTOSAR Per-Instance Parameters" on page 4-310

Configure Model Workspace Parameters as AUTOSAR Shared Parameters

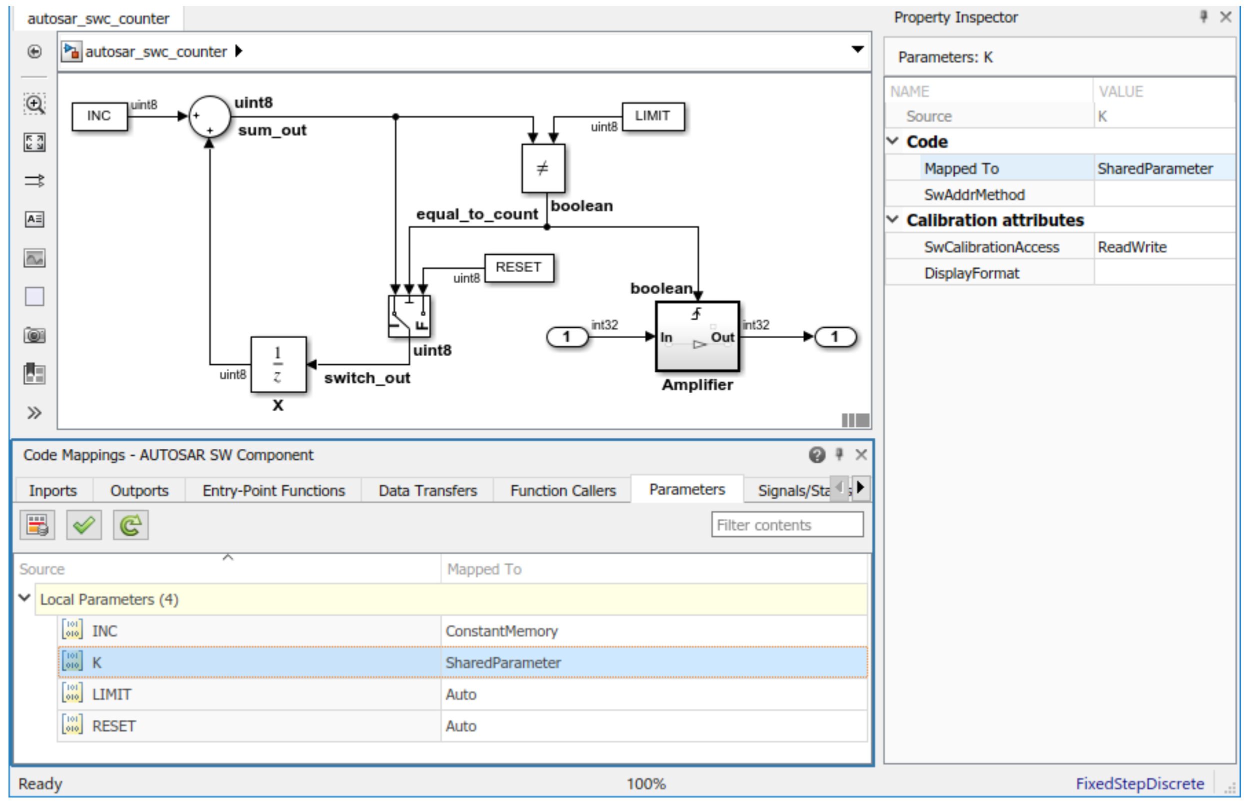
To model AUTOSAR shared parameters in Simulink:

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_swc_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_swc_counter`, clear the **Argument** property for parameter K. Leave the parameter **StorageClass** set to `Auto`.



3 In AUTOSAR code perspective, open Code Mappings editor and select the **Parameters** tab. In the list of available parameters, select `K`. Selecting a parameter displays the parameter attributes in Property Inspector. Use Property Inspector to modify the parameter attributes. In the **Mapped To** drop-down list, select parameter type `SharedParameter`. For more information about parameter code and calibration attributes, see “Map Model Workspace Parameters to AUTOSAR Component Internal Parameters” on page 4-75.



When you generate code:

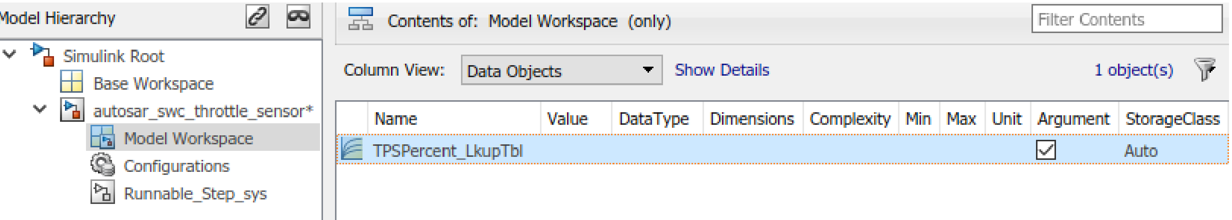
- Exported `arxml` files contain `SHARED-PARAMETERS` descriptions for parameters that you configured as `SharedParameter`.
- Generated C code contains `Rte_CData` calls where shared parameters are used.

```
autosar_swc_counter_B.Gain = Rte_CData_K() *
  Rte_IRead_Runnable_Step_RPort_InData();
```

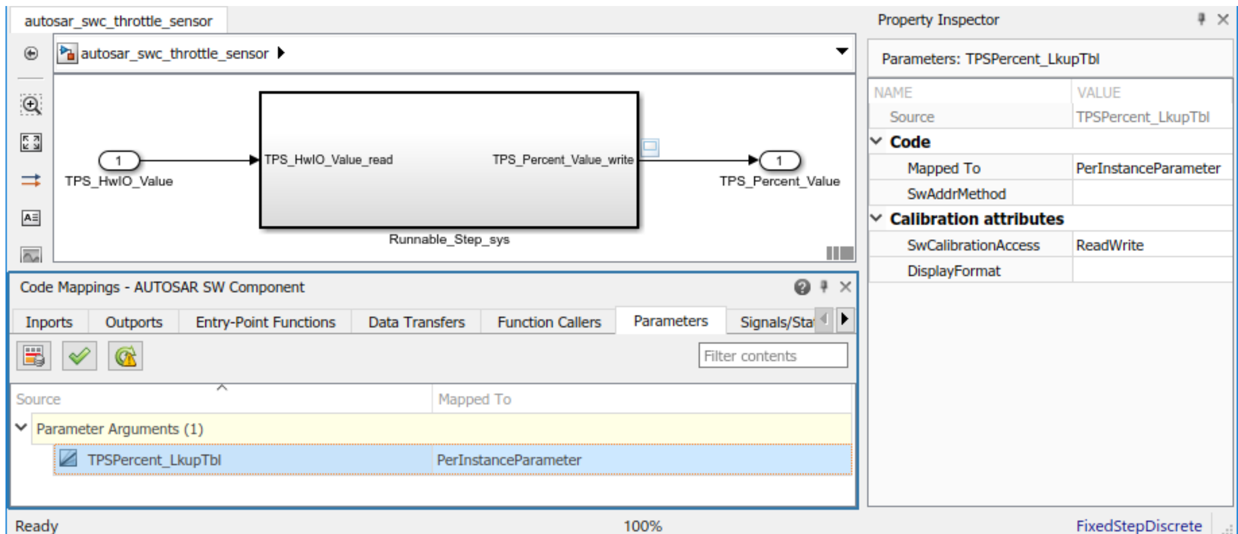
Configure Model Workspace Parameters as AUTOSAR Per-Instance Parameters

To model AUTOSAR per-instance parameters in Simulink:

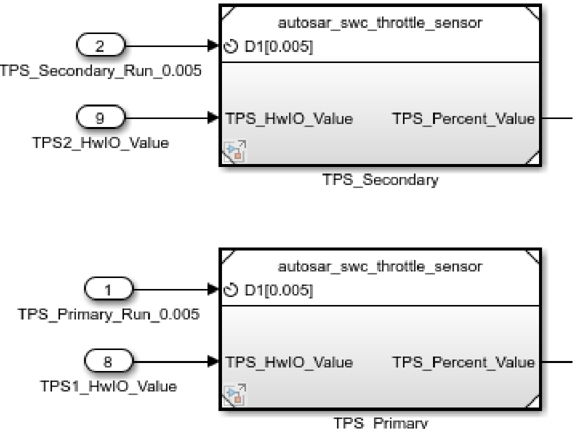
1. Open an AUTOSAR model that contains a model workspace parameter for which you want to generate an AUTOSAR `PerInstanceParameter`. This example uses model `autosar_swc_throttle_sensor`. This model is part of AUTOSAR composition model `autosar_composition`, which contains two instances of `autosar_swc_throttle_sensor`.
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`.



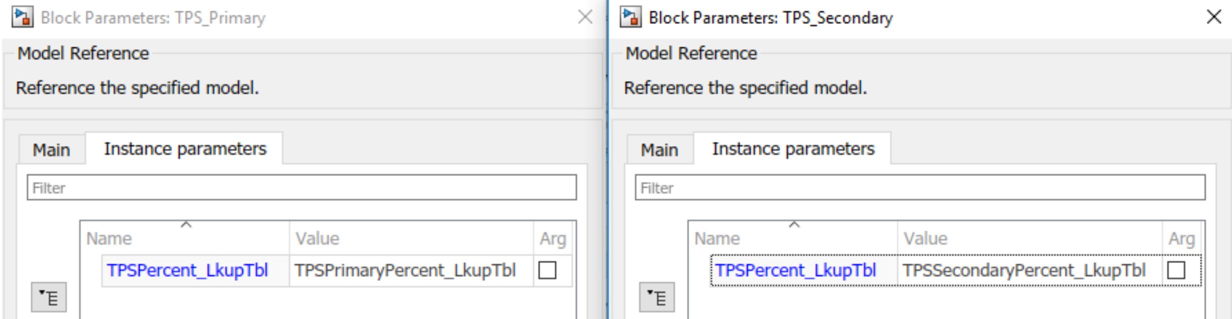
3. In AUTOSAR code perspective, open Code Mappings editor and select the **Parameters** tab. Select parameter `TPSPercent_LkupTbl`. Selecting a parameter displays the parameter attributes in Property Inspector. Use Property Inspector to modify the parameter attributes. In the **Mapped To** drop-down list, select parameter type `PerInstanceParameter`. For more information about parameter code and calibration attributes, see “Map Model Workspace Parameters to AUTOSAR Component Internal Parameters” on page 4-75.



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_IWrite_Runnable_Step_TPS_Percent_Value(self, look1_iflf_linlcpw((float32)
  rtb_DataTypeConversion, (Rte_CData_TPSPercent_LkupTbl(self))->BP1,
  (Rte_CData_TPSPercent_LkupTbl(self))->Table, 10U));
```

See Also

`getParameter` | `mapParameter`

Related Examples

- “Map Model Workspace Parameters to AUTOSAR Component Internal Parameters” on page 4-75
- “Model AUTOSAR Component Behavior” on page 2-38

More About

- “Shared and Per-Instance Parameters” on page 2-42

Configure AUTOSAR Release 4.x Data Types

AUTOSAR Release 4.0 introduced a new 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 R4.x data types, see "Release 4.x Data Types" on page 2-55.

The software supports AUTOSAR R4.x 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 R4.x 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.

In this section...
"Control Application Data Type Generation" on page 4-314
"Configure DataTypeMappingSet Package and Name" on page 4-315
"Initialize Data with ApplicationValueSpecification" on page 4-317

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 date 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 **ImplementationDataType Reference** (`XMLOptions` property `ImplementationDataTypeReference`), 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.

To set the `ImplementationDataTypeReference` property in the MATLAB Command Window, use an AUTOSAR property `set` function call similar to the following:

```
hModel = 'autosar_swc_expfcns';
open_system(hModel);
arProps=autosar.api.getAUTOSARProperties(hModel);
set(arProps,'XmlOptions','ImplementationTypeReference','NotAllowed');
get(arProps,'XmlOptions','ImplementationTypeReference')
```

To set the `ImplementationDataTypeReference` property in AUTOSAR Dictionary, select **XML Options**. Select a value for parameter **ImplementationDataType Reference**. 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 AUTOSAR Dictionary or the AUTOSAR property `set` function.

```
hModel = 'autosar_swc_expfcns';
open_system(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 UUID="...">
    <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_expfcns';
open_system(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','ComponentQualifiedName');
ib = get(propObj,swc,'Behavior','PathType','FullyQualified');
set(propObj,ib,'DataTypeMapping',newMappingSet);

% Force generation of application data types
set(propObj,'XmlOptions','ImplementationTypeReference','NotAllowed');

% Build
rtwbuild(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, R4.1 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.
- If you select AUTOSAR schema 4.0 or later for a model that contains AUTOSAR parameters typed by application data type, 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`.

Automatic AUTOSAR Data Type Generation

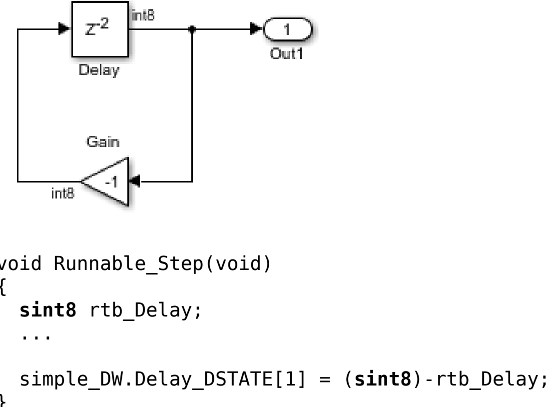
When you generate AUTOSAR-compliant C code for an AUTOSAR component model, Embedded Coder generates AUTOSAR platform data types in the code. AUTOSAR type generation allows you to generate AUTOSAR platform data types for top models, referenced models, and shared utilities without configuring Simulink 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 2-52.

When you build your AUTOSAR model, C code generation replaces Simulink data types with corresponding AUTOSAR data types, based on your AUTOSAR schema version.

Simulink Data Type	AUTOSAR Data Type	
	R4.x Platform Type	R2.x/3.x Primitive Type
boolean	boolean	Boolean
single	float32	Float
double	float64	Double
int8	sint8	Sint8
int16	sint16	Sint16
int32	sint32	Sint32
uint8	uint8	Uint8
uint16	uint16	Uint16
uint32	uint32	Uint32

For example, suppose that you create a simple AUTOSAR model containing Gain and Delay blocks, set the AUTOSAR schema version to 4.0 or higher, 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

More About

- “Model AUTOSAR Data Types” on page 2-52

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 measurement and calibration.

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.

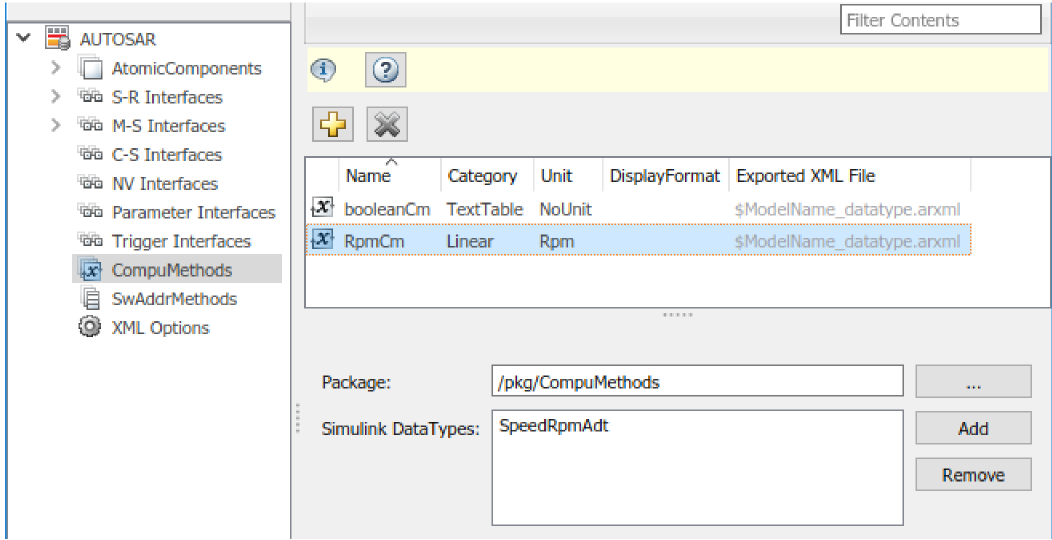
In this section...

"Configure AUTOSAR CompuMethod Properties" on page 4-320
"Create AUTOSAR CompuMethods" on page 4-322
"Configure CompuMethod Direction for Linear Functions" on page 4-323
"Export CompuMethod Unit References" on page 4-325
"Modify Linear Scaling for SCALE_LINEAR_AND_TEXTTABLE CompuMethod" on page 4-325
"Configure Rational Function CompuMethod for Dual-Scaled Parameter" on page 4-327

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 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.

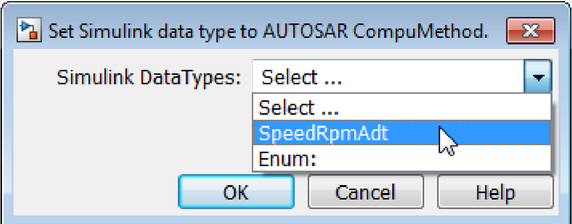
- **Name** — Specify name text
- **Category** — Select `Identical`, `Linear`, `RatFunc`, `TextTable`, or `LinearAndTextTable` (see "CompuMethod Categories for Data Types" on page 2-59)
- **Unit** — Select from units available in the model
- **DisplayFormat** — Optionally specify format to be used by measurement and calibration 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 4-266.
- **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 4-150.

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**.

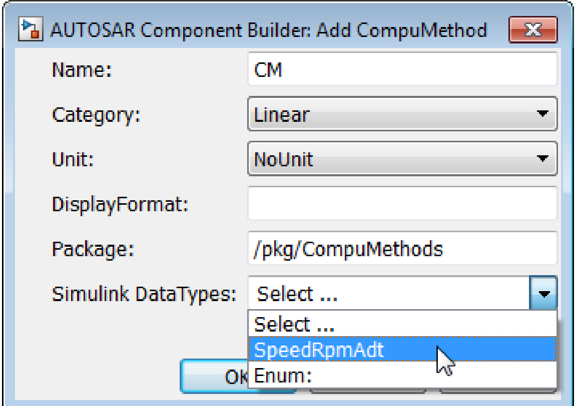


To set the **Simulink DataTypes** property programmatically, open the model and use an AUTOSAR property `set` function call similar to the following:

```
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 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 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

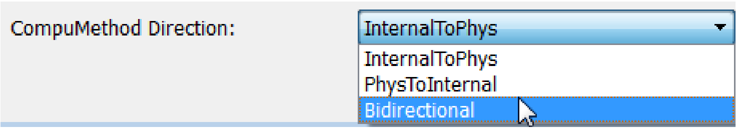
For designs originated in Simulink, you can control properties for an exported `CompuMethod`, including the direction of `CompuMethod` conversion between internal and physical representations of a value. Using either AUTOSAR Dictionary or the AUTOSAR property `set` function, you can specify one of the 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_swc_expfcns';
open_system(hModel);
arProps=autosar.api.getAUTOSARProperties(hModel);
set(arProps,'XmlOptions','CompuMethodDirection','Bidirectional');
get(arProps,'XmlOptions','CompuMethodDirection')
```

To specify CompuMethod direction in 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 measurement and calibration tool use of exported AUTOSAR data.

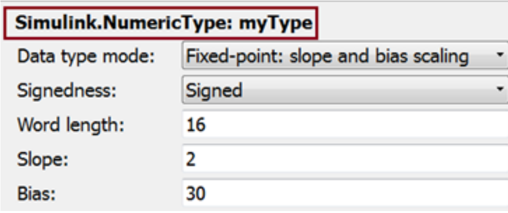
Modify Linear Scaling for SCALE_LINEAR_AND_TEXTTABLE CompuMethod

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.



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
```

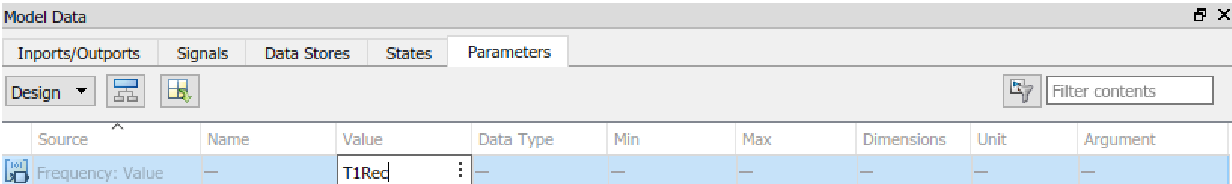
```
>> get(arProps,'/simple_ar_package/simple_ar_dt/COMPU_myType','CellOfEnums')
ans =
    'SensorError'    'SignalNotAvailable'
>> % Get internal values for COMPU_myType TEXTTABLE scales
>> get(arProps,'/simple_ar_package/simple_ar_dt/COMPU_myType','IntValues')
ans =
   350   351
```

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 outport.
2. Open the Model Data Editor (**View > 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`.

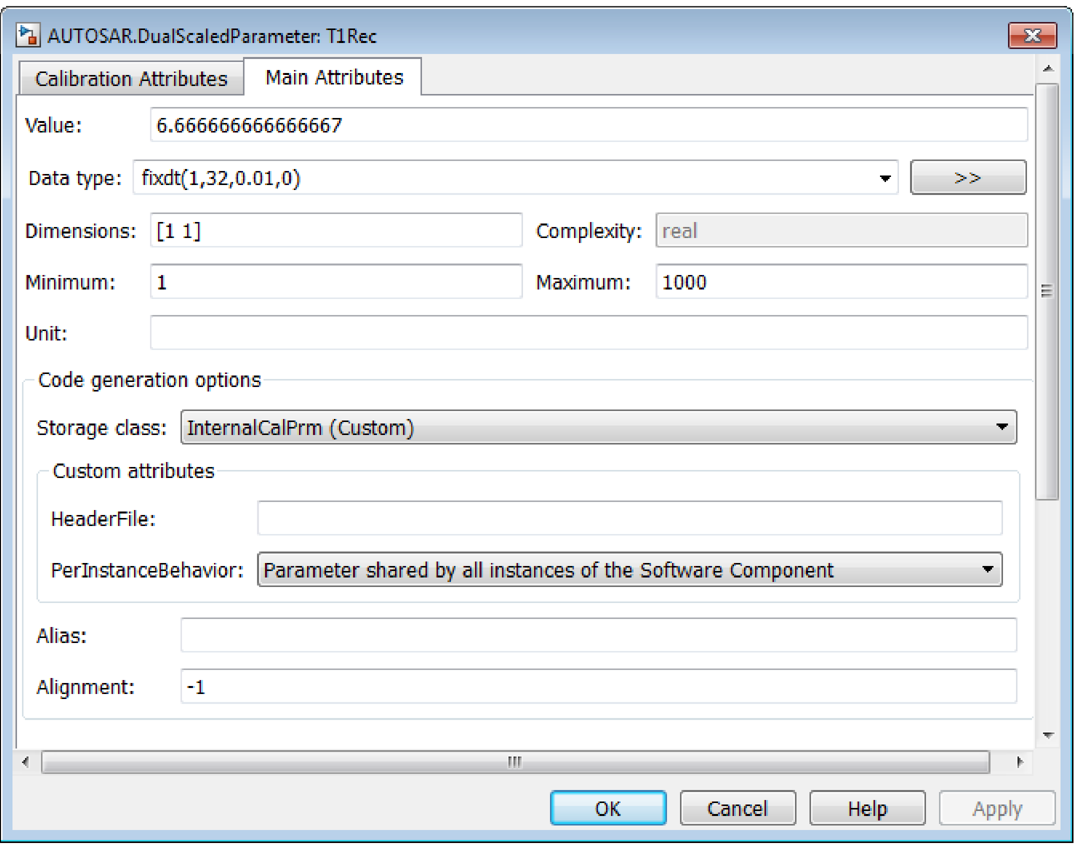


3. Create the `T1Rec` data object. In the Model Data Editor, to the right of the value `T1Rec`, click the action button ⋮ and select **Create**.

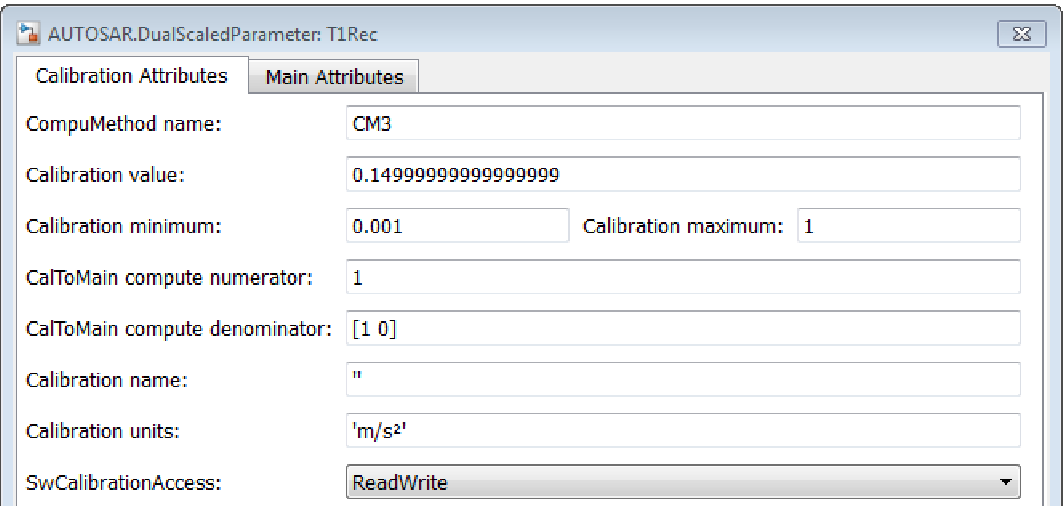
   In the Create New Data dialog box, set **Value** to `AUTOSAR.DualScaledParameter` and click **Create**. An `AUTOSAR.DualScaledParameter` data object appears in the base workspace. The dual-scaled parameter property dialog box opens.
4. Configure the attributes of the dual-scaled parameter `T1Rec`. Execute 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';
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.Alias = '';
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/s²';
```

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. The attributes include **CompuMethod name** (`T1Rec.CompuMethodName`), which allows you to specify the name of the AUTOSAR `CompuMethod` for this data type.



7 If `CompuMethod` direction is not already set to bidirectional in the AUTOSAR properties, use 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/ApplDataTypes/
                                 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/ApplDataTypes/
                                                             T1Rec_DualScaled</TYPE-TREF>
  ...
</PARAMETER-DATA-PROTOTYPE>
```

See Also

Related Examples

- “Import AUTOSAR Software Component” on page 3-27

- “Configure AUTOSAR Package for Component, Interface, CompuMethod, or SwAddrMethod” on page 4-150
- “Configure AUTOSAR XML Options” on page 4-62
- “Configure AUTOSAR Code Generation” on page 5-12

More About

- “CompuMethod Categories for Data Types” on page 2-59
- “AUTOSAR Component Configuration” on page 4-3

Configure AUTOSAR Internal Data Constraints Export

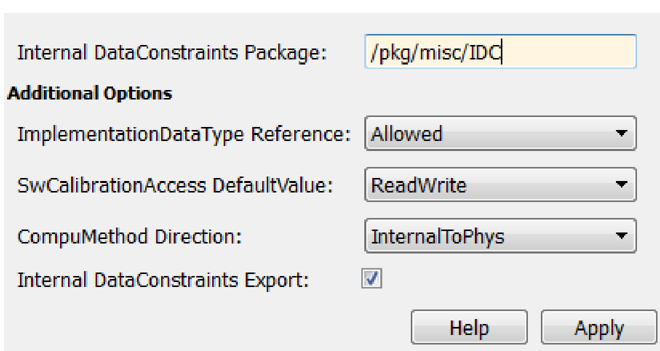
AUTOSAR applications use data constraints to implement limits on data types and provide a controlled range of possible values. Internal data constraints represent minimum and maximum values for implementation data types, reflecting the internal or machine view of the data.

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 AUTOSAR Dictionary. Select **Code > C/C++ Code > Configure 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**.



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 UUID="...">
    <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 4-62.

See Also

Related Examples

- “Configure AUTOSAR Release 4.x Data Types” on page 4-314
- “Model AUTOSAR Data Types” on page 2-52
- “Configure AUTOSAR Code Generation” on page 5-12

More About

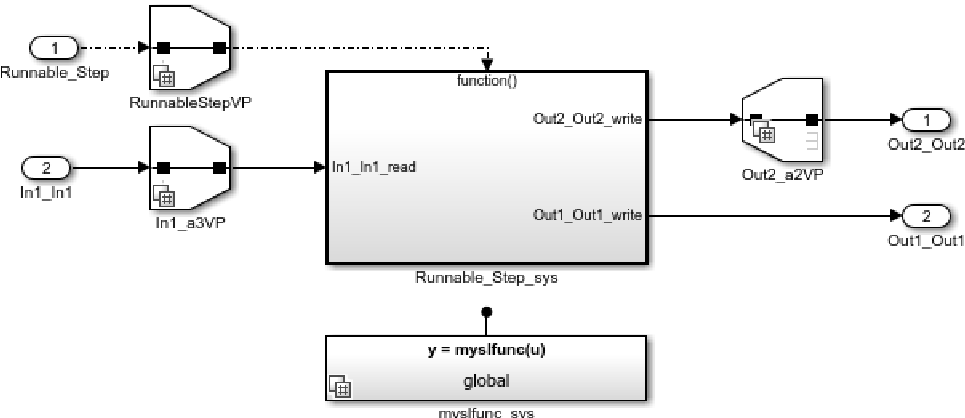
- “Release 4.x Data Types” on page 2-55
- “AUTOSAR Component Configuration” on page 4-3

Configure Variants for AUTOSAR 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, 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.

For example, here is a model that contains two Variant Source blocks and a Variant Sink block. You can open the model from *`matlabroot`*`/help/toolbox/autosar/examples/mAutosarInlineVariant.slx`.

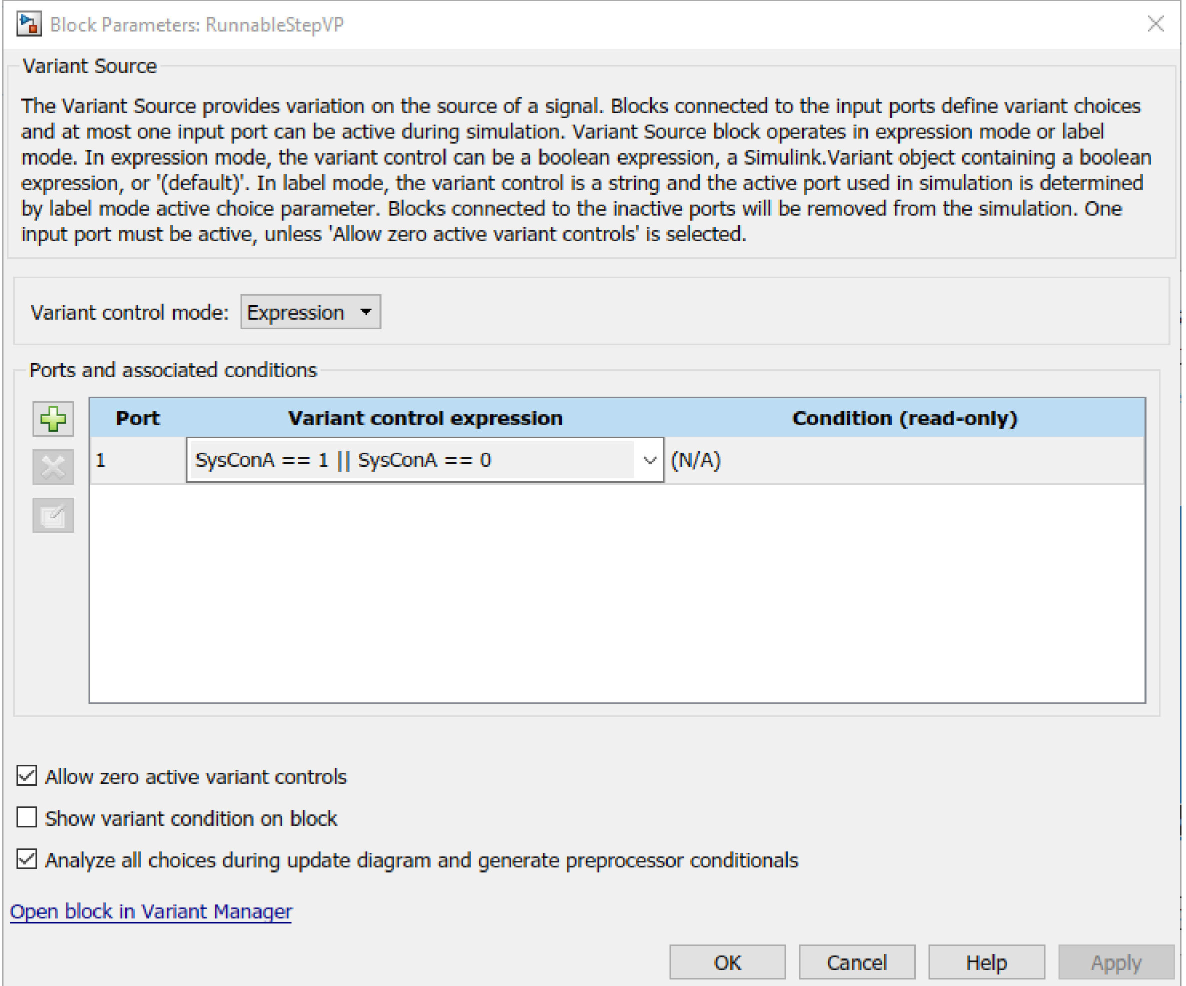


To model an AUTOSAR system constant, the model defines `AUTOSAR.Parameter` data object `SysConA`:

```
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 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/SysConA</
          == 0 ||
          <SYSC-REF DEST="SW-SYSTEMCONST">/mInlineVariant_pkg/mInlineVariant_dt/SystemConstants/SysConA</
          == 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/SysConA</
          == 0 ||
          <SYSC-REF DEST="SW-SYSTEMCONST">/mInlineVariant_pkg/mInlineVariant_dt/SystemConstants/SysConA</
          == 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) and “Variant Systems” (Simulink).

See Also

`AUTOSAR.Parameter` | Variant Sink | Variant Source

Related Examples

- “Model AUTOSAR Variants” on page 2-44

More About

- “Variant Systems” (Embedded Coder)
- “System Constants” on page 2-40

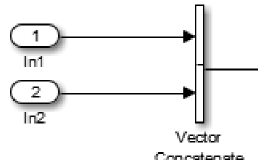
Configure Variants for AUTOSAR Array Sizes

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 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`.

```
SymDimA = AUTOSAR.Parameter;
SymDimA.CoderInfo.StorageClass = 'custom';
SymDimA.CoderInfo.CustomStorageClass = 'SystemConstant';
SymDimA.DataType = 'uint8';
SymDimA.Value = 5;
```

In the dialog box for inport block `In1`, enter the parameter name, `SymDimA`, in the **Port dimensions** field.

Port dimensions (-1 for inherited):

SymDimA

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.

Example 4.3. Generated C Code

```
/* SignalConversion: '<Root>/ConcatBufferAtVector ConcatenateIn1' */
  for (i = 0; i < Rte_SysCon_SymDimA; i++) {
    rtb_VectorConcatenate[i] = tmp[i];
```

Example 4.4. Exported `arxml` Code

```
<MAX-NUMBER-OF-ELEMENTS BINDING-TIME="PRE-COMPILE-TIME">
    <SYSC-REF DEST="SW-SYSTEMCONST">/pkg/dt/SC/SymDimA</SYSC-REF>
</MAX-NUMBER-OF-ELEMENTS>
```

See Also

`AUTOSAR.Parameter`

Related Examples

- “Implement Dimension Variants for Array Sizes in Generated Code” (Embedded Coder)
- “Model AUTOSAR Variants” on page 2-44

More About

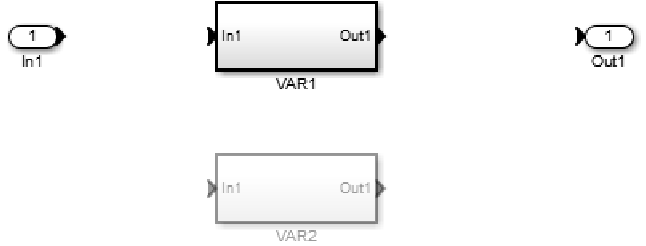
- “System Constants” on page 2-40

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:

- Use a Variant Subsystem block to represent variant implementations of a subsystem or model and define variant condition logic.
- Use `AUTOSAR.Parameter` data objects with storage class `SystemConstant` to model AUTOSAR system constants. The system constants represent the condition values that determine the active subsystem or model implementation.

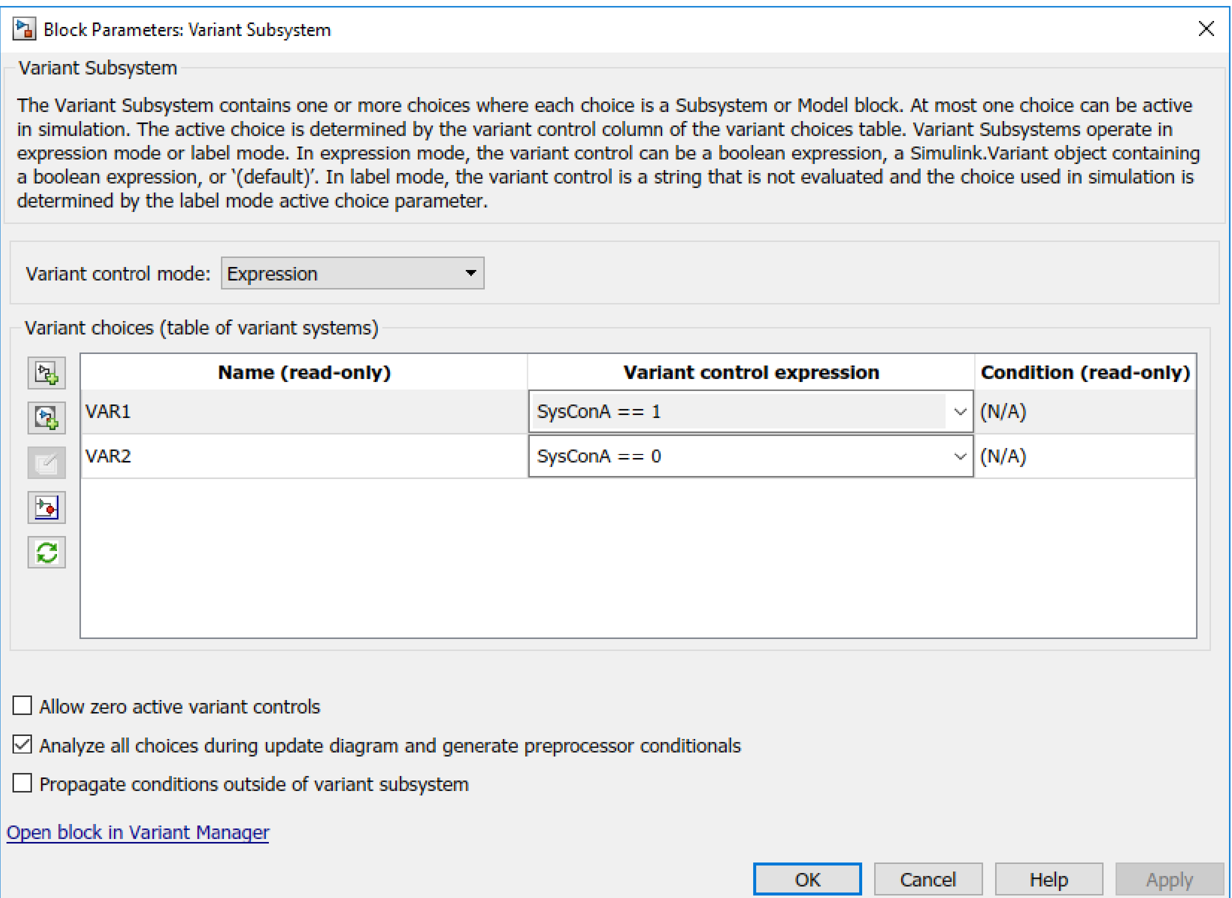
For example, suppose that you implement a Variant Subsystem block. The variant choices are subsystems VAR1 and VAR2. The blocks are not connected because connectivity is determined during simulation, based on the active variant.



To model an AUTOSAR system constant, define `AUTOSAR.Parameter` data object `SysConA`:

```
SysConA = AUTOSAR.Parameter;
SysConA.CoderInfo.StorageClass = 'Custom';
SysConA.CoderInfo.CustomStorageClass = 'SystemConstant';
SysConA.DataType = 'uint8';
SysConA.Value = 1;
```

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. SysConA 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">/basic_pkg/basic_dt/SystemConstants/SysConA</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">/basic_pkg/basic_dt/SystemConstants/SysConA</SYSC-REF>
          == 0</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

`AUTOSAR.Parameter` | Variant Subsystem

Related Examples

- “Model AUTOSAR Variants” on page 2-44

More About

- “System Constants” on page 2-40

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 `SwSystemconsts` 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 `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.

Typical steps include:

1. Get a list of the `PredefinedVariants` or `SwSystemconstantValueSets` 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 SwSystemconstantValueSets.

   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 SwSystemconstantValueSets A and B, which together provide values for SwSystemconsts in the AUTOSAR software component.

```
>> createComponentAsModel(obj,compNames{1},'ModelPeriodicRunnablesAs','AtomicSubsystem',...
   'SystemConstValueSets',{'/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 SwSystemconstantValueSets.

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 2-44
- “Configure Variants for AUTOSAR Ports and Runnables” on page 4-335

More About

- “System Constants” on page 2-40

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.

In this section...
“AUTOSAR Property and Map Functions” on page 4-347
“Tree View of AUTOSAR Configuration” on page 4-348
“Properties of AUTOSAR Elements” on page 4-349
“Specify AUTOSAR Element Location” on page 4-353

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 4-355.

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	• 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	• Runnables (add/delete) • Events (add/delete) • PIM (add/delete) • IRV (add/delete) • Parameters (add/delete) • IncludedDataTypeSets • DataTypeMapping • Name

AUTOSAR Element Class	Properties
DataReceiverPort DataSenderPort DataSenderReceiverPort ClientPort ServerPort ModeReceiverPort NvDataReceiverPort NvDataSenderPort NvDataSenderReceiverPort ParameterReceiverPort TriggerReceiverPort	• Interface • Name
Runnable	• symbol • canBeInvokedConcurrently • SwAddrMethod • Name
TimingEvent	• Period • StartOnEvent • DisabledMode • Name
DataReceivedEvent DataReceiveErrorEvent OperationInvokedEvent	• Trigger • StartOnEvent • DisabledMode • Name
ModeSwitchEvent	• Trigger • Activation • StartOnEvent • DisabledMode • Name
InitEvent	• StartOnEvent • Name

AUTOSAR Element Class	Properties
IrvData	• Type • SwAddrMethod • SwCalibrationAccess • DisplayFormat • SwAlignment • Name
ParameterData	• Type • SwAddrMethod • SwCalibrationAccess • DisplayFormat • SwAlignment • Kind • Name
SenderReceiverInterface NvDataInterface ParameterInterface	• DataElements (add/delete) • IsService • Name
FlowData	• Type • SwAddrMethod • SwCalibrationAccess • DisplayFormat • SwAlignment • Name
ModeSwitchInterface	• ModeGroup (add/delete) • IsService • Name
ModeDeclarationGroupElement	• ModeGroup • SwCalibrationAccess • Name

AUTOSAR Element Class	Properties
ClientServerInterface	• Operations (add/delete) • IsService • Name
TriggerInterface	• 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
```

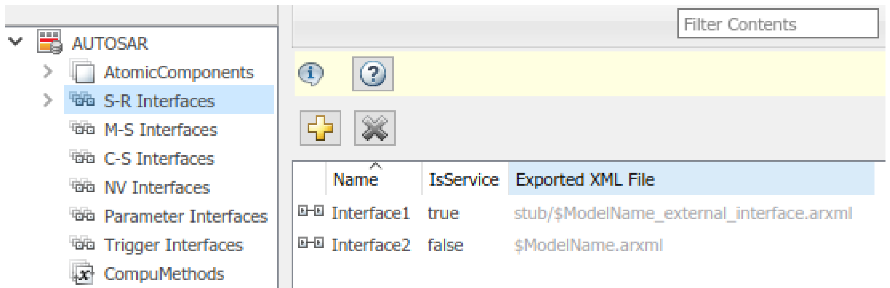
For AUTOSAR property functions other than `addPackageableElement`, you can specify a partially-qualified path that does not include the package hierarchy, for example:

```
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_expfcns` to `true`. In this case, specifying the name `Interface1` is enough to locate the element.

```
hModel = 'autosar_swc_expfcns';
open_system(hModel);
arProps = autosar.api.getAUTOSARProperties(hModel);
set(arProps,'Interface1','IsService',true);
```

Here is the resulting display in the **S-R Interfaces** view in AUTOSAR Dictionary.



If you added a Sender-Receiver Interface to a component package, you would specify a fully qualified path, for example:

```
hModel = 'autosar_swc_expfcns';
open_system(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 AUTOSAR Dictionary and 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 4-356
   a. "Configure AUTOSAR Software Component Name and Type" on page 4-356
   b. "Configure AUTOSAR Ports" on page 4-357
   c. "Configure AUTOSAR Runnables and Events" on page 4-361
   d. "Configure AUTOSAR Inter-Runnable Variables" on page 4-366
2. "Configure AUTOSAR Interfaces" on page 4-368
   a. "Configure AUTOSAR Sender-Receiver Interfaces" on page 4-369
   b. "Configure AUTOSAR Client-Server Interfaces" on page 4-371
   c. "Configure AUTOSAR Mode-Switch Interfaces" on page 4-374
3. "Configure AUTOSAR XML Export" on page 4-376

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','ComponentQualifiedName');
```

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','ComponentQualifiedName');
```

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 data transfers, entry-point functions, 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 Code Mappings editor.

Configure AUTOSAR Software Component

- "Configure AUTOSAR Software Component Name and Type" on page 4-356
- "Configure AUTOSAR Ports" on page 4-357
- "Configure AUTOSAR Runnables and Events" on page 4-361
- "Configure AUTOSAR Inter-Runnable Variables" on page 4-366

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_swc_expfcns';
open_system(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 4-357
- “Configure Client-Server Ports” on page 4-359
- “Configure and Map Mode Receiver Ports” on page 4-359

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 4-369.

```
% Open model
hModel = 'autosar_swc_expfcns';
open_system(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 Simulink inport
[arPortName,arDataElementName,arDataAccessMode]=getInport(slMap,'RPort_DE2')

arPortName =
    'RPort'

arDataElementName =
  0×0 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

This example:

1. Opens a model.
2. Finds AUTOSAR client or server ports.
3. Loops through the ports and lists associated client-server interfaces.
4. Modifies the associated interface for a port.

See also “Configure AUTOSAR Client-Server Interfaces” on page 4-371.

```
% Open model
hModel = 'mControllerWithInterface_server';
addpath (fullfile(matlabroot,'/help/toolbox/autosar/examples'));
open_system(hModel);

% Use AUTOSAR property functions
arProps = autosar.api.getAUTOSARProperties(hModel);

% Find AUTOSAR ports - specify ServerPort or ClientPort
arPortType = 'ServerPort';
aswcPath = find(arProps,[],'AtomicComponent','PathType','FullyQualified');
sPorts=find(arProps,aswcPath{1},arPortType,'PathType','FullyQualified');

% Loop through ports and list their associated interfaces
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 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

This example:

1. Opens a model.
2. Finds AUTOSAR mode receiver ports.
3. Loops through the ports and lists associated mode-switch interfaces.

4 Modifies the associated interface for a port.
5 Maps a Simulink inport to an AUTOSAR mode receiver port.

See also “Configure AUTOSAR Mode-Switch Interfaces” on page 4-374.

```
% Add path to model and mode definition files and open model
addpath (fullfile(matlabroot,'/help/toolbox/autosar/examples'));
hModel = 'mAutosarMsConfigAfter';
open_system(hModel);

% Use 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 =
  0×0 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 4-361
- "Configure and Map Runnables" on page 4-363
- "Configure Events for Runnable Activation" on page 4-364
- "Gather Information for AUTOSAR Custom Scheduler Script" on page 4-365

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_expfcns';
open_system(hModel);

% Use AUTOSAR property functions
arProps = autosar.api.getAUTOSARProperties(hModel);

% Find AUTOSAR runnables
swc = get(arProps,'XmlOptions','ComponentQualifiedName');
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};
    eventStartOn = get(arProps,event,'StartOnEvent');
    fprintf('AUTOSAR event %s triggers %s\n',event,eventStartOn);
end

AUTOSAR event /pkg/swc/ASWC/IB/Event_t_1tic_A triggers ASWC/IB/myRunnable
AUTOSAR event /pkg/swc/ASWC/IB/Event_t_1tic_B triggers ASWC/IB/Runnable2
AUTOSAR event /pkg/swc/ASWC/IB/Event_t_10tic triggers ASWC/IB/Runnable3

% Modify AUTOSAR event name
set(arProps,events{1},'Name','myEvent');
events = find(arProps,ib,'TimingEvent','PathType','FullyQualified');
eventStartOn = get(arProps,events{1},'StartOnEvent');
fprintf('AUTOSAR event %s triggers %s\n',events{1},eventStartOn);

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 4-364.

```
% Open model
hModel = 'autosar_swc_counter';
open_system(hModel);

% Use AUTOSAR property functions
arProps = autosar.api.getAUTOSARProperties(hModel);

% Add AUTOSAR initialization and periodic runnables
initRunnable = 'myInitRunnable';
periodicRunnable = 'myPeriodicRunnable';
swc = get(arProps,'XmlOptions','ComponentQualifiedName')
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'}
```

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 4-371.

For `ModeSwitchEvent` syntax, see "Configure AUTOSAR Mode-Switch Interfaces" on page 4-374.

```
% Open model
hModel = 'autosar_swc_expfcns';
open_system(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 =
    {'ASWC/IB/Runnable_Init'}    {'ASWC/IB/Runnable1'}
    {'ASWC/IB/Runnable2'}    {'ASWC/IB/Runnable3'}

events =
    {'ASWC/IB/Event_t_1tic_B'}    {'ASWC/IB/Event_t_10tic'}    {'ASWC/IB/myTimingEvent'}
    {'ASWC/IB/myDREvent'}    {'ASWC/IB/myDREEvent'}
```

Gather Information for AUTOSAR Custom Scheduler Script

This example:

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.

```
% Example of how to extract timing information for runnables
% to prepare for hooking up a custom scheduler

% Use AUTOSAR property functions
arProps = autosar.api.getAUTOSARProperties(hModel);

swc = get(arProps,'XmlOptions','ComponentQualifiedName');

% Get AUTOSAR internal behavior
ib = get(arProps,swc,'Behavior');

% Get AUTOSAR events and runnables
events = get(arProps,ib,'Events');
runnables = get(arProps,ib,'Runnables');

% Loop through events
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

% Loop through runnables
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
```

Running the example code on the example model `autosar_swc_expfcns` generates the following output:

```
Event Event_t_1tic_A triggers runnable Runnable1 with period 1
Event Event_t_1tic_B triggers runnable Runnable2 with period 1
Event Event_t_10tic 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 *`matlabroot`*`/help/toolbox/autosar/examples/mMultitasking_4rates.slx` 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 4-366
- "Configure Inter-Runnable Variable for Data Transfer with Rate Transition" on page 4-367

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_expfcns';
```

```
open_system(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','ComponentQualifiedName')
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);

% 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 model
hModel = 'mMultitasking_4rates';
addpath (fullfile(matlabroot,'/help/toolbox/autosar/examples'));
```

```
open_system(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','ComponentQualifiedName')
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);

% Map Simulink RT 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')

% To pass validation, remove redundant IRV in AUTOSAR configuration
irvs = get(arProps,ib,'IRV');
delete(arProps,[ib,'/IRV3'])
irvs = get(arProps,ib,'IRV')

swc =
    '/mMultitasking_4rates_pkg/mMultitasking_4rates_swc/mMultitasking_4rates'

ib =
    'mMultitasking_4rates/Behavior'

irvs =
    {'mMultitasking_4rates/Behavior/IRV1'}    {'mMultitasking_4rates/Behavior/IRV2'}
    {'mMultitasking_4rates/Behavior/IRV3'}

arIrvName =
    'myIrv'

arDataAccessMode =
    'Explicit'

irvs =
    {'mMultitasking_4rates/Behavior/IRV1'}    {'mMultitasking_4rates/Behavior/IRV2'}
    {'mMultitasking_4rates/Behavior/myIrv'}
```

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) — introduced in AUTOSAR Release 4.0

- Nonvolatile (NV) data — introduced in AUTOSAR Release 4.0

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 4-369
- “Configure AUTOSAR Client-Server Interfaces” on page 4-371
- “Configure AUTOSAR Mode-Switch Interfaces” on page 4-374

Configure AUTOSAR Sender-Receiver Interfaces

- “Configure and Map Sender-Receiver Interface” on page 4-369
- “Configure Sender-Receiver Data Element Properties” on page 4-370

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 4-361.

```
% Open model
hModel = 'autosar_swc_expfcns';
open_system(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(slMap,'RPort_DE2',rPortName,de2,rDataAccessMode);
[arPortName,arDataElementName,arDataAccessMode]=getInport(slMap,'RPort_DE2')

% Map Simulink outport PPort_DE1 to AUTOSAR sender port myPPort and data element myDE1
sDataAccessMode = 'ImplicitSend';
[arPortName,arDataElementName,arDataAccessMode]=getOutport(slMap,'PPort_DE1')
mapOutport(slMap,'PPort_DE1',pPortName,de1,sDataAccessMode);
[arPortName,arDataElementName,arDataAccessMode]=getOutport(slMap,'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_expfcns';
open_system(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 4-371
- “Configure Client Properties” on page 4-373

Configure Server Properties

This example:

1. Opens a model.
2. Adds an AUTOSAR client-server interface to the model.
3. Adds an operation.
4. Creates a server port.
5. Creates a server runnable.
6. Maps a Simulink function to the AUTOSAR server runnable.

```
% Open model
hModel = 'mControllerWithInterface_server';
addpath (fullfile(matlabroot,'/help/toolbox/autosar/examples'));
```

```
open_system(hModel);

% Use 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');

% 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'}
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')

% 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')
ib = get(arProps,swc,'Behavior')
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);
arRunnableName=getFunction(slMap,'readData')

ifPkg =
    '/ControllerWithInterface_ar_pkg/ControllerWithInterface_ar_if'

args =
    {'Op'     }    {'In' }
    {'Data'   }    {'Out'}
    {'ERR'    }    {'Out'}
    {'NegCode'}    {'Out'}

ans =
    {'myCsIf/readData/Op'}    {'myCsIf/readData/Data'}
```

```
    {'myCsIf/readData/ERR'}    {'myCsIf/readData/NegCode'}

swc =
    '/pkg/swc/SWC_Controller'

ib =
    'SWC_Controller/ControllerWithInterface_ar'

arRunnableName =
    'Runnable_myReadData'
```

Configure Client Properties

This example:

1. Opens a model.
2. Adds an AUTOSAR client-server 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 model
hModel = 'mControllerWithInterface_client';
addpath (fullfile(matlabroot,'/help/toolbox/autosar/examples'));
open_system(hModel);

% Use 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')

% 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'}
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')

% 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')

ifPaths =
    {'/pkg/if/csInterface'}    {'/pkg/if/myCsIf'}

args =
    {'Op'     }    {'In' }
    {'Data'   }    {'Out'}
    {'ERR'    }    {'Out'}
    {'NegCode'}    {'Out'}

ans =
    {'myCsIf/readData/Op'}    {'myCsIf/readData/Data'}
    {'myCsIf/readData/ERR'}    {'myCsIf/readData/NegCode'}

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 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.

```
% Add path to model and mode definition files and open model
addpath (fullfile(matlabroot,'/help/toolbox/autosar/examples'));
hModel = 'mAutosarMsConfig';
open_system(hModel);

% Use AUTOSAR property functions
arProps = autosar.api.getAUTOSARProperties(hModel);

% File mdgModes.m declares AUTOSAR mode declaration group mdgModes for use with the M-S interface.
% See matlabroot/help/toolbox/autosar/examples/mdgModes.m, which must be on the MATLAB path.
% 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')

% 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','ComponentQualifiedName');
ib = get(arProps,swc,'Behavior');
runnables = get(arProps,ib,'Runnables')
add(arProps,ib,'Events',msEventName,'Category','ModeSwitchEvent',...
    'Activation', 'OnTransition', ...
    'StartOnEvent', [ib '/' msRunnable]);
% Separate code, below, sets ModeSwitchEvent port and trigger values.

% 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')

% Export mode declaration group information to AUTOSAR data type package in XML
mdgPkg = get(arProps,'XmlOptions','DataTypePackage');
mdgPath = [mdgPkg '/' modeGroup]
initMode = [mdgPath '/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 M-S interface
set(arProps,[ifPkg '/' ifName '/' modeGroup],'ModeGroup',mdgPath)

% Set port and trigger for AUTOSAR ModeSwitchEvent
expTrigger = {[mrPortName '.STARTUP'], [mrPortName '.SHUTDOWN']}
set(arProps,[ib '/' msEventName],'Trigger',expTrigger)

% Use AUTOSAR map functions
slMap=autosar.api.getSimulinkMapping(hModel);

% Map Simulink inport MRPort to AUTOSAR mode receiver port myMRPort and mode group mdgModes
msDataAccessMode = 'ModeReceive';
[arPortName,arDataElementName,arDataAccessMode]=getInport(slMap,'MRPort');
mapInport(slMap,'MRPort',mrPortName,modeGroup,msDataAccessMode);
[arPortName,arDataElementName,arDataAccessMode]=getInport(slMap,'MRPort')

% To pass validation, set inport Runnable1 sample time to -1 (inherited)
set_param([hModel,'/Runnable1'],'SampleTime','-1')
```

```
ifPaths =
    {'/pkg/if/myMsIf'}
```

```
runnables =
    {'ASWC/Behavior/Runnable_Init'}    {'ASWC/Behavior/Runnable1'}
    {'ASWC/Behavior/Runnable2'}    {'ASWC/Behavior/Runnable3'}

events =
    {'ASWC/Behavior/Event_t_1tic_B'}    {'ASWC/Behavior/Event_t_10tic'}
    {'ASWC/Behavior/myMSEvent'}

mdgPath =
    '/pkg/dt/mdgModes'

initMode =
    '/pkg/dt/mdgModes/STARTUP'

expTrigger =
    {'myMRPort.STARTUP'}    {'myMRPort.SHUTDOWN'}

arPortName =
    'myMRPort'
arDataElementName =
    'mdgModes'
arDataAccessMode =
    'ModeReceive'
```

Configure AUTOSAR XML Export

- “Configure XML Export Options” on page 4-376
- “Configure AUTOSAR Package Paths” on page 4-377

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 4-377.

```
% Open model
hModel = 'autosar_swc_counter';
open_system(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 4-140.

To configure other XML export options, see “Configure XML Export Options” on page 4-376.

```
% Open model
hModel = 'autosar_swc_counter';
open_system(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 4-347

More About

- “AUTOSAR Component Configuration” on page 4-3

Limitations and Tips

The following limitations apply to AUTOSAR component development.

In this section...
"AUTOSAR Client Block in Referenced Model" on page 4-378
"Use the Merge Block for Inter-Runnable Variables" on page 4-378

AUTOSAR Client Block in Referenced Model

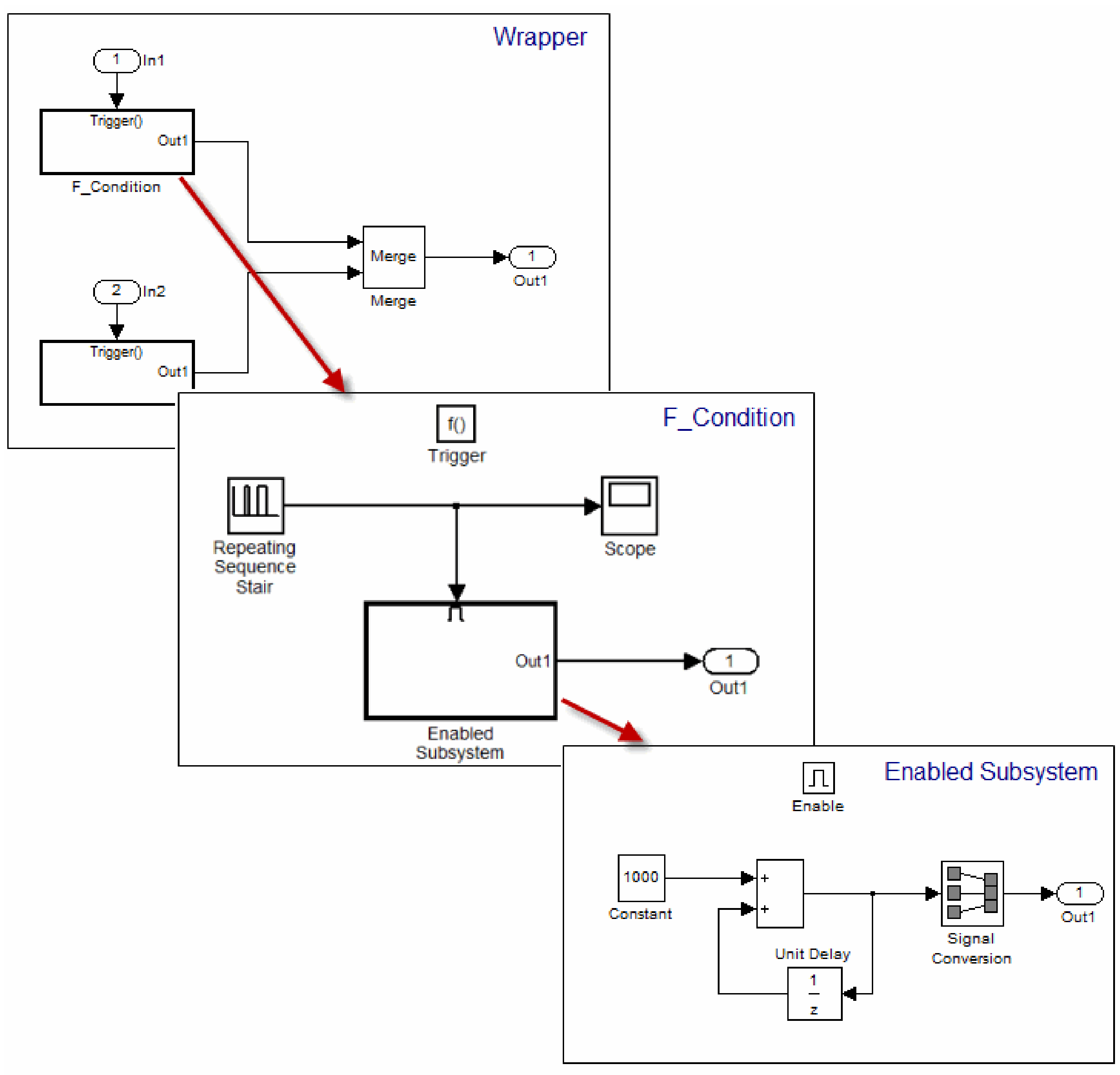
The software does not support the use of an AUTOSAR client block, such as Function Caller or Invoke AUTOSAR Server Operation, in a referenced model.

Use the Merge Block for Inter-Runnable Variables

You can use the Merge block to merge inter-runnable variables. However, you must do the following:

- Connect the output signal of the Merge block to either one root outport or one or more subsystems.
- If the output signal of the Merge block is connected to the inputs of one or more subsystems, assign the same signal name to the Merge block's output and inputs.

In addition, the signal from the function-call subsystem outport that enters a Merge block must not be conditionally computed. Consider the following example.



The output from the subsystem `F_condition` is the conditional output from `Enabled Subsystem`. When you try to validate or build the model, the software generates an error.

If you use an S-Function block instead of the Enabled Subsystem block, the software generates a *warning* when you validate or build the model.

AUTOSAR Code Generation

- "Generate AUTOSAR C or C++ Code and XML Descriptions" on page 5-2
- "Configure AUTOSAR Code Generation" on page 5-12
- "Configure AUTOSAR Adaptive Code Generation" on page 5-18
- "Code Generation with AUTOSAR Code Replacement Library" on page 5-20
- "Verify AUTOSAR C or C++ Code with SIL and PIL" on page 5-23
- "Import and Simulate AUTOSAR Code from Previous Releases" on page 5-24
- "Limitations and Tips" on page 5-25

Generate AUTOSAR C or C++ Code and XML Descriptions

Generate AUTOSAR-compliant C or C++ code and export AUTOSAR XML (`arxml`) descriptions from AUTOSAR classic or adaptive component

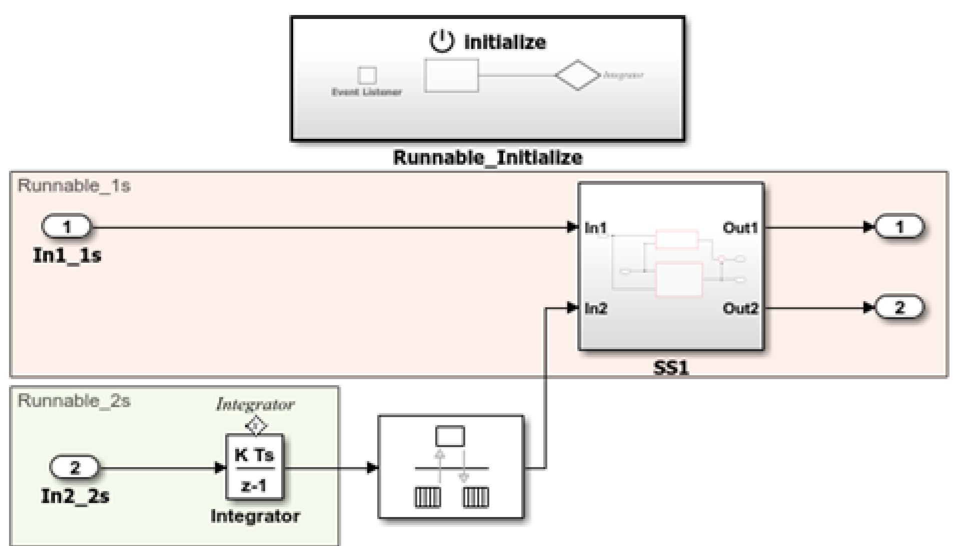
AUTOSAR Blockset software supports AUTomotive Open System ARchitecture (AUTOSAR), an open and standardized automotive software architecture. Automobile manufacturers, suppliers, and tool developers jointly develop AUTOSAR components. To develop AUTOSAR components in Simulink, follow this general workflow:

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

Generating C Code and XML Descriptions for Classic Platform

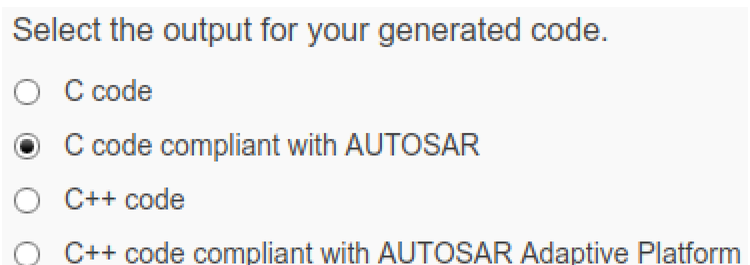
If you are licensed for Simulink Coder and Embedded Coder, you can generate AUTOSAR-compliant C code and `arxml` software descriptions for a component model configured for the AUTOSAR Classic Platform. Open a model from which you want to generate AUTOSAR C code and descriptions. This example uses AUTOSAR example model `autosar_swc`.

```
open_system('autosar_swc');
```



To refine model configuration settings for code generation, you can use Embedded Coder Quick Start (recommended) or follow the manual procedures described in “Configure AUTOSAR Code Generation” on page 5-12. This example uses Embedded Coder Quick Start. In the model window, click **Code > C/C++ Code > Embedded Coder Quick Start**.

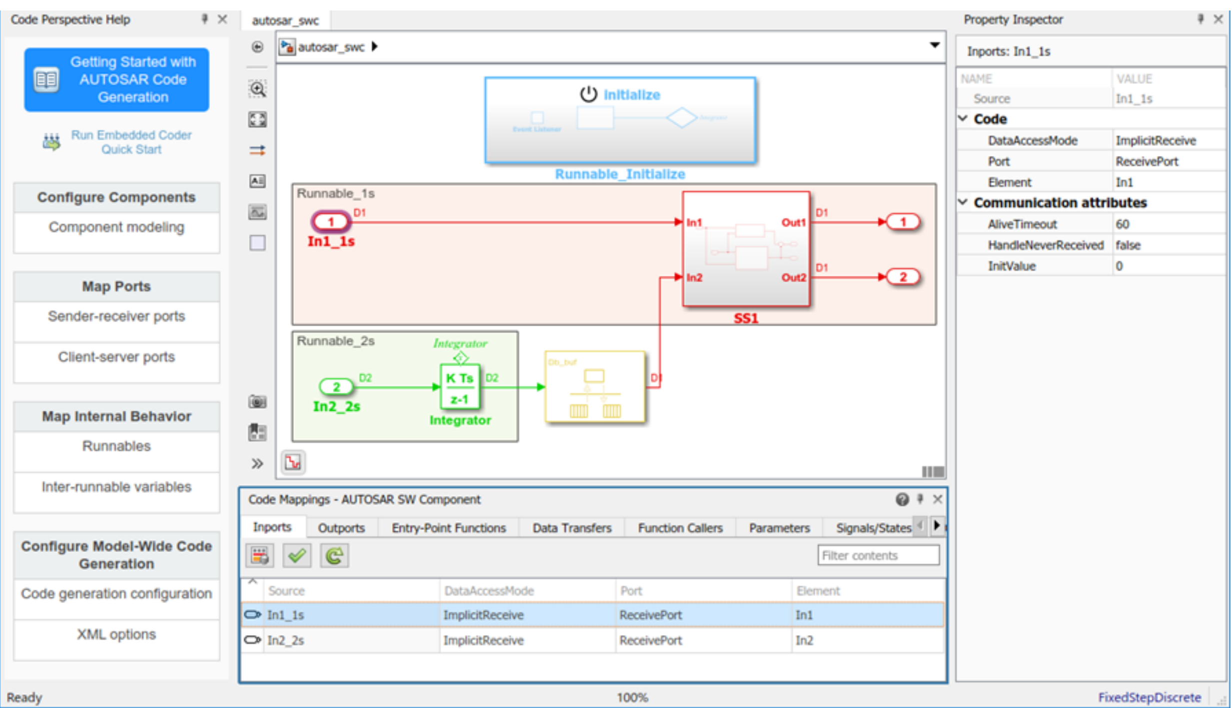
Work through the quick-start procedure. In the Output window, select output option **C code compliant with AUTOSAR**.



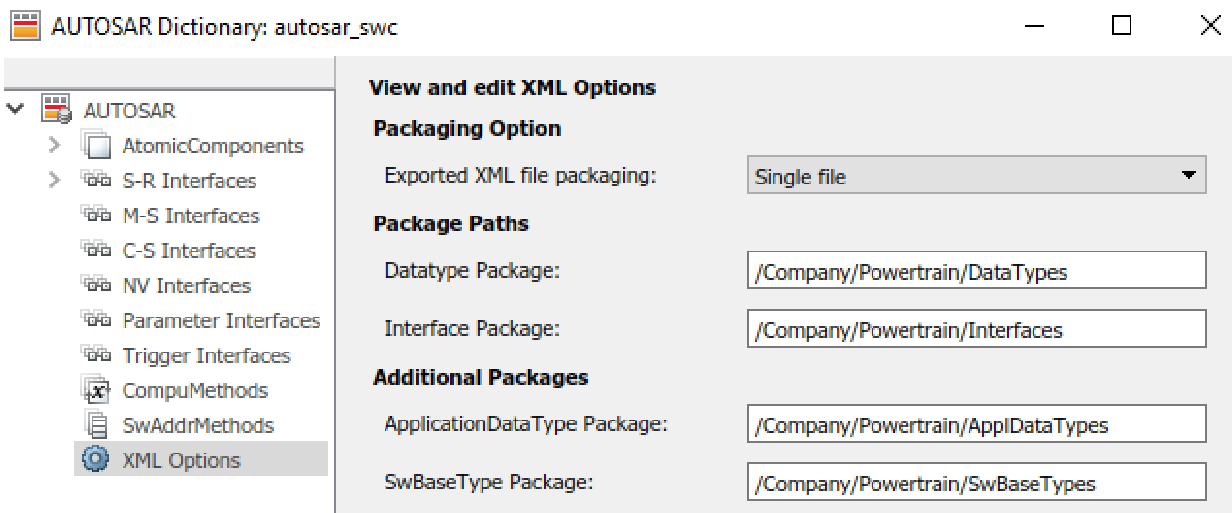
The quick-start software takes the following steps to configure an AUTOSAR software component model:

1. Configures code generation settings for the model. If the AUTOSAR target is not already selected, the software sets model configuration parameter **System target file** to `autosar.tlc` and **Generate XML for schema version** to a default schema value.
2. If no AUTOSAR mapping exists, creates a mapped AUTOSAR software component for the model.
3. Performs a model build.

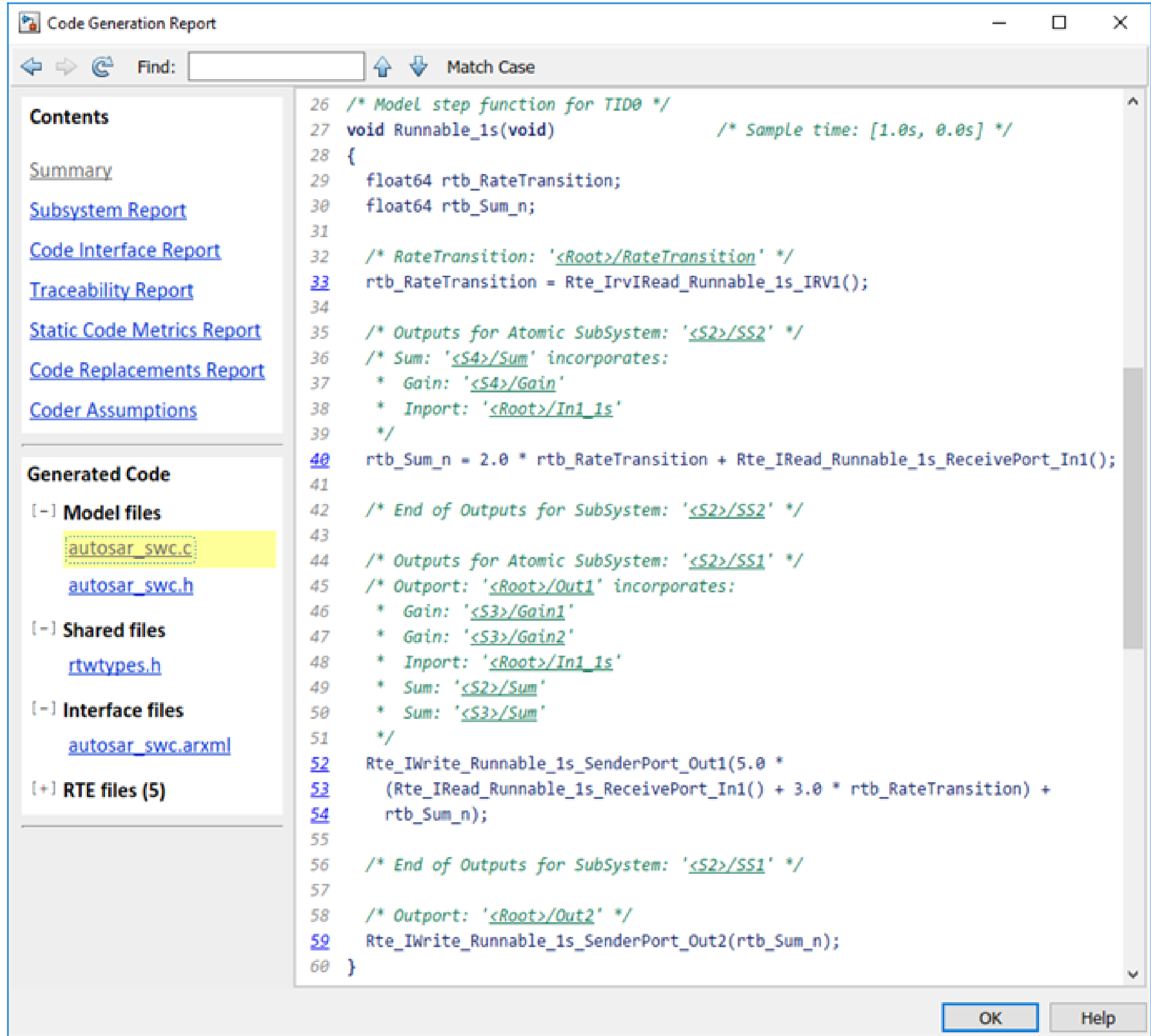
In the last window, when you click **Finish**, your model opens in the AUTOSAR code perspective.



Before generating code, check the settings of AUTOSAR XML export parameters in AUTOSAR Dictionary. Click **Code** > **C/C++ Code** > **Configure AUTOSAR Dictionary** and select **XML Options**. This example sets **Exported XML file packaging** to `Single file`. so that `arxml` is exported into a single file, *`modelname`*`.arxml`.



To generate AUTOSAR-compliant C code and `arxml` software descriptions, build the model. In the model window, press **Ctrl+B**, or click the **Code** menu and select **C/C++ Code > Build Model**. The build process generates C code and `arxml` descriptions to the model build folder, `autosar_swc_autosar_rtw`. When the build completes, a code generation report opens.

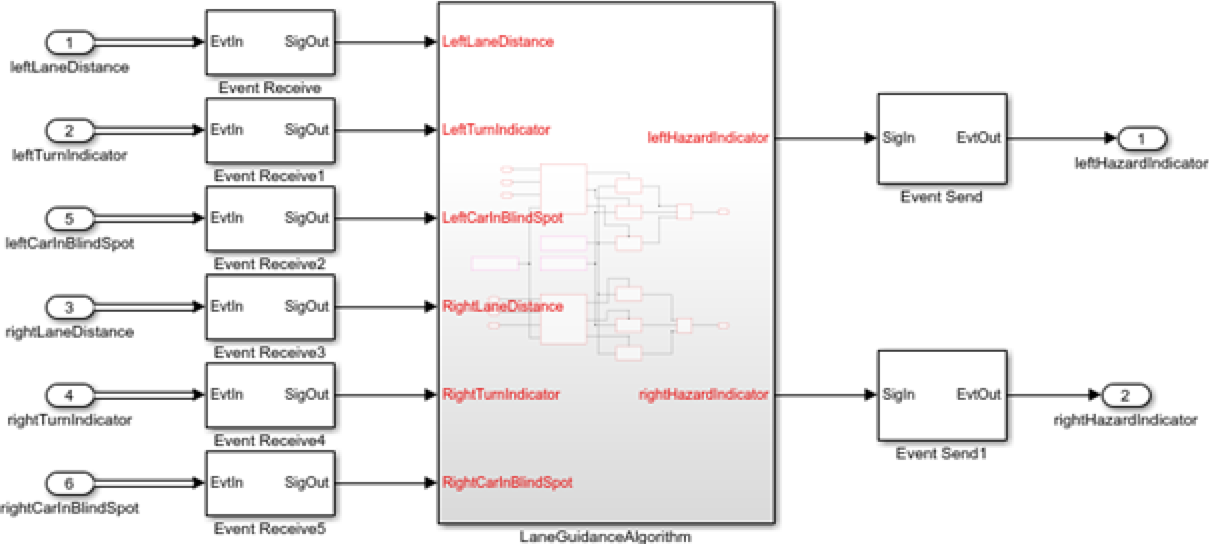


Generating C++ Code and XML Descriptions for Adaptive Platform

If you are licensed for Simulink Coder and Embedded Coder, you can generate AUTOSAR-compliant C++ code and `arxml` software descriptions for a component model configured

for the AUTOSAR Adaptive Platform. Open a model from which you want to generate AUTOSAR C++ code and 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 Embedded Coder Quick Start (recommended). This example uses Embedded Coder Quick Start. In the model window, click **Code** > **C/C++ Code** > **Embedded Coder Quick Start**.

Work through the quick-start procedure. In the Output window, select output option **C++ code compliant with AUTOSAR Adaptive Platform**.

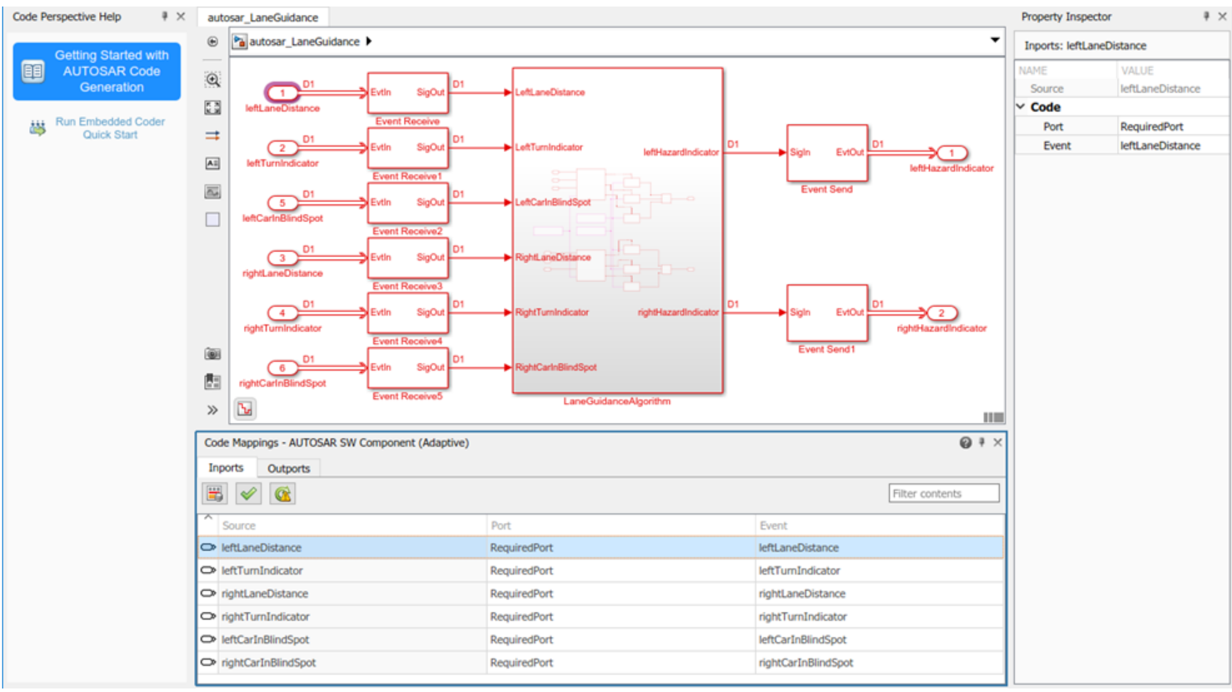
Select the output for your generated code.

- ○ C code
- ○ C code compliant with AUTOSAR
- ○ C++ code
- ◉ C++ code compliant with AUTOSAR Adaptive Platform

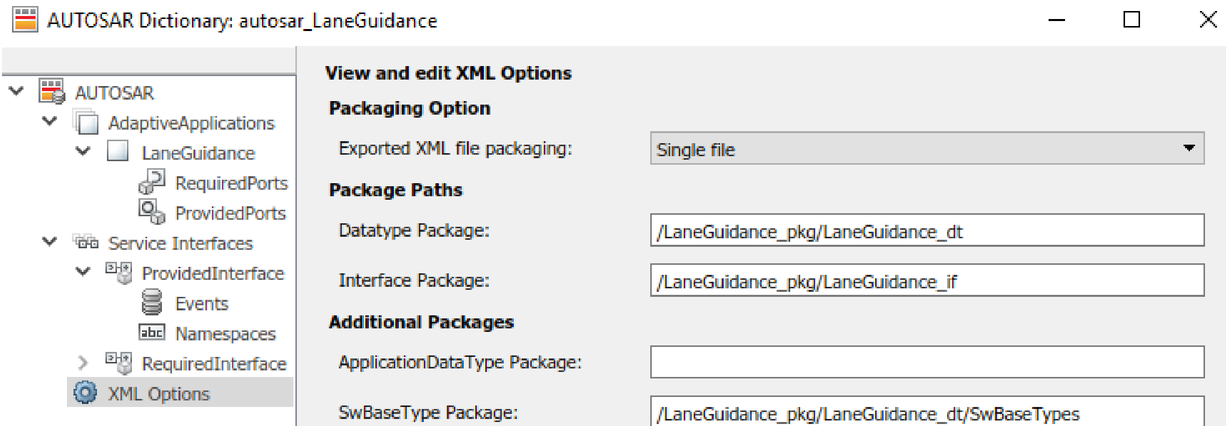
The quick-start software takes the following steps to configure an AUTOSAR adaptive software component model:

1. Configures code generation settings for the model. If the AUTOSAR target is not already selected, the software sets model configuration parameter **System target file** to `autosar_adaptive.tlc`.
2. If no AUTOSAR mapping exists, creates a mapped AUTOSAR adaptive software component for the model.
3. Performs a model build.

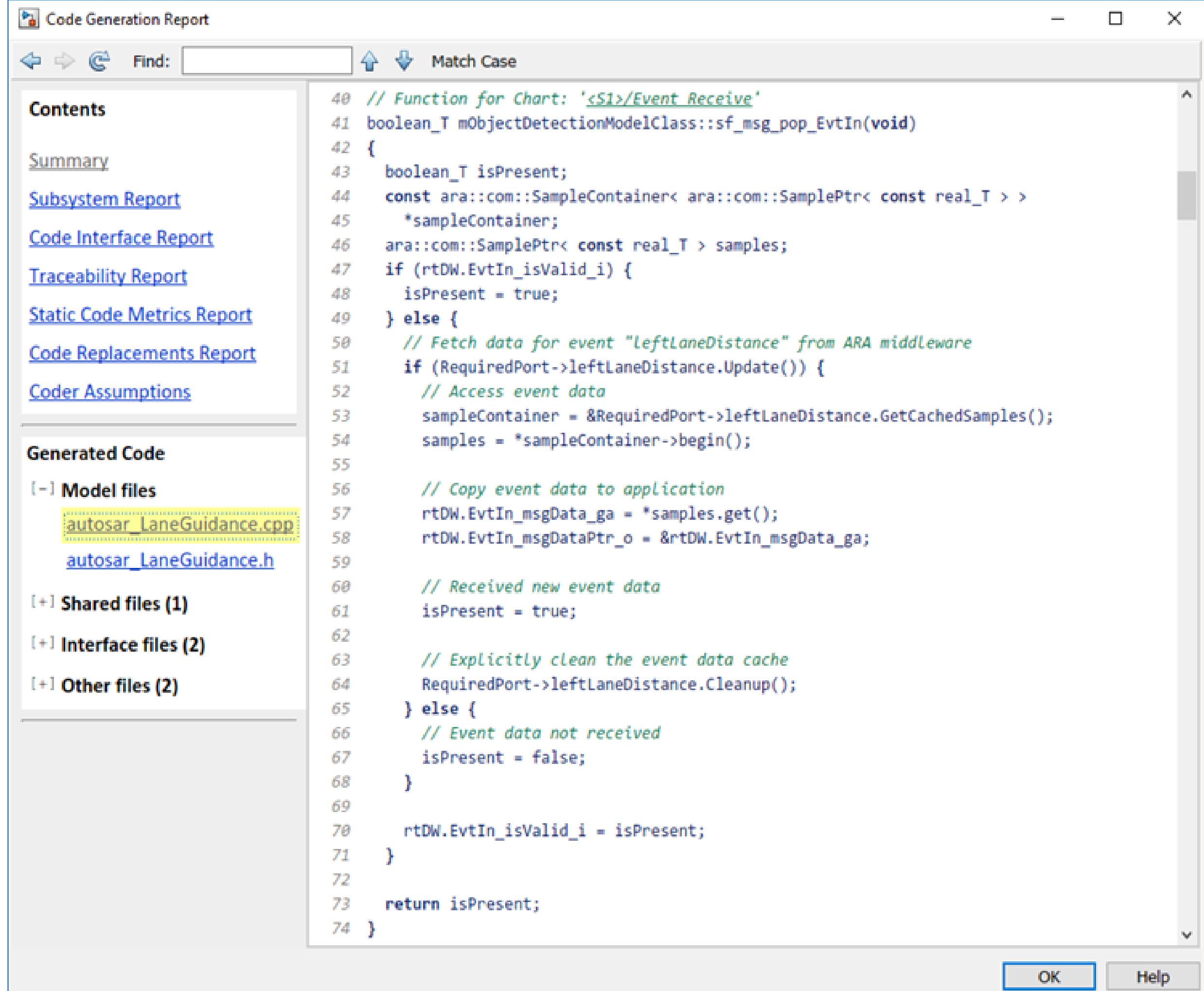
In the last window, when you click **Finish**, your model opens in the AUTOSAR code perspective.



Before generating code, check the settings of AUTOSAR XML export parameters in AUTOSAR Dictionary. Click **Code** > **C/C++ Code** > **Configure AUTOSAR Dictionary** and select **XML Options**. This example sets **Exported XML file packaging** to `Single file`. so that `arxml` is exported into a single file, *`modelname`*`.arxml`.



To generate AUTOSAR-compliant C++ code and `arxml` software descriptions, build the model. In the model window, press **Ctrl+B**, or click the **Code** menu and select **C/C++ Code** > **Build Model**. The build process generates C++ code and `arxml` descriptions to the model build folder, `autosar_LaneGuidance_autosar_adaptive`. When the build completes, a code generation report opens.



Related Links

- “Code Generation” (Classic Platform)
- “Code Generation” (Adaptive Platform)
- “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 AUTOSAR:

1. Examine AUTOSAR code generation parameters on the **Code Generation > AUTOSAR Code Generation Options** pane of the Configuration Parameters dialog box.
2. Examine AUTOSAR XML export options using AUTOSAR Dictionary or AUTOSAR property functions.
3. Build the model.

In this section...
“Select an AUTOSAR Schema” on page 5-12
“Specify Maximum SHORT-NAME Length” on page 5-13
“Configure AUTOSAR Compiler Abstraction Macros” on page 5-14
“Root-Level Matrix I/O” on page 5-15
“Inspect AUTOSAR XML Options” on page 5-16
“Generate AUTOSAR C and XML Files” on page 5-16

Select an AUTOSAR Schema

The software supports the following AUTOSAR Classic Platform schema versions for import and export of `arxml` files and generation of AUTOSAR-compatible C code.

Schema Version Value	Schema Revisions Supported for Import	Export Schema Revision
`4.3` (default)	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
`3.2`	3.2.1, 3.2.2	3.2.2
`3.1`	3.1.1, 3.1.2, 3.1.3, 3.1.4	3.1.4

Schema Version Value	Schema Revisions Supported for Import	Export Schema Revision
3.0	3.0.1, 3.0.2, 3.0.3, 3.0.4, 3.0.5, 3.0.6	3.0.2
2.1	2.1 (XSD rev 0014, 0015, 0017, 0018)	2.1 (XSD rev 0017)

Selecting the AUTOSAR system target file for your model for the first time sets the schema version parameter to the default value, `4.3`.

When you import `arxml` code into Simulink, the `arxml` importer detects the schema version and sets the schema version parameter in the model. For example, if you import `arxml` code based on schema 4.0 revision 4.0.1, 4.0.2, or 4.0.3, the importer sets the schema version parameter to `4.0`.

When you export your AUTOSAR software component, code generation exports XML that is compliant with the current schema version value. For example, if **Generate XML file for schema version** equals `4.0`, export uses the export schema revision listed above for schema 4.0, that is, revision 4.0.3.

If you need to change the schema version, you must do so before exporting your AUTOSAR software component. To select a schema version, on the **Code Generation > AUTOSAR Code Generation Options** pane of the Configuration Parameters dialog box, from the **Generate XML file for schema version** drop-down list, select the schema version that you require.

Note The AUTOSAR model parameters on the **AUTOSAR Code Generation Options** pane must be set 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, for schema version 4.x or later, or 32 characters, for earlier schema versions. Even for earlier schema versions, your AUTOSAR authoring tool may support the use of longer `SHORT-NAME` elements, for example, to name ports and interfaces.

Use the **Maximum SHORT-NAME length** parameter to specify a maximum length for SHORT-NAME elements exported by the code generator. On the **Code Generation > AUTOSAR Code Generation Options** pane of the Configuration Parameters dialog box, in the **Maximum SHORT-NAME length** field, specify a positive number of characters less than or equal to 128. 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 configure AUTOSAR compiler macro generation, in the **Code Generation > AUTOSAR Code Generation Options** pane of the Configuration Parameters dialog box, select **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).

Example 5.1. Example

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

The software supports matrix I/O at the root-level by generating code that implements matrices as one-dimensional arrays. However, this behavior is not the default. To configure root-level matrix I/O, on the **Code Generation > AUTOSAR Code Generation Options** pane of the Configuration Parameters dialog box, select **Support root-level matrix I/O using one-dimensional arrays**.

Inspect AUTOSAR XML Options

Examine the XML options that you configured using AUTOSAR Dictionary. If you have not yet configured them, see “Configure AUTOSAR XML Options” on page 4-62.

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 model (**Ctrl+B**).

The build process generates AUTOSAR-compliant C code and AUTOSAR XML descriptions to the model build folder. The following table shows which XML files are generated, based on the value of the **Exported XML file packaging** option configured in AUTOSAR Dictionary.

Exported XML File Packaging Value	Exported File Name	By Default Contains...
`Single file`	`*modelname*.arxml`	All AUTOSAR elements.
`Modular`	`*modelname*_component.arxml`	Software components. This is the main `arxml` file exported for the Simulink model. In addition to AUTOSAR software components, the file includes elements for which AUTOSAR packages (AR-PACKAGEs) are not configured, and AR-PACKAGEs that do not align with the package paths in the other exported `arxml` files. For more information on AR-PACKAGEs and their location in modular exported `arxml` files, see “Configure AUTOSAR Packages” on page 4-138.
	`*modelname*_datatype.arxml`	Data types and related elements.
	`*modelname*-_implementation.arxml`	Software component implementation.

Exported XML File Packaging Value	Exported File Name	By Default Contains...
	*modelname*_interface.arxml	Interfaces, including S-R, C-S, M-S, NV, and other interfaces.
	*modelname*_behavior.arxml	Software component internal behavior (generated only for schema 3.x or earlier).

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 AAT and the Simulink model-based design environment, the code generator preserves AUTOSAR elements and their UUIDs across arxml import and export. For more information, see “Round-Trip Preservation of AUTOSAR XML File Structure and Element Information” on page 3-46.

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 or C++ Code and XML Descriptions” on page 5-2.

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. Examine AUTOSAR XML export options using AUTOSAR Dictionary or AUTOSAR property functions.
2. Build the model.

In this section...
"Inspect AUTOSAR Adaptive XML Options" on page 5-18
"Generate AUTOSAR Adaptive C++ and XML Files" on page 5-18

Inspect AUTOSAR Adaptive XML Options

Examine the XML options that you configured using AUTOSAR Dictionary. If you have not yet configured them, see "Configure AUTOSAR Adaptive XML Options" on page 4-98.

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 model (**Ctrl+B**).

The build process generates AUTOSAR-compliant C++ code and AUTOSAR XML descriptions to the model build folder. The following table shows which XML files are generated, based on the value of the **Exported XML file packaging** option configured in AUTOSAR Dictionary.

Exported XML File Packaging Value	Exported File Name	By Default Contains...
`Single file`	*`modelname`*`.arxml`	All AUTOSAR elements.

Exported XML File Packaging Value	Exported File Name	By Default Contains...
Modular	*modelname*_component.arxml	Software components. This is the main arxml file exported for the Simulink model. In addition to AUTOSAR software components, the file includes elements for which AUTOSAR packages (AR-PACKAGEs) are not configured, and AR-PACKAGEs that do not align with the package paths in the other exported arxml files. For more information on AR-PACKAGEs and their location in modular exported arxml files, see "Configure AUTOSAR Packages" on page 4-138.
	*modelname*_datatype.arxml	Data types and related elements.
	modelname-_implementation.arxml	Software component implementation.
	*modelname*_interface.arxml	Interfaces, including adaptive service interfaces.
	*modelname*_behavior.arxml	Software component internal behavior (generated only for schema 3.x or earlier).

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 usually defined early in the design process. You must, however, merge the internal behavior file because this information is part of the model implementation.

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 or C++ Code and XML Descriptions" on page 5-2.

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. The AUTOSAR 4.0 code replacement library is intended for use with AUTOSAR schema version 4.0 or later.

Code Replacement Library for AUTOSAR Code Generation

The AUTOSAR 4.0 code replacement library enables you to customize the code generator to produce C or C++ code that closely aligns with the AUTOSAR standard. Considering 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`.

For more information, see “Choose a Code Replacement Library” (Embedded Coder).

Configure Code Generator to Use AUTOSAR 4.0 Code Replacement Library

To configure the code generator for your model to use the AUTOSAR code replacement library, open the Configuration Parameters dialog box. Select **Code Generation** > **Interface** > **Code replacement library** > **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).

Code Replacement Library Checks

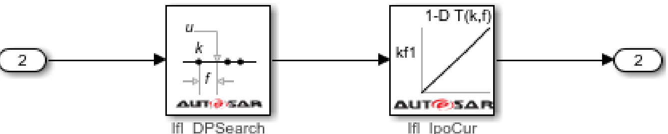
Code replacement requires that the combination of types for input, breakpoint, table, and output types are compatible with the AUTOSAR specification. Floating-point (IFL) replacement only supports single types while fixed-point (IFX) replacement supports uint8, uint16, int8, int16 and associated fixed-point 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

This example shows how to replace code generated for Simulink lookup table blocks with functions that are compatible with AUTOSAR floating-point interpolation (IFL) library routines. If you want to replace code with fixed-point interpolation (IFX) library routines, you can change the type and reconfigure the block.

1. 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:



2 Configure the code generator for your model to use the AUTOSAR 4.0 code replacement library. In the Configuration Parameters dialog box, select **Code Generation** > **Interface** > **Code replacement library** > **AUTOSAR 4.0**. Alternatively, from the command line or programmatically, use `set_param` to set the `CodeReplacementLibrary` parameter to `'AUTOSAR 4.0'`.

3 Optionally, you can configure the model for the code generator 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'`.

4 Build the model and review the generated code for expected code replacements.

For example:

```
/* 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(Ifl_IpoCur_f32(&rtb_Prelookup,
    Rte_CData_Lcom_4_single()));
}
```

Verify AUTOSAR C or C++ Code with SIL and PIL

You can carry out model-based verification of AUTOSAR software components using software-in-the-loop (SIL) and processor-in-the-loop (PIL) simulations. Use SIL for verification of generated source code on your host computer, and PIL for verification of object code on your production target hardware. For example:

- You can run a top model that is configured for an AUTOSAR system target file (`autosar.tlc` or `autosar_adaptive.tlc`) using the `Software-in-the-Loop (SIL)` and `Processor-in-the-Loop (PIL)` simulation modes.
- You can use Model block SIL or PIL to test AUTOSAR top-model code. In the Model block, set parameter **Code interface** to `Top model`.

For more information, see “Simulation with Top Model” (Embedded Coder) and “Simulation with Model Blocks” (Embedded Coder).

Note You can create a SIL or PIL block for a component configured for the AUTOSAR system target file. For more information about configuring and running simulations with SIL or PIL blocks, see “Simulation with Blocks From Subsystems” (Embedded Coder). However, SIL and PIL block verification does not support code generated for Simulink Function and Function Caller blocks, for example, in AUTOSAR client-server configurations.

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:

- “Workflow” (Embedded Coder)
- “Import AUTOSAR Code from Previous Releases” (Embedded Coder)

Limitations and Tips

The following limitations apply to AUTOSAR code generation.

In this section...

"Generate Code Only Check Box" on page 5-25
"AUTOSAR Compiler Abstraction Macros" on page 5-25
"Relative File Paths in AUTOSAR Code Descriptors (Schema Versions 3.x and Earlier)" on page 5-26
"Schedule Editor Explicit Partitions" on page 5-26

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

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

Relative File Paths in AUTOSAR Code Descriptors (Schema Versions 3.x and Earlier)

When you build a Simulink model for an AUTOSAR system target file, using AUTOSAR schema version 3.x or earlier, the code generator produces a CODE-DESCRIPTORS element within the SWC_IMPLEMENTATION element. The CODE-DESCRIPTORS element contains XFILE elements that provide descriptions of the generated code.

For example, if you build the model autosar_swc_counter, the generated file autosar_swc_counter_implementation.arxml has the following XFILE element:

```
<XFILE>
  <SHORT-NAME>autosar_swc_counter_c</SHORT-NAME>
  <CATEGORY>GeneratedFile</CATEGORY>
  <URL>autosar_swc_counter_autosar_rtw\autosar_swc_counter.c</URL>
  <TOOL>Embedded Coder</TOOL>
  <TOOL-VERSION>n.n</TOOL-VERSION>
</XFILE>
```

However, the URL element does not specify an absolute path. The path is *relative* to the build folder. Therefore, before you use the AUTOSAR XML in a run-time environment to generate code, you must place the XML in the parent folder.

Schedule Editor Explicit Partitions

AUTOSAR code generation does not support explicit partitions created with Schedule Editor. If an AUTOSAR model has explicit partitions, AUTOSAR validation errors out.

AUTOSAR Composition and ECU Software Simulation

- “Import AUTOSAR Composition to Simulink” on page 6-2
- “Combine and Simulate AUTOSAR Software Components” on page 6-7
- “Configure Calls to AUTOSAR Diagnostic Event Manager Service” on page 6-13
- “Configure Calls to AUTOSAR NVRAM Manager Service” on page 6-20
- “Configure AUTOSAR Basic Software Service Implementations for Simulation” on page 6-27
- “Simulate AUTOSAR Basic Software Services and Run-Time Environment” on page 6-32

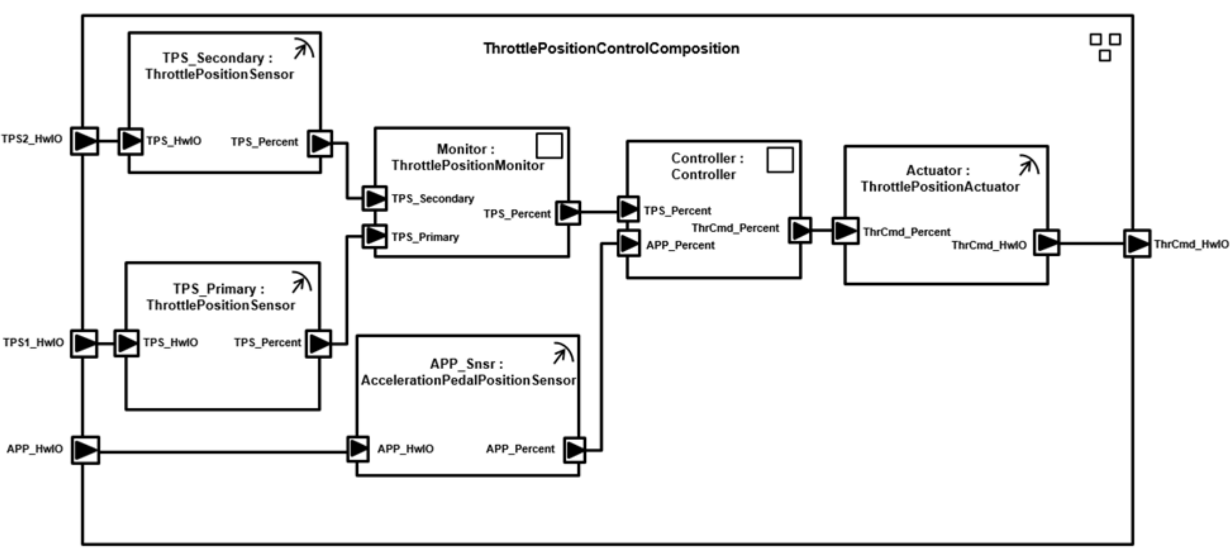
Import AUTOSAR Composition to Simulink

Create Simulink® representation of AUTOSAR composition imported from AUTOSAR authoring tool `arxml` file

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`.



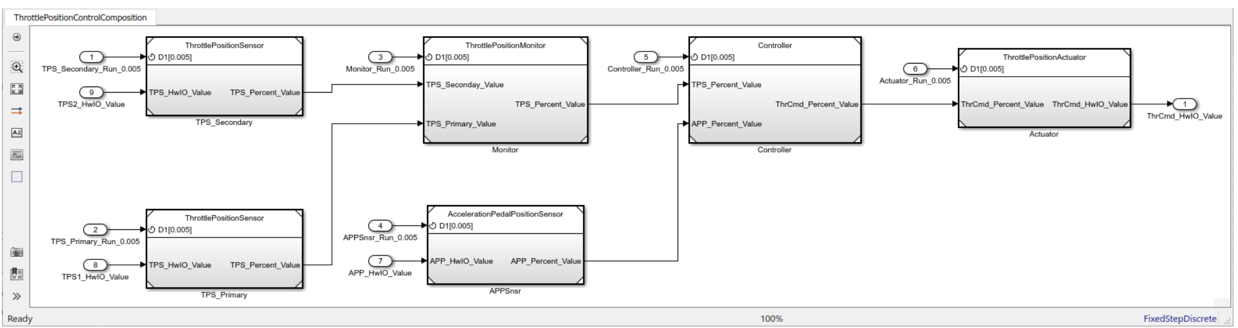
Use the MATLAB function `createCompositionAsModel` to import the `arxml` description and create an initial Simulink representation of the AUTOSAR composition.

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

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.

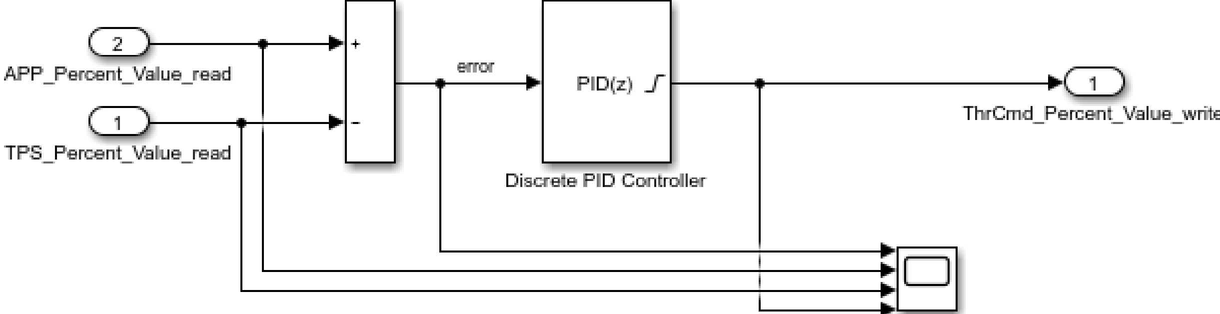


Develop AUTOSAR Component Algorithms, Simulate, and Generate Code

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. 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. (To explore this implementation, open model `Controller.slx`.)



As you develop AUTOSAR components, you can:

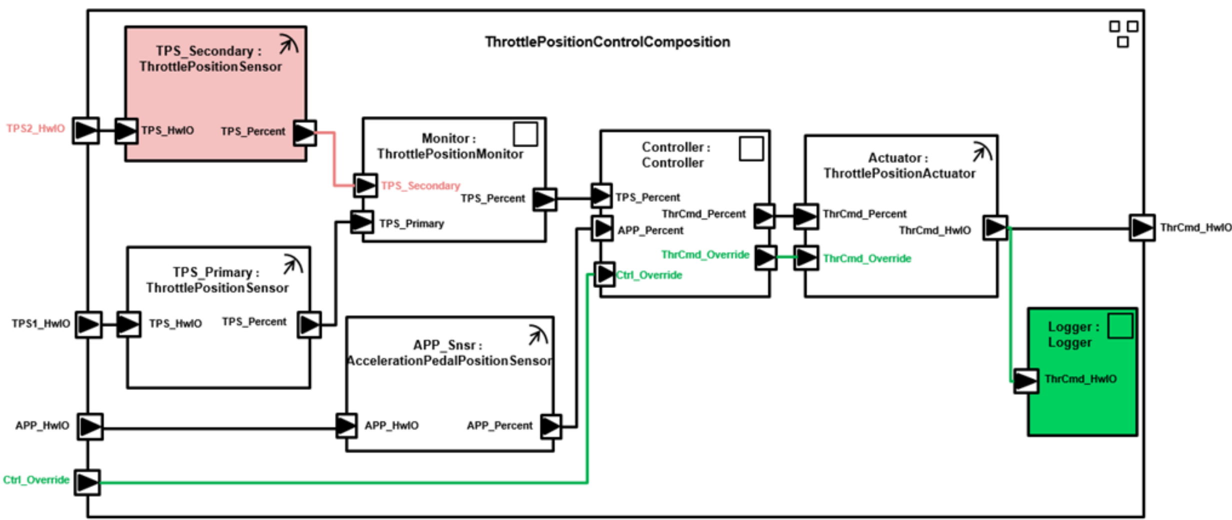
- Simulate component models individually or together in a containing composition.
- Generate `arxml` component description files and algorithmic C code for testing in Simulink or integration into an AUTOSAR run-time environment. (AUTOSAR code generation requires Simulink Coder and Embedded Coder.)

For more information on developing, simulating, and building AUTOSAR components, see example “Design AUTOSAR Components, Simulate, and Generate Code” on page 4-121.

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`.



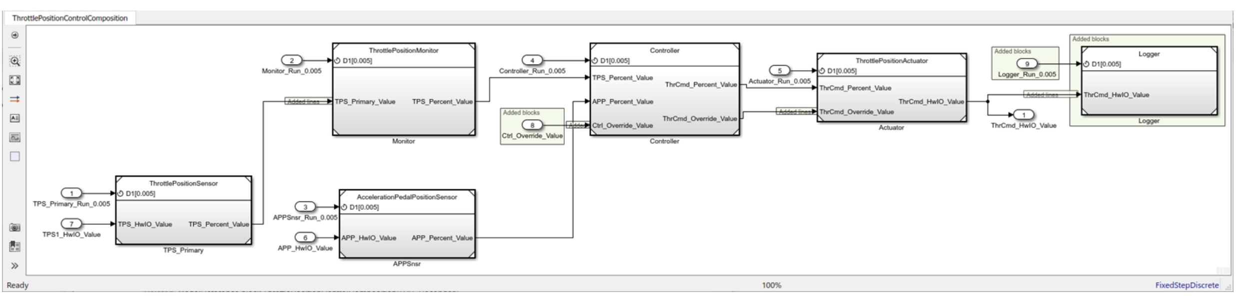
Use the MATLAB function `updateModel` to import the architectural revisions from the `arxml` file. The function updates the AUTOSAR composition model with the changes and reports the results.

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

```
Updating model 'ThrottlePositionSensor' for component 1 of 6:
/Company/Components/ThrottlePositionSensor
### Updating model ThrottlePositionSensor
### Saving original model as ThrottlePositionSensor_backup.slx
### Creating HTML report ThrottlePositionSensor_update_report.html
Updating model 'ThrottlePositionMonitor' for component 2 of 6:
/Company/Components/ThrottlePositionMonitor
### Updating model ThrottlePositionMonitor
### Saving original model as ThrottlePositionMonitor_backup.slx
### Creating HTML report ThrottlePositionMonitor_update_report.html
Creating model 'Logger' for component 3 of 6: /Company/Components/Logger
Updating model 'Controller' for component 4 of 6: /Company/Components/Controller
### Updating model Controller
### Saving original model as Controller_backup.slx
### Creating HTML report Controller_update_report.html
Updating model 'AccelerationPedalPositionSensor' for component 5 of 6:
/Company/Components/AccelerationPedalPositionSensor
### Updating model AccelerationPedalPositionSensor
### Saving original model as AccelerationPedalPositionSensor_backup.slx
```

```
### Creating HTML report AccelerationPedalPositionSensor_update_report.html
Updating model 'ThrottlePositionActuator' for component 6 of 6:
/Company/Components/ThrottlePositionActuator
### Updating model ThrottlePositionActuator
### Saving original model as ThrottlePositionActuator_backup.slx
### Creating HTML report ThrottlePositionActuator_update_report.html
Updating 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 3-39
- “Design AUTOSAR Components, Simulate, and Generate Code” on page 4-121

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.

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.

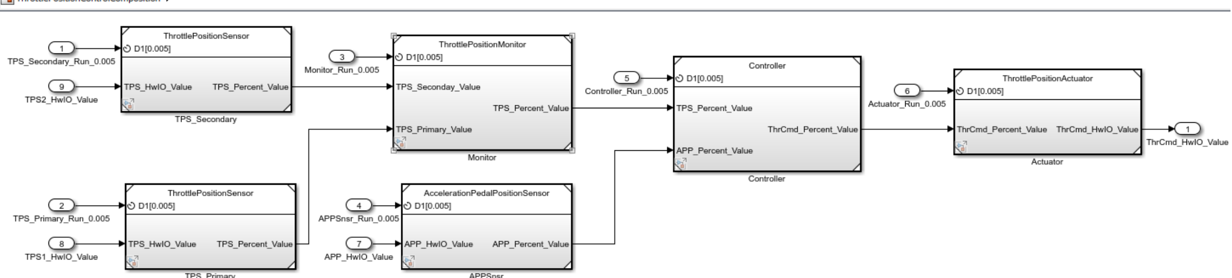
In this section...
“Import AUTOSAR Composition as Model (Classic Platform)” on page 6-7
“Create Composition Model for Simulating AUTOSAR Components” on page 6-8
“Alternatives for AUTOSAR System-Level Simulation” on page 6-10

Import AUTOSAR Composition as Model (Classic Platform)

A composition is an AUTOSAR software component that aggregates related groups of software components. Compositions support component scalability 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 AUTOSAR importer function `createCompositionAsModel`. This function call creates composition model `ThrottlePositionControlComposition`.

```
addpath(fullfile(matlabroot,'examples','autosarblockset'));
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 6-2.

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` (open).

- `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:

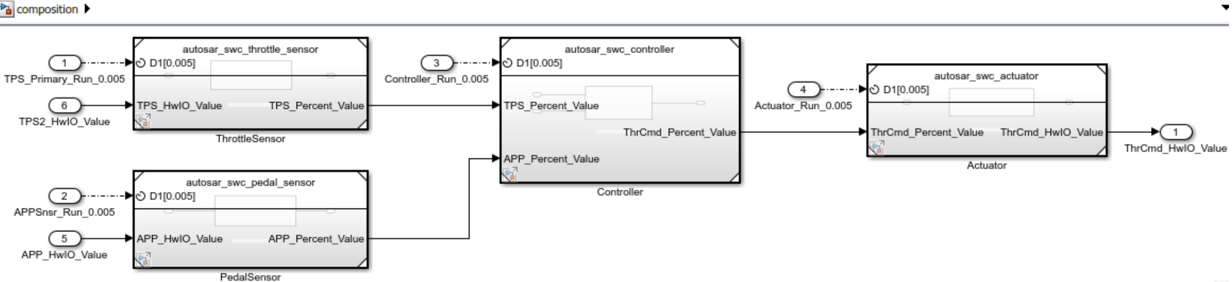
   a Open the component model separately and verify that it simulates.

   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');
```

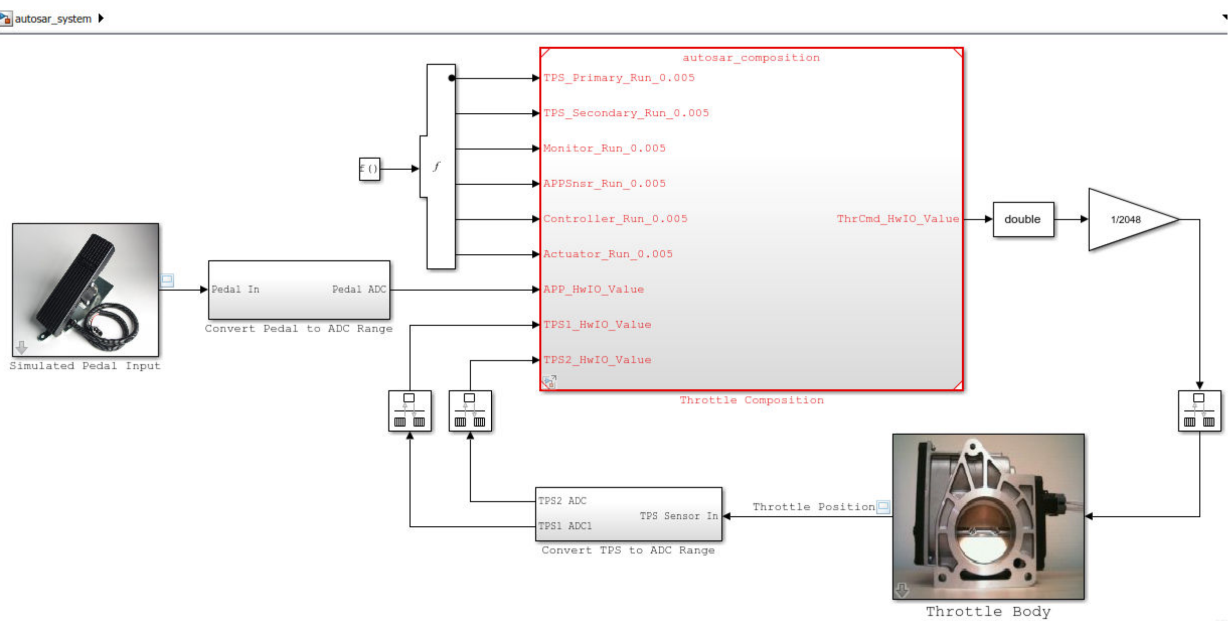
For more information, see “Design AUTOSAR Components, Simulate, and Generate Code” on page 4-121.

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 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. For a system-level model, use a Simulink model or a Simulink Test test harness model.

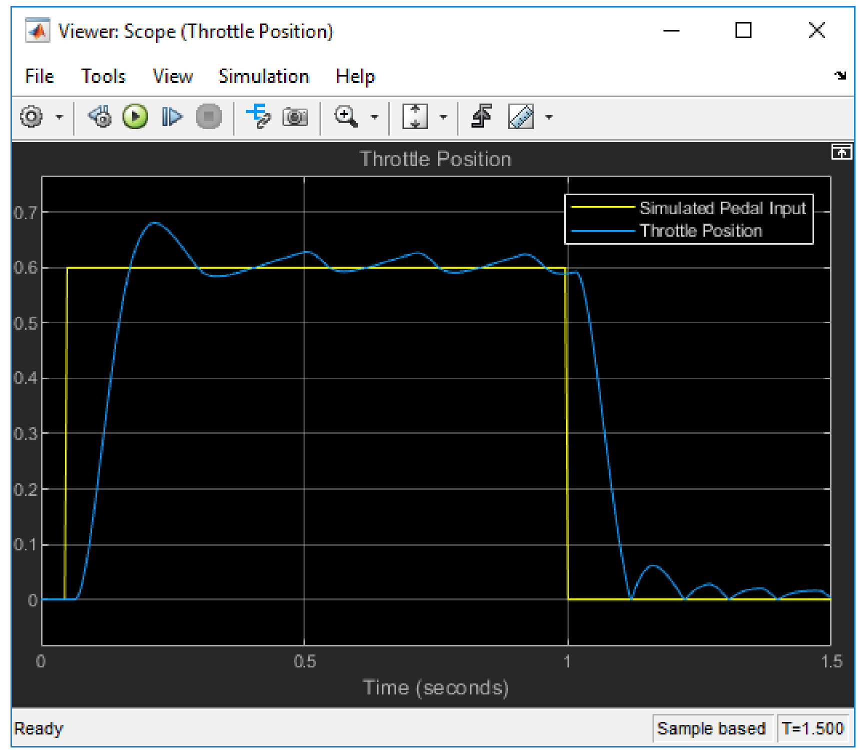
For an example of a closed-loop simulation, open example model `autosar_system`. This model provides a system-level test harness for the AUTOSAR composition model `autosar_composition`.

```
open_system('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 AUTOSAR Components, Simulate, and Generate Code” on page 4-121.

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 6-32.

See Also

`createCompositionAsModel`

Related Examples

- “Import AUTOSAR Composition to Simulink” on page 6-2
- “Design AUTOSAR Components, Simulate, and Generate Code” on page 4-121
- “Testing AUTOSAR Compositions” (Simulink Test)
- “Simulate AUTOSAR Basic Software Services and Run-Time Environment” on page 6-32

More About

- “AUTOSAR Software Components and Compositions” on page 1-11

Configure Calls to AUTOSAR Diagnostic Event Manager Service

The AUTOSAR standard defines important services as part of Basic Software (BSW) that runs in the AUTOSAR Runtime Environment (RTE). Examples include the NVRAM Manager (NvM) and the Diagnostic Event Manager (Dem). In the AUTOSAR RTE, AUTOSAR software components typically access BSW services using client-server or sender-receiver communication.

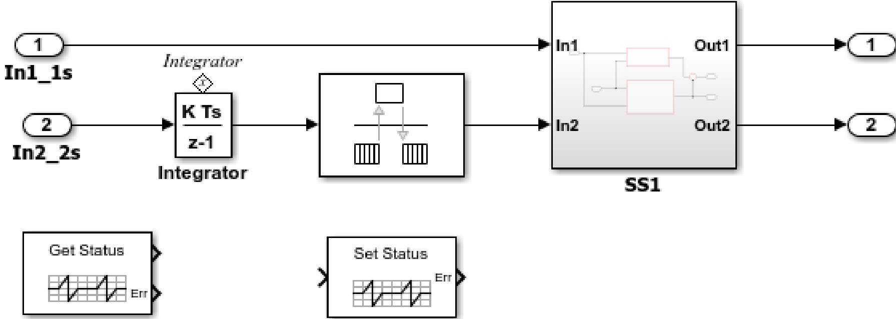
To support system-level modeling of AUTOSAR components and services, AUTOSAR Blockset provides an AUTOSAR Basic Software block library. The library contains preconfigured blocks for modeling component calls to AUTOSAR BSW services and reference implementations of the BSW services. For information about using the blocks to model client calls to AUTOSAR BSW service interfaces, see “Model AUTOSAR Basic Software Service Calls” on page 2-67.

For a live-script example of simulating AUTOSAR BSW services, see example “Simulate AUTOSAR Basic Software Services and Run-Time Environment” on page 6-32.

Here is an example of configuring client calls to Dem service interfaces in your AUTOSAR software component.

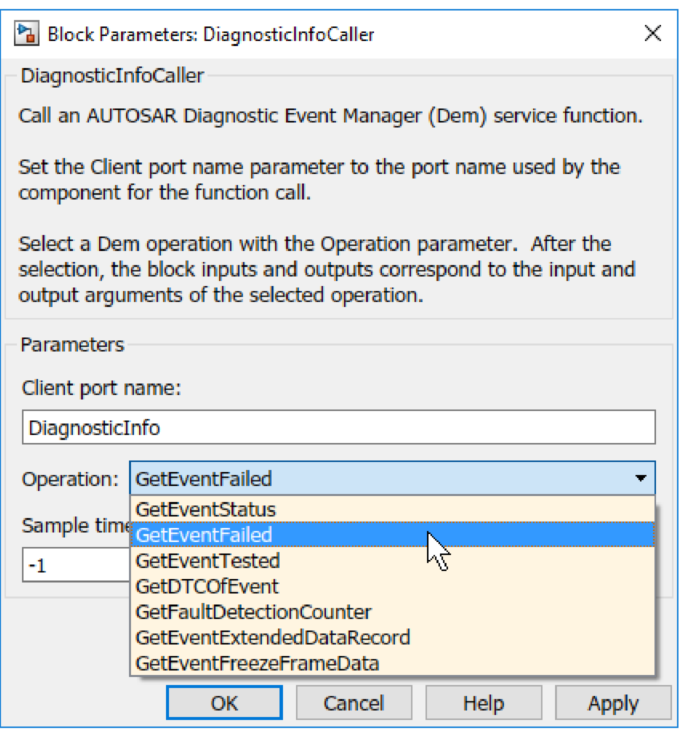
1. Open a model that is configured for AUTOSAR code generation. Using the Library Browser or by typing block names in the model window, add Dem blocks to the model. This example adds the blocks DiagnosticInfoCaller and DiagnosticMonitorCaller to a writable copy of the example model `autosar_swc`.

   As you insert each block, you are prompted for a client port name, which is the name of the AUTOSAR client port used by this component to call the BSW interface. For the purposes of this example, accept the default names, `DiagnosticInfo` and `DiagnosticMonitor`.

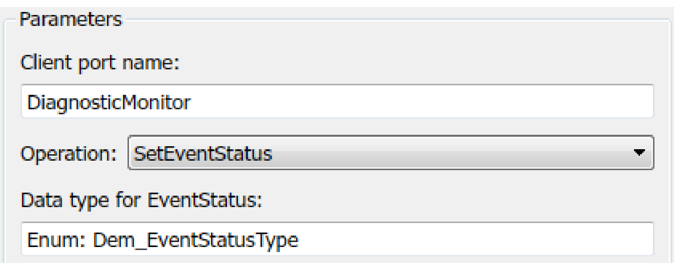


2 Open each block and examine the parameters, especially **Operation**. If you select a different operation and click **Apply**, the software updates the block inputs and outputs to match the arguments of the selected operation.

   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 6-32.)



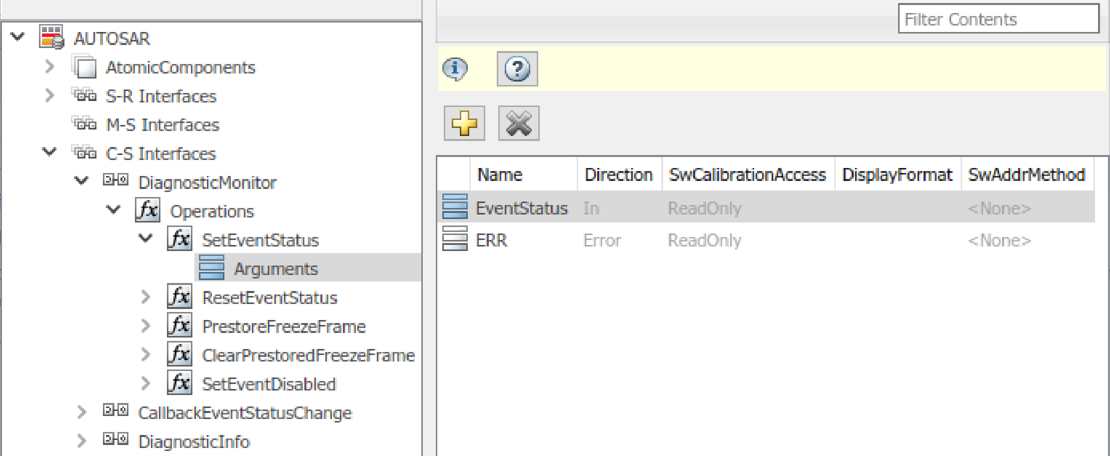
For some Dem operations, such as `GetDTCOfEvent` and `SetEventStatus`, the block parameters dialog box displays a data type parameter. The parameter specifies an enumerated data type for a function input that represents a Dem format type or event status. Default data types are provided, such as `Dem_DTCFormatType` or `Dem_EventStatusType`. For more information about format type or event status values, see the AUTOSAR standard *Specification of Diagnostic Event Manager*.



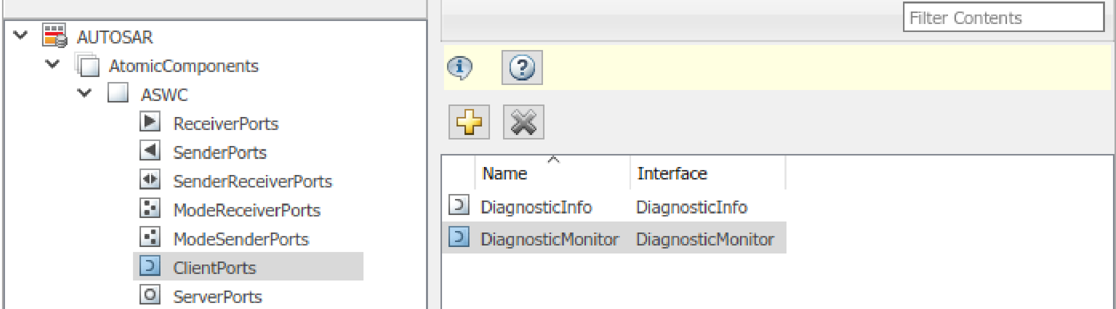
3 Open Code Mappings editor. To update the Simulink to AUTOSAR mapping of the model with changes to Simulink function callers, click the **Update** button. The software creates AUTOSAR client-service interfaces, operations, and ports, and maps each Simulink function caller to an AUTOSAR client port and operation.

For example, for the DiagnosticMonitorCaller block in this example, for which the `SetEventStatus` operation is selected:

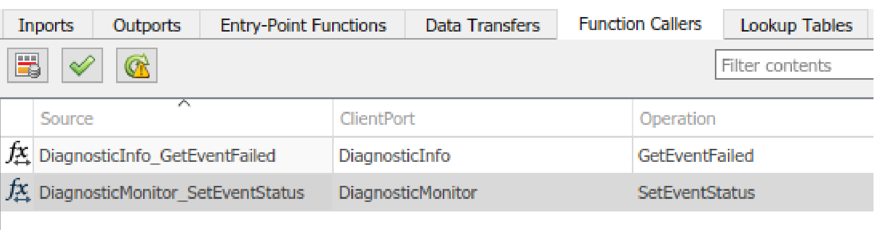
- The software creates C-S interface `DiagnosticMonitor`, and under `DiagnosticMonitor`, its supported operations. For each operation, arguments are provided with read-only properties. Here are the arguments for the `DiagnosticMonitor` operation `SetEventStatus` displayed in AUTOSAR Dictionary.



- 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.



- Code Mappings editor maps the `DiagnosticMonitor` function caller block to AUTOSAR client port `DiagnosticMonitor` and AUTOSAR operation `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:

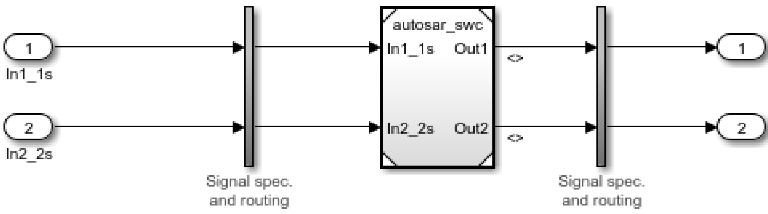
```
<SERVER-CALL-POINTS>
...
    <SYNCHRONOUS-SERVER-CALL-POINT UUID="...">
        <SHORT-NAME>SC_DiagnosticMo_334e61e63627b44b</SHORT-NAME>
        <OPERATION-IREF>
            <CONTEXT-R-PORT-REF DEST="R-PORT-PROTOTYPE">
              /Company/Powertrain/Components/ASWC/DiagnosticMonitor
            </CONTEXT-R-PORT-REF>
```

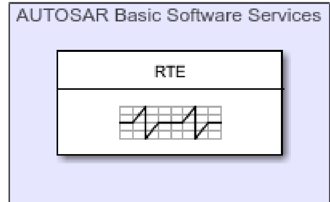
```
            <TARGET-REQUIRED-OPERATION-REF DEST="CLIENT-SERVER-OPERATION">
              /AUTOSAR/Services/Dem/DiagnosticMonitor/SetEventStatus
            </TARGET-REQUIRED-OPERATION-REF>
        </OPERATION-IREF>
        <TIMEOUT>1.0E-06</TIMEOUT>
    </SYNCHRONOUS-SERVER-CALL-POINT>
</SERVER-CALL-POINTS>
```

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 example, in the model window, select **Analysis > Test Harness > Create for Model**. In the Create Test Harness dialog box, click **OK**. The software compiles the model, adds a Diagnostic Service Component block, and creates ports and other elements required for simulation.





   For more information, see “Configure AUTOSAR Basic Software Service Implementations for Simulation” on page 6-27 and “Simulate AUTOSAR Basic Software Services and Run-Time Environment” on page 6-32.

See Also

Diagnostic Service Component | DiagnosticInfoCaller | DiagnosticMonitorCaller

Related Examples

- “Configure AUTOSAR Basic Software Service Implementations for Simulation” on page 6-27
- “Simulate AUTOSAR Basic Software Services and Run-Time Environment” on page 6-32
- “Configure AUTOSAR Client-Server Communication” on page 4-197

More About

- “Model AUTOSAR Basic Software Service Calls” on page 2-67
- “Model AUTOSAR Communication” on page 2-26

Configure Calls to AUTOSAR NVRAM Manager Service

The AUTOSAR standard defines important services as part of Basic Software (BSW) that runs in the AUTOSAR Runtime Environment (RTE). Examples include the NVRAM Manager (NvM) and the Diagnostic Event Manager (Dem). In the AUTOSAR RTE, AUTOSAR software components typically access BSW services using client-server or sender-receiver communication.

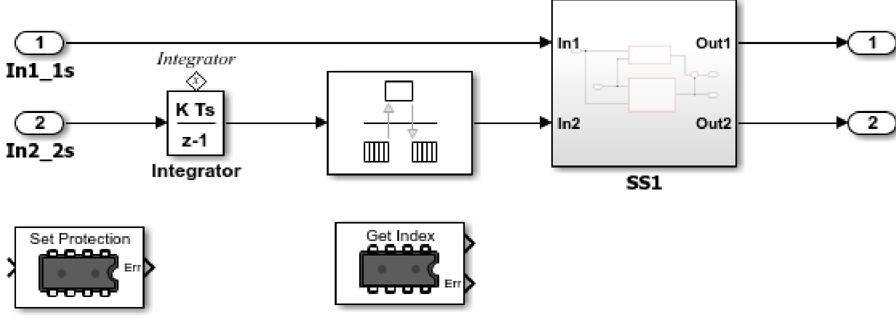
To support system-level modeling of AUTOSAR components and services, AUTOSAR Blockset provides an AUTOSAR Basic Software block library. The library contains preconfigured blocks for modeling component calls to AUTOSAR BSW services and reference implementations of the BSW services. For information about using the blocks to model client calls to AUTOSAR BSW service interfaces, see “Model AUTOSAR Basic Software Service Calls” on page 2-67.

For a live-script example of simulating AUTOSAR BSW services, see example “Simulate AUTOSAR Basic Software Services and Run-Time Environment” on page 6-32.

Here is an example of configuring client calls to NvM service interfaces in your AUTOSAR software component.

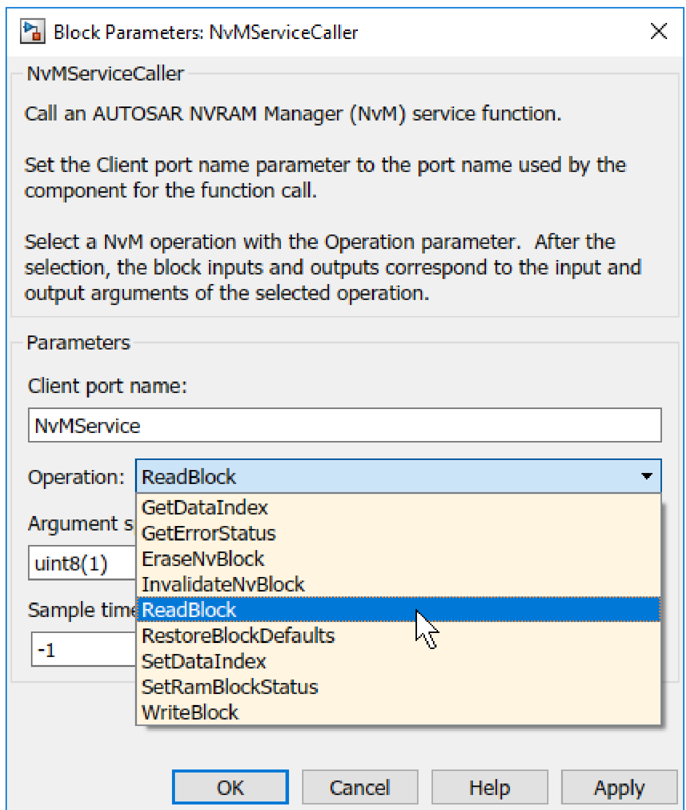
1. Open a model that is configured for AUTOSAR code generation. Using the Library Browser or by typing block names in the model window, add NvM blocks to the model. This example adds the blocks NvMAdminCaller and NvMServiceCaller to a writable copy of the example model `autosar_swc`.

   As you insert each block, you are prompted for a client port name, which is the name of the AUTOSAR client port used by this component to call the BSW interface. For the purposes of this example, accept the default names, `NvMAdmin` and `NvMService`.

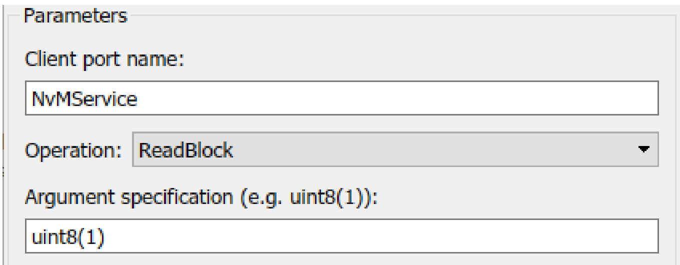


2 Open each block and examine the parameters, especially **Operation**. If you select a different operation and click **Apply**, the software updates the block inputs and outputs to match the arguments of the selected operation.

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 6-32.)



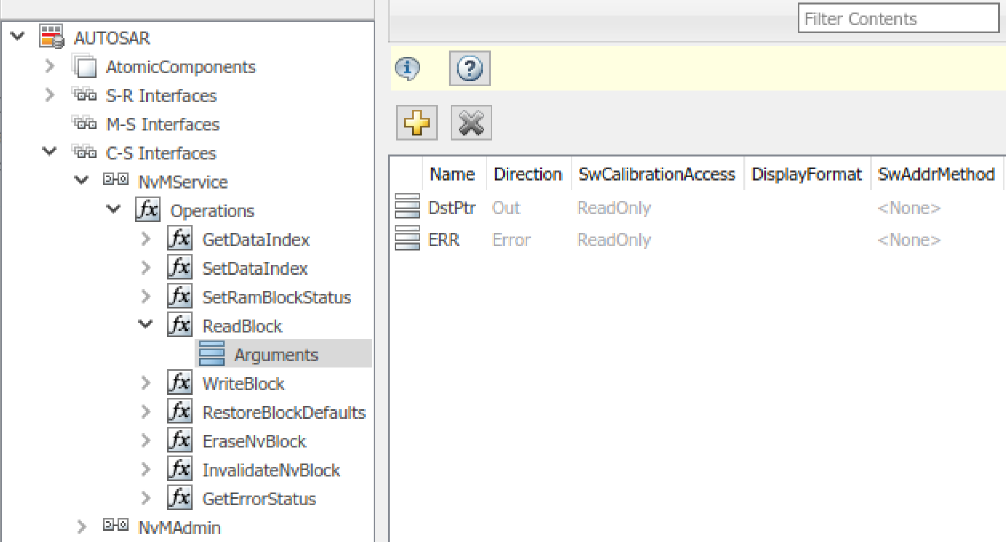
For some NvM operations, such as `ReadBlock` and `WriteBlock`, the block parameters dialog box displays an argument specification parameter. The parameter specifies data type and dimension information for data to be read or written by the operation, set to `uint8(1)` by default. You can specify array and bus data types.



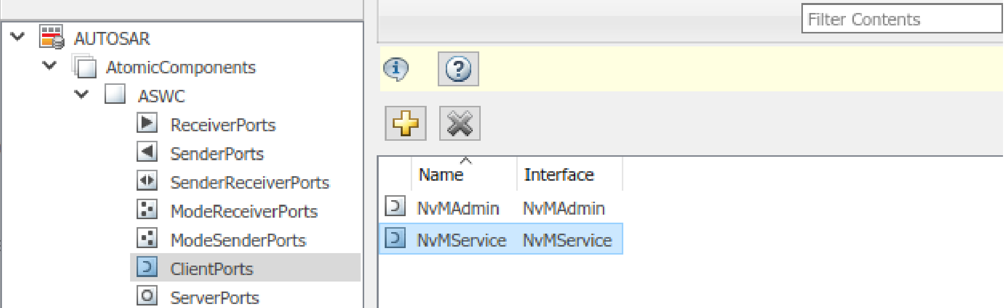
3 Open Code Mappings editor. To update the Simulink to AUTOSAR mapping of the model with changes to Simulink function callers, click the **Update** button . The software creates AUTOSAR client-service interfaces, operations, and ports, and maps each Simulink function caller to an AUTOSAR client port and operation.

For example, for the NvMServiceCaller block in this example, for which the `ReadBlock` operation is selected:

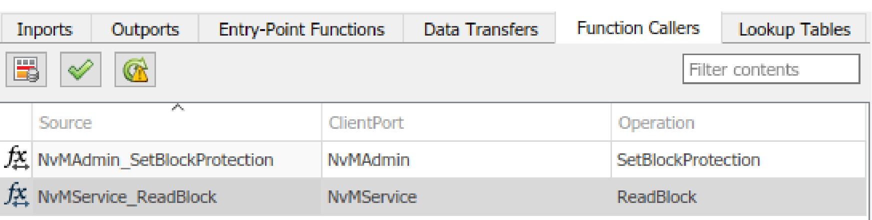
- The software creates C-S interface `NvMService`, and under `NvMService`, its supported operations. For each operation, arguments are provided with read-only properties. Here are the arguments for the `NvMService` operation `ReadBlock` displayed in AUTOSAR Dictionary.



- 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.

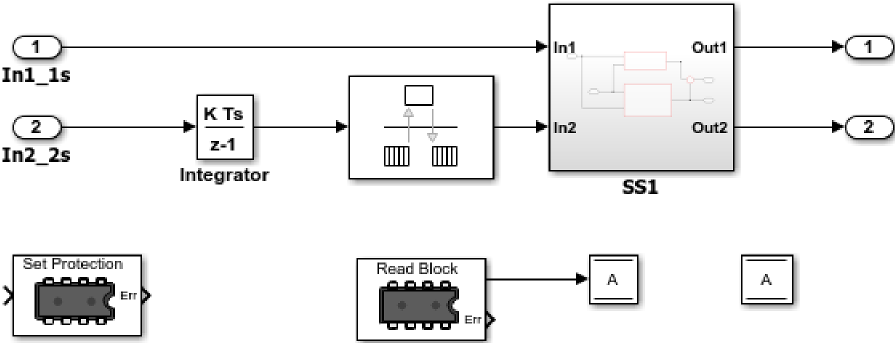


- Code Mappings editor maps the `NvMService` function caller block to AUTOSAR client port `NvMService` and AUTOSAR operation `ReadBlock`.



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 outport 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>/NvMServiceCaller' */
Rte_Call_NvMService_ReadBlock(&rtDW.A);
...
/* FunctionCaller: '<Root>/NvMAdminCaller' */
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:

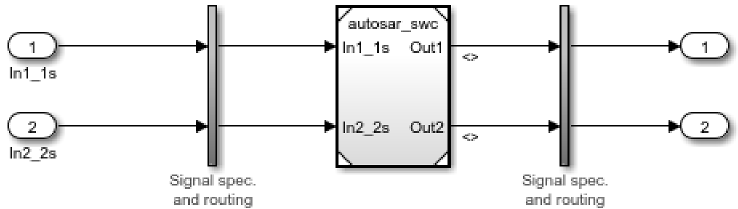
```
<SERVER-CALL-POINTS>
...
    <ASYNCHRONOUS-SERVER-CALL-POINT UUID="...">
        <SHORT-NAME>SC_NvMService_ReadBlock</SHORT-NAME>
        <OPERATION-IREF>
            <CONTEXT-R-PORT-REF DEST="R-PORT-PROTOTYPE">
              /Company/Powertrain/Components/ASWC/NvMService
            </CONTEXT-R-PORT-REF>
            <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>
```

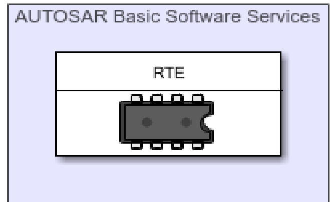
5. To simulate the component model, create a containing composition, system, or test harness model. In that containing model, insert reference implementations of the NvM `ReadBlock` and `SetBlockProtection` service operations.

   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 example, in the model window, select **Analysis > Test Harness > Create for Model**. In the Create Test Harness dialog box, click **OK**. The software compiles the model, adds a NVRAM Service Component block, and creates ports and other elements required for simulation.





For more information, see "Configure AUTOSAR Basic Software Service Implementations for Simulation" on page 6-27 and "Simulate AUTOSAR Basic Software Services and Run-Time Environment" on page 6-32.

See Also

NVRAM Service Component | NvMAdminCaller | NvMServiceCaller

Related Examples

- "Configure AUTOSAR Basic Software Service Implementations for Simulation" on page 6-27
- "Simulate AUTOSAR Basic Software Services and Run-Time Environment" on page 6-32
- "Configure AUTOSAR Client-Server Communication" on page 4-197

More About

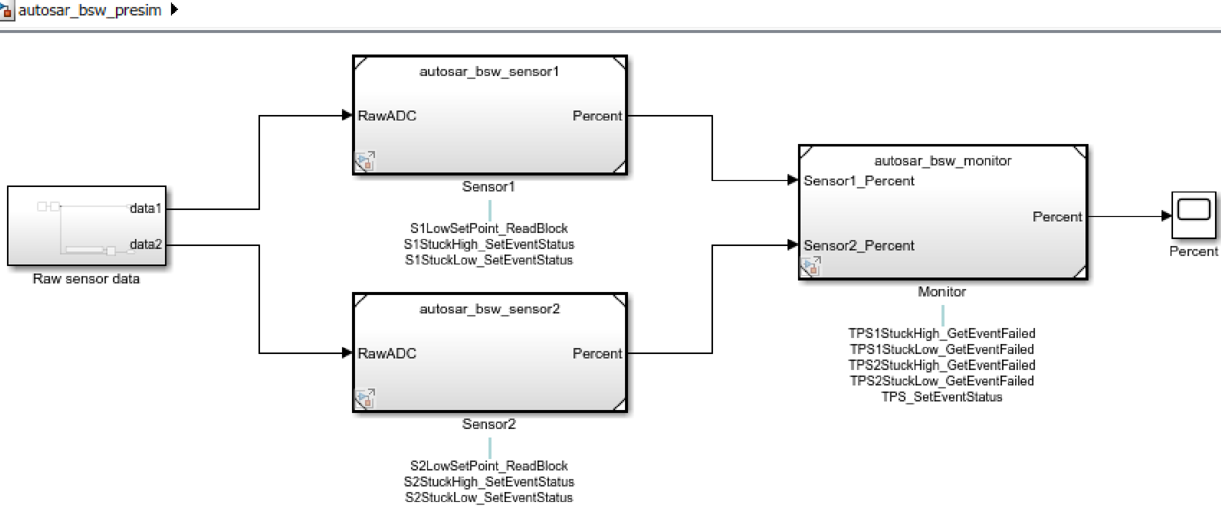
- “Model AUTOSAR Basic Software Service Calls” on page 2-67
- “Model AUTOSAR Communication” on page 2-26

Configure AUTOSAR Basic Software Service Implementations for Simulation

AUTOSAR Blockset provides reference implementations of NVRAM Manager (NvM) and Diagnostic Event Manager (Dem) 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 the AUTOSAR NvM service or AUTOSAR Dem service. Follow the procedure described in "Configure Calls to AUTOSAR NVRAM Manager Service" on page 6-20 or "Configure Calls to AUTOSAR Diagnostic Event Manager Service" on page 6-13.
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 6-32. The referenced component models call NvM service operation `ReadBlock` and Dem service operations `SetEventStatus` and `GetEventFailed`.

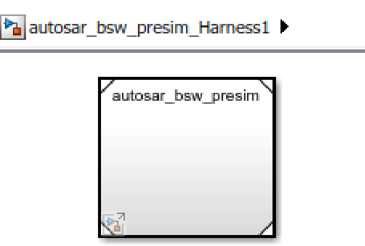


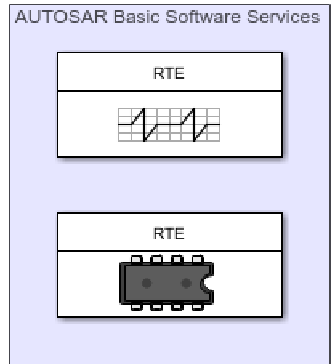
Alternatively, as shown in the next step, you can use Simulink Test to create a harness model.

3 In the containing model, provide reference implementations of the NvM or Dem service operations that your AUTOSAR component models call. For NvM and Dem service operations, the AUTOSAR Basic Software block library provides NVRAM Service Component and Diagnostic Service Component blocks.

   You can insert a Service Component block in either of two ways:

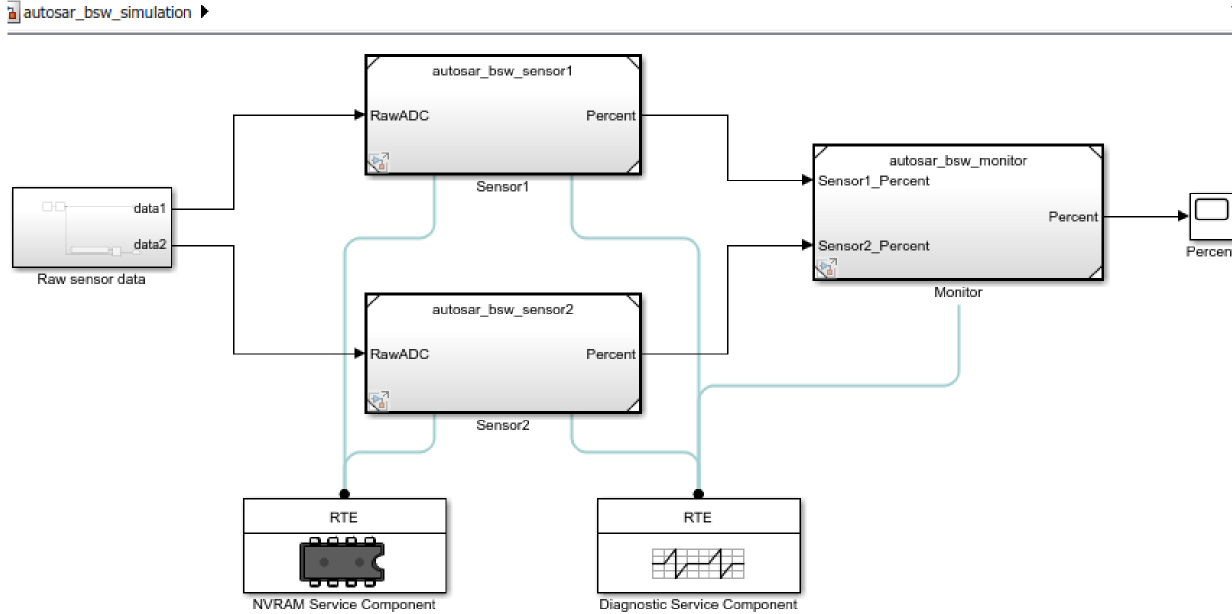
   - Automatically insert the block by creating a Simulink Test harness model. In an AUTOSAR component model or a containing model, select **Analysis** > **Test Harness** > **Create for Model**. In the Create Test Harness dialog box, click **OK**. The software compiles the model, adds an NVRAM or Diagnostic Service Component block, and creates ports and other elements required for simulation. For example, here is a test harness created for the integration model in example “Simulate AUTOSAR Basic Software Services and Run-Time Environment” on page 6-32.





- 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 6-32 uses these `add_block` commands to add NVRAM Service Component and Diagnostic Service Component blocks to a containing model.

```
add_block('autosarlibnvm/NVRAM Service Component',...
  'autosar_bsw_presim/NVRAM Service Component');
add_block('autosarlibdem/Diagnostic Service Component',...
  'autosar_bsw_presim/Diagnostic Service Component');
```



4 Each service component block has prepopulated parameters. Examine the parameter settings and consider if modifications are required, based on how you are using the NvM or Dem service operations. For more information, see NVRAM Service Component and Diagnostic Service Component.

5 Simulate the containing model. The simulation exercises the AUTOSAR NvM and Dem service calls in the component models. For a sample simulation, see example "Simulate AUTOSAR Basic Software Services and Run-Time Environment" on page 6-32.

See Also

Diagnostic Service Component | NVRAM Service Component

Related Examples

- "Simulate AUTOSAR Basic Software Services and Run-Time Environment" on page 6-32
- "Configure Calls to AUTOSAR Diagnostic Event Manager Service" on page 6-13

- “Configure Calls to AUTOSAR NVRAM Manager Service” on page 6-20

More About

- “Model AUTOSAR Basic Software Service Calls” on page 2-67
- “Model AUTOSAR Communication” on page 2-26

Simulate AUTOSAR Basic Software Services and Run-Time Environment

Simulate AUTOSAR component calls to Basic Software memory and diagnostic services using reference implementations

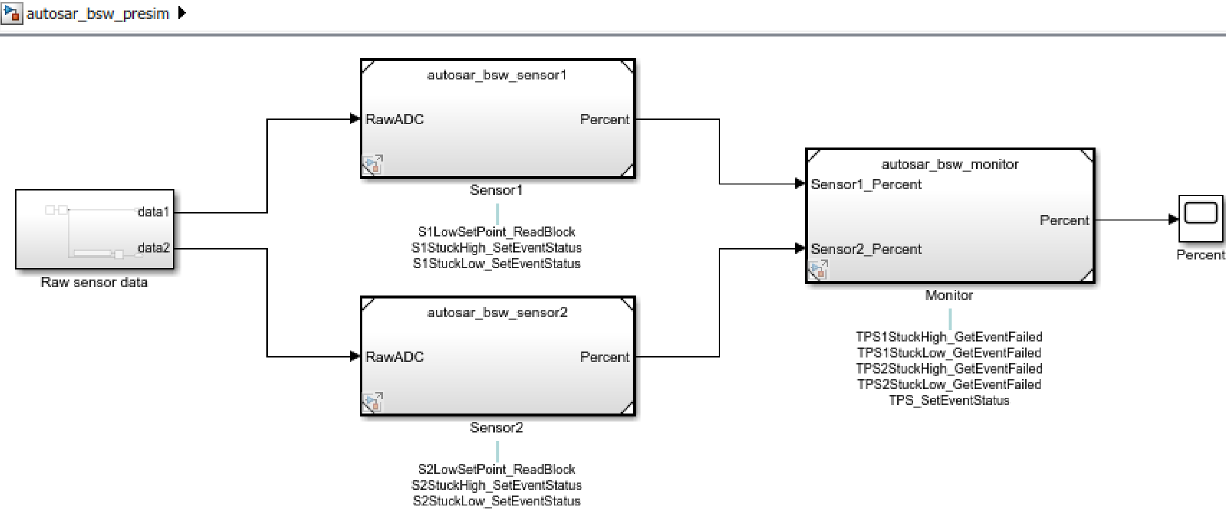
Configure Calls to AUTOSAR Basic Software Services

The AUTOSAR standard defines Basic Software (BSW) services that run in the AUTOSAR run-time environment. The services include NVRAM Manager (NvM) and Diagnostic Event Manager (Dem) services. In the AUTOSAR run-time environment, AUTOSAR software components typically access BSW services using client-server or sender-receiver communication.

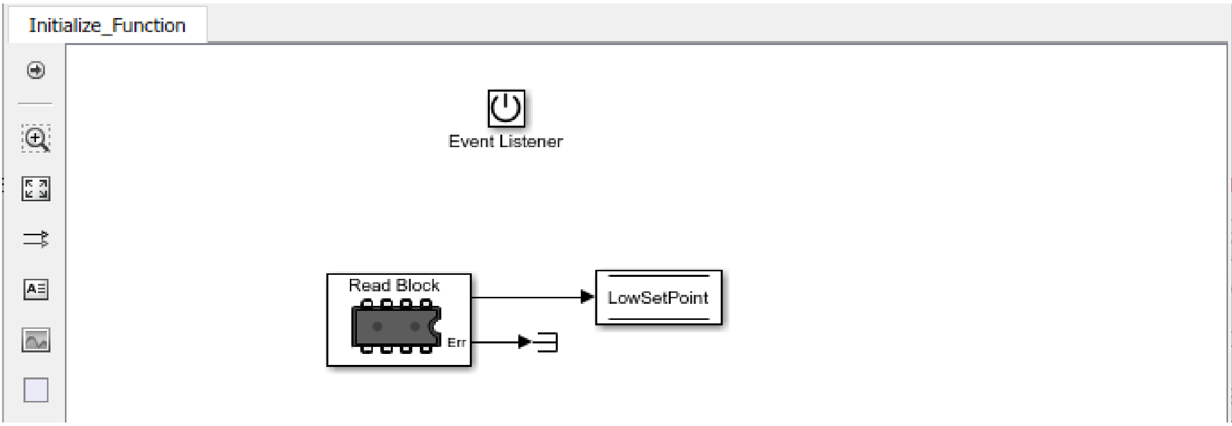
In your AUTOSAR software component model, to implement client calls to NvM and Dem service interfaces, you drag and drop preconfigured NvM and Dem caller blocks. Each block has prepopulated parameters, such as **Client port name** and **Operation**. You configure the block parameters, for example, to select an NvM or Dem 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 6-20 and “Configure Calls to AUTOSAR Diagnostic Event Manager Service” on page 6-13.

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.

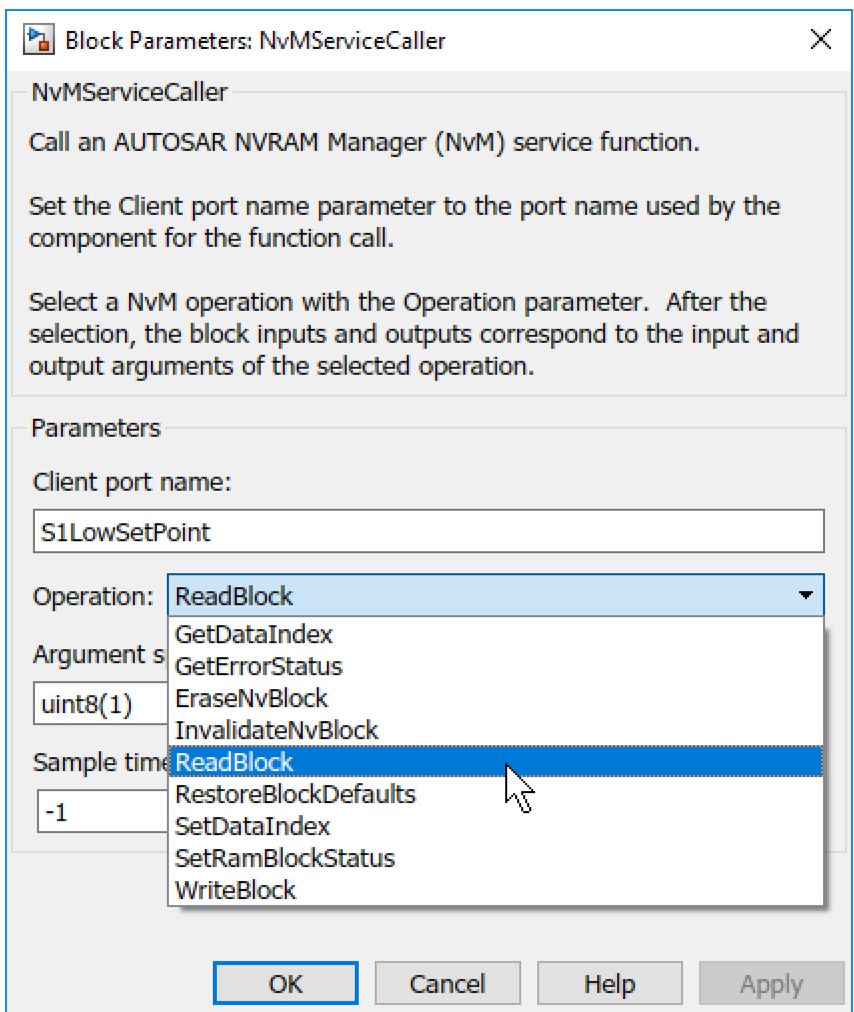
```
open_system('autosar_bsw_presim');
```



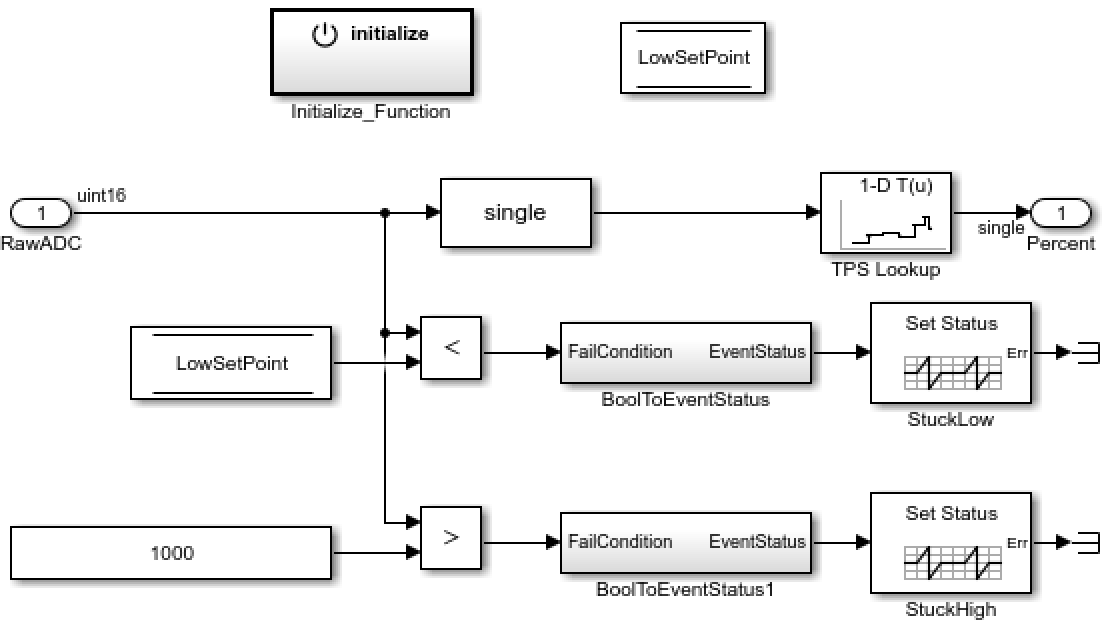
Sensor components `autosar_bsw_sensor1` and `autosar_bsw_sensor2` each contain an Initialize Function block, which calls the NvM service interface `NvMService`. The calls are implemented using the Basic Software library block NvMServiceCaller. Each block is configured to call the `NvMService` operation `ReadBlock`. The `ReadBlock` calls use client ports `S1LowSetPoint` and `S2LowSetPoint`. Here is the Initialize Function block for `autosar_bsw_sensor1`.



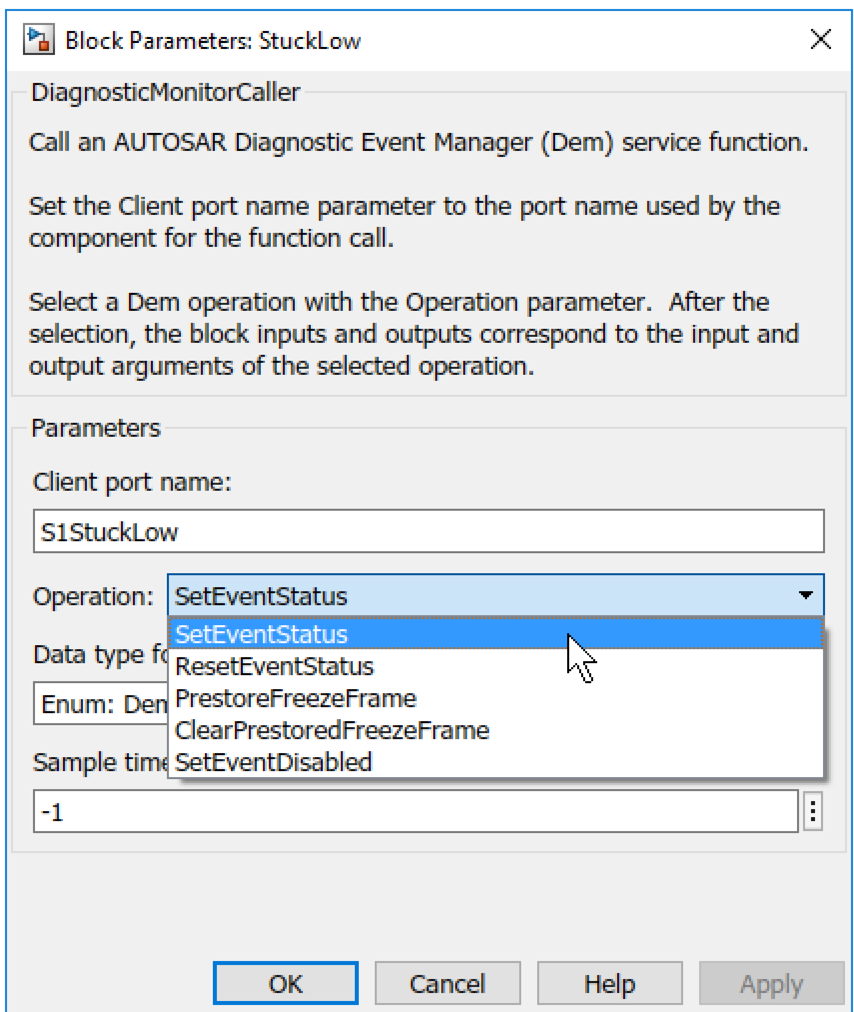
Here is the NvMServiceCaller block dialog box for the `ReadBlock` call in the Initialize Function block. For more information, see NvMServiceCaller.



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`.



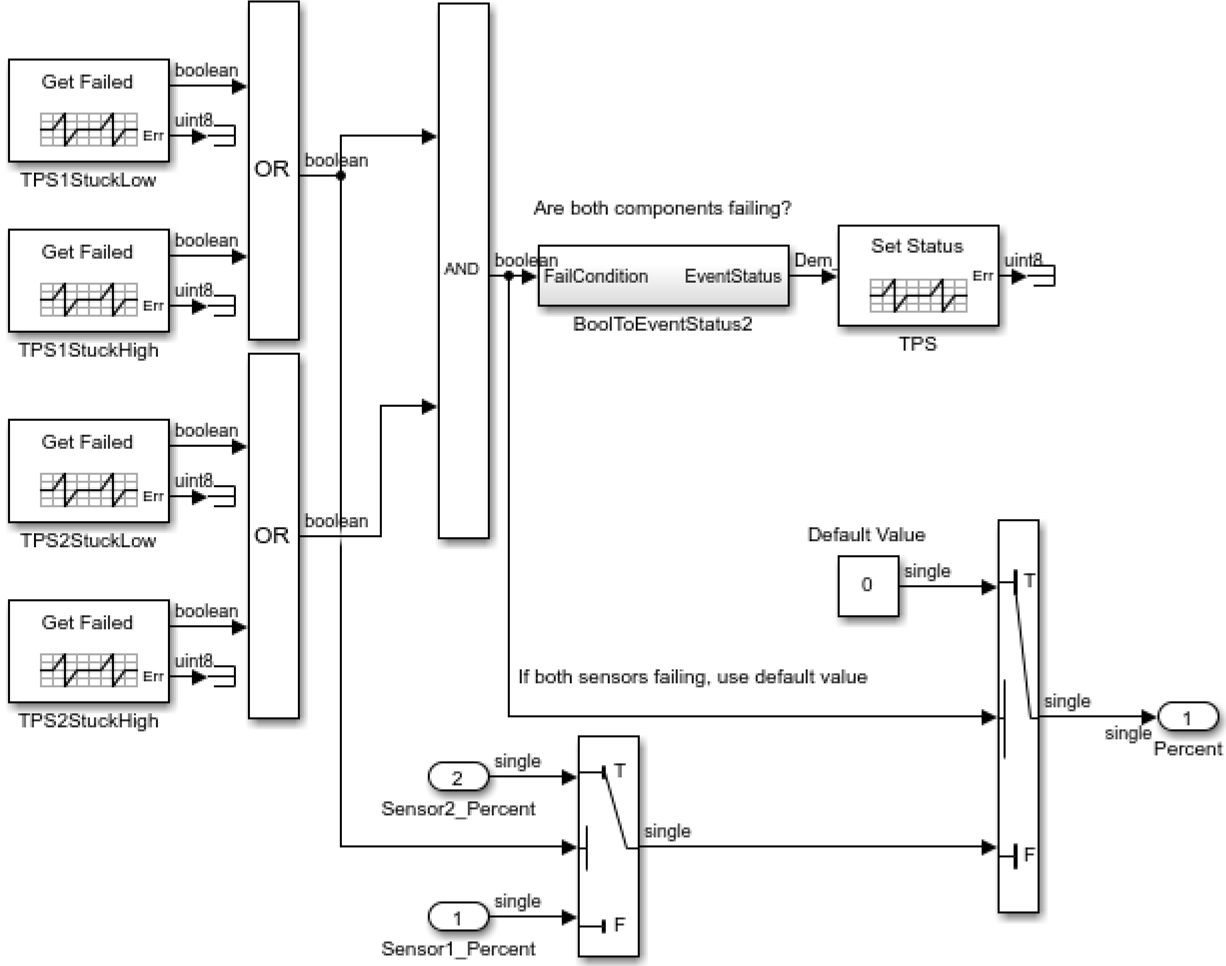
Here is the DiagnosticMonitorCaller block dialog box for the `StuckLow` call in the first sensor component. For more information, see DiagnosticMonitorCaller.



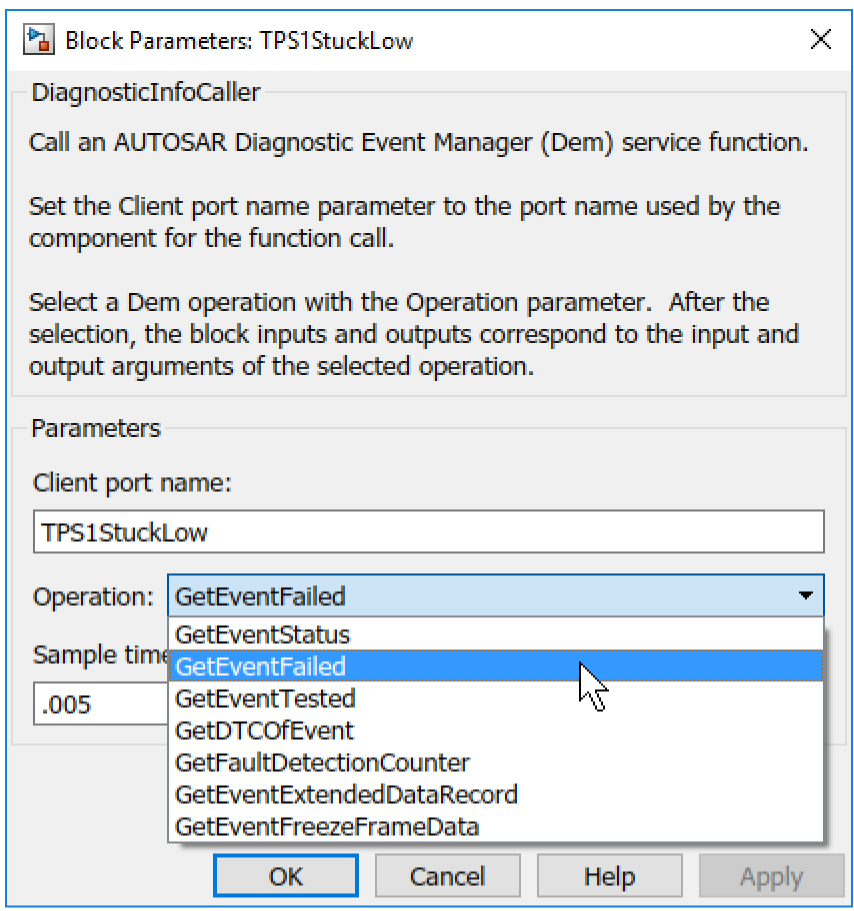
Monitor component `autosar_bsw_monitor` contains a call to the Dem service interface `DiagnosticMonitor` and four calls to the DEM service interface `DiagnosticInfo`.

- 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 are licensed for Simulink Coder and Embedded Coder, 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 `rtwbuild('autosar_bsw_monitor')`.

To see the results of the model build, examine the code generation report.

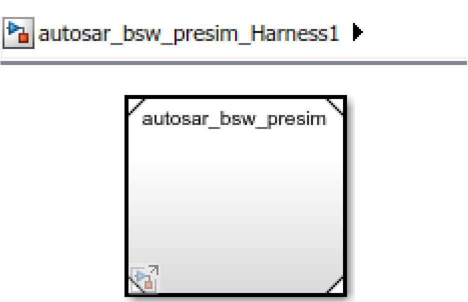
Configure Reference Implementations of AUTOSAR Basic Software Services for Simulation

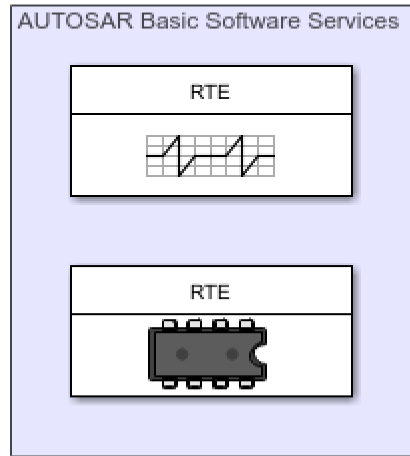
To simulate an AUTOSAR component model that calls BSW services, create a containing composition, system, or harness model. In that containing model, provide reference implementations of the NvM and Dem service operations called by the component.

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 and Dem service operations. To support simulation of component calls to the NvM and Dem 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

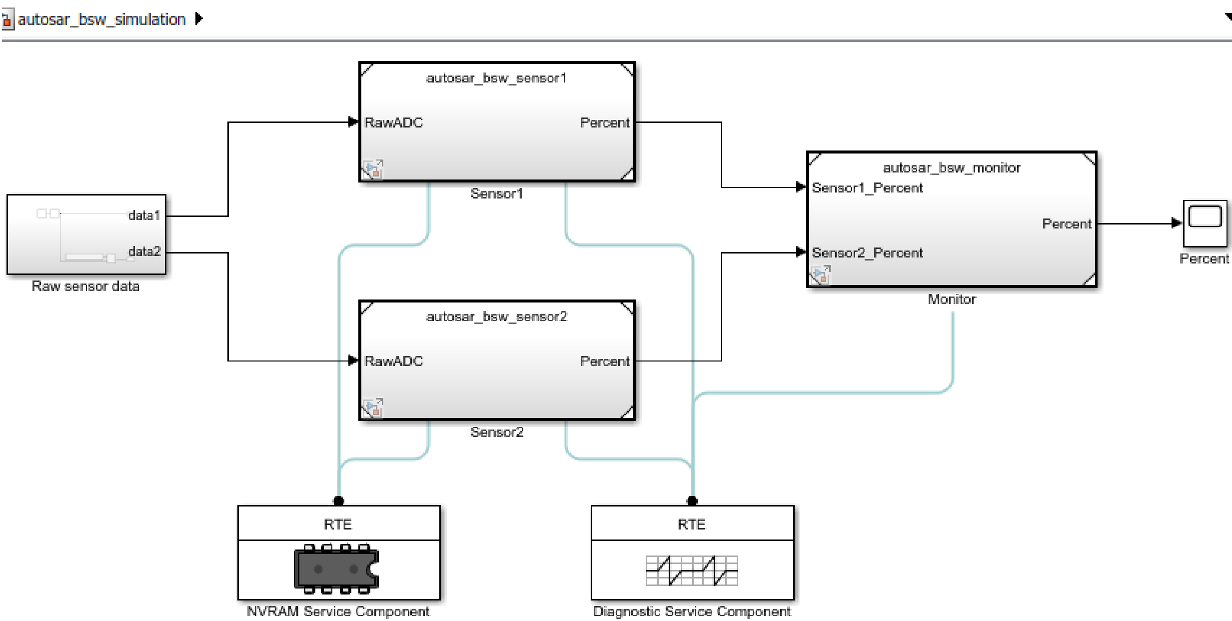
To insert Service Component blocks automatically for a model that calls BSW NvM and Dem services, open the model (or a containing model) and create a Simulink Test harness. Select **Analysis > Test Harness > Create for Model**. In the Create Test Harness dialog box, click **OK**. The software compiles the model, adds NVRAM and Diagnostic Service Component blocks, and creates ports and other elements required for simulation. For example, here is a test harness created for the throttle position integration model.



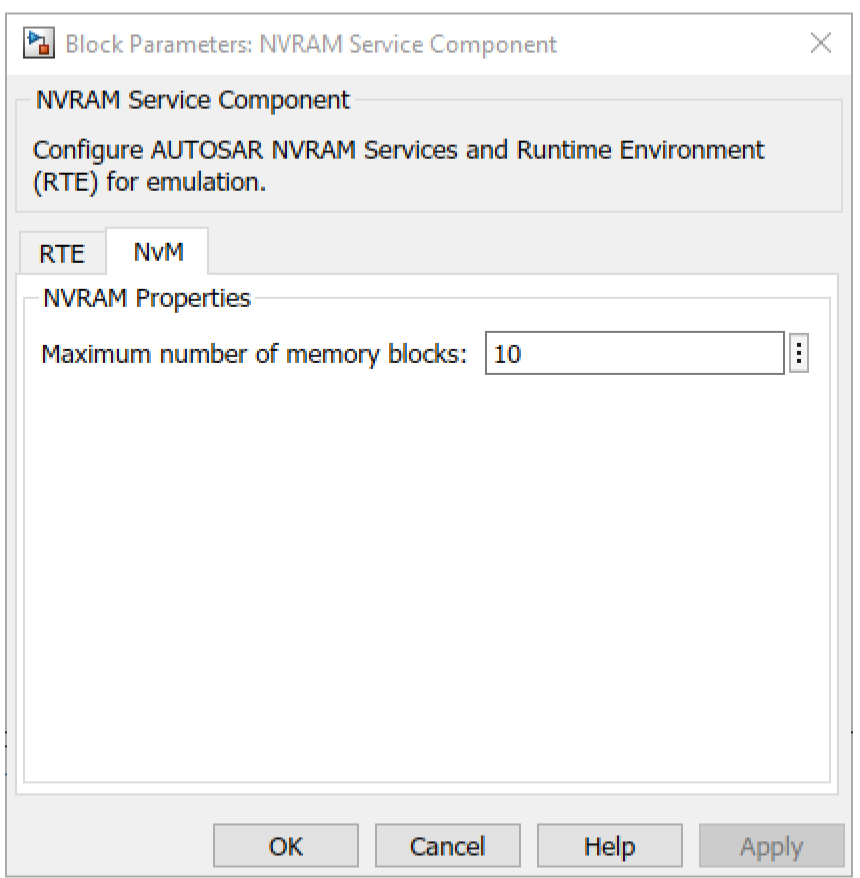


To insert Service Component blocks manually for the NvM and Dem service calls in this example, open the integration model. Using the Library Browser or `add_block` commands, or by typing block names in the model window, add the NVRAM and Diagnostic Service Component blocks to the model.

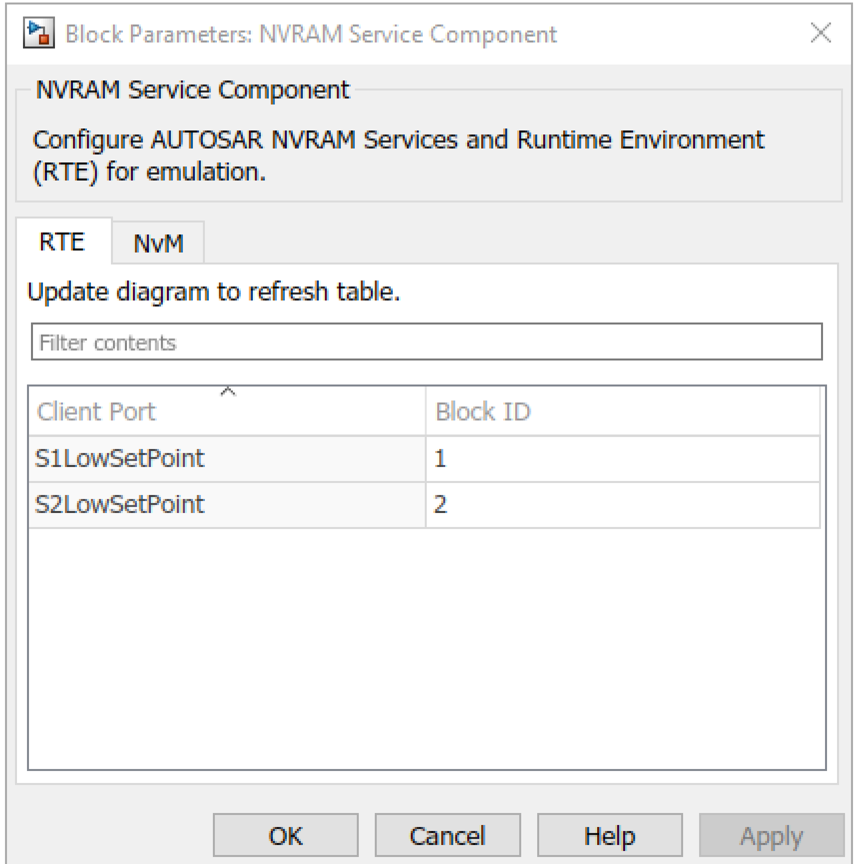
```
open_system('autosar_bsw_presim');
add_block('autosarlibnvm/NVRAM Service Component','autosar_bsw_presim/NVRAM Service Cor
add_block('autosarlibdem/Diagnostic Service Component','autosar_bsw_presim/Diagnostic S
set_param('autosar_bsw_presim','SimulationCommand','update');
```



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.

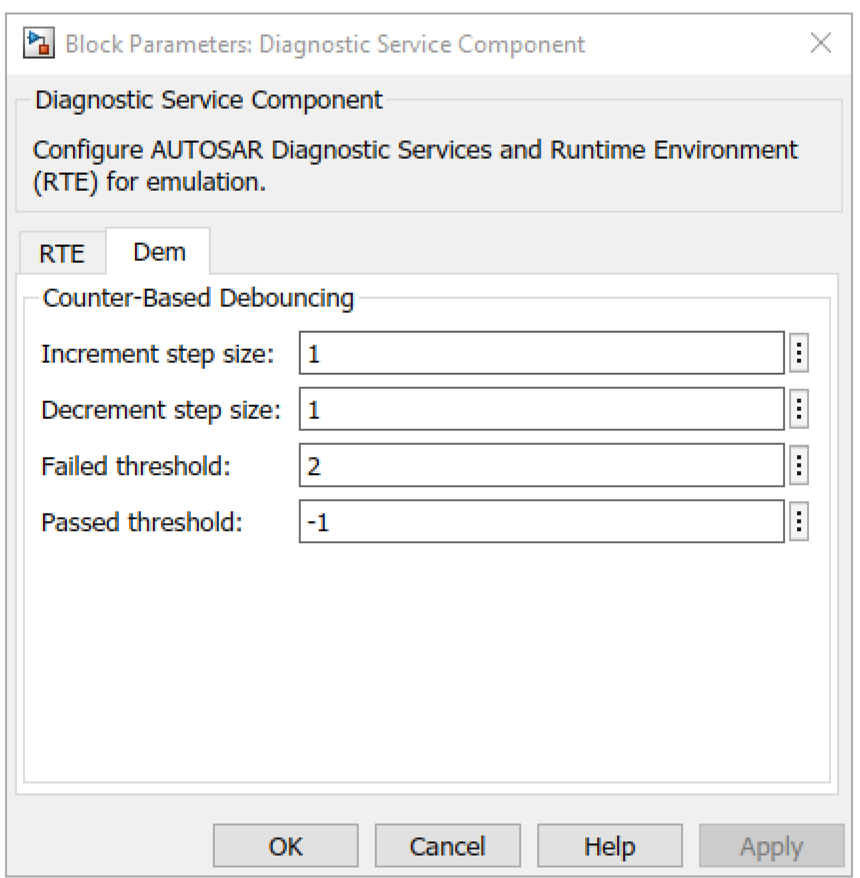


The RTE tab table lists component client ports and their mapping to NvM service block IDs. Each row in the table represents a call into NvM services from a Basic Software caller block. Calls that act on the same NvM block typically use the same block ID. This example maps the NvM `ReadBlock` client ports to different block IDs.

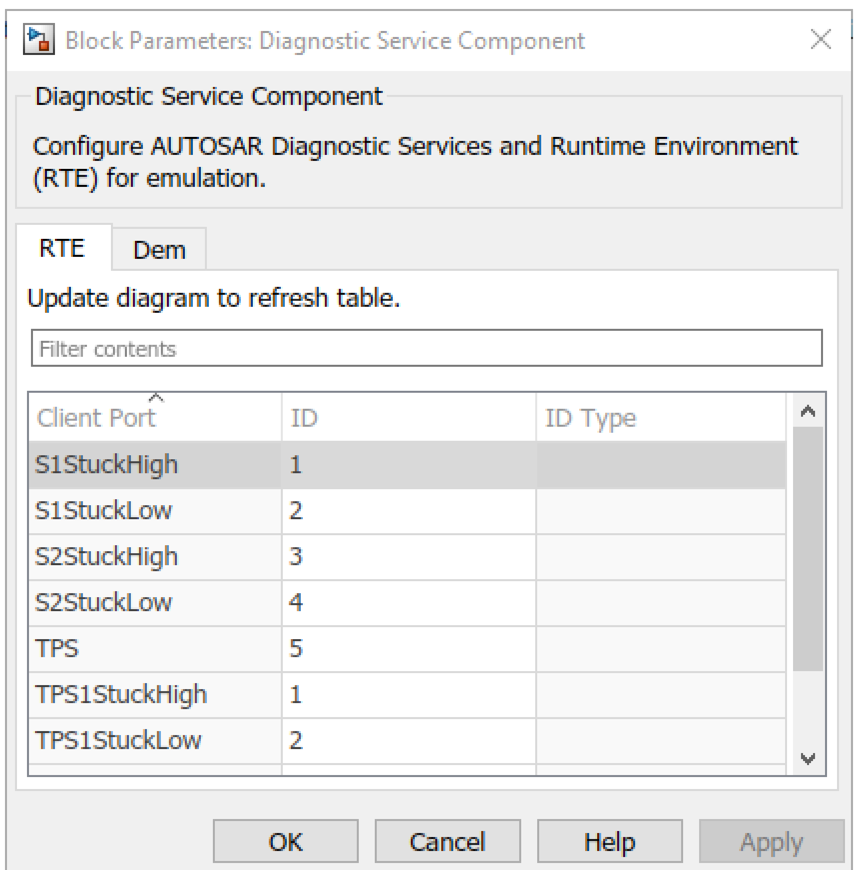


The Diagnostic Service Component block has prepopulated parameters, including RTE parameters and **Counter-Based Debouncing** parameters. Examine the parameter settings and consider if any require modifying, based on how you are using the Dem service operations.

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 service 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 manually adding NVRAM and Diagnostic Service Component blocks to 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 2-67

- "Configure Calls to AUTOSAR NVRAM Manager Service" on page 6-20
- "Configure Calls to AUTOSAR Diagnostic Event Manager Service" on page 6-13
- "Configure AUTOSAR Basic Software Service Implementations for Simulation" on page 6-27