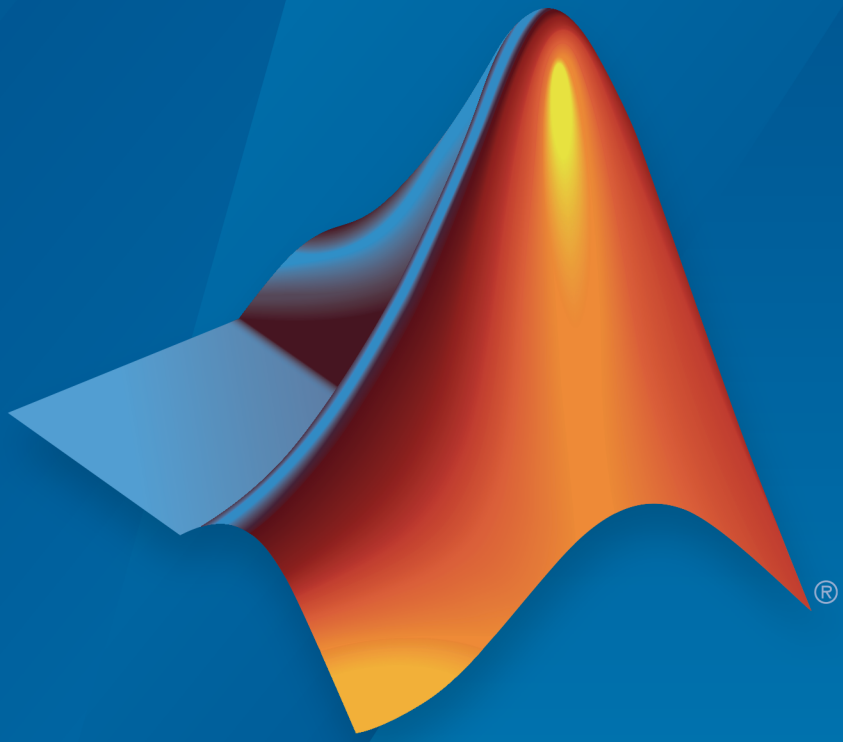
Embedded Coder®

AUTOSAR



MATLAB® & SIMULINK®

R2016b

MathWorks®

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.
3 Apple Hill Drive
Natick, MA 01760-2098

Embedded Coder® AUTOSAR

© COPYRIGHT 2014–2016 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

October 2014	Online only	New for Version 6.7 (Release 2014b)
March 2015	Online only	Revised for Version 6.8 (Release 2015a)
September 2015	Online only	Revised for Version 6.9 (Release 2015b)
October 2015	Online only	Rereleased for Version 6.8.1 (Release 2015aSP1)
March 2016	Online only	Revised for Version 6.10 (Release 2016a)
September 2016	Online only	Revised for Version 6.11 (Release 2016b)

Check Bug Reports for Issues and Fixes

Software is inherently complex and is not free of errors. The output of a code generator might contain bugs, some of which are not detected by a compiler. MathWorks reports critical known bugs brought to its attention on its Bug Report system at www.mathworks.com/support/bugreports/. Use the Saved Searches and Watched Bugs tool with the search phrase "Incorrect Code Generation" to obtain a report of known bugs that produce code that might compile and execute, but still produce wrong answers.

The bug reports are an integral part of the documentation for each release. Examine periodically all bug reports for a release, as such reports may identify inconsistencies between the actual behavior of a release you are using and the behavior described in this documentation.

In addition to reviewing bug reports, you should implement a verification and validation strategy to identify potential bugs in your design, code, and tools.

Contents

1 Overview of AUTOSAR Support

AUTOSAR Standard 1-2

Support Package for AUTOSAR Standard 1-3

Workflows for AUTOSAR 1-4

Sample Workflows 1-6

AUTOSAR Terminology 1-7

2 Modeling Patterns for AUTOSAR

Simulink Modeling Patterns for AUTOSAR 2-2

Model AUTOSAR Software Components 2-3
- About AUTOSAR Software Components 2-3
- Multi-Instance Software Components 2-4
- Multi-Runnable Software Components 2-5
- Startup, Reset, and Shutdown 2-9

Model AUTOSAR Communication 2-11
- About AUTOSAR Communication 2-11
- Sender-Receiver Interface 2-12
- Client-Server Interface 2-13
- Mode-Switch Interface 2-15
- Nonvolatile Data Interface 2-20
- Parameter Interface 2-20
- Trigger Interface 2-21

Model AUTOSAR Basic Software (BSW) Service Calls **2-22**

Model AUTOSAR Calibration Parameters and Lookup Tables **2-24**
About AUTOSAR Calibration Parameters **2-24**
Import and Export Calibration Parameters **2-24**
Calibration Parameters for STD_AXIS and COM_AXIS Lookup Tables **2-25**

Model AUTOSAR Component Behavior **2-28**
AUTOSAR Elements for Modeling Component Behavior . . . **2-28**
Runnables **2-28**
Inter-Runnable Variables **2-29**
System Constants **2-30**
Per-Instance Memory **2-31**
Static and Constant Memory **2-32**

Model AUTOSAR Data Types **2-34**
About AUTOSAR Data Types **2-34**
Enumerated Data Types **2-38**
Structure Parameters **2-39**
Release 2.x and 3.x Data Types **2-39**
Release 4.x Data Types **2-40**
CompuMethod Categories for Data Types **2-43**

Model AUTOSAR Variants **2-46**
Variants in Ports and Runnables **2-46**
Variants in Array Sizes **2-47**
Variants in Runnable Condition Logic **2-48**
Predefined Variants and System Constant Value Sets **2-48**

3 AUTOSAR Component Creation

AUTOSAR arxml Importer Tool **3-2**

Import AUTOSAR Software Component **3-4**
General arxml Import Workflow **3-4**
Import Multi-Runnable Software Component **3-6**

Import AUTOSAR Software Component Updates **3-8**
Update Model with AUTOSAR Software Component Changes **3-8**
AUTOSAR Update Report Section Examples **3-10**

Round-Trip Preservation of AUTOSAR Elements and UUIDs **3-14**

Create AUTOSAR Software Component in Simulink **3-15**

Import or Update Shared AUTOSAR Reference Element Definitions **3-22**

Limitations and Tips **3-24**
Cannot Save Importer Objects in MAT-Files **3-24**
ApplicationRecordDataType and ImplementationDataType Element Names Must Match **3-24**

4 AUTOSAR Component Development

AUTOSAR Interface Configuration **4-3**

Configure AUTOSAR Component Using AUTOSAR Properties Explorer **4-6**

Map Model Elements Using Simulink-AUTOSAR Mapping Explorer **4-60**

Configure AUTOSAR Package for Interface or CompuMethod **4-67**

Configure AUTOSAR Packages **4-70**
AR-PACKAGE Structure **4-70**
Configure AUTOSAR Packages and Paths **4-72**
Control AUTOSAR Elements Affected by Package Path Modifications **4-75**
Export AUTOSAR Packages **4-75**
AR-PACKAGE Location in Exported ARXML Files **4-79**

Configure AUTOSAR Provide-Require Port **4-83**

Configure AUTOSAR Port aliveTimeout Value **4-86**

Configure AUTOSAR Receiver Port for IsUpdated Service **4-87**

Configure AUTOSAR Sender Port for Data Element Invalidation **4-90**

Configure AUTOSAR S-R Interface Port for End-To-End Protection **4-94**

Configure AUTOSAR Receiver Port for DataReceiveErrorEvent **4-97**

Configure AUTOSAR Client-Server Communication **4-101**
Configure AUTOSAR Server **4-101**
Configure AUTOSAR Client **4-112**
Configure AUTOSAR Client-Server Error Handling **4-120**
Concurrency Constraints for AUTOSAR Server Runnables **4-126**
Configure and Map AUTOSAR Server and Client Programmatically **4-128**

Configure AUTOSAR Mode-Switch Communication **4-130**
Configure Mode Receiver Port and Mode-Switch Event for Mode User **4-130**
Configure Mode Sender Port and Mode Switch Point for Application Mode Manager **4-137**

Configure AUTOSAR Nonvolatile Data Communication . . **4-141**

Configure Receiver for AUTOSAR Parameter Communication **4-144**

Configure Receiver for AUTOSAR External Trigger Event Communication **4-148**

Configure Calls to AUTOSAR Diagnostic Event Manager (Dem) Service **4-153**

Configure Calls to AUTOSAR NVRAM Manager (NvM) Service **4-161**

Configure AUTOSAR Internal Calibration Parameters . . 4-167

Configure AUTOSAR Port-Based Calibration Parameters 4-169

Configure AUTOSAR Calibration Component 4-170

Configure STD_AXIS and COM_AXIS Lookup Tables for AUTOSAR Measurement and Calibration 4-174

Configure COM_AXIS Lookup Table Using AUTOSAR.Parameter Objects 4-184

Configure AUTOSAR Data for Measurement and Calibration 4-190
About Software Data Definition Properties (`SwDataDefProps`) 4-190
Configure SwCalibrationAccess 4-191
Configure DisplayFormat 4-194
Configure SwAddrMethod 4-198
Configure SwAlignment 4-202
Export SwImplPolicy 4-203
Export SwRecordLayout for Lookup Table Data 4-203

Configure AUTOSAR Multiple Runnables 4-205
Configure Multiple Runnables Using Rate-Based Multitasking 4-205
Configure Multiple Runnables Using Function Calls 4-211

Configure AUTOSAR Initialize, Reset, or Terminate Runnables 4-215

Add Top-Level Asynchronous Trigger to Periodic Rate-Based System 4-222

Configure AUTOSAR Initialization Runnable (R4.1) 4-226

Configure Disabled Mode for AUTOSAR Runnable Event 4-229

Configure AUTOSAR Per-Instance Memory 4-230

Configure AUTOSAR Static or Constant Memory 4-234

Configure AUTOSAR Release 4.x Data Types **4-237**
Control Application Data Type Generation **4-237**
Configure DataTypeMappingSet Package and Name **4-238**
Initialize Data with ApplicationValueSpecification **4-240**

Configure AUTOSAR CompuMethods **4-241**
Configure AUTOSAR CompuMethod Properties **4-241**
Create AUTOSAR CompuMethods **4-243**
Configure CompuMethod Direction for Linear Functions . . **4-244**
Export CompuMethod Unit References **4-246**
Modify Linear Scaling for SCALE_LINEAR_AND_TEXTTABLE CompuMethod **4-246**
Configure Rational Function CompuMethod for Dual-Scaled Parameter **4-248**

Configure AUTOSAR Internal Data Constraints Export . . **4-254**

Configure AUTOSAR Variants in Ports and Runnables . . **4-256**

Configure AUTOSAR Variants in Array Sizes **4-259**

Configure AUTOSAR Variants in Runnable Condition Logic **4-261**

Control AUTOSAR Variants with Predefined Value Combinations **4-264**

Configure and Map AUTOSAR Component Programmatically **4-267**
AUTOSAR Property and Map Functions **4-267**
Tree View of AUTOSAR Configuration **4-268**
Properties of AUTOSAR Elements **4-269**
Specify AUTOSAR Element Location **4-272**

AUTOSAR Property and Map Function Examples **4-274**
Configure AUTOSAR Software Component **4-275**
Configure AUTOSAR Interfaces **4-287**
Configure AUTOSAR XML Export **4-293**

Limitations and Tips **4-296**
AUTOSAR Client Block in Referenced Model **4-296**
Use the Merge Block for Inter-Runnable Variables **4-296**

5 AUTOSAR Code Generation

Export AUTOSAR Component XML and C Code **5-2**
Inspect XML Options **5-2**
Select an AUTOSAR Schema **5-2**
Specify Maximum SHORT-NAME Length **5-4**
Configure AUTOSAR Compiler Abstraction Macros **5-4**
Root-Level Matrix I/O **5-6**
Export AUTOSAR Software Component **5-6**

Code Generation with AUTOSAR Library **5-8**
AUTOSAR Code Replacement Library **5-8**
Supported AUTOSAR Library Routines **5-9**
Configure Code Generator to Use AUTOSAR Code Replacement Library **5-9**
Replace Code with Functions Compatible with AUTOSAR IFL and IFX Library Routines **5-9**
Required Algorithm Property Settings for IFL/IFX Function and Block Mappings **5-10**
Code Replacement Checks for AUTOSAR Lookup Table Functions **5-27**

Verify AUTOSAR C Code with SIL and PIL **5-29**

Limitations and Tips **5-30**
Generate Code Only Check Box **5-30**
AUTOSAR Compiler Abstraction Macros **5-30**
Relative File Paths in AUTOSAR Code Descriptors (Schema Versions 3.x and Earlier) **5-31**

6 Functions — Alphabetical List

7 Blocks — Alphabetical List

1

Overview of AUTOSAR Support

- "AUTOSAR Standard" on page 1-2
- "Support Package for AUTOSAR Standard" on page 1-3
- "Workflows for AUTOSAR" on page 1-4
- "Sample Workflows" on page 1-6
- "AUTOSAR Terminology" on page 1-7

AUTOSAR Standard

Embedded Coder® software supports *AUTomotive Open System ARchitecture* (AUTOSAR), an open and standardized automotive software architecture. Automobile manufacturers, suppliers, and tool developers jointly develop AUTOSAR components.

The AUTOSAR standard addresses:

- Architecture – Three layers, *Application*, *Runtime Environment* (RTE), and *Basic Software*, enable decoupling of AUTOSAR software components 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.
- Methodology – Specification of code formats and description file templates, for example.
- Application Interfaces – Specification of interfaces for typical automotive applications.

For more information, see:

- `www.autosar.org` for details on the AUTOSAR standard.
- "Modeling Patterns for AUTOSAR" to model AUTOSAR software components and related concepts in Simulink®.
- "Workflows for AUTOSAR" on page 1-4 to use the code generator to produce code and description files that are compliant with AUTOSAR.
- `http://www.mathworks.com/automotive/standards/autosar.html` to learn about using MathWorks® products and third-party tools for AUTOSAR.

Support Package for AUTOSAR Standard

Embedded Coder software provides add-on support for the AUTOSAR standard via the Embedded Coder Support Package for AUTOSAR Standard. With the support package installed, you can perform a wide range of AUTOSAR-related workflows in Simulink, including:

- Create and modify an AUTOSAR configuration for a model
- Model AUTOSAR elements
- Generate `arxml` and AUTOSAR-compatible C code from a model

To install support packages:

- On the MATLAB® **Home** tab, in the **Environment** section, click **Add-Ons > Get Hardware Support Packages**.

To uninstall support packages:

- On the MATLAB **Home** tab, in the **Environment** section, click **Add-Ons > Manage Add-Ons**.

To update existing support packages:

- On the MATLAB **Home** tab, in the **Environment** section, click **Add-Ons > Check for Updates > Hardware Support Packages**.

For more information, see “Support Package Installation”.

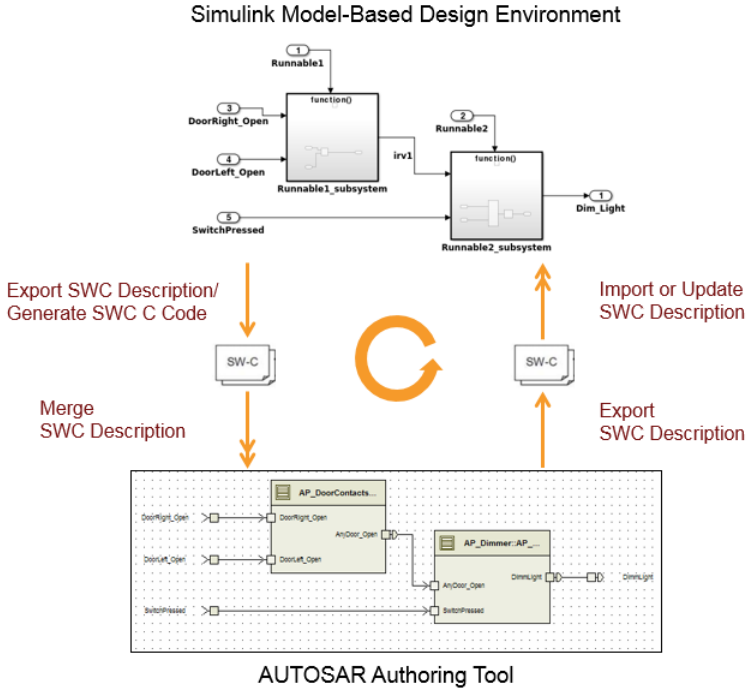
Workflows for AUTOSAR

This topic describes how you use Embedded Coder software to configure a Simulink representation of an AUTOSAR application for model-based design, and subsequently generate AUTOSAR-compliant code from the model.

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.

The following diagram shows the round-trip workflow.



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 3-4
   - "Configure AUTOSAR Internal Calibration Parameters" on page 4-167

2. Develop the model using Simulink model-based design. This process includes configuring AUTOSAR elements, mapping Simulink model elements to AUTOSAR component elements, and validating the AUTOSAR interface. See:

   - "AUTOSAR Interface Configuration" on page 4-3
   - "Configure AUTOSAR Multiple Runnables" on page 4-205
   - "Configure AUTOSAR Client-Server Communication" on page 4-101
   - "Configure and Map AUTOSAR Component Programmatically" on page 4-267

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

   - "Export AUTOSAR Software Component" on page 5-6
   - "Generate AUTOSAR-Compliant Code for Multiple Runnable Entities"

   You can also verify your generated code in a simulation. See "Verify AUTOSAR C Code with SIL and PIL" on page 5-29.

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

   You can use the authoring tool to export specifications, which can be imported back into Simulink. If there are AUTOSAR software component changes in the `arxml` code, you can update the model to reflect the changes. See "Import AUTOSAR Software Component Updates" on page 3-8.

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 Simulink model-based design and use Simulink to create a customized AUTOSAR component. See "Create AUTOSAR Software Component in Simulink" on page 3-15. Subsequent tasks in the workflow are as listed above.

Sample Workflows

Embedded Coder provides the following examples to demonstrate AUTOSAR workflows.

Example	How to ...
"Generate Code That Is Compliant with the AUTOSAR Standard"	From a rate-based Simulink model, configure AUTOSAR elements, and generate AUTOSAR-compliant code and export AUTOSAR XML files.
"Generate AUTOSAR-Compliant Code for Multiple Runnable Entities"	From a function-call based Simulink model with multiple runnables, configure AUTOSAR elements, and generate AUTOSAR-compliant code and export AUTOSAR XML files.
"Model for AUTOSAR Platform"	Model AUTOSAR atomic software components in Simulink, using rate-based and function-call-based approaches.
"AUTOSAR Property and Map Function Examples" on page 4-274	Programmatically add AUTOSAR elements to a model, configure AUTOSAR properties, and map Simulink elements to AUTOSAR elements.
"Configure AUTOSAR Client-Server Communication" on page 4-101	Model AUTOSAR servers and clients in Simulink for simulation and code generation.
"Configure AUTOSAR Mode-Switch Communication" on page 4-130	Configure AUTOSAR mode-switch interfaces, including mode receiver ports and mode-switch events, in Simulink.

AUTOSAR Terminology

Term	Notes

AUTOSAR Runtime Environment (RTE)	• Layer between Application and Basic Software layers
	• Realizes communication between:
	  • AUTOSAR software components
	  • AUTOSAR software components and Basic Software

AUTOSAR Software Component	• A software component containing one or more algorithms, which communicates with its environment through ports • Connected to the AUTOSAR Runtime Environment (RTE) • Relocatable (not tied to a particular ECU)
Characteristics	Values of characteristics can be changed on an ECU through a calibration data management tool or an offline calibration tool.
Client-Server Interface	• PortInterface for client-server communication • Defines operations provided by server and used by client
Composite data types	Category of data types, such as one of the following: • Array — Contains more than one element of the same type, and has zero-based indexing • Record — Non-empty set of objects, where each object has a unique identifier
ComSpec	Defines specific communication attributes.
DataElementPrototype (data element)	Data value (signal) exchanged between a sender and a receiver.
Data types	• Either primitive or composite • Types data elements, arguments of operations in a Client-Server Interface, and constants

Term	Notes
ErrorStatus	Indicates errors detected by communication system. Runtime Environment defines the following macros for sender-receiver communication: • `RTE_E_OK: no errors` • `RTE_E_INVALID: data element invalid` • `RTE_E_MAX_AGE_EXCEEDED: data element outdated`
OperationPrototype (operation)	• Invoked by a client • Provides value for each argument with direction `in` or `inout`, which must be of the corresponding data type • Client expects to receive a response to the invoked operation, part of which is a value with direction `out` or `inout`
PortInterface	• Characterizes information provided or required by a port • Can be either Sender-Receiver Interface or Client-Server Interface
Primitive data types	Category of data types that allow a direct mapping to C intrinsic types.
Provide port (PPort)	Port providing data or service of a server.
Require port (RPort)	Port requiring data or service of a server.

Term	Notes
RTEEvent	Event or situation that triggers execution of a runnable by the Runtime Environment (RTE). The software supports the following RTEEvents: • TimingEvent • DataReceivedEvent • DataReceiveErrorEvent • ExternalTriggerOccurredEvent • ModeSwitchEvent • OperationInvokedEvent (applicable to server operations) • InitEvent
Runnable entity (runnable)	Part of AUTOSAR Software-Component that can be executed and scheduled independently of other runnable entities (runnables).
Sender-Receiver Interface	• PortInterface for sender-receiver communication • Defines data elements sent by sending component (with `Provide` port providing Sender-Receiver Interface) or received by receiving component (with `Require` requiring Sender-Receiver Interface)
Sender-Receiver Annotation	Annotation of data elements in a port that implements Sender-Receiver Interface.
Sensor Actuator Software Component	AUTOSAR software component dedicated to the control of a sensor or actuator.
Service	Logical entity of Basic Software that offers functionality, which is used by various AUTOSAR software components.

Modeling Patterns for AUTOSAR

- "Simulink Modeling Patterns for AUTOSAR" on page 2-2
- "Model AUTOSAR Software Components" on page 2-3
- "Model AUTOSAR Communication" on page 2-11
- "Model AUTOSAR Basic Software (BSW) Service Calls" on page 2-22
- "Model AUTOSAR Calibration Parameters and Lookup Tables" on page 2-24
- "Model AUTOSAR Component Behavior" on page 2-28
- "Model AUTOSAR Data Types" on page 2-34
- "Model AUTOSAR Variants" on page 2-46

Simulink Modeling Patterns for AUTOSAR

The following topics present Simulink modeling patterns for common AUTOSAR elements. You can leverage these modeling patterns when developing models for AUTOSAR-compliant code generation.

- "Model AUTOSAR Software Components" on page 2-3
- "Model AUTOSAR Communication" on page 2-11
- "Model AUTOSAR Basic Software (BSW) Service Calls" on page 2-22
- "Model AUTOSAR Calibration Parameters and Lookup Tables" on page 2-24
- "Model AUTOSAR Component Behavior" on page 2-28
- "Model AUTOSAR Data Types" on page 2-34
- "Model AUTOSAR Variants" on page 2-46

Model AUTOSAR Software Components

In Simulink, you can flexibly model the structure and behavior of AUTOSAR software components. 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 leverage Simulink modeling styles, such as rate-based single-tasking, rate-based multitasking, and function-call based.

In this section...
"About AUTOSAR Software Components" on page 2-3
"Multi-Instance Software Components" on page 2-4
"Multi-Runnable Software Components" on page 2-5
"Startup, Reset, and Shutdown" on page 2-9

About AUTOSAR Software Components

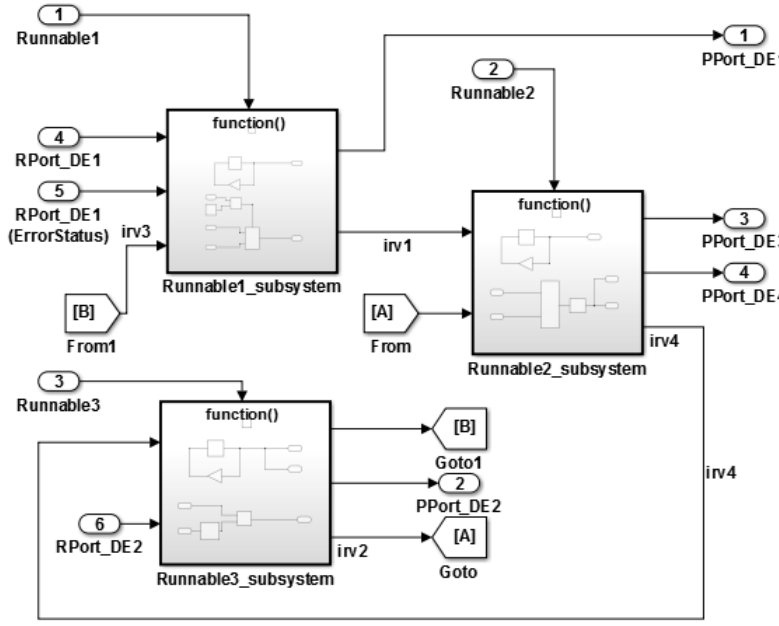
In AUTOSAR, application software consists of separate units, AUTOSAR *software components* (SWCs).

Note: An AUTOSAR software component is sometimes referred to as *atomic*, because it is never split across more than one Electronic Control Unit (ECU). Do not confuse *atomic* in this context with the Simulink concept of atomic subsystems.

For information about AUTOSAR calibration components, see "Model AUTOSAR Calibration Parameters and Lookup Tables" on page 2-24

An AUTOSAR software component interacts with other AUTOSAR software components via well-defined connection points called *ports*. The behavior of an AUTOSAR software component is implemented by one or more *runnable entities* (runnables).

In Simulink, you represent an AUTOSAR software component using a model. For example, in the example model `rtwdemo_autosar_multirunnables`, shown below, the model represents an AUTOSAR software component, and function-call subsystems represent runnables that implement its behavior. The function-call subsystems are labeled `Runnable1_subsystem`, `Runnable2_subsystem`, and `Runnable3_subsystem`.



You also can implement a multi-runnable component using Simulink rate-based multitasking, without function-call subsystems. For more information, see “Multi-Runnable Software Components” on page 2-5.

For examples of different ways to model AUTOSAR atomic software components, see “Model for AUTOSAR Platform”.

Multi-Instance Software 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**.
- 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 if the model uses exported function modeling style and contains any of the following blocks:

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

Multi-Runnable Software Components

- "Model Multiple Runnables Using Rate-Based Multitasking" on page 2-6
- "Model Multiple Runnables Using Function Calls" on page 2-7

Simulink supports the following styles for modeling an AUTOSAR software component that contains multiple runnable entities:

- *Rate-based multitasking modeling style* — Create or import model content with multiple 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 one initialization runnable and *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.

  Select this modeling style if you want multiple runnable entities with rate-based multitasking execution semantics.
- *Function-call-based modeling style* — 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 one initialization runnable and *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 this modeling style if you want multiple runnable entities with function-call execution semantics.

For examples of different ways to model AUTOSAR multi-runnable software components, see “Model for AUTOSAR Platform”.

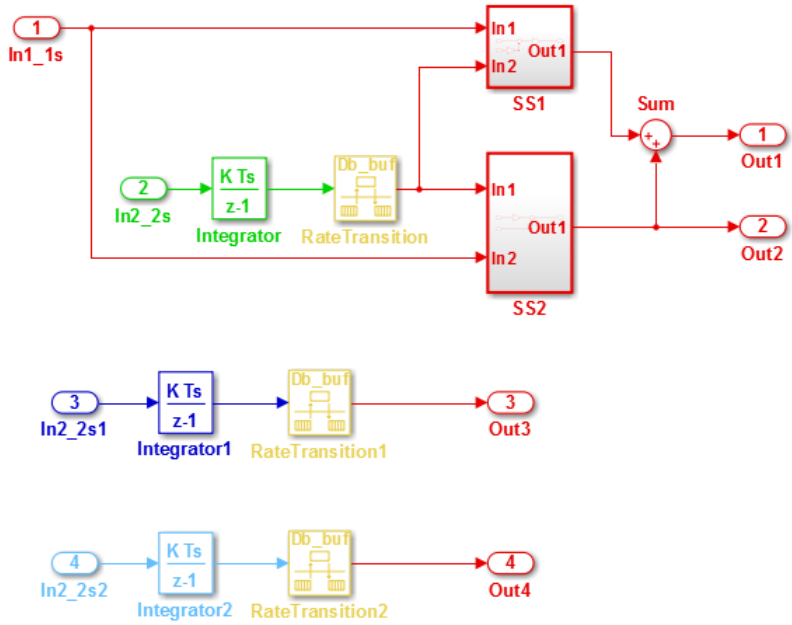
Model Multiple Runnables Using Rate-Based Multitasking

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 the `createComponentAsModel` method of the class arxml.importer with property `CreateInternalBehavior` omitted or set to `false`.)
- 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/ecoder/examples/autosar/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.

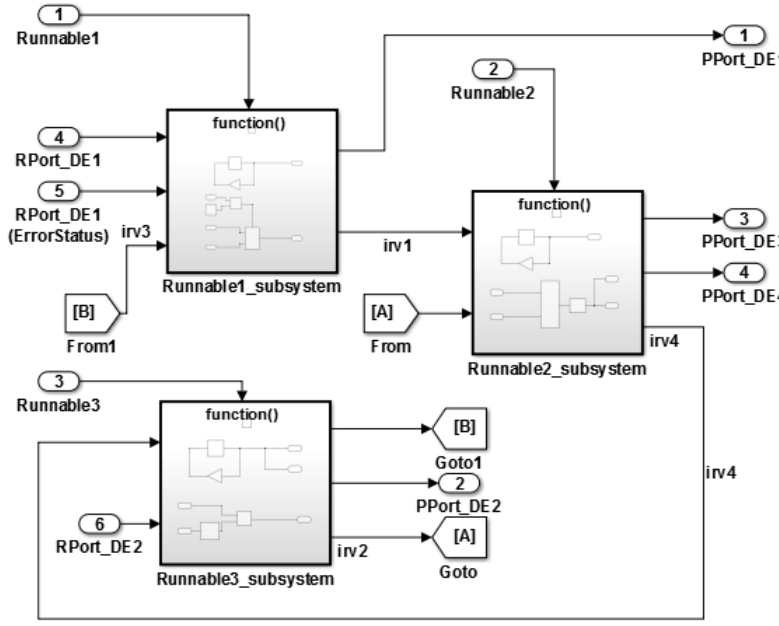


When you generate code, the model C code contains rate-grouped model step functions corresponding to AUTOSAR runnables, one for each discrete rate in the model. (The periodic step functions must be called in the manner of a rate-monotonic scheduler.) For more information, see “Configure Multiple Runnables Using Rate-Based Multitasking” on page 4-205.

Model Multiple Runnables Using Function Calls

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. 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 `rtwdemo_autosar_multirunnables`, which you can open at the MATLAB command line.) The model represents an AUTOSAR software component, and the function-call subsystems labeled `Runnable1_subsystem`, `Runnable2_subsystem`, and `Runnable3_subsystem` represent runnables that implement its behavior. The signal lines `irv1`, `irv2`, `irv3`, and `irv4` represent AUTOSAR IRVs.



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 “Configure Multiple Runnables Using Function Calls” on page 4-211.

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. Use the **Event** parameter of the Event Listener block to specify the event type —initialize, reset, or terminate—of the function. 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 leverage 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 4-215.

See Also

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

Related Examples

- “Import AUTOSAR Software Component” on page 3-4

- “Model for AUTOSAR Platform”
- “Configure Multiple Runnables Using Rate-Based Multitasking” on page 4-205
- “Configure Multiple Runnables Using Function Calls” on page 4-211
- “Configure AUTOSAR Initialize, Reset, or Terminate Runnables” on page 4-215
- “Add Top-Level Asynchronous Trigger to Periodic Rate-Based System” on page 4-222
- “Export AUTOSAR Component XML and C Code” on page 5-2

More About

- “AUTOSAR Interface Configuration” on page 4-3

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, and trigger communication.

In this section...
"About AUTOSAR Communication" on page 2-11
"Sender-Receiver Interface" on page 2-12
"Client-Server Interface" on page 2-13
"Mode-Switch Interface" on page 2-15
"Nonvolatile Data Interface" on page 2-20
"Parameter Interface" on page 2-20
"Trigger Interface" on page 2-21

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:

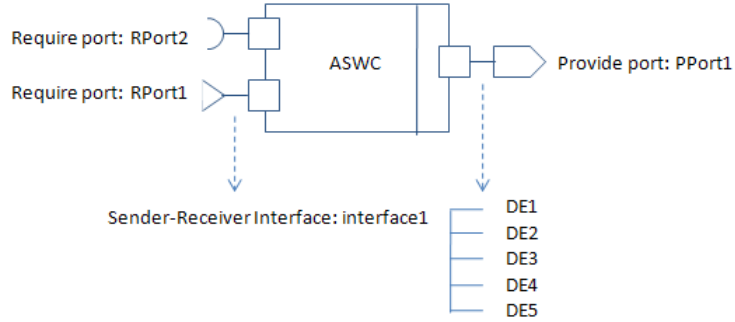
- Sender-Receiver
- Client-Server
- Mode-Switch
- Nonvolatile Data
- Parameter
- Trigger

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*.

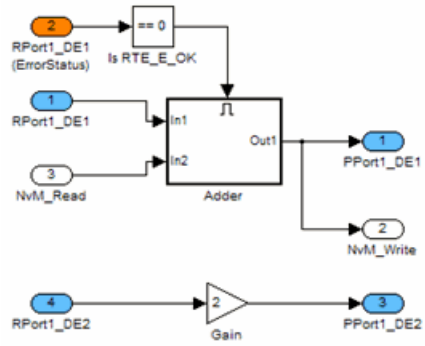
Sender-Receiver Interface

A Sender-Receiver Interface consists of one or more data elements. Although a `Require`, `Provide`, or `Provide-Require` port may 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 4-83.

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

Use the Configure AUTOSAR Interface dialog box to specify the AUTOSAR settings for each inport and outport. For information on how you specify settings, see "AUTOSAR Interface Configuration" on page 4-3.

Client-Server Interface

AUTOSAR allows client-server communication between:

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

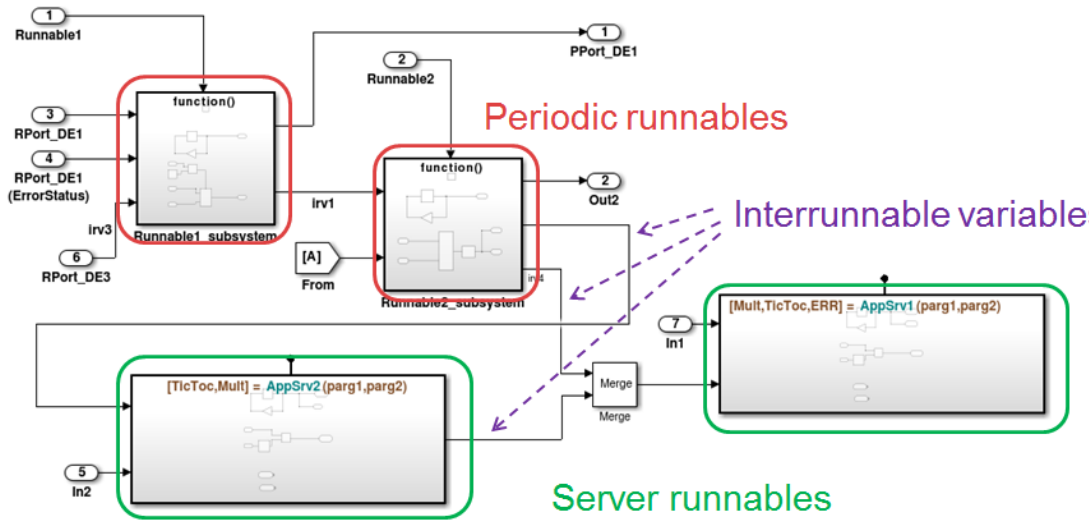
An AUTOSAR Client-Server Interface defines the interaction between a software component that *provides* the interface and a software component that *requires* the

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:

- Use `Simulink Function` blocks at the root level of a model to model AUTOSAR servers.
- Use `Function Caller` blocks to model AUTOSAR client invocations.
- 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.

The following 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 run-time environment (RTE).)

For more information, see "Configure AUTOSAR Client-Server Communication" on page 4-101.

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 2-15
- "Application Mode Manager" on page 2-19

Mode User

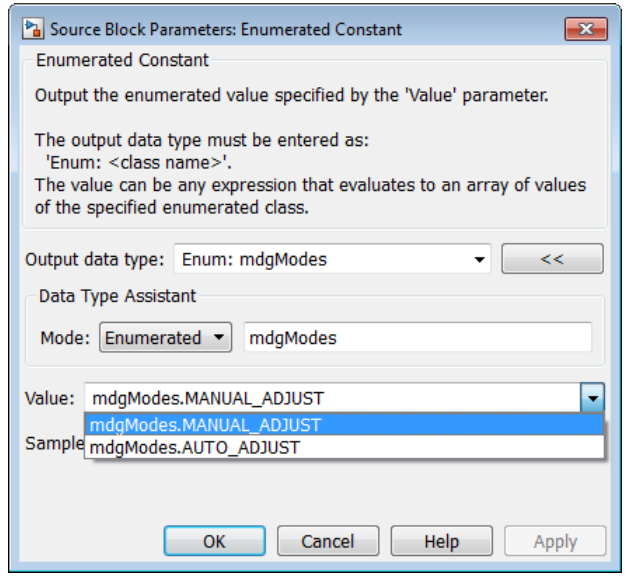
To model an AUTOSAR mode user software component in Simulink:

- 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.

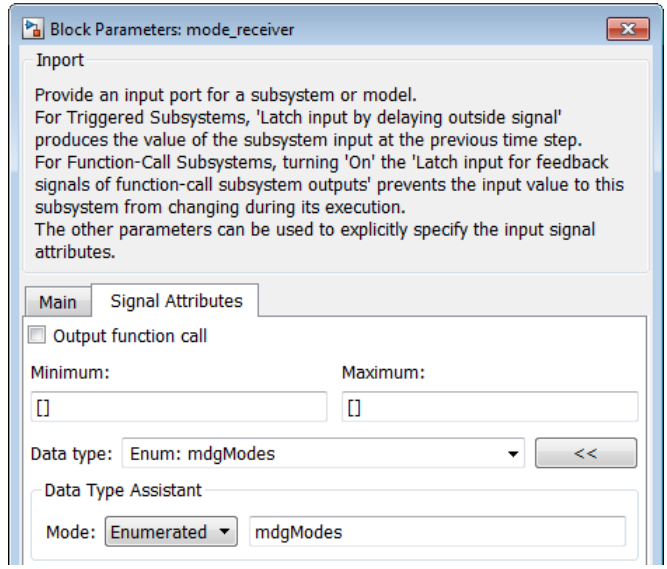
To model an AUTOSAR software component mode-receiver port, general steps might include:

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`.

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

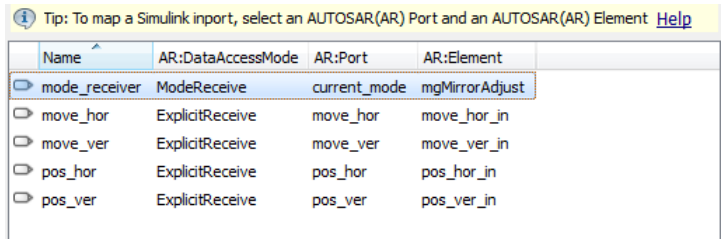


2 Apply the enumeration data type to a Simulink inport that represents an AUTOSAR mode-receiver port. In the Inport block properties dialog below, enumerated type `mdgModes` is specified as the inport data type.



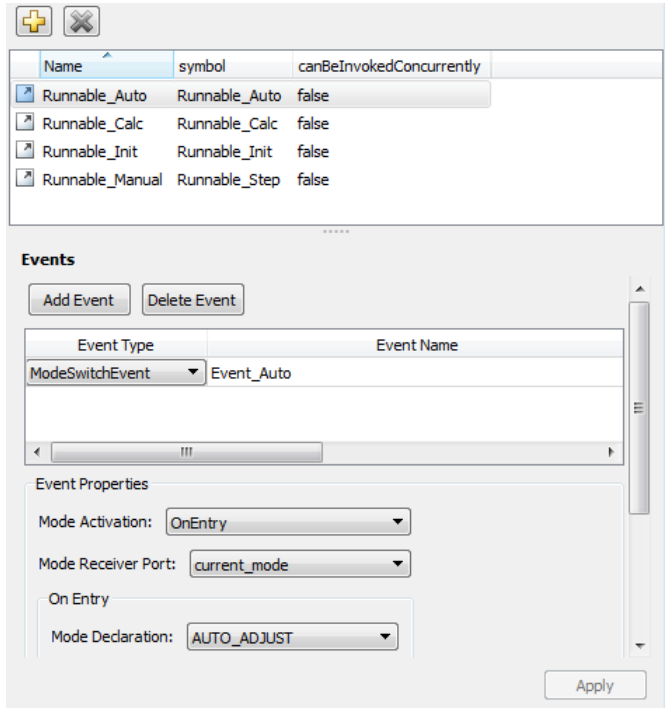
3 Use the Configure AUTOSAR Interface dialog box in the **Simulink-AUTOSAR Mapping** view (or equivalent API commands) to specify the mapping of the Simulink inport to the AUTOSAR mode-receiver port.

In the following example, in the Inports view of the **Simulink-AUTOSAR Mapping** Explorer, 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 or exported runnable, general steps might include:

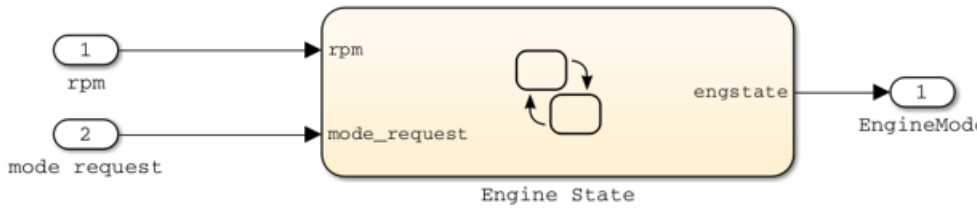
1. Use the Configure AUTOSAR Interface dialog box in the **AUTOSAR Properties** view (or equivalent API commands) to edit, add, or remove AUTOSAR mode-switch interfaces and mode-receiver ports.
2. In your model, choose or add a runnable that you want to be activated by a mode-switch event.
3. In the Runnables view of the **AUTOSAR Properties** Explorer, select the runnable that you want to be activated by a mode-switch event and 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 4-130.

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 datatype is an enumeration class, 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 new 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 the Simulink to AUTOSAR mapping interface, 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 4-130.

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 2-12.
- Generate C and `arxml` code for AUTOSAR NV data interfaces and ports.

For more information , see "Configure AUTOSAR Nonvolatile Data Communication" on page 4-141.

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. If you model AUTOSAR

integrated and distributed lookups as described in "Configure STD_AXIS and COM_AXIS Lookup Tables for AUTOSAR Measurement and Calibration" on page 4-174, you can map Simulink lookup tables to AUTOSAR port-based parameters.

For more information, see "Configure Receiver for AUTOSAR Parameter Communication" on page 4-144.

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 is activated by the event.

For more information, see "Configure Receiver for AUTOSAR External Trigger Event Communication" on page 4-148.

Related Examples

- "Configure AUTOSAR Provide-Require Port" on page 4-83
- "Configure AUTOSAR Client-Server Communication" on page 4-101
- "Configure AUTOSAR Mode-Switch Communication" on page 4-130
- "Configure AUTOSAR Nonvolatile Data Communication" on page 4-141
- "Configure Receiver for AUTOSAR Parameter Communication" on page 4-144
- "Configure Receiver for AUTOSAR External Trigger Event Communication" on page 4-148

More About

- "AUTOSAR Interface Configuration" on page 4-3

Model AUTOSAR Basic Software (BSW) 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, Embedded Coder Support Package for AUTOSAR Standard 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 `CallbackEventStatusChangeCaller`, `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, you synchronize 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 (Dem) Service” on page 4-153 and “Configure Calls to AUTOSAR NVRAM Manager (NvM) Service” on page 4-161.

See Also

`CallbackEventStatusChangeCaller` | `DiagnosticInfoCaller` | `DiagnosticMonitorCaller` | `NvMAdminCaller` | `NvMServiceCaller`

Related Examples

- “Configure Calls to AUTOSAR Diagnostic Event Manager (Dem) Service” on page 4-153

- "Configure Calls to AUTOSAR NVRAM Manager (NvM) Service" on page 4-161

More About

- "Model AUTOSAR Communication" on page 2-11

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...

"About AUTOSAR Calibration Parameters" on page 2-24

"Import and Export Calibration Parameters" on page 2-24

"Calibration Parameters for STD_AXIS and COM_AXIS Lookup Tables" on page 2-25

About 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 can be accessed by AUTOSAR software components. You can import a calibration component from `arxml` files into Simulink or create a calibration component in Simulink.
- Internal calibration parameters, which are defined and accessed by only one AUTOSAR software component. See "Configure AUTOSAR Internal Calibration Parameters" on page 4-167

The software supports `arxml` import, `arxml` export, and C code generation for both types of calibration parameters.

Import and Export Calibration Parameters

You can import calibration parameters into the MATLAB base workspace.

For example, to import parameters from an AUTOSAR calibration component description, use arxml.importer.createCalibrationComponentObjects.

To provide your Simulink model with access to these parameters, assign the imported parameters to block parameters.

For more information, see “Import AUTOSAR Software Component” on page 3-4.

You can specify the type of calibration parameter exported by configuring properties of the corresponding block parameter in the base workspace.

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 high-performance 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 an n-D 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 AUTOSAR parameter access modes — `Shared`, `PerInstance`, `Const`, or `PortParameter`. If the access mode is `PortParameter`, the parameter is also mapped to an AUTOSAR parameter receiver port.

  - 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 can create AUTOSAR parameters for lookup tables graphically, using the AUTOSAR Properties Explorer, or programmatically, using AUTOSAR property functions.

  - To model an AUTOSAR lookup table in a STD_AXIS configuration, create an n-D Lookup Table block. Store all of the data in a single `Simulink.LookupTable` object. Use the object in the n-D Lookup Table 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 Prelookup and Interpolation Using Prelookup blocks. Store each set of table data in a `Simulink.LookupTable` object and each breakpoint vector in a `Simulink.Breakpoint` object. Use each `Simulink.LookupTable` object in an Interpolation 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 n-D Lookup Table or Prelookup blocks. Alternatively, configure input signals to n-D Lookup Table or Prelookup blocks with static global memory.

  - Use the Simulink-AUTOSAR Mapping Explorer, Lookup Tables view, to map the Simulink lookup tables to AUTOSAR parameters with AUTOSAR parameter access modes. If suitable AUTOSAR parameters do not exist, use the AUTOSAR Properties Explorer to create either software component parameters or parameter interface data elements.

- Generate `arxml` and C code with STD_AXIS and COM_AXIS lookup table content.

  In the Simulink Configuration Parameters dialog box, **Interface** pane, select the AUTOSAR 4.0 code replacement library for C code generation.

  The generated C code contains appropriate `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 STD_AXIS and COM_AXIS Lookup Tables for AUTOSAR Measurement and Calibration" on page 4-174.

See Also

Simulink.LookupTable | Simulink.Breakpoint | arxml.importer.createCalibrationComponentObjects | `Interpolation Using Prelookup` | `n-D Lookup Table` | `Prelookup`

Related Examples

- "Configure STD_AXIS and COM_AXIS Lookup Tables for AUTOSAR Measurement and Calibration" on page 4-174
- "Import AUTOSAR Software Component" on page 3-4
- "Export AUTOSAR Component XML and C Code" on page 5-2

More About

- "AUTOSAR Interface 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 2-28
"Runnables" on page 2-28
"Inter-Runnable Variables" on page 2-29
"System Constants" on page 2-30
"Per-Instance Memory" on page 2-31
"Static and Constant Memory" on page 2-32

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:

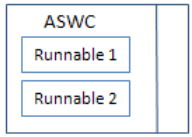
- Runnables and the events to which they respond
- Inter-runnable variables, used to communicate data between runnables in the same component
- System constants, used to specify system-level constant values that are available for reference in component algorithms
- Per-instance memory, used to specify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

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.

The following figure shows an AUTOSAR software component with two runnables, `Runnable 1` and `Runnable 2`. Each runnable is triggered by RTEEvents, events generated by the AUTOSAR run-time environment (RTE). For example, TimingEvent is an RTEEvent that is generated periodically.

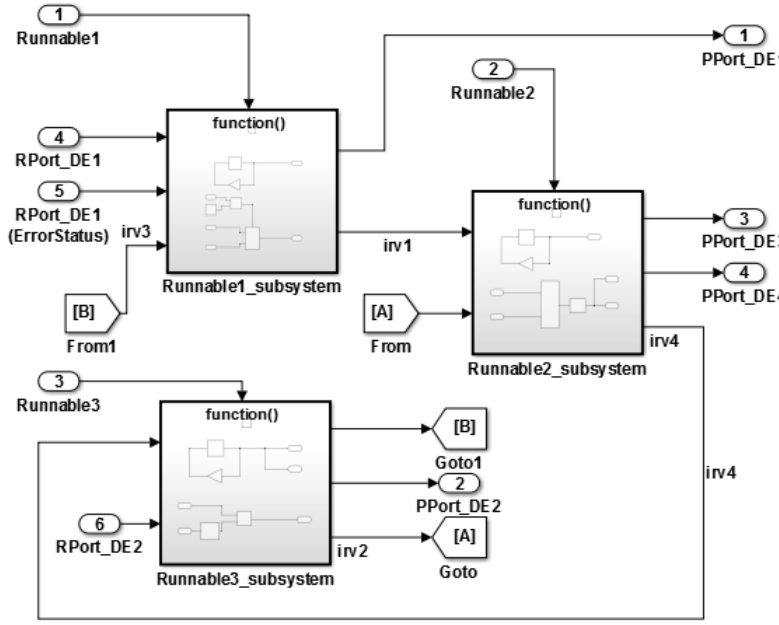


A component also can contain a single runnable, represented by a model, and can be single-rate or multirate.

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

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 AUTOSAR Variants in Runnable Condition Logic" on page 4-261.

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 AUTOSAR-typed per-instance memory, create an `AUTOSAR.Signal` data object in the base workspace. Open the data object, set the **Storage class** to `PerInstanceMemory`, and configure its per-instance memory attributes. Then create a `Data Store Memory` block that references the `AUTOSAR.Signal` object.

For more information, see "Configure AUTOSAR Per-Instance Memory" on page 4-230.

Static and Constant Memory

AUTOSAR supports Static Memory and Constant Memory 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` code with 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 or Constant Memory in Simulink, use signals or parameters that map to AUTOSAR signal or parameter data objects. In each signal or parameter data object, set the **Storage class** attribute to `ExportedGlobal` or a custom storage class that generates a global variable in the model code. Additionally, you can configure software calibration access to the data using the **SwCalibrationAccess** attribute of the parameter or signal data object.

To define the memory location of static or constant memory, you can create AUTOSAR-compatible memory sections. To map a Simulink memory section to an AUTOSAR memory section, use `cscdesigner` to create a reference to the `SwAddrMethod` memory-section in the AUTOSAR package for your custom data package. For convenience, you can use `AUTOSAR4.Signal` and `AUTOSAR4.Parameter` data objects, with **Storage class** set to `Global`, to specify an AUTOSAR memory section for each signal or parameter.

For more information, see “Configure AUTOSAR Static or Constant Memory” on page 4-234 and “Configure AUTOSAR Data for Measurement and Calibration” on page 4-190.

See Also

`AUTOSAR.Parameter` | `AUTOSAR.Signal` | `AUTOSAR4.Parameter` | `AUTOSAR4.Signal` | `Data Store Memory`

Related Examples

- “Configure AUTOSAR Multiple Runnables” on page 4-205
- “Configure AUTOSAR Per-Instance Memory” on page 4-230
- “Configure AUTOSAR Static or Constant Memory” on page 4-234

More About

- "AUTOSAR Interface Configuration" on page 4-3

Model AUTOSAR Data Types

In Simulink, you can model AUTOSAR data types used in data elements, operation arguments, calibration parameters, and inter-runnable variables.

In this section...
"About AUTOSAR Data Types" on page 2-34
"Enumerated Data Types" on page 2-38
"Structure Parameters" on page 2-39
"Release 2.x and 3.x Data Types" on page 2-39
"Release 4.x Data Types" on page 2-40
"CompuMethod Categories for Data Types" on page 2-43

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
- Inter-runnable variables

The data types fall into two categories:

- Primitive data types, which allow a direct mapping to C intrinsic types.
- Composite data types, which map to C arrays and structures.

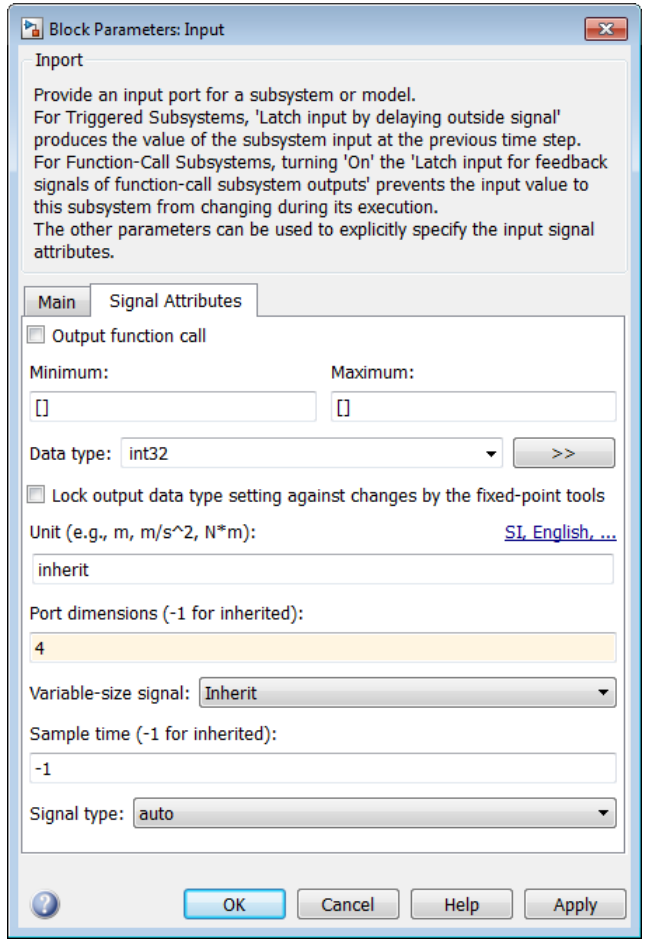
You can use Simulink data types to define AUTOSAR primitive types.

AUTOSAR Data Type	Simulink Data Type
`boolean`	`boolean`
`float32`	`single`
`float64`	`double`

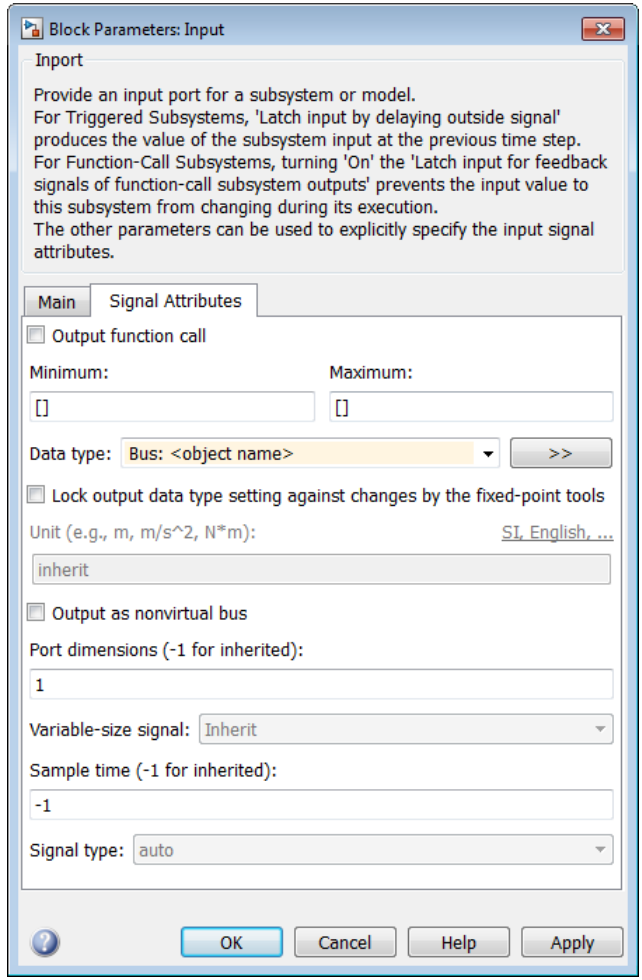
AUTOSAR Data Type	Simulink Data Type
sint8	int8
sint16	int16
sint32	int32
uint8	uint8
uint16	uint16
uint32	uint32

AUTOSAR composite data types are arrays and records, which are represented in Simulink by wide signals and bus objects, respectively. In the Inport or Outport Block Parameters dialog box, use the **Signal Attributes** pane to configure wide signals and bus objects.

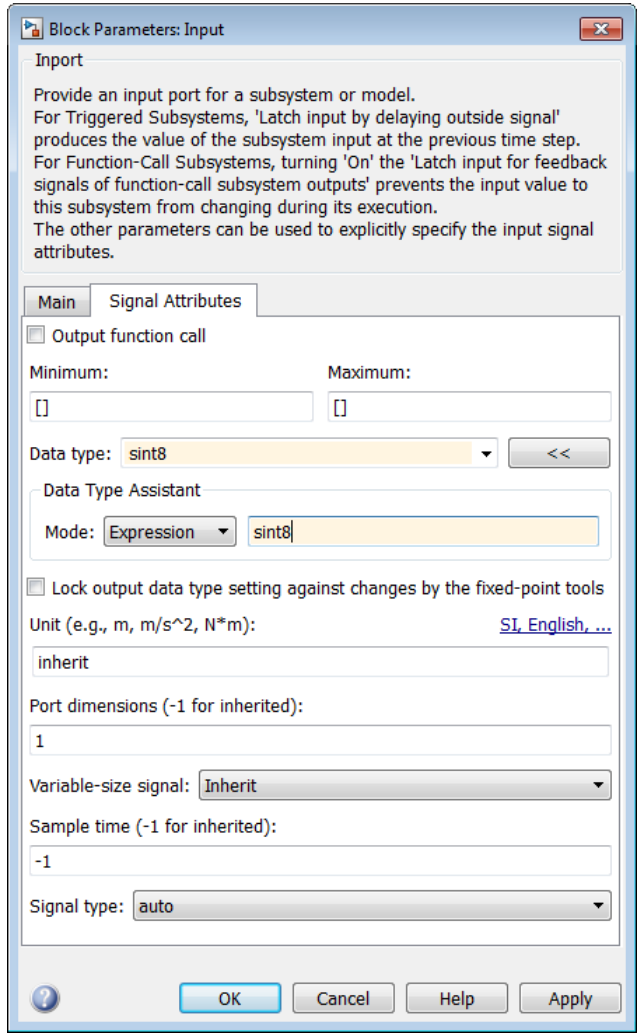
The following figure shows how to specify a wide signal, which corresponds to an AUTOSAR composite array.



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



You can use the **Data Type Assistant** on the **Signal Attributes** pane of the Inport or Outport Block Parameters dialog box to specify the data types of data elements and arguments of an operation prototype. If you select **Mode** to be `Built in`, then you can specify the data type to be, for example, `boolean`, `single`, or `int8`. Alternatively, if you select **Mode** to be `Expression`, you can specify an (alias) expression for data type. As an example, the following figure shows an alias `sint8`, corresponding to an AUTOSAR data type, in the **Data type** field.



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 number 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 "Organize Block Parameter Values into Structures in the Generated Code" in Simulink Coder™ documentation. Otherwise, the software displays the following behavior:

- When you validate the AUTOSAR interface, 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 2-40.

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';`

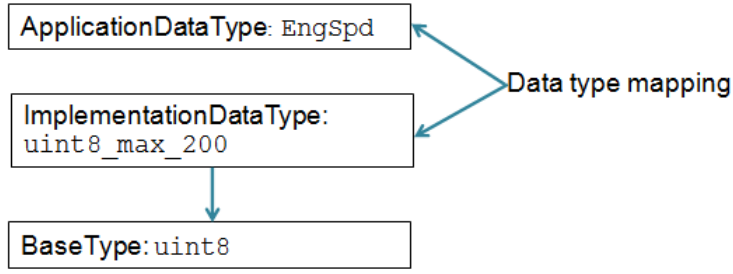
Primitive type (fixed point), for example, `myFixPt`	`myFixPt = Simulink.NumericType;`
	`myFixPt.DataTypeMode = …;`
	`myFixPt.IsAlias = true;`
	`myFixPt.HeaderFile = 'Rte_Type.h';`

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 (integer, Boolean, real, and so on).



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.

- 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 2-43.

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 4-237.

- "R4.x Data Types in Simulink Originated Workflow" on page 2-41
- "R4.x Data Types in Round-Trip Workflow" on page 2-42
- "Application Data Type Physical Constraint Mapping" on page 2-43

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`	`<IMPLEMENTATION-DATA-TYPE>` `<SHORT-NAME>myInt16</SHORT-NAME>`	Not generated

Simulink Data Type	AUTOSAR XML	
	Implementation Type	Application Type

Covers Boolean, integer, real
`myInt16 = Simulink.AliasType;`
`myInt16.BaseType = 'int16';`
	`<CATEGORY>VALUE</CATEGORY>`
	…	

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 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 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 `ApplicationDataType`s 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` `PhysConstr`s in the `arxml` files.

CompuMethod Categories for Data Types

AUTOSAR computation methods (`CompuMethods`) are used for the conversion of internal values into their physical representation, and vice versa. The `category` attribute of a `CompuMethod` represents a specialization of the `CompuMethod`, which may impose semantic constraints. The `CompuMethod` categories produced by the code generator include:

- `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 `CompuMethod`s 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
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 `CompuMethod`s for code generation, see “Configure AUTOSAR CompuMethods” on page 4-241.

Related Examples

- “Organize Block Parameter Values into Structures in the Generated Code”
- “Application Data Type Physical Constraint Mapping” on page 2-43
- “Configure AUTOSAR Release 4.x Data Types” on page 4-237
- “Configure AUTOSAR CompuMethods” on page 4-241

More About

- "AUTOSAR Interface 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 code inside 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 code inside an AUTOSAR runnable, use `Variant Subsystem` or `Model Variants` blocks.
- Resolve modeled variants by using predefined variants and system constant value sets imported from `arxml` files.

In this section...

"Variants in Ports and Runnables" on page 2-46
"Variants in Array Sizes" on page 2-47
"Variants in Runnable Condition Logic" on page 2-48
"Predefined Variants and System Constant Value Sets" on page 2-48

Variants in 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 workspace variables for modeling with variation points, `Variant Sink` blocks, and `Variant Source` blocks to propagate variant conditions.
- Model AUTOSAR elements with variation points.
  - To define variant condition logic, use `Simulink.Variant` data objects.
  - To represent AUTOSAR system constants, use `AUTOSAR.Parameter` data objects with storage class `SystemConstant`.
  - To propagate variant conditions for the AUTOSAR elements, use Variant Sink and Variant Source blocks.
- 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 previously imported AUTOSAR ports and runnables with variation points.

For more information, see "Configure AUTOSAR Variants in Ports and Runnables" on page 4-256.

Variants in 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 AUTOSAR Variants in Array Sizes" on page 4-259.

Variants in Runnable Condition Logic

AUTOSAR software components can specify variant condition logic inside an AUTOSAR runnable. You can model variant condition logic inside a runnable by using the following Simulink elements:

- `Variant Subsystem` or `Model Variants` blocks, containing variant implementations with associated condition logic.
- `AUTOSAR.Parameter` data objects to model AUTOSAR system constants, representing the conditional values associated with the variant condition logic.
- `Simulink.Variant` data objects in the base workspace to define the variant condition logic.

For more information, see "Configure AUTOSAR Variants in Runnable Condition Logic" on page 4-261.

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 `SwSystemconstantValueSet`s for controlling variation points in the component, you can resolve the variation points at model creation time. Specify a `PredefinedVariant` or `SwSystemconstantValueSet`s with which the importer can initialize `SwSystemconst` data.

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 `PredefinedVariant`s and `SwSystemconstantValueSet`s to `arxml` code.

For more information, see “Control AUTOSAR Variants with Predefined Value Combinations” on page 4-264.

Related Examples

- “Configure AUTOSAR Variants in Ports and Runnables” on page 4-256
- “Configure AUTOSAR Variants in Array Sizes” on page 4-259
- “Configure AUTOSAR Variants in Runnable Condition Logic” on page 4-261
- “Control AUTOSAR Variants with Predefined Value Combinations” on page 4-264

More About

- “Variant Systems”
- “AUTOSAR Interface Configuration” on page 4-3

AUTOSAR Component Creation

- "AUTOSAR arxml Importer Tool" on page 3-2
- "Import AUTOSAR Software Component" on page 3-4
- "Import AUTOSAR Software Component Updates" on page 3-8
- "Round-Trip Preservation of AUTOSAR Elements and UUIDs" on page 3-14
- "Create AUTOSAR Software Component in Simulink" on page 3-15
- "Import or Update Shared AUTOSAR Reference Element Definitions" on page 3-22
- "Limitations and Tips" on page 3-24

AUTOSAR arxml Importer Tool

The AUTOSAR `arxml` importer tool imports AUTOSAR software component description files produced by an AUTOSAR authoring tool (AAT) into a Simulink model. The tool first parses `arxml` code that describes an AUTOSAR software component. 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 the 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 tool validates the XML in the imported `arxml` files. If XML validation fails for a file, the tool 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 AUTOSAR elements and their UUIDs across `arxml` import and export. For more information, see “Round-Trip Preservation of AUTOSAR Elements and UUIDs” on page 3-14.
- 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 3-8.

The AUTOSAR `arxml` importer tool is implemented as an `arxml.importer` class. For a complete list of methods, see the arxml.importer class reference page.

Related Examples

- “Import AUTOSAR Software Component” on page 3-4

- “Export AUTOSAR Component XML and C Code” on page 5-2

More About

- “Workflows for AUTOSAR” on page 1-4

Import AUTOSAR Software Component

In Simulink, you can import an AUTOSAR software component description from `arxml` code and create a model representing the AUTOSAR software component. You use the AUTOSAR `arxml` importer tool, which is implemented as an arxml.importer class. For more information, see "AUTOSAR arxml Importer Tool" on page 3-2.

In this section...

"General arxml Import Workflow" on page 3-4

"Import Multi-Runnable Software Component" on page 3-6

General arxml Import Workflow

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

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

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

   If you enter the `arxml.importer` constructor 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 component ASWC is `/pkg/swc/ASWC`.

   ```
   obj =

   The file "H:\wrk\mr_component.arxml" contains:
     1 Application-Software-Component-Type:
       '/pkg/swc/ASWC'

     0 Sensor-Actuator-Software-Component-Type.
     0 CalPrm-Component-Type.
     0 Client-Server-Interface.
   >>
   ```

   Each software component requires an `arxml.importer` object. To specify a different main software component file and update the list of components, use arxml.importer.setFile.

2 To import a parsed atomic software component into a Simulink model, call one of the following methods. If you have not specified all dependencies for the components, you will see errors.

- arxml.importer.createComponentAsModel — Creates and configures a Simulink model mapping corresponding to the specified atomic software component description.

  For example:

  ```
  createComponentAsModel(obj,'/pkg/swc/ASWC')
  ```

  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. For example:

  ```
  createComponentAsModel(obj,'/pkg/swc/ASWC','DataDictionary','ardata.sldd')
  ```

  For a multi-runnable AUTOSAR software component, you can set the `CreateInternalBehavior` property on the model creation to control whether the created model is a rate-based model (`false`, the default) or a function-call-based model (`true`). For more information, see "Import Multi-Runnable Software Component" on page 3-6 and "Multi-Runnable Software Components" on page 2-5.

  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 arxml.importer.createComponentAsModel.

- arxml.importer.createCalibrationComponentObjects — Creates Simulink calibration objects corresponding to the specified AUTOSAR calibration component description.

  For example:

  ```
  [success] = createCalibrationComponentObjects(obj,'/ComponentType/MyCalibComp1',...
  'CreateSimulinkObject', true)
  ```

  To import Simulink calibration objects for AUTOSAR data into a Simulink data dictionary, you can set the `DataDictionary` property on the calibration objects creation. If the specified dictionary does not already exist, the importer creates it. For example:

  ```
  createComponentAsModel(obj,'/ComponentType/MyCalibComp1','DataDictionary','ardata.sldd')
  ```

- arxml.importer.updateModel — Updates an AUTOSAR model with changes found in `arxml` code.

  For example:

  ```
  open_system('mySWC')
  obj = arxml.importer('updatedSWC.arxml')
  updateModel(obj,'mySWC');
  ```

  For more information on updating a model with `arxml` code changes, see "Import AUTOSAR Software Component Updates" on page 3-8.

After you import your software component into Simulink, you can modify the model. For parameters from a calibration component, after importing the parameters into the MATLAB workspace or a Simulink data dictionary, assign the calibration parameters to block parameters in your model.

To refine the configuration of the imported AUTOSAR software component, see "AUTOSAR Interface Configuration" on page 4-3.

To configure AUTOSAR code generation options and XML export options, see "Export AUTOSAR Component XML and C Code" on page 5-2.

Import Multi-Runnable Software Component

The AUTOSAR `arxml` importer tool can import an AUTOSAR software component with multiple runnable entities into a Simulink model. Use the `createComponentAsModel` method of the class arxml.importer. Use the `CreateInternalBehavior` property on the model creation to specify the type of model to create, either rate-based (`false`, the default) or function-call-based (`true`). For more information on multi-runnable modeling styles, see "Multi-Runnable Software Components" on page 2-5.

If `CreateInternalBehavior` is omitted or set to the default value, `false`, the importer creates a rate-based model. If the `arxml` code contains periodic runnables, the importer adds rate-based model content, including periodic functions and data transfer lines with rate transitions, and maps them to corresponding periodic runnables and IRVs imported from the AUTOSAR software component.

If `CreateInternalBehavior` is set to `true`, the importer creates a function-call-based model. 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.

The following call directs the importer to import a multi-runnable AUTOSAR software component and map it into a new function-call-based model:

```
obj = arxml.importer('mySWC.arxml')
createComponentAsModel(obj,'/pkg/swc/ASWC','CreateInternalBehavior',true)
```

Note: When you import an AUTOSAR component into a model, in most cases, the importer requires that an AUTOSAR port be accessed by a runnable in order to generate the corresponding Simulink elements. For example, if a sender-receiver or client-server port in `arxml` code is not accessed by a runnable, the importer does not create the corresponding inports, outports, or Simulink functions.

An exception is when you import a single-runnable AUTOSAR component to create a rate-based, single-tasking model (setting property `'CreateInternalBehavior'` to `false`). In this case, the importer creates inports, outports, and so on even when an AUTOSAR port is not accessed by the runnable.

See Also

arxml.importer

Related Examples

- “Import AUTOSAR Software Component Updates” on page 3-8
- “Round-Trip Preservation of AUTOSAR Elements and UUIDs” on page 3-14
- “Export AUTOSAR Component XML and C Code” on page 5-2

More About

- “AUTOSAR arxml Importer Tool” on page 3-2
- “Workflows for AUTOSAR” on page 1-4

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 outside of Simulink. Using the method arxml.importer.updateModel, you can import the modified `arxml` description and update the model to reflect the changes. The update generates an HTML report that details automatic updates applied to the model, and additional manual changes that you must perform.

In this section...

"Update Model with AUTOSAR Software Component Changes" on page 3-8
"AUTOSAR Update Report Section Examples" on page 3-10

Update Model with AUTOSAR Software Component Changes

To update a model with AUTOSAR software component changes described in `arxml` code:

1. Open a model for which you previously imported or exported `arxml` code. For example:

   ```
   >> open_system('mySWC')
   ```

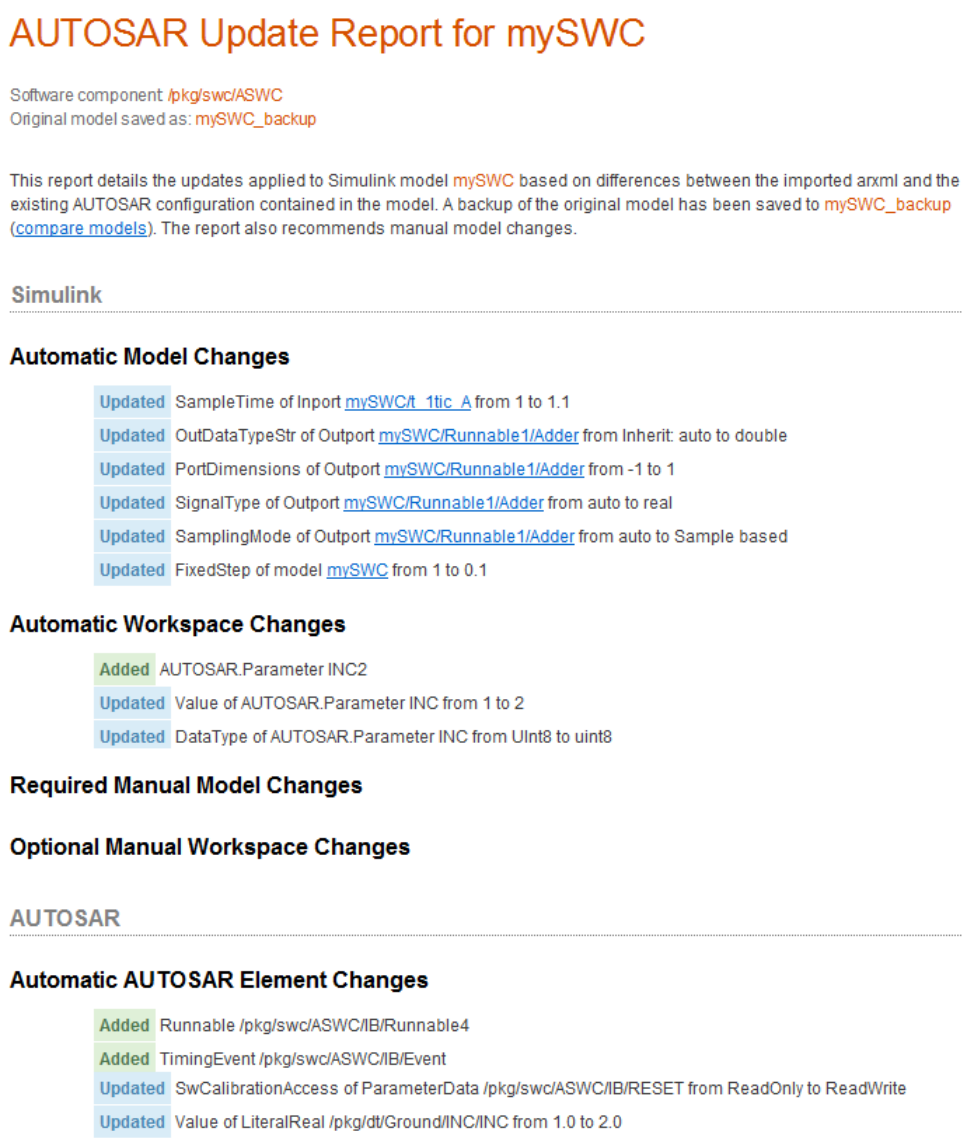
2. Issue MATLAB commands to import `arxml` code into the model and update the model with changes.

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

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

   ```
   >> obj = arxml.importer('updatedSWC.arxml');
   >> updateModel(obj,'mySWC');
   ### Updating model mySWC
   ### Saving original model as mySWC_backup.slx
   ### Creating HTML report mySWC_update_report.html
   >>
   ```

   The AUTOSAR Update Report opens.



AUTOSAR Update Report for mySWC

Software component: /pkg/swc/ASWC
Original model saved as: mySWC_backup

This report details the updates applied to Simulink model mySWC 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 mySWC_backup (compare models). The report also recommends manual model changes.

Simulink

Automatic Model Changes

Updated	SampleTime of Inport mySWC/t_1tic_A from 1 to 1.1
Updated	OutDataTypeStr of Outport mySWC/Runnable1/Adder from Inherit: auto to double
Updated	PortDimensions of Outport mySWC/Runnable1/Adder from -1 to 1
Updated	SignalType of Outport mySWC/Runnable1/Adder from auto to real
Updated	SamplingMode of Outport mySWC/Runnable1/Adder from auto to Sample based
Updated	FixedStep of model mySWC from 1 to 0.1

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

Optional Manual Workspace Changes

AUTOSAR

Automatic AUTOSAR Element Changes

Added	Runnable /pkg/swc/ASWC/IB/Runnable4
Added	TimingEvent /pkg/swc/ASWC/IB/Event
Updated	SwCalibrationAccess of ParameterData /pkg/swc/ASWC/IB/RESET from ReadOnly to ReadWrite
Updated	Value of LiteralReal /pkg/dt/Ground/INC/INC from 1.0 to 2.0

3 Examine the report. Verify that the `arxml` importer has updated the model content, workspace, and configuration based on the `arxml` changes. If the report lists any

required manual model changes, such as deleting a Simulink block, perform the required changes.

4 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` method with the same `arxml` files.

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 3-10
- "Automatic Workspace Changes" on page 3-11
- "Required Manual Model Changes" on page 3-11
- "Automatic AUTOSAR Element Changes" on page 3-12

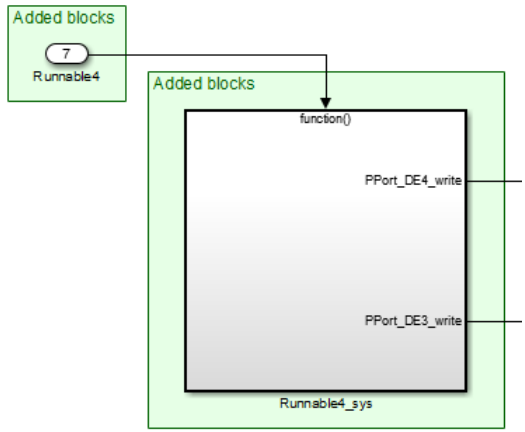
Automatic Model Changes

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	SampleTime of Inport mySWC/t_1tic_A from 1 to 1.1
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
Updated	FixedStep of model mySWC from 1 to 0.1

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

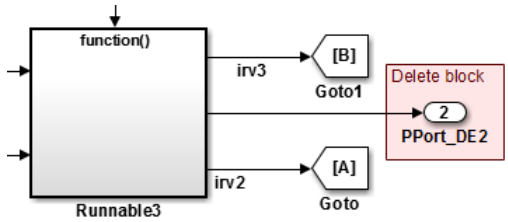
Required Manual Model Changes

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	Period of TimingEvent /pkg/swc/ASWC/IB/Event_t_1tic_A from 1.0 to 1.1
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

arxml.importer.updateModel

Related Examples

- “Import AUTOSAR Software Component” on page 3-4
- “Export AUTOSAR Component XML and C Code” on page 5-2

More About

- "Workflows for AUTOSAR" on page 1-4

Round-Trip Preservation of AUTOSAR Elements and UUIDs

To support the round trip of AUTOSAR elements between an AUTOSAR authoring tool (AAT) and the Simulink model-based design environment, Embedded Coder preserves AUTOSAR elements and their UUIDs across `arxml` import and export.

- When `arxml` files created by an AAT are imported into a Simulink model, AUTOSAR element information is preserved, including UUIDs (for Identifiables), properties, references, and packages.
- After import, you can view and edit AUTOSAR objects in a Simulink model window without losing the information imported from the AAT. For example, you can use the Configure AUTOSAR Interface dialog box to edit object name and property values (preserving the original object UUID), add new objects such as Interfaces and Ports (creating new UUIDs), and delete objects (retiring UUIDs). These actions do not perturb the imported objects and their relationships.
- When `arxml` files are exported from a Simulink model, the elements are generated back into `arxml` code with their UUIDs and other information preserved.

As a result, the `arxml` files exported from Simulink can more easily be merged back into the AAT environment. Existing elements retain their UUIDs, while new elements created in Simulink get new UUIDs.

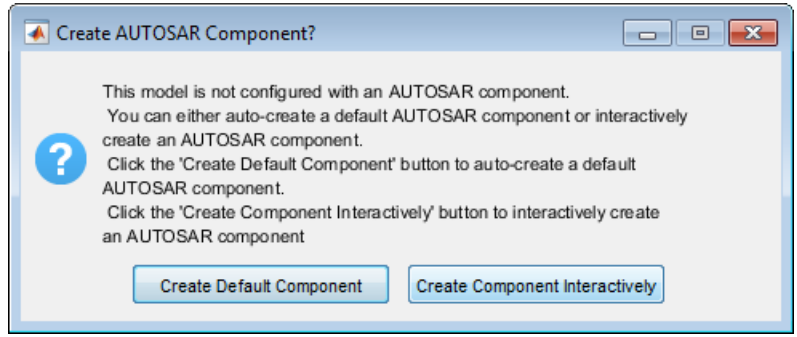
Create AUTOSAR Software Component in Simulink

As an alternative to importing an AUTOSAR software component from an AUTOSAR authoring tool (AAT), you can create an AUTOSAR software component in Simulink.

1. Open a Simulink model that is not configured for AUTOSAR.
2. Click **Simulation > Model Configuration Parameters** to open the Configuration Parameters dialog box.
3. On the **Code Generation** pane, use **System target file** to select the target for AUTOSAR code generation, `autosar.tlc`. Click **Apply**.
4. In the model window, select **Code > C/C++ Code > Configure Model as AUTOSAR Component**. This opens the **Create AUTOSAR Component?** dialog box. The dialog box offers two paths for creating an AUTOSAR software component:

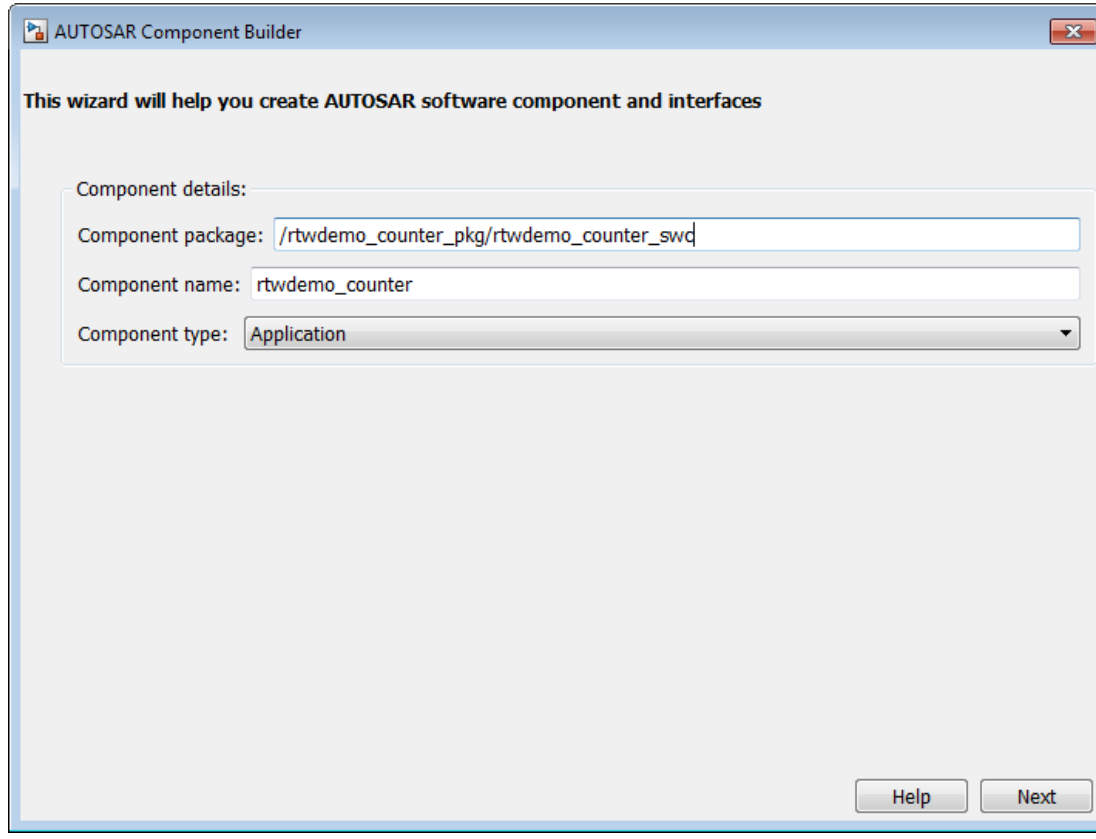
   - **Create Default Component** — Automatically create an AUTOSAR component with default settings and open it in the Configure AUTOSAR Interface dialog box.
   - **Create Component Interactively** — Interactively create an AUTOSAR component using the AUTOSAR Component Builder dialog box.

Note: In most situations, **Create Default Component** is the recommended path. Default component creation automatically creates a starting AUTOSAR configuration in which existing Simulink model content is mapped to a framework of AUTOSAR packages and elements. Interactive component creation can be time consuming and more prone to error.



5. Choose one of the two paths. If you click **Create Default Component**, a new AUTOSAR component opens in the Configure AUTOSAR Interface dialog box. For information about using this dialog box, see "AUTOSAR Interface Configuration" on page 4-3. If you click **Create Component Interactively**, the AUTOSAR Component Builder dialog box opens.
6. In the initial view of the AUTOSAR Component Builder dialog box, you specify the following items:

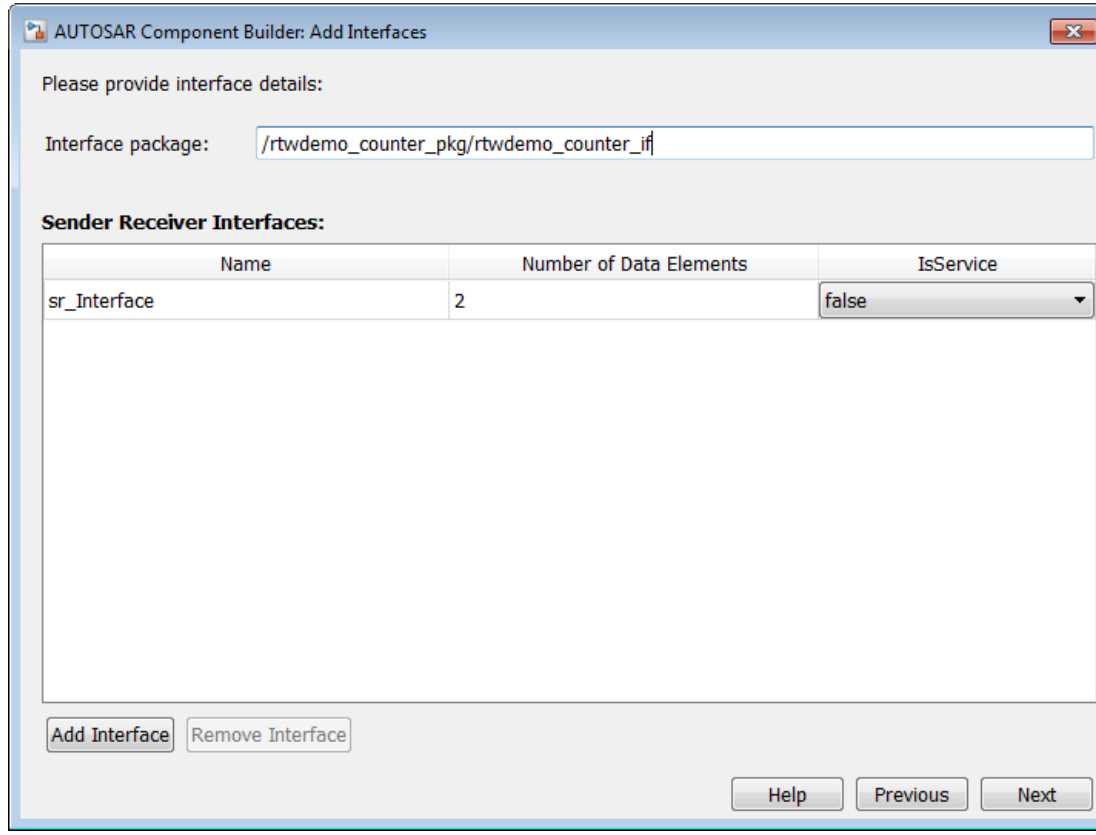
   - Path for the AUTOSAR component package.
   - Name for the AUTOSAR component.
   - AUTOSAR component type: `Application` for an AUTOSAR application software component, or `Sensor Accuator` for an AUTOSAR sensor/actuator software component.



Click **Next** to go to the Add Sender Receiver Interfaces view.

7 In the Add Sender Receiver Interfaces view of the AUTOSAR Component Builder dialog box, you can:

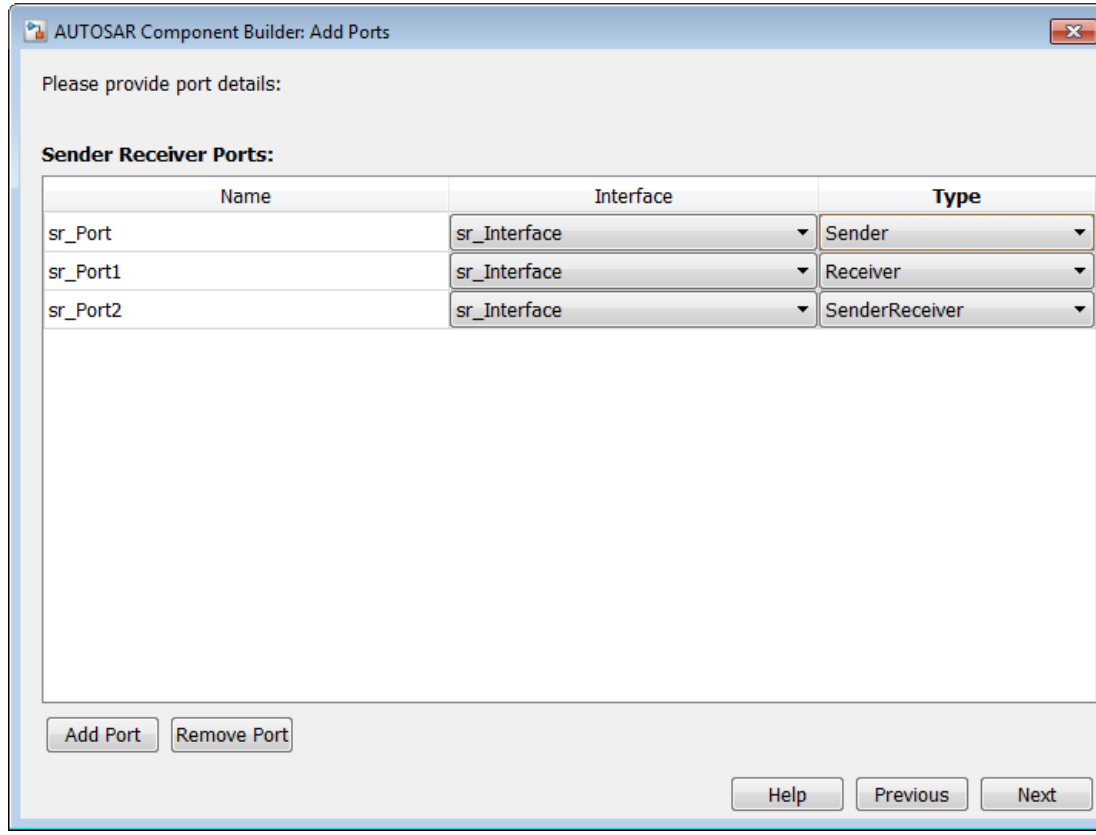
- Modify the name of the Interface package.
- Click **Add Interface** to add more interfaces to the displayed list.
- Click **Remove Interface** to remove a selected interface.
- For each listed interface, edit the name and the number of data elements it contains, and select whether the interface is a service.



Click **Next** to go to the Add Sender Receiver Ports view.

8 In the Add Sender Receiver Ports view of the AUTOSAR Component Builder dialog box, you can:

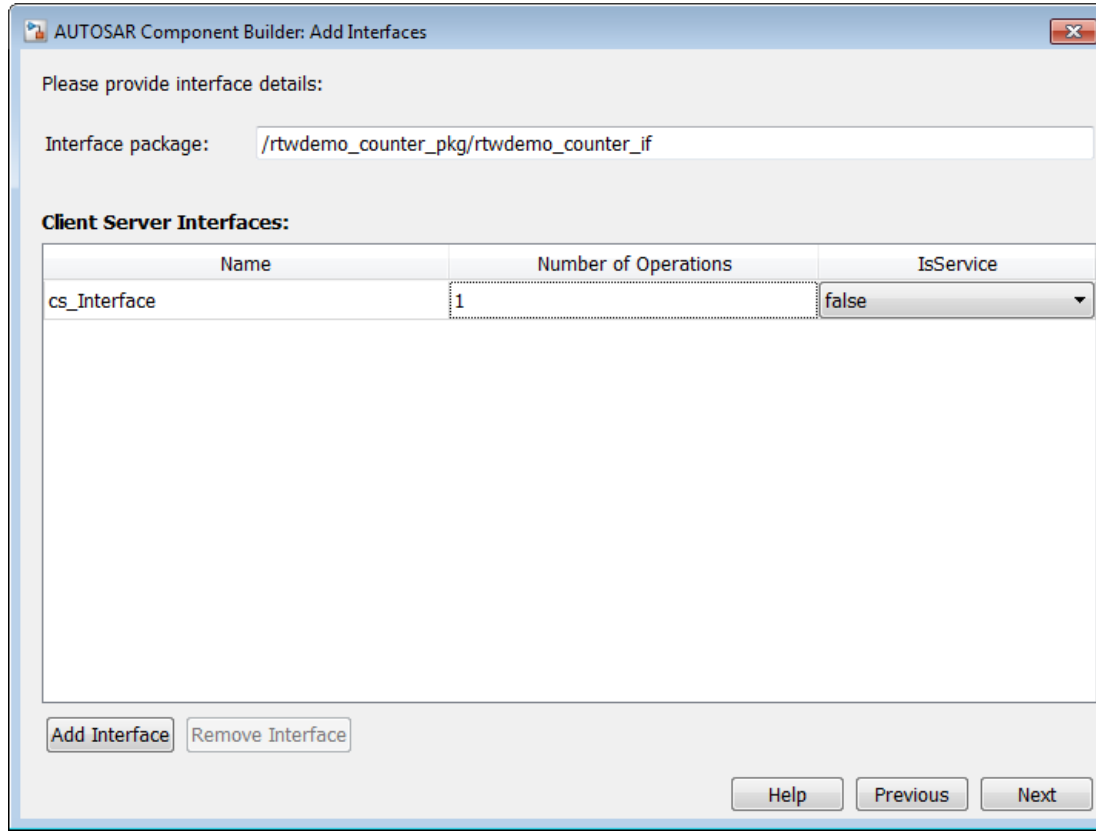
- Click **Add Port** to add more sender, receiver, or sender-receiver ports to the displayed list.
- Click **Remove Port** to remove a selected port.
- For each listed port, edit the name, select the associated S-R interface, and select whether the port type is `Sender`, `Receiver`, or `SenderReceiver`. (AUTOSAR sender-receiver ports require configuring the model for AUTOSAR schema version 4.1 or higher.)



Click **Next** to go to the Add Client Server Interfaces view.

9 In the Add Client Server Interfaces view of the AUTOSAR Component Builder dialog box, you can:

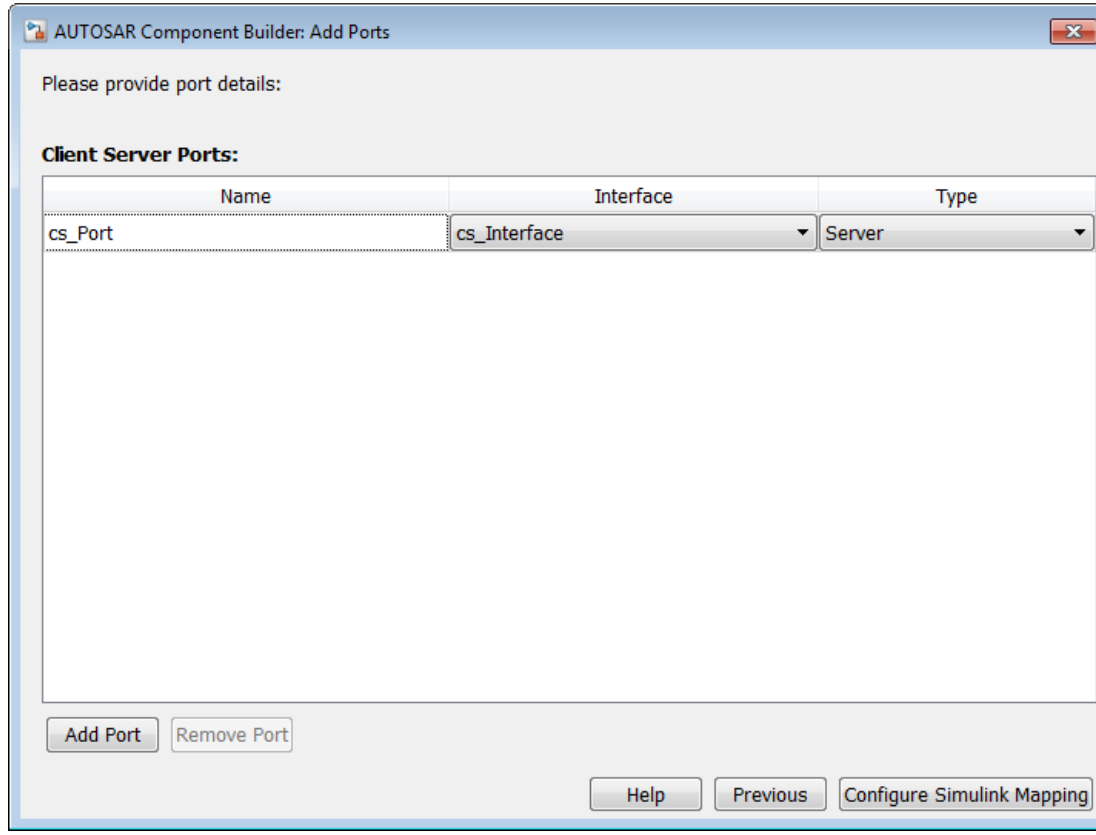
- Modify the name of the Interface package.
- Click **Add Interface** to add more interfaces to the displayed list.
- Click **Remove Interface** to remove a selected interface.
- For each listed interface, edit the name and the number of operations it contains, and select whether the interface is a service.



Click **Next** to go to the Add Client Server Ports view.

10 In the Add Client Server Ports view of the AUTOSAR Component Builder dialog box, you can:

- Click **Add Port** to add more ports to the displayed list.
- Click **Remove Port** to remove a selected port.
- For each listed port, edit the name, select the associated C-S interface, and select whether the port type is `Client` or `Server`.



Click **Configure Simulink Mapping** to open the Configure AUTOSAR Interface dialog box. To continue, see “AUTOSAR Interface Configuration” on page 4-3.

Import or Update Shared AUTOSAR Reference Element Definitions

You can import external AUTOSAR element definitions, defined in `arxml` files, for sharing among multiple AUTOSAR components and services. 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 centralized way.

Suppose that you have many AUTOSAR software components that use similar packageable AUTOSAR elements in similar ways. You can define sets of *reference elements* in `arxml` files, and your software components can share them on a read-only basis. Each software component can import the element definitions it requires and reference them. When you build the model, exported `arxml` code contains references to the shared elements, but not their definitions. Their definitions remain in the reference element `arxml` source files.

If definitions of reference elements change, you modify them in the `arxml` files, and then import the updated definitions into the affected software components.

To set up and share AUTOSAR reference element definitions:

1. Create one or more `arxml` files containing definitions of AUTOSAR packageable elements for components to share. Elements that are supported for reference use in Simulink include:
   - `CompuMethod`, `Unit`, and `PhysicalDimension`
   - `ImplementationDataType` and `SwBaseType`
   - `SwSystemConst`, `SwSystemConstValueSet`, and `PredefinedVariant`
   - `SwRecordLayout`
   - `SwAddrMethod`
2. For each component that must add external definitions, or update previously imported definitions with revisions, open the model and use importer method `updateReferences`. For example:

```
>> open_system('mySWC')
>> obj = arxml.importer(fullfile(pathToFile,'ExternalElements.arxml'));
>> updateReferences(obj,'mySWC');
### Updating references in model mySWC
### Saving original model as mySWC_backup
### Creating HTML report mySWC_update_report.html
>>
```

Optionally, using property-value pairs, you can specify subsets of elements to import. For example:

- `CompuMethod`s in a specific AUTOSAR package.
- Definitions in per-element definition files.
- A single element definition specified using a path to a packageable element.

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

When you import a read-only 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 some forms of 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, but not their definitions. The definitions remain centralized in the reference element `arxml` source files.

Limitations and Tips

The following limitations apply to AUTOSAR component creation.

In this section...
“Cannot Save Importer Objects in MAT-Files” on page 3-24
“ApplicationRecordDataType and ImplementationDataType Element Names Must Match” on page 3-24

Cannot Save Importer Objects in MAT-Files

If you try to save an `arxml.importer` object in a MAT-file, you lose the information. If you reload the MAT-file, then the object is null (handle = –1), because of the Java® objects that compose the `arxml.importer` object.

ApplicationRecordDataType and ImplementationDataType Element Names Must Match

The element name of an imported `ApplicationRecordDataType` must match the element name of the corresponding `ImplementationDataType`. For example, if an imported `ApplicationRecordDataType` has element `PVAL_1` and the corresponding `ImplementationDataType` has element `IPVAL_1`, the software flags the mismatch and instructs you to rename the elements to match.

AUTOSAR Component Development

- "AUTOSAR Interface Configuration" on page 4-3
- "Configure AUTOSAR Component Using AUTOSAR Properties Explorer" on page 4-6
- "Map Model Elements Using Simulink-AUTOSAR Mapping Explorer" on page 4-60
- "Configure AUTOSAR Package for Interface or CompuMethod" on page 4-67
- "Configure AUTOSAR Packages" on page 4-70
- "Configure AUTOSAR Provide-Require Port" on page 4-83
- "Configure AUTOSAR Port aliveTimeout Value" on page 4-86
- "Configure AUTOSAR Receiver Port for IsUpdated Service" on page 4-87
- "Configure AUTOSAR Sender Port for Data Element Invalidation" on page 4-90
- "Configure AUTOSAR S-R Interface Port for End-To-End Protection" on page 4-94
- "Configure AUTOSAR Receiver Port for DataReceiveErrorEvent" on page 4-97
- "Configure AUTOSAR Client-Server Communication" on page 4-101
- "Configure AUTOSAR Mode-Switch Communication" on page 4-130
- "Configure AUTOSAR Nonvolatile Data Communication" on page 4-141
- "Configure Receiver for AUTOSAR Parameter Communication" on page 4-144
- "Configure Receiver for AUTOSAR External Trigger Event Communication" on page 4-148
- "Configure Calls to AUTOSAR Diagnostic Event Manager (Dem) Service" on page 4-153
- "Configure Calls to AUTOSAR NVRAM Manager (NvM) Service" on page 4-161
- "Configure AUTOSAR Internal Calibration Parameters" on page 4-167
- "Configure AUTOSAR Port-Based Calibration Parameters" on page 4-169
- "Configure AUTOSAR Calibration Component" on page 4-170

- "Configure STD_AXIS and COM_AXIS Lookup Tables for AUTOSAR Measurement and Calibration" on page 4-174
- "Configure COM_AXIS Lookup Table Using AUTOSAR.Parameter Objects" on page 4-184
- "Configure AUTOSAR Data for Measurement and Calibration" on page 4-190
- "Configure AUTOSAR Multiple Runnables" on page 4-205
- "Configure AUTOSAR Initialize, Reset, or Terminate Runnables" on page 4-215
- "Add Top-Level Asynchronous Trigger to Periodic Rate-Based System" on page 4-222
- "Configure AUTOSAR Initialization Runnable (R4.1)" on page 4-226
- "Configure Disabled Mode for AUTOSAR Runnable Event" on page 4-229
- "Configure AUTOSAR Per-Instance Memory" on page 4-230
- "Configure AUTOSAR Static or Constant Memory" on page 4-234
- "Configure AUTOSAR Release 4.x Data Types" on page 4-237
- "Configure AUTOSAR CompuMethods" on page 4-241
- "Configure AUTOSAR Internal Data Constraints Export" on page 4-254
- "Configure AUTOSAR Variants in Ports and Runnables" on page 4-256
- "Configure AUTOSAR Variants in Array Sizes" on page 4-259
- "Configure AUTOSAR Variants in Runnable Condition Logic" on page 4-261
- "Control AUTOSAR Variants with Predefined Value Combinations" on page 4-264
- "Configure and Map AUTOSAR Component Programmatically" on page 4-267
- "AUTOSAR Property and Map Function Examples" on page 4-274
- "Limitations and Tips" on page 4-296

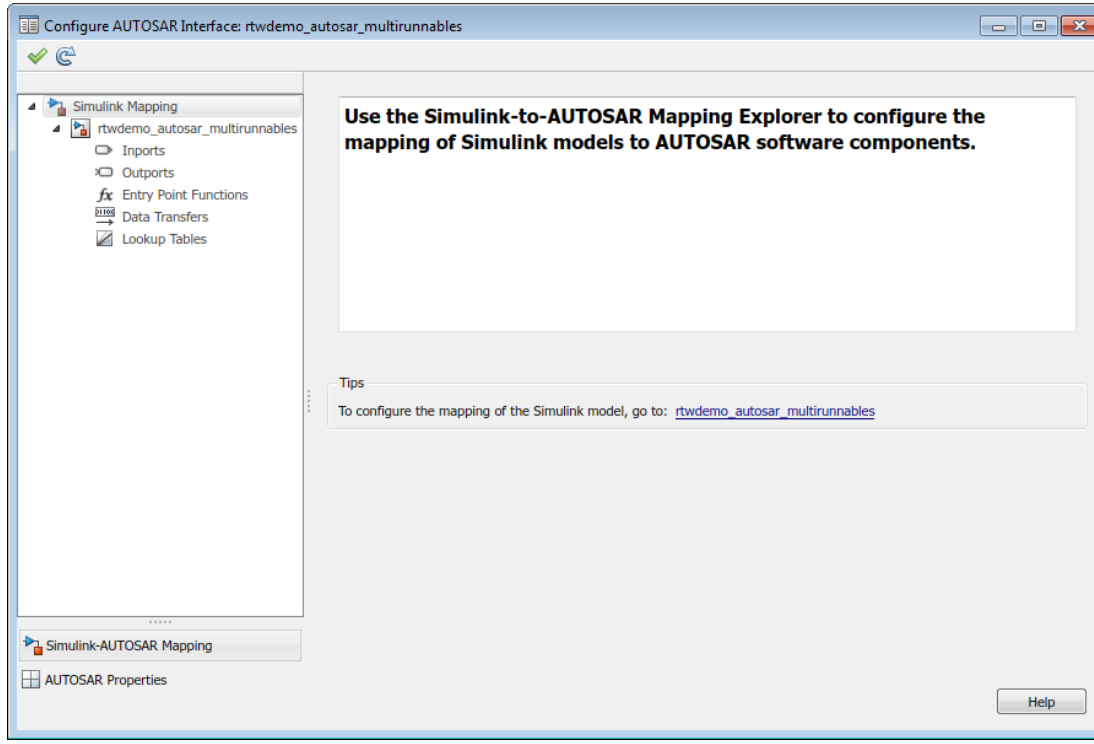
AUTOSAR Interface Configuration

After you have imported an AUTOSAR software component into a Simulink model, using `arxml.importer`, or created a Simulink representation of an AUTOSAR software component, using the AUTOSAR Component Builder, open the model and use the Configure AUTOSAR Interface dialog box to further develop the AUTOSAR component. The Configure AUTOSAR Interface dialog box provides two distinct views, which can be used separately and together to configure the AUTOSAR interface:

- Simulink-AUTOSAR Mapping Explorer — This view displays model inports, outports, entry-point functions, data transfers, and lookup tables in a tree format. Use this view to map model elements to AUTOSAR elements and interfaces from a Simulink model perspective.
- AUTOSAR Properties Explorer — This view displays a mapped AUTOSAR component and its elements, interfaces, computation methods, and XML options in a tree format. 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 4-267.

In a model for which the AUTOSAR system target file (`autosar.tlc`) has been selected, you can open the Configure AUTOSAR Interface dialog box by selecting **Code > C/C++ Code > Configure Model as AUTOSAR Component**.



As you progressively configure the model representation of the AUTOSAR component, you can:

- Freely switch between the Simulink and AUTOSAR perspectives, by clicking **Simulink-AUTOSAR Mapping** or **AUTOSAR Properties**.
- Click the **Validate** icon to validate the AUTOSAR interface configuration.
- Click the **Synchronize** icon to update the Simulink to AUTOSAR mapping of the model with changes to Simulink data transfers, entry-point functions, and function callers.
- Use the **Filter Contents** field (where available) to selectively display some elements, while omitting others, in the current view.

Note: Configuring an AUTOSAR interface requires an Embedded Coder license. If Embedded Coder is not licensed, the Configure AUTOSAR Interface dialog box runs in read-only mode.

Related Examples

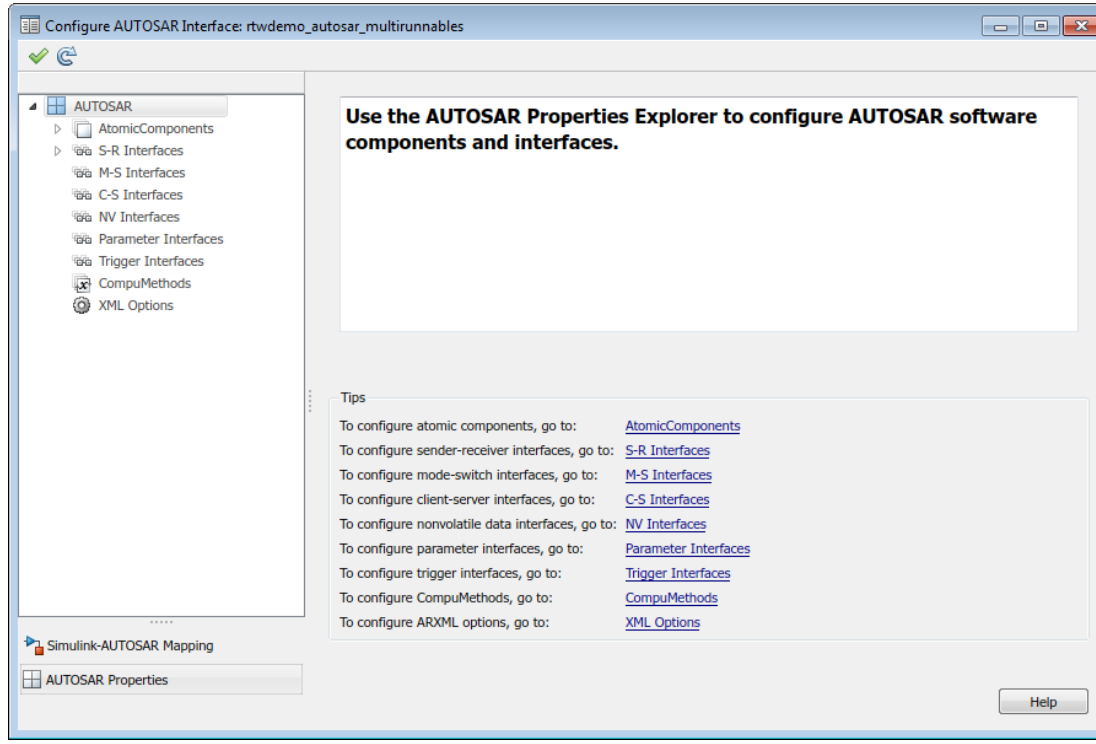
- “Configure AUTOSAR Component Using AUTOSAR Properties Explorer” on page 4-6
- “Map Model Elements Using Simulink-AUTOSAR Mapping Explorer” on page 4-60
- “Configure and Map AUTOSAR Component Programmatically” on page 4-267

Configure AUTOSAR Component Using AUTOSAR Properties Explorer

In Simulink, you can use the Configure AUTOSAR Interface dialog box to graphically configure an AUTOSAR software component and map Simulink model elements to AUTOSAR component elements. For more information, see "AUTOSAR Interface Configuration" on page 4-3.

To configure AUTOSAR elements from an AUTOSAR component perspective in Simulink:

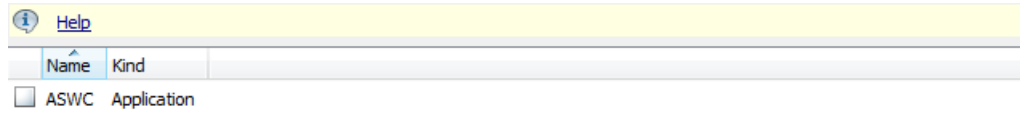
1. Open a model for which the AUTOSAR system target file (`autosar.tlc`) has been selected.
2. Open the Configure AUTOSAR Interface dialog box by selecting **Code > C/C++ Code > Configure Model as AUTOSAR Component**. If the AUTOSAR Properties Explorer is not already selected, click **AUTOSAR Properties**. The top-level view shows the types of AUTOSAR elements for which properties can be configured — atomic software components, communication interfaces, computation methods, and XML options.



3 In the left-hand pane of the Configure AUTOSAR Interface dialog box, under **AUTOSAR**, select **AtomicComponents**.

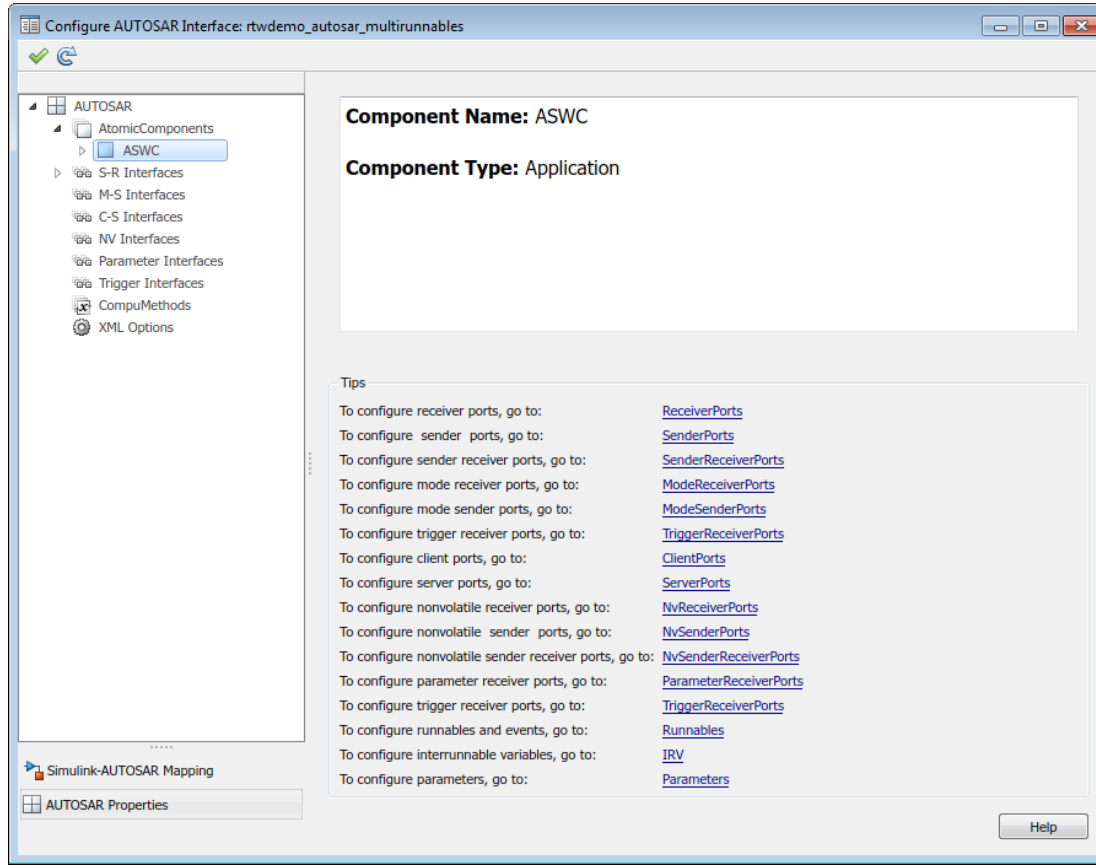
The Atomic Components view of the AUTOSAR Properties Explorer displays atomic components and their types. You can:

- Select an AUTOSAR component and then select a menu value for its kind: `Application` for an AUTOSAR application software component, or `Sensor Accuator` for an AUTOSAR sensor/actuator software component.
- Rename an AUTOSAR component by clicking its name and then editing the name text.



4 In the left-hand pane of the Configure AUTOSAR Interface dialog box, expand **AtomicComponents** and select an AUTOSAR component.

The component view of the AUTOSAR Properties Explorer displays the name and type of the selected component, and lists the types of AUTOSAR component elements for which properties can be configured — ports, runnables and events, and inter-runnable variables, and parameters.



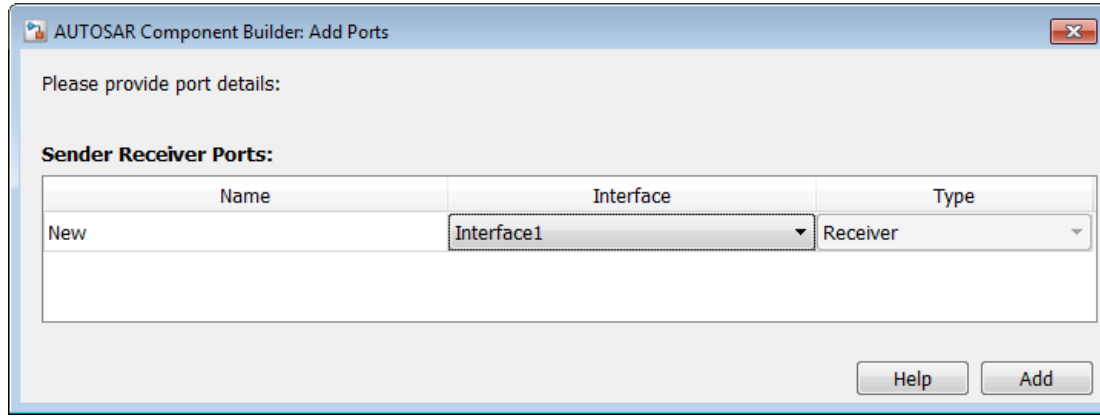
5 In the left-hand pane of the Configure AUTOSAR Interface dialog box, expand the component and select **ReceiverPorts**.

The Receiver Ports view of the AUTOSAR Properties Explorer 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.

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

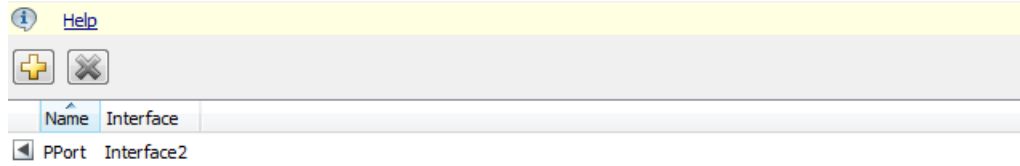


6 In the left-hand pane of the Configure AUTOSAR Interface dialog box, select **SenderPorts**.

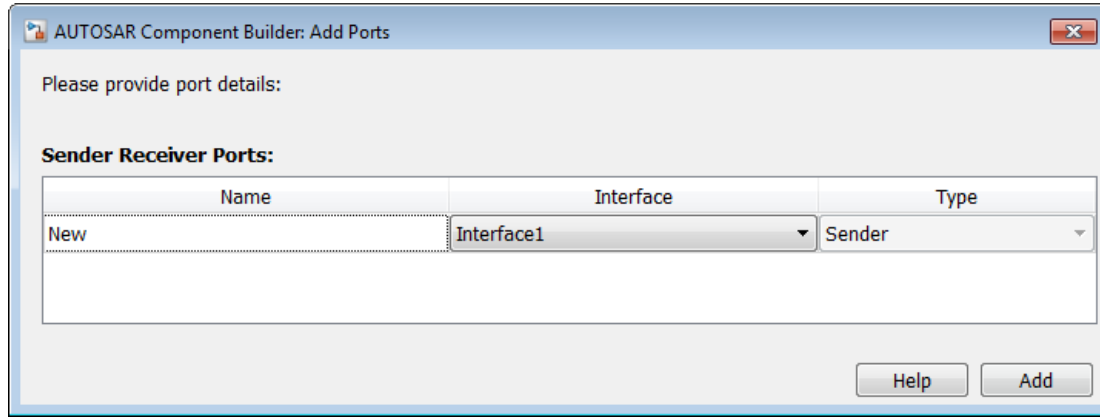
The Sender Ports view of the AUTOSAR Properties Explorer lists sender ports and their properties. You can:

- 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.

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



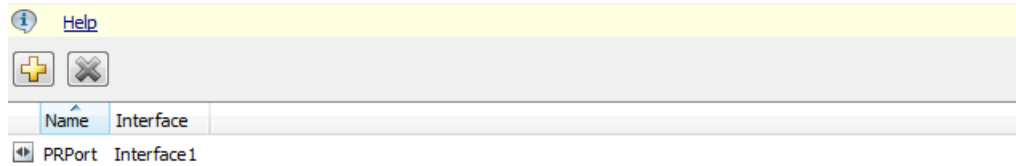
7 In the left-hand pane of the Configure AUTOSAR Interface dialog box, select **SenderReceiverPorts**.

The Sender-Receiver Ports view of the AUTOSAR Properties Explorer lists sender-receiver ports and their properties. You can:

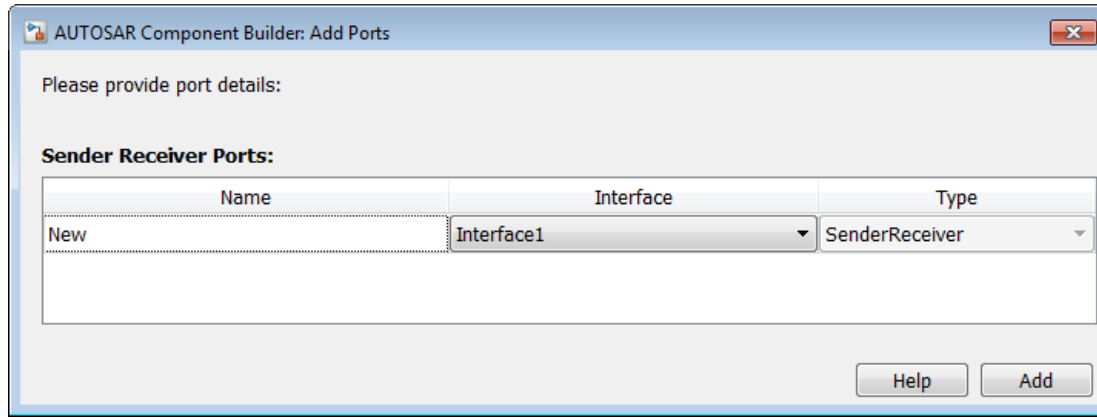
- Select an AUTOSAR sender-receiver port, and view and optionally reselect its associated S-R interface.
- Rename an AUTOSAR sender-receiver port by clicking its name and then editing the name text.

- Click the **Add** icon to open an Add Ports dialog box to add a port.
- Select a port and then click the **Delete** icon 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** in the Configuration Parameters dialog box.



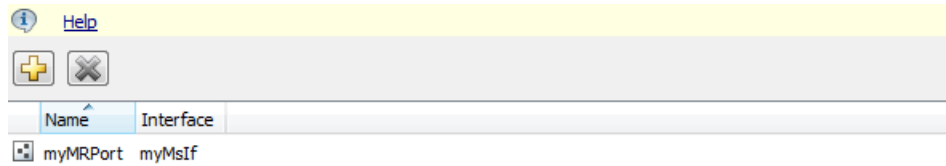
The Add Ports dialog box lets you add a sender-receiver port and associate it with an existing S-R interface. Click **Add** to add the port and return to the Sender-Receiver Ports view.



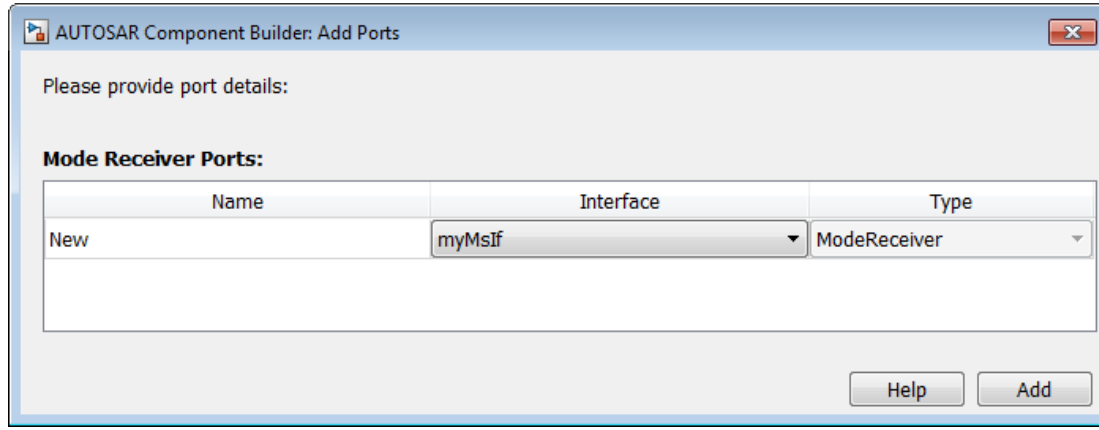
8 In the left-hand pane of the Configure AUTOSAR Interface dialog box, select **ModeReceiverPorts**.

The Mode Receiver Ports view of the AUTOSAR Properties Explorer 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** icon to open an Add Ports dialog box to add a port.
- Select a port and then click the **Delete** icon 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.



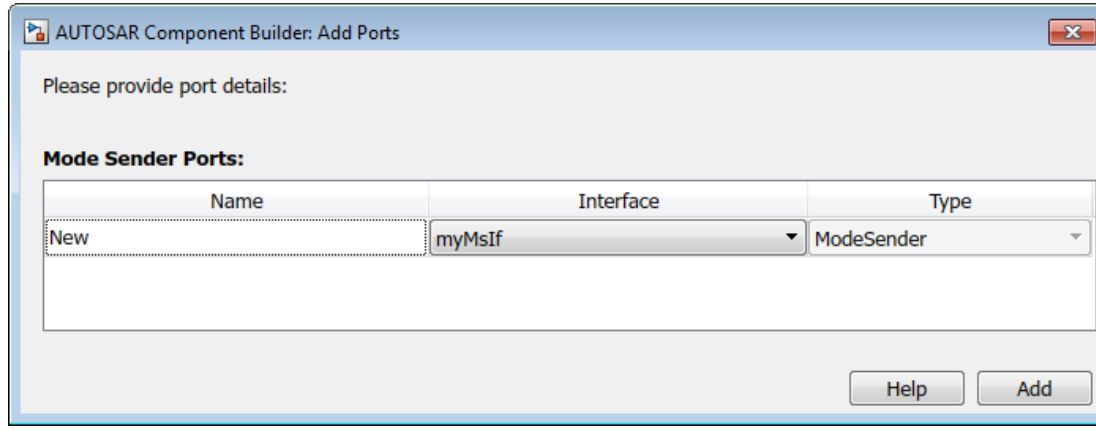
For more information about adding and configuring mode-receiver ports, see “Configure AUTOSAR Mode-Switch Communication” on page 4-130.

9 In the left-hand pane of the Configure AUTOSAR Interface dialog box, select **ModeSenderPorts**.

The Mode Sender Ports view of the AUTOSAR Properties Explorer 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** icon to open an Add Ports dialog box to add a port.
- Select a port and then click the **Delete** icon to remove it.

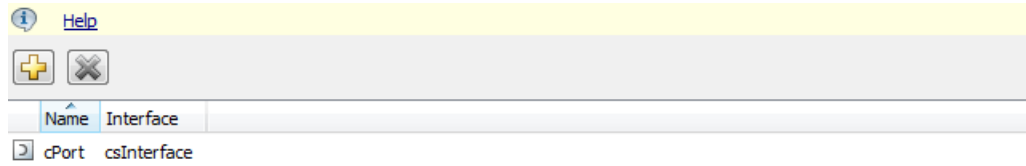
The Add Ports dialog box lets you add a mode sender port and associate it with an existing M-S interface. If an M-S interface does not exist in the component, you must create one before adding the port. Click **Add** to add the port and return to the Mode Sender Ports view.



10 In the left-hand pane of the Configure AUTOSAR Interface dialog box, select **ClientPorts**.

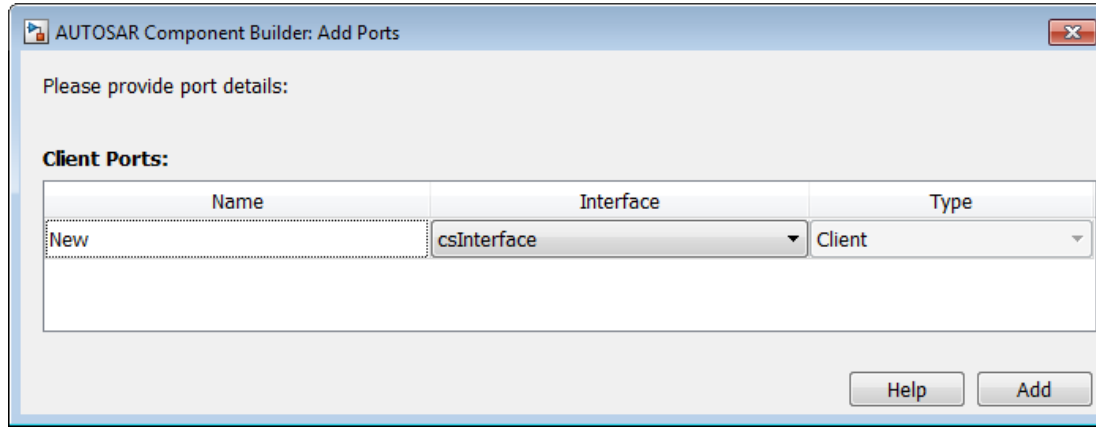
The Client Ports view of the AUTOSAR Properties Explorer lists client ports and their properties. You can:

- 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** icon to open an Add Ports dialog box to add a client port.
- Select a port and then click the **Delete** icon 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.

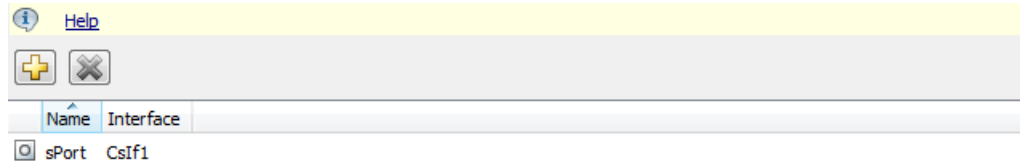


For more information, see “Configure AUTOSAR Client-Server Communication” on page 4-101.

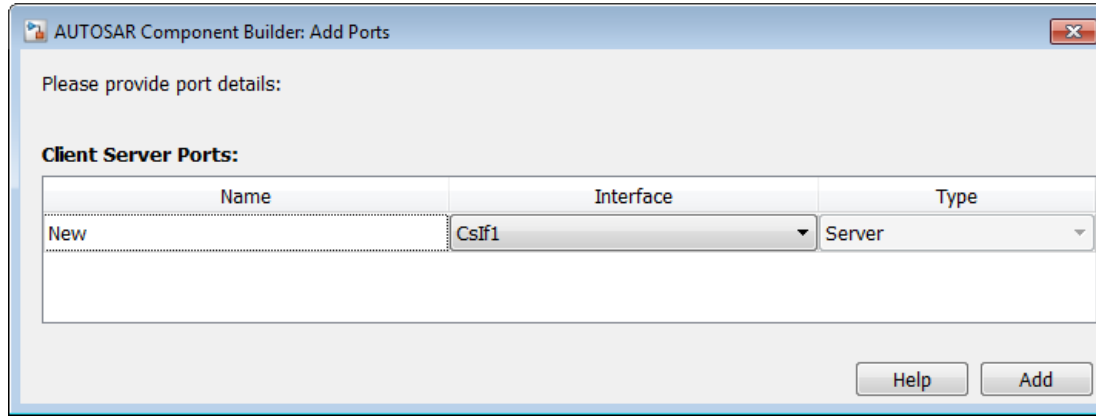
11 In the left-hand pane of the Configure AUTOSAR Interface dialog box, select **ServerPorts**.

The Server Ports view of the AUTOSAR Properties Explorer lists client 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. Set the name to the name of the associated server function.
- Click the **Add** icon to open an Add Ports dialog box to add a server port.
- Select a port and then click the **Delete** icon to remove it.



The Add Ports dialog box lets you add a server port and associate it with an existing C-S interface. Set **Name** to the name of the associated server function. 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.



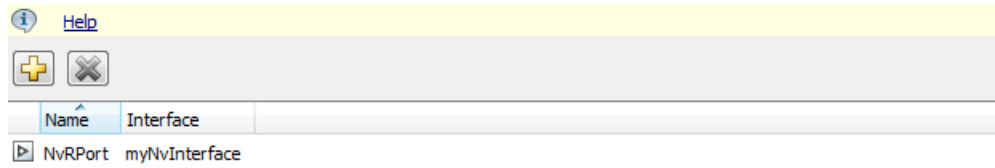
For more information, see “Configure AUTOSAR Client-Server Communication” on page 4-101.

12 In the left-hand pane of the Configure AUTOSAR Interface dialog box, expand the component and select **NvReceiverPorts**.

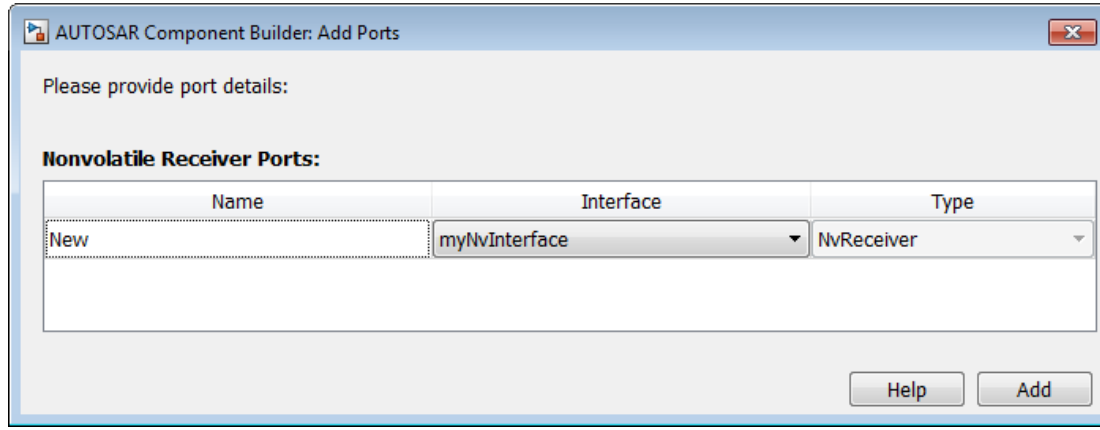
The NV Receiver Ports view of the AUTOSAR Properties Explorer lists nonvolatile (NV) receiver ports and their properties. You can:

- 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** icon to open an Add Ports dialog box to add a port.
- Select a port and then click the **Delete** icon 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.



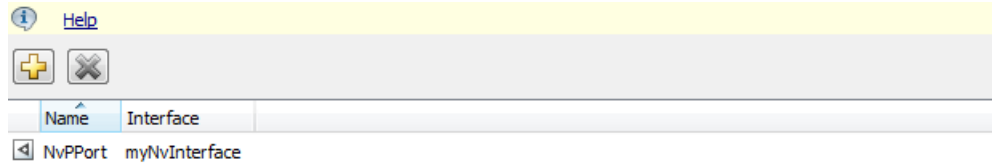
For more information, see “Configure AUTOSAR Nonvolatile Data Communication” on page 4-141.

13 In the left-hand pane of the Configure AUTOSAR Interface dialog box, select **NvSenderPorts**.

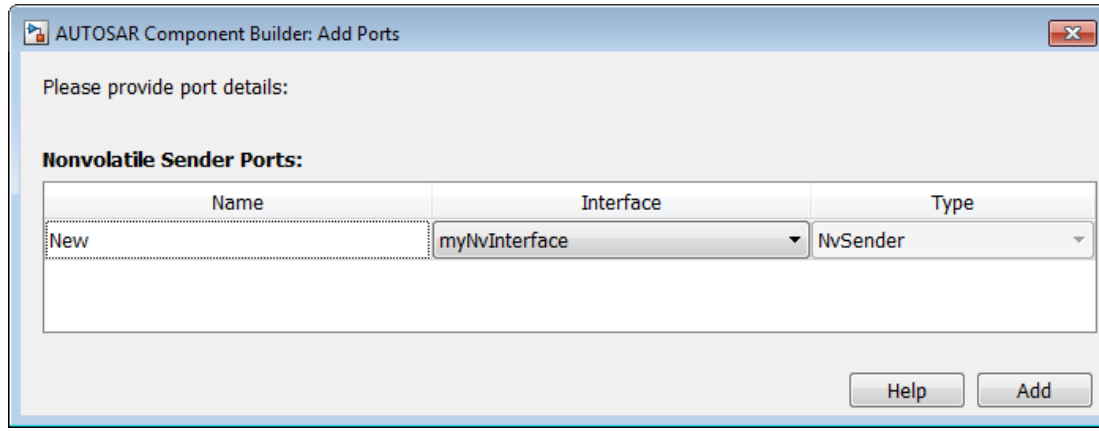
The NV Sender Ports view of the AUTOSAR Properties Explorer lists nonvolatile (NV) sender ports and their properties. You can:

- 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** icon to open an Add Ports dialog box to add a port.
- Select a port and then click the **Delete** icon 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.



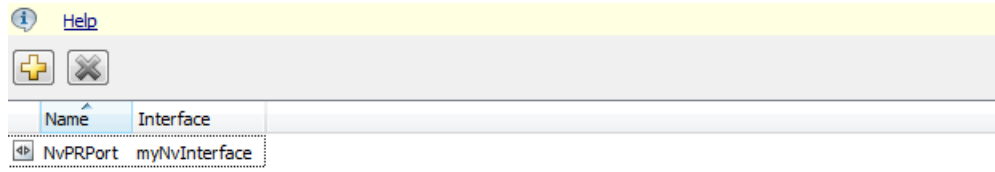
For more information, see “Configure AUTOSAR Nonvolatile Data Communication” on page 4-141.

14 In the left-hand pane of the Configure AUTOSAR Interface dialog box, select **NvSenderReceiverPorts**.

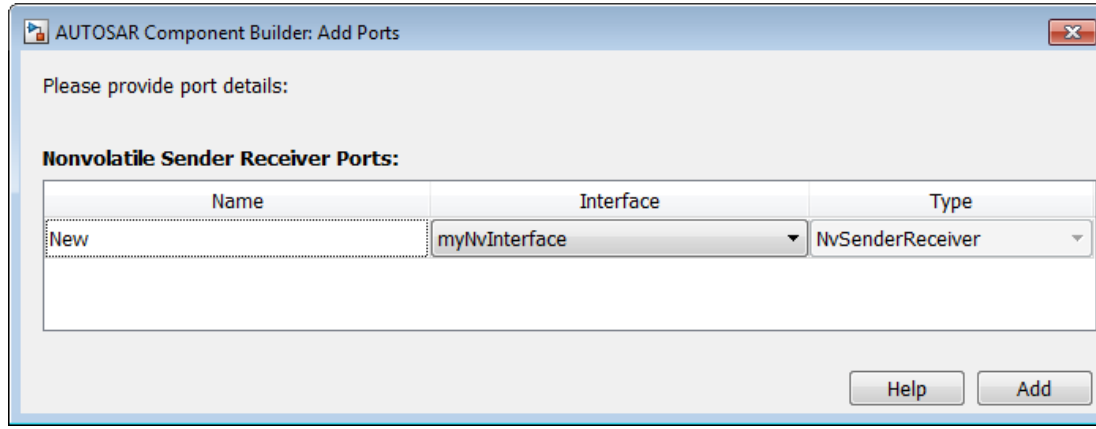
The NV Sender-Receiver Ports view of the AUTOSAR Properties Explorer lists nonvolatile (NV) sender-receiver ports and their properties. You can:

- 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** icon to open an Add Ports dialog box to add a port.
- Select a port and then click the **Delete** icon 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** 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.

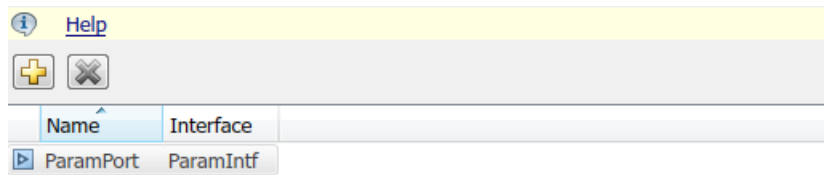


For more information, see "Configure AUTOSAR Nonvolatile Data Communication" on page 4-141.

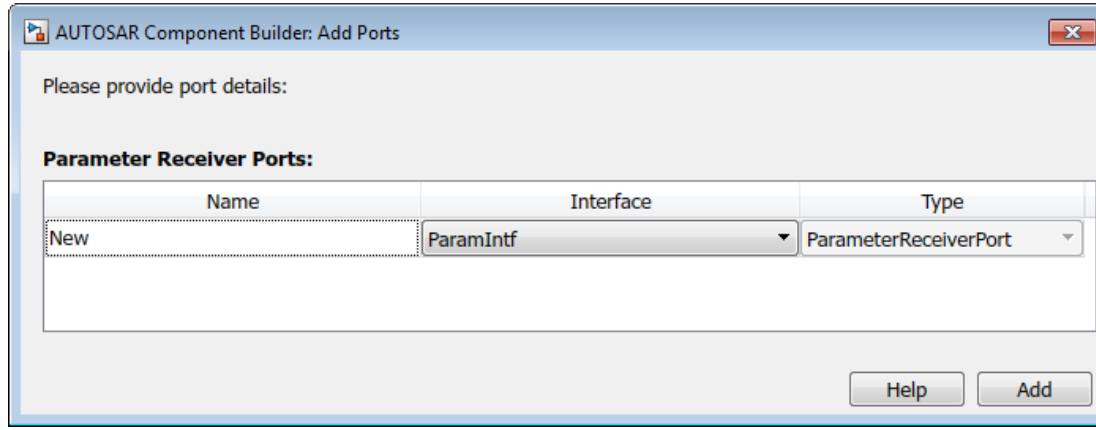
15 In the left-hand pane of the Configure AUTOSAR Interface dialog box, expand the component and select **ParameterReceiverPorts**.

The Parameter Receiver Ports view of the AUTOSAR Properties Explorer 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** icon to open an Add Ports dialog box to add a port.
- Select a port and then click the **Delete** icon 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.

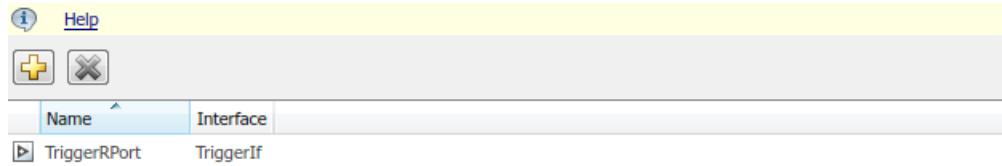


For more information, see “Configure Receiver for AUTOSAR Parameter Communication” on page 4-144.

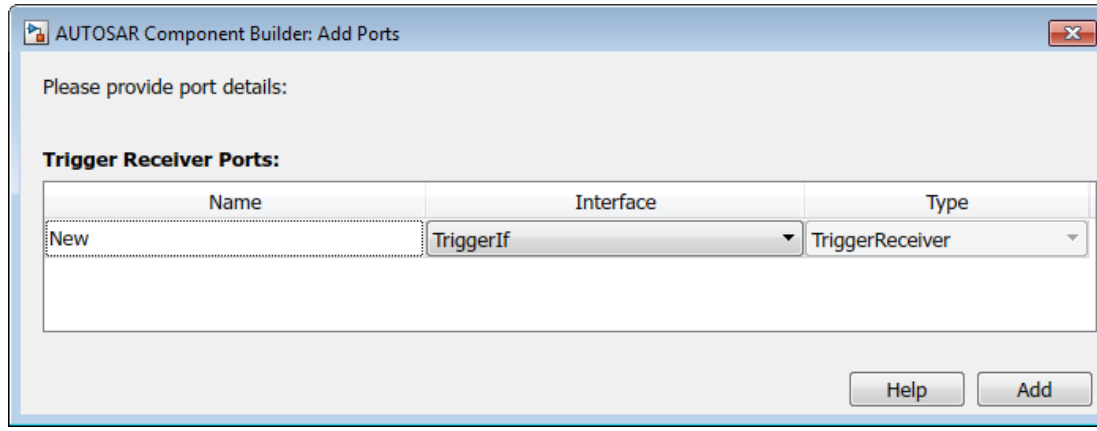
16 In the left-hand pane of the Configure AUTOSAR Interface dialog box, expand the component and select **TriggerReceiverPorts**.

The Trigger Receiver Ports view of the AUTOSAR Properties Explorer 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** icon to open an Add Ports dialog box to add a port.
- Select a port and then click the **Delete** icon 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.



For more information, see “Configure Receiver for AUTOSAR External Trigger Event Communication” on page 4-148.

17 In the left-hand pane of the Configure AUTOSAR Interface dialog box, select **Runnables**.

The Runnables view of the AUTOSAR Properties Explorer 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.

rtwdemo_autosar_multirunnables.arxml

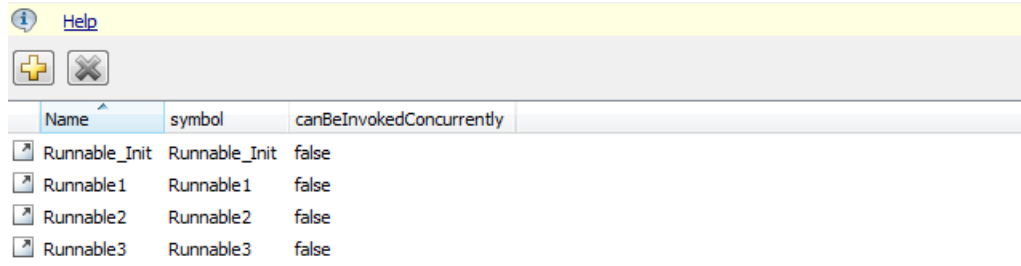
```
<RUNNABLE-ENTITY UUID="65432c3e-34c7-5e82-4229-f6d04927eb78">
    <SHORT-NAME>Runnable1</SHORT-NAME>
...
    <SYMBOL>test_symbol</SYMBOL>
...
</RUNNABLE-ENTITY>
```

rtwdemo_autosar_multirunnables.c

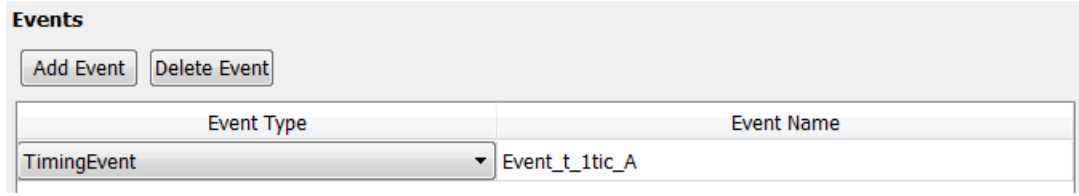
```
/* Output function for RootInportFunctionCallGenerator:
   '<Root>/RootFcnCall_InsertedFor_Runnable1_at_outport_1' */
void test_symbol(void)
{
...
}
```

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

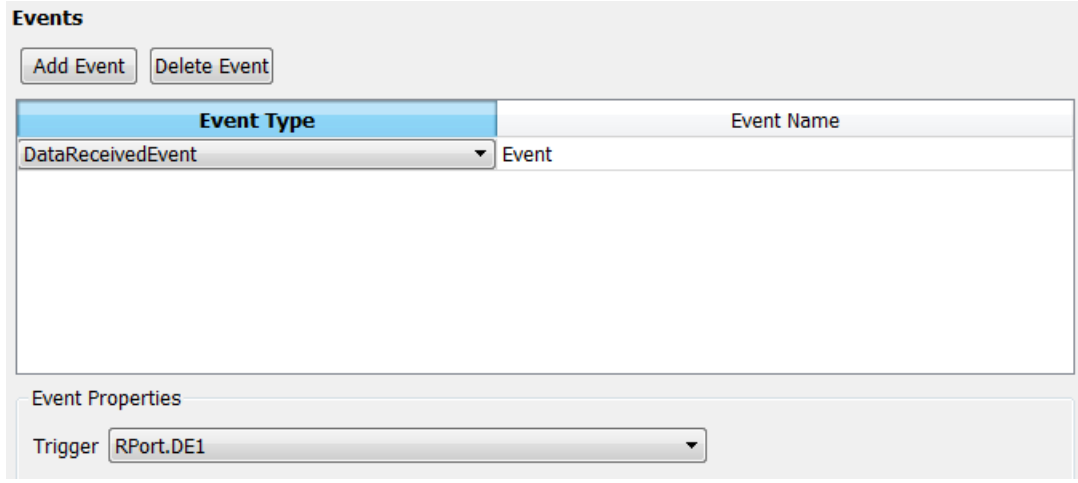
- For an AUTOSAR server runnable, set the runnable property `canBeInvokedConcurrently` to designate whether to enforce concurrency constraints. For nonserver runnables, leave `canBeInvokedConcurrently` set to `false`. For more information, see "Concurrency Constraints for AUTOSAR Server Runnables" on page 4-126.
- Click the **Add** icon to add an AUTOSAR runnable.
- Select an AUTOSAR runnable and then click the **Delete** icon 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.

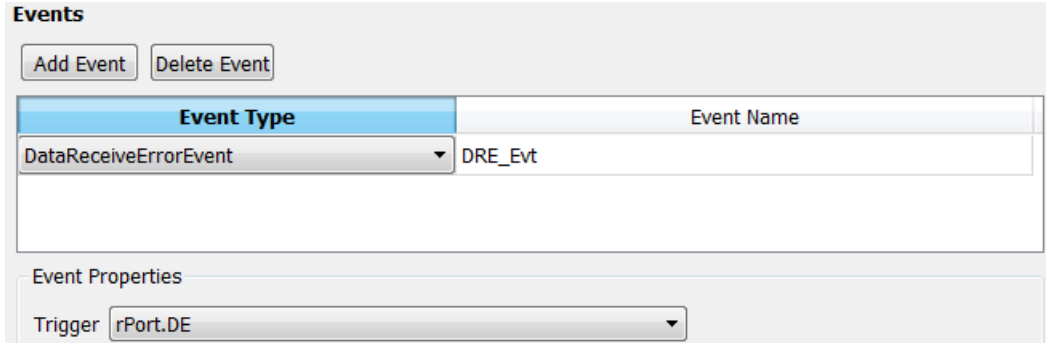


If you select an event of type `DataReceivedEvent`, the runnable is activated by a `DataReceivedEvent`. Select the event name to display its **Trigger** property. Select a trigger for the event from the list of available trigger ports.

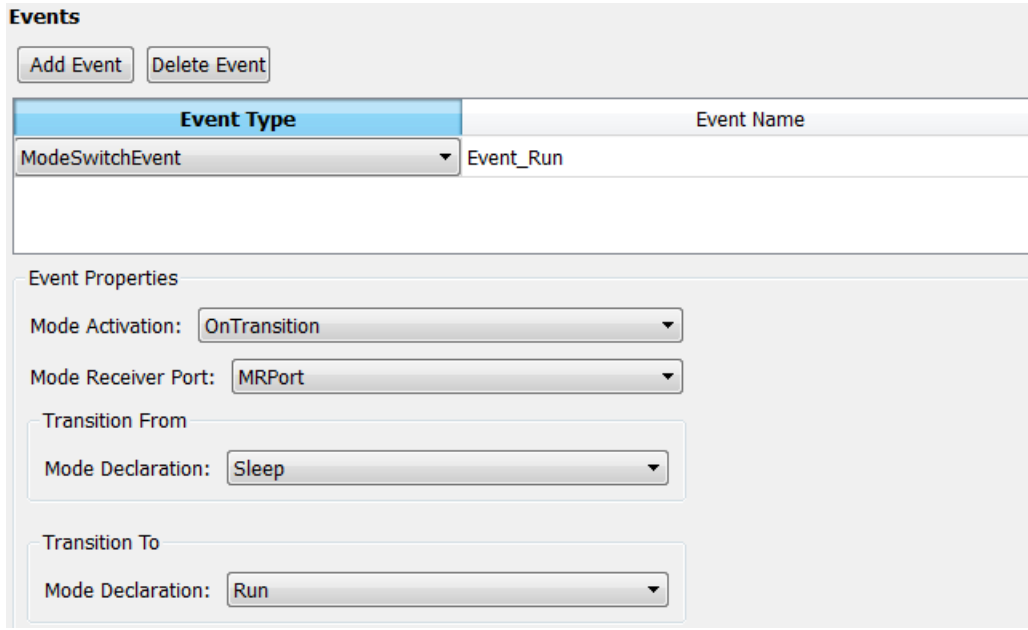


If you select an event of type `DataReceiveErrorEvent`, the runnable is activated by a `DataReceiveErrorEvent`. Select the event name to display its **Trigger** property. Select a trigger for the event from the list of available trigger ports. (For

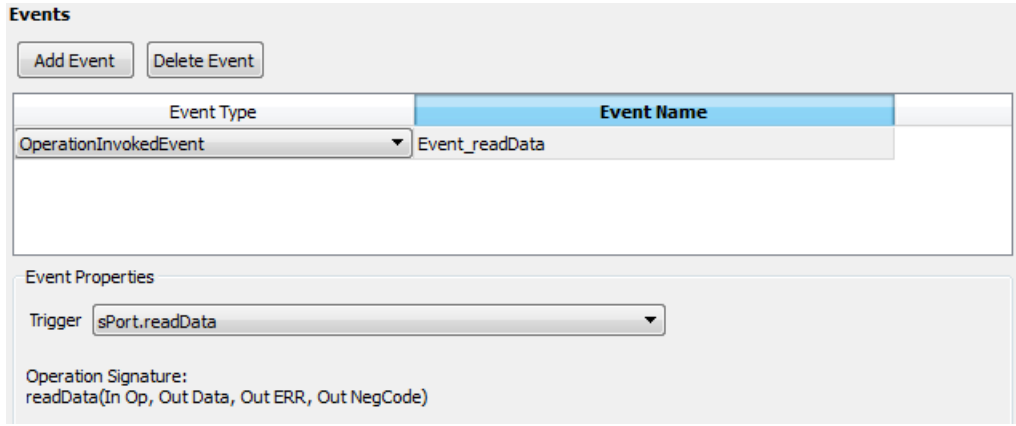
more information on using a `DataReceiveErrorEvent`, see "Configure AUTOSAR Receiver Port for DataReceiveErrorEvent" on page 4-97.)



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 4-130.)

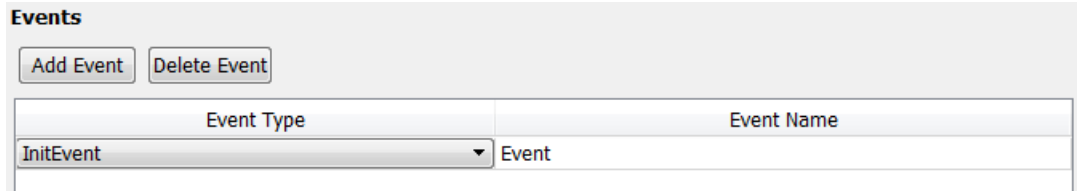


If you select an event of type `OperationInvokedEvent`, the runnable becomes an AUTOSAR server runnable. Select the event name to display its **Trigger** property. Select a trigger for the event from the list of available server port and operation combinations. The **Operation Signature** is displayed below the **Trigger** property. (For more information on using an `OperationInvokedEvent`, see "Configure AUTOSAR Client-Server Communication" on page 4-101.)

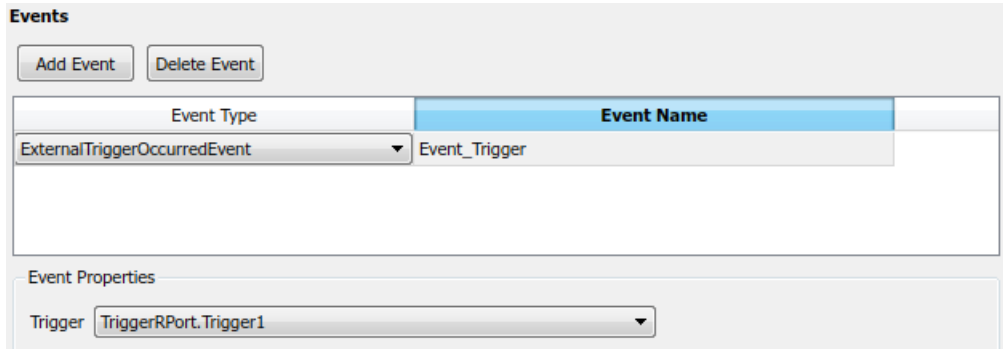


If you select an event of type `InitEvent`, you can rename the event by clicking its name and then editing the name text. (For more information on using an `InitEvent`, see "Configure AUTOSAR Initialization Runnable (R4.1)" on page 4-226.)

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** in the Configuration Parameters dialog box.



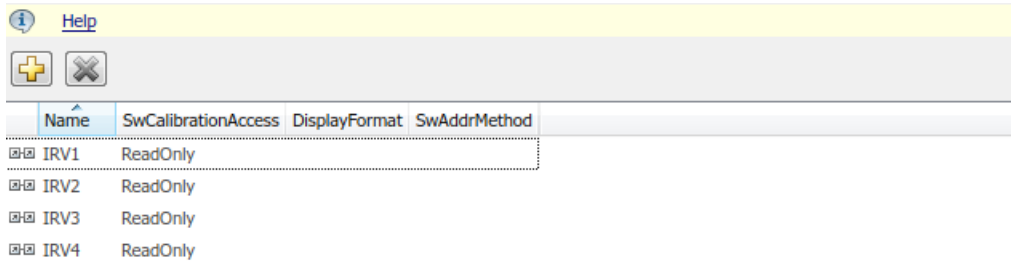
If you select an event of type `ExternalTriggerOccurredEvent`, the runnable is activated when an AUTOSAR software component or service signals an external trigger event. Select the event name to display its **Trigger** property. Select a trigger for the event from the list of available trigger receiver port and trigger combinations. (For more information on using an `ExternalTriggerOccurredEvent`, see "Configure Receiver for AUTOSAR External Trigger Event Communication" on page 4-148.)



18 In the left-hand pane of the Configure AUTOSAR Interface dialog box, select **IRV**.

The IRV view of the AUTOSAR Properties Explorer lists inter-runnable variables for the AUTOSAR component. You can:

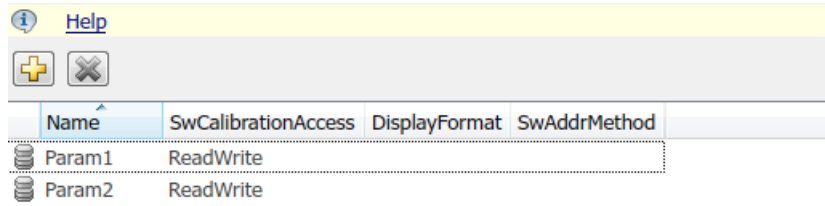
- Rename an AUTOSAR IRV by clicking its name and then editing the name text.
- Specify the level of measurement and calibration tool access to IRV data. Select an IRV and set its **SwCalibrationAccess** value to `ReadOnly`, `ReadWrite`, or `NotAccessible`.
- Optionally specify the format to be used by 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 4-194.
- 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 SwAddrMethod" on page 4-198.
- Click the **Add** icon to add an AUTOSAR IRV.
- Select an AUTOSAR IRV and then click the **Delete** icon to remove it.



19 In the left-hand pane of the Configure AUTOSAR Interface dialog box, select **Parameters**.

The Parameters view of the AUTOSAR Properties Explorer lists internal parameters (that is, parameters that are not port-based) for the AUTOSAR component. You can:

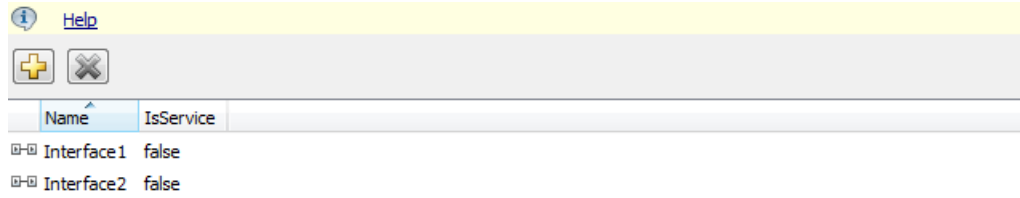
- Rename an AUTOSAR parameter by clicking its name and then editing the name text.
- Specify the level of measurement and calibration tool access to parameters. Select a parameter and set its **SwCalibrationAccess** value to `ReadOnly`, `ReadWrite`, or `NotAccessible`.
- Optionally specify the format 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 4-194.
- Optionally specify a software address method for the parameter data. Select or enter a value for **SwAddrMethod**. AUTOSAR software components use `SwAddrMethod`s to group data in memory for access by measurement and calibration tools. For more information, see “Configure SwAddrMethod” on page 4-198.
- Click the **Add** icon to add an AUTOSAR internal parameter.
- Select an AUTOSAR internal parameter and then click the **Delete** icon to remove it.



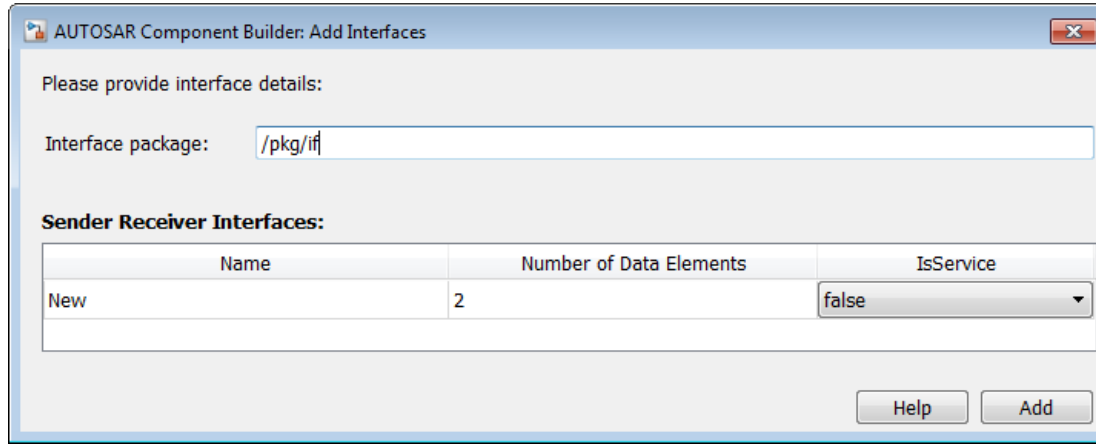
20 In the left-hand pane of the Configure AUTOSAR Interface dialog box, select **S-R Interfaces**.

The S-R Interfaces view of the AUTOSAR Properties Explorer 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** icon 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** icon 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.



21 In the left-hand pane of the Configure AUTOSAR Interface dialog box, expand **S-R Interfaces** and select an S-R interface from the list.

The S-R interface view of the AUTOSAR Properties Explorer 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:

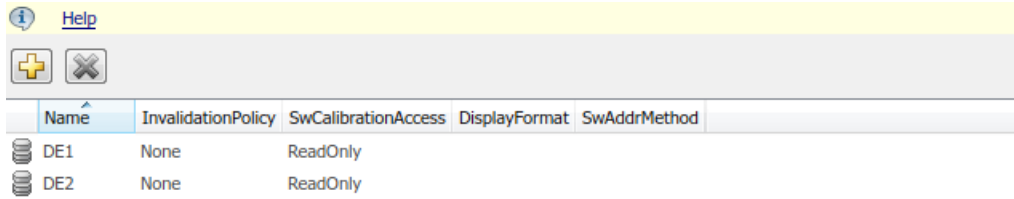
- Enter a package path in the **Package** parameter field.
- Click the button to the right of the **Package** field to open the AUTOSAR Package Browser. Use the browser to navigate to an existing package or create a new package. When you select a package in the browser and click **Apply**, the interface **Package** parameter value is updated with your selection. For more information about the AUTOSAR Package Browser, see “Configure AUTOSAR Package for Interface or CompuMethod” on page 4-67.

22 In the left-hand pane of the Configure AUTOSAR Interface dialog box, expand the selected interface and select **DataElements**.

The Data Elements view of the AUTOSAR Properties Explorer lists AUTOSAR sender-receiver interface data elements and their properties. You can:

- Select an S-R interface data element and edit the name value.
- If signal invalidation is configured for the data element on a sender port, specify an invalidation policy. Set **InvalidationPolicy** to `Keep`, `Replace`, or `DontInvalidate`. The default is `None`. For more information, see "Configure AUTOSAR Sender Port for Data Element Invalidation" on page 4-90.
- 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 4-194.

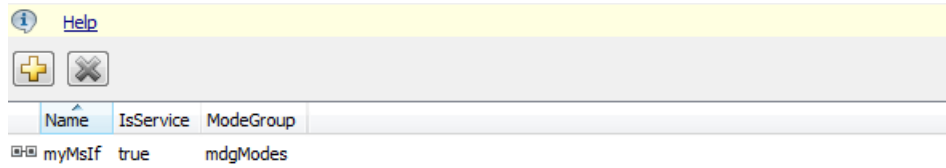
- Optionally specify a software address method for the data element. Select or enter a value for **SwAddrMethod**. AUTOSAR software components use `SwAddrMethod`s to group data in memory for access by measurement and calibration tools. For more information, see "Configure SwAddrMethod" on page 4-198.
- Click the **Add** icon to add a data element.
- Select a data element and then click the **Delete** icon to remove it.



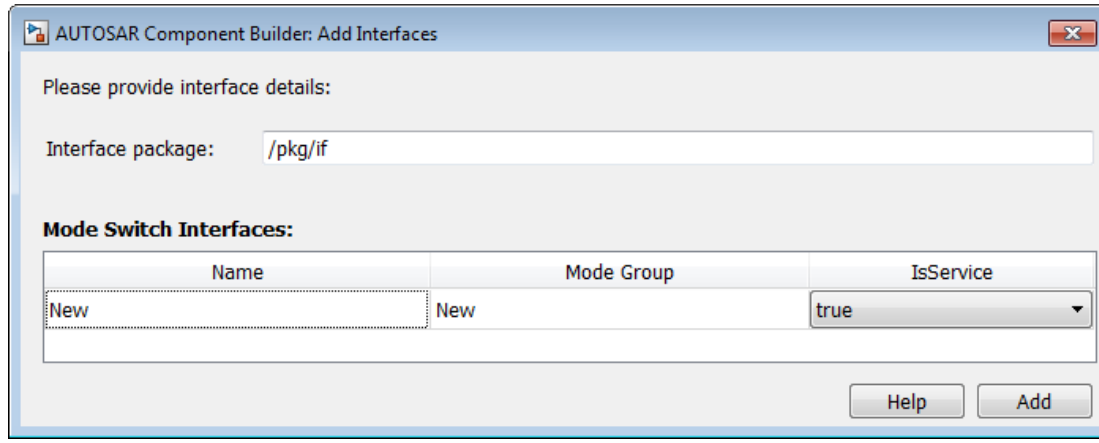
23 In the left-hand pane of the Configure AUTOSAR Interface dialog box, select **M-S Interfaces**.

The M-S Interfaces view of the AUTOSAR Properties Explorer 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 4-130.
- Rename an M-S interface by clicking its name and then editing the name text.
- Click the **Add** icon 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** icon 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.



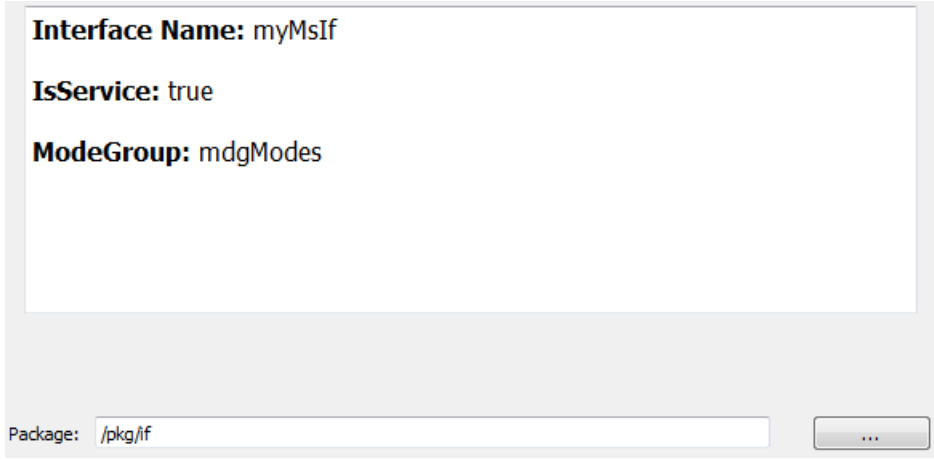
For more information, see “Configure AUTOSAR Mode-Switch Communication” on page 4-130.

24 In the left-hand pane of the Configure AUTOSAR Interface dialog box, expand **M-S Interfaces** and select an M-S interface from the list.

The M-S interface view of the AUTOSAR Properties Explorer 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 Interface or CompuMethod" on page 4-67.



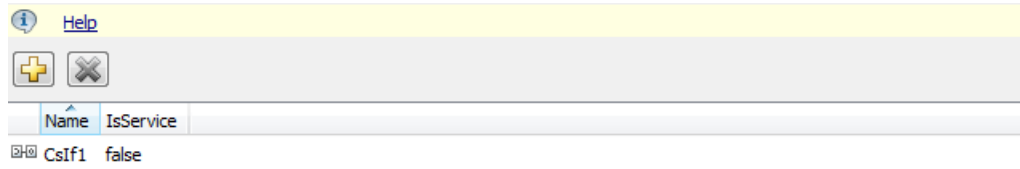
For more information, see "Configure AUTOSAR Mode-Switch Communication" on page 4-130.

25 In the left-hand pane of the Configure AUTOSAR Interface dialog box, select **C-S Interfaces**.

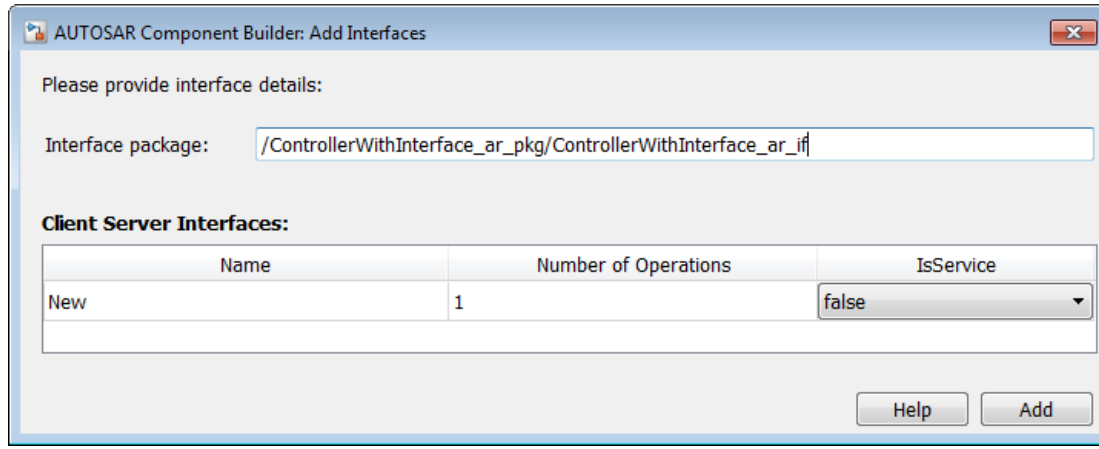
The C-S Interfaces view of the AUTOSAR Properties Explorer lists AUTOSAR client-server interfaces and their properties. You can:

- 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** icon 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** icon 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.



For more information, see “Configure AUTOSAR Client-Server Communication” on page 4-101.

26 In the left-hand pane of the Configure AUTOSAR Interface dialog box, expand **C-S Interfaces** and select a C-S interface from the list.

The C-S interface view of the AUTOSAR Properties Explorer 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 Interface or CompuMethod" on page 4-67.

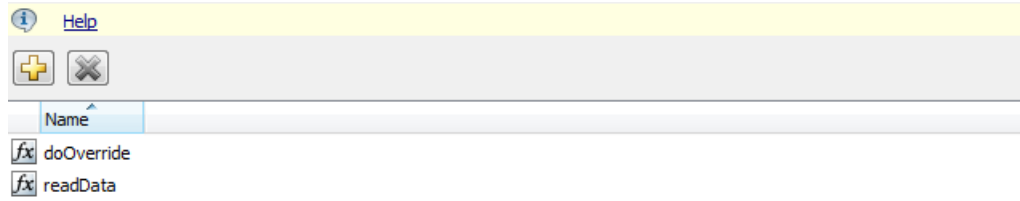
For more information, see "Configure AUTOSAR Client-Server Communication" on page 4-101.

27 In the left-hand pane of the Configure AUTOSAR Interface dialog box, expand the selected interface and select **Operations**.

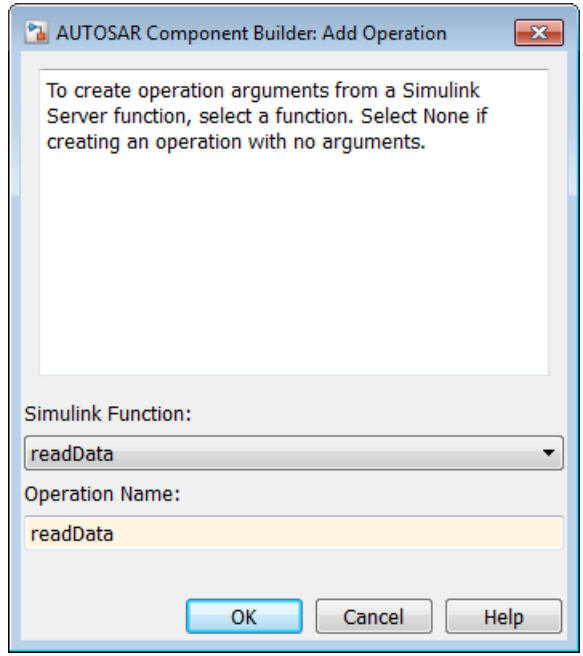
The Operations view of the AUTOSAR Properties Explorer lists AUTOSAR client-server interface operations. You can:

- Select a C-S interface operation and edit the name value.

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



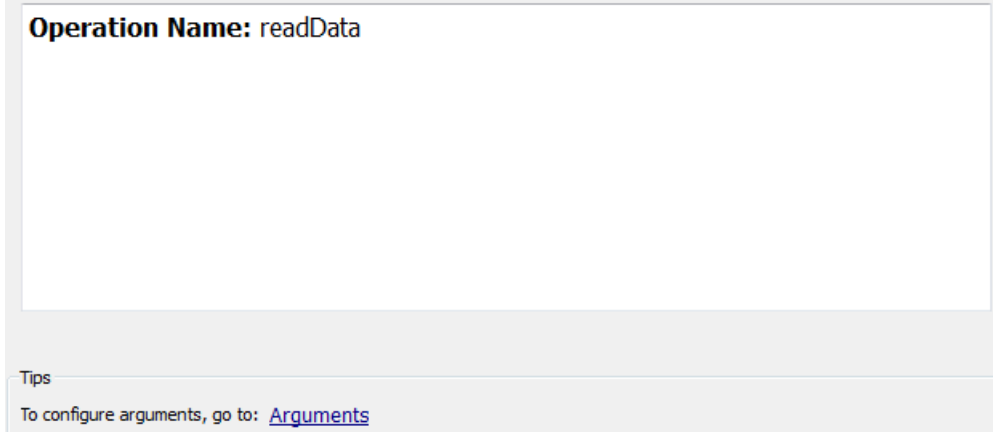
The Add Operation dialog box lets you specify the name of a new C-S interface operation. To create operation arguments from a Simulink function, select the associated Simulink function among those present in the configuration. Select `None` if you are creating an operation without arguments.



For more information, see “Configure AUTOSAR Client-Server Communication” on page 4-101.

28 In the left-hand pane of the Configure AUTOSAR Interface dialog box, expand **Operations** and select an operation from the list.

The operation view of the AUTOSAR Properties Explorer displays the name of the selected C-S operation.



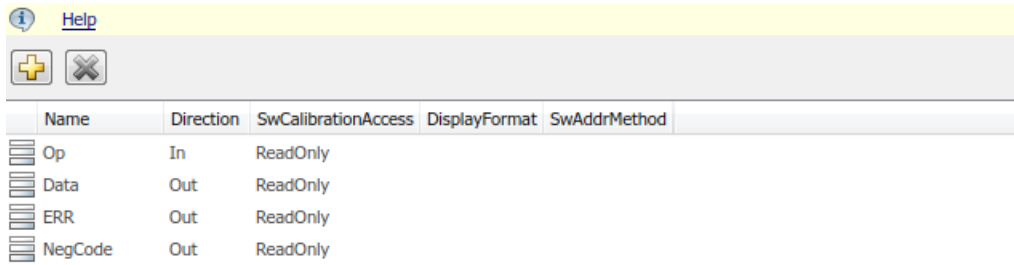
For more information, see “Configure AUTOSAR Client-Server Communication” on page 4-101.

29 In the left-hand pane of the Configure AUTOSAR Interface dialog box, expand the selected operation and select **Arguments**.

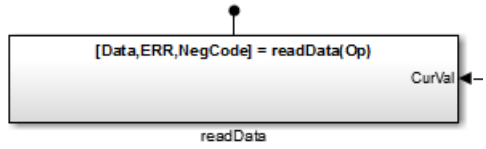
The Arguments view of the AUTOSAR Properties Explorer lists AUTOSAR client-server operation arguments and their properties. You can:

- 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 4-120.
- 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 4-194.
- 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 SwAddrMethod" on page 4-198.
- Click the **Add** icon to add an argument.
- Select an argument and then click the **Delete** icon to remove it.



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

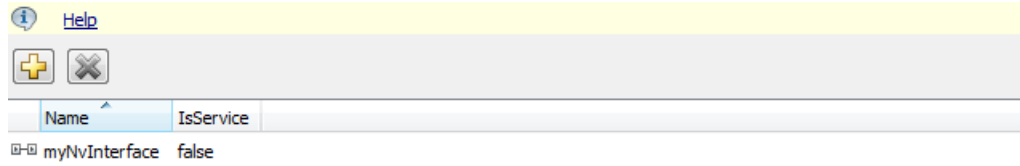


For more information, see "Configure AUTOSAR Client-Server Communication" on page 4-101.

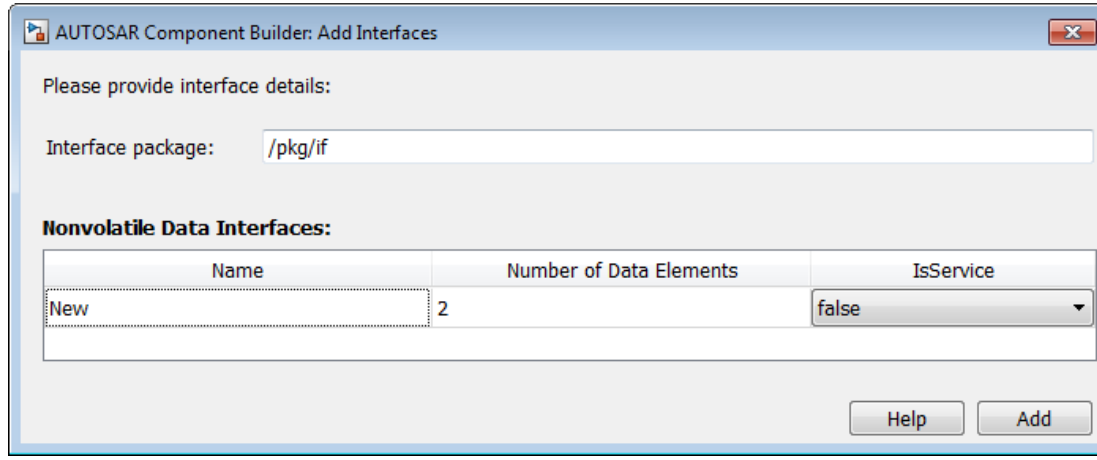
30 In the left-hand pane of the Configure AUTOSAR Interface dialog box, select **NV Interfaces**.

The NV Interfaces view of the AUTOSAR Properties Explorer lists AUTOSAR nonvolatile (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** icon to open an Add Interfaces dialog box to add one or more NV interfaces.
- Select an NV interface and then click the **Delete** icon 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.



For more information, see “Configure AUTOSAR Nonvolatile Data Communication” on page 4-141.

31 In the left-hand pane of the Configure AUTOSAR Interface dialog box, expand **NV Interfaces** and select an NV interface from the list.

The NV interface view of the AUTOSAR Properties Explorer displays the name of the selected nonvolatile (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:

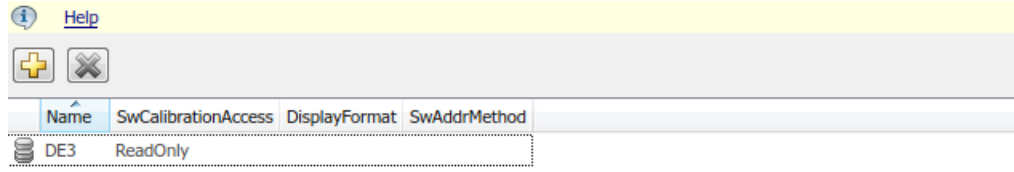
- Enter a package path in the **Package** parameter field.
- Click the button to the right of the **Package** field to open the AUTOSAR Package Browser. Use the browser to navigate to an existing package or create a new package. When you select a package in the browser and click **Apply**, the interface **Package** parameter value is updated with your selection. For more information about the AUTOSAR Package Browser, see “Configure AUTOSAR Package for Interface or CompuMethod” on page 4-67.

For more information, see “Configure AUTOSAR Nonvolatile Data Communication” on page 4-141.

32 In the left-hand pane of the Configure AUTOSAR Interface dialog box, expand the selected interface and select **DataElements**.

The Data Elements view of the AUTOSAR Properties Explorer lists AUTOSAR nonvolatile (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 4-194.
- Optionally specify a software address method for the data element. Select or enter a value for **SwAddrMethod**. AUTOSAR software components use `SwAddrMethod`s to group data in memory for access by measurement and calibration tools. For more information, see “Configure SwAddrMethod” on page 4-198.
- Click the **Add** icon to add a data element.
- Select a data element and then click the **Delete** icon to remove it.

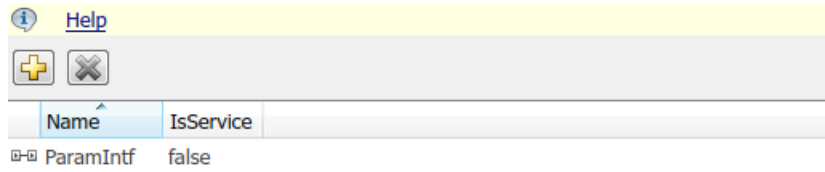


For more information, see “Configure AUTOSAR Nonvolatile Data Communication” on page 4-141.

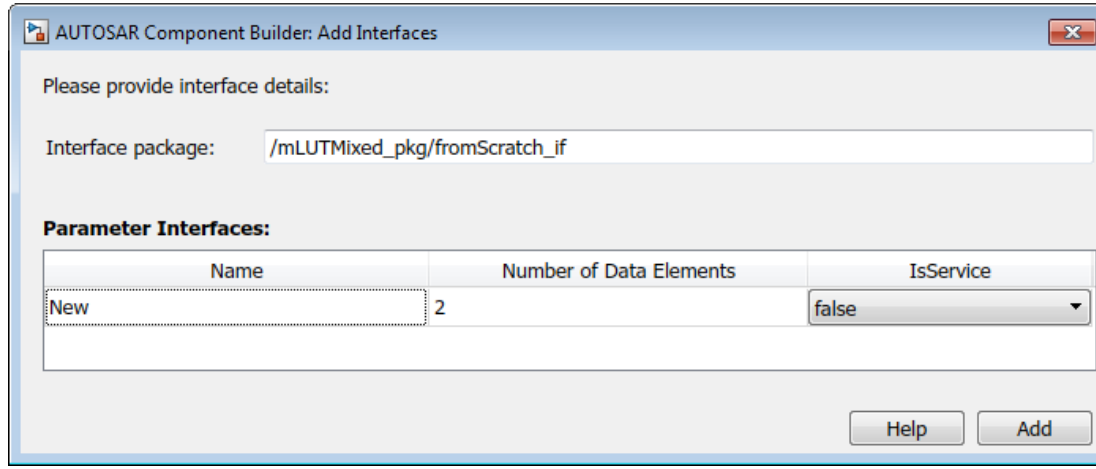
33 In the left-hand pane of the Configure AUTOSAR Interface dialog box, select **Parameter Interfaces**.

The Parameter Interfaces view of the AUTOSAR Properties Explorer 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** icon to open an Add Interfaces dialog box to add one or more parameter interfaces.
- Select a parameter interface and then click the **Delete** icon 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.



For more information, see "Configure Receiver for AUTOSAR Parameter Communication" on page 4-144.

34 In the left-hand pane of the Configure AUTOSAR Interface dialog box, expand **Parameter Interfaces** and select a parameter interface from the list.

The parameter interface view of the AUTOSAR Properties Explorer displays the name of the selected parameter 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 Interface or CompuMethod" on page 4-67.

For more information, see “Configure Receiver for AUTOSAR Parameter Communication” on page 4-144.

35 In the left-hand pane of the Configure AUTOSAR Interface dialog box, expand the selected interface and select **DataElements**.

The Data Elements view of the AUTOSAR Properties Explorer 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 4-194.
- Optionally specify a software address method for the data element. Select or enter a value for **SwAddrMethod**. AUTOSAR software components use

`SwAddrMethod`s to group data in memory for access by measurement and calibration tools. For more information, see "Configure SwAddrMethod" on page 4-198.

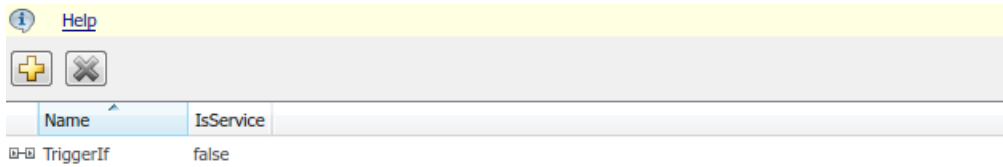
- Click the **Add** icon to add a data element.
- Select a data element and then click the **Delete** icon to remove it.

For more information, see "Configure Receiver for AUTOSAR Parameter Communication" on page 4-144.

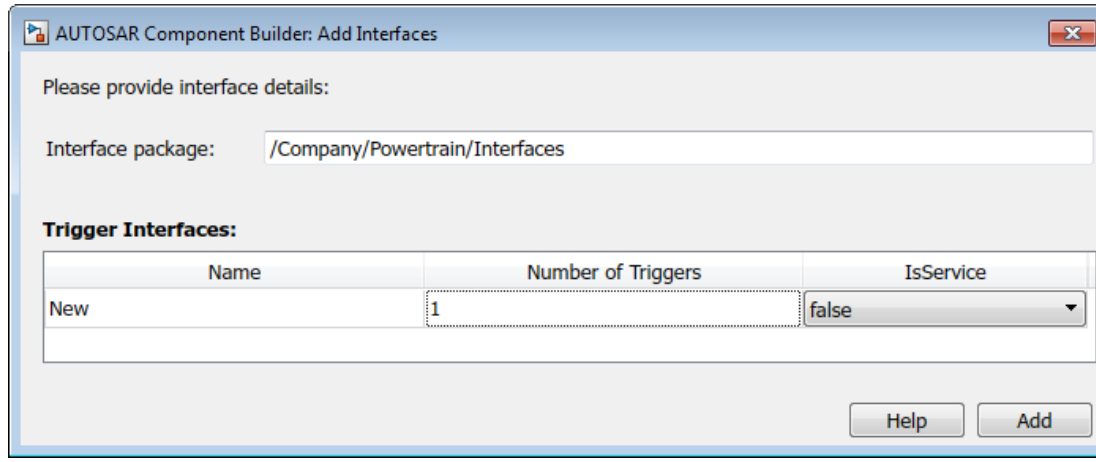
36 In the left-hand pane of the Configure AUTOSAR Interface dialog box, select **Trigger Interfaces**.

The Trigger Interfaces view of the AUTOSAR Properties Explorer 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** icon to open an Add Interfaces dialog box to add one or more trigger interfaces.
- Select a trigger interface and then click the **Delete** icon 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.



For more information, see “Configure Receiver for AUTOSAR External Trigger Event Communication” on page 4-148.

37 In the left-hand pane of the Configure AUTOSAR Interface dialog box, expand **Trigger Interfaces** and select a trigger interface from the list.

The trigger interface view of the AUTOSAR Properties Explorer 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 Interface or CompuMethod” on page 4-67.

For more information, see “Configure Receiver for AUTOSAR External Trigger Event Communication” on page 4-148.

38 In the left-hand pane of the Configure AUTOSAR Interface dialog box, expand the selected interface and select **Triggers**.

The Triggers view of the AUTOSAR Properties Explorer 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
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 if there is a new value

CseCode	Time Base
CSE1000	Nondeterministic (no fixed scaling)

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

For more information, see “Configure Receiver for AUTOSAR External Trigger Event Communication” on page 4-148.

39 In the left-hand pane of the Configure AUTOSAR Interface dialog box, select **CompuMethods**.

The CompuMethods view of the AUTOSAR Properties Explorer displays CompuMethods and their properties. You can:

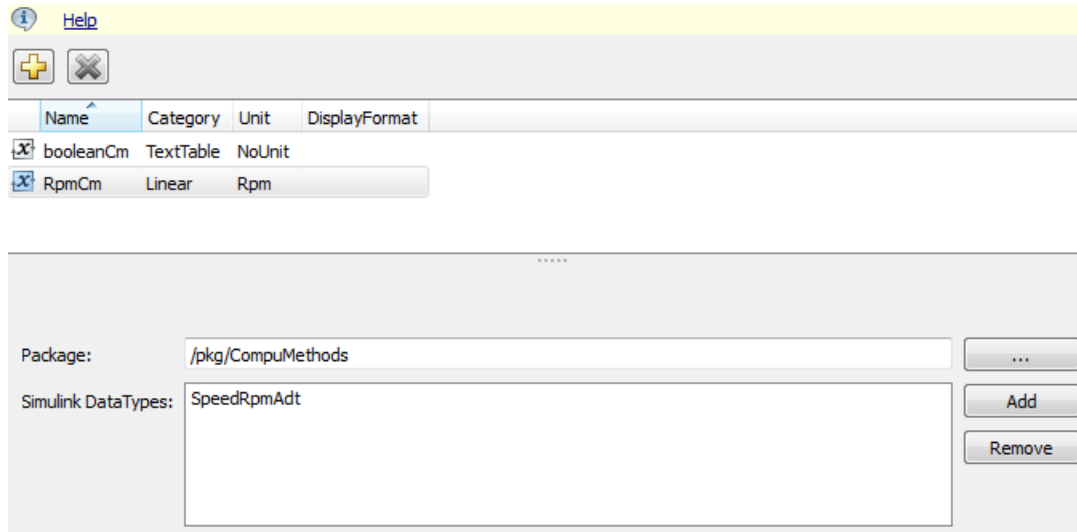
- Select a CompuMethod and modify properties, such as name, category, unit, display format for measurement and calibration, AUTOSAR package to be generated for the CompuMethod, and a list of Simulink data types that reference the CompuMethod. For property descriptions, see “Configure AUTOSAR CompuMethods” on page 4-241.
- Click the **Add** icon to open an Add CompuMethod dialog box to add a CompuMethod.
- Select a CompuMethod and then click the **Delete** icon to remove it.

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

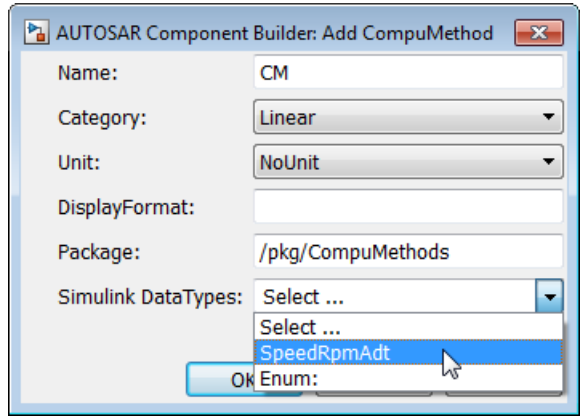
- Enter a package path in the **Package** parameter field.
- Click the button to the right of the **Package** field to open the AUTOSAR Package Browser. Use the browser to navigate to an existing package or create a new

package. When you select a package in the browser and click **Apply**, the interface **Package** parameter value is updated with your selection. For more information about the AUTOSAR Package Browser, see "Configure AUTOSAR Package for Interface or CompuMethod" on page 4-67.

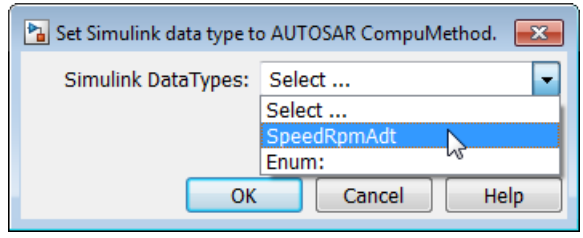
To associate a CompuMethod with a Simulink data type used in the model, select a CompuMethod and click the **Add** button to the right of **Simulink DataTypes**. This action opens a dialog box with a list of available data types. Select a data type and click OK to add it to the **Simulink DataTypes** 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.



The Set Simulink data type to AUTOSAR CompuMethod 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.



For more information, see "Configure AUTOSAR CompuMethods" on page 4-241.

40 In the left-hand pane of the Configure AUTOSAR Interface dialog box, select **XML Options**.

The XML Options view of the AUTOSAR Properties Explorer displays XML export parameters and their values. You can:

- Specify the granularity of file packaging for exported XML. 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 any elements for which AUTOSAR packages (AR-PACKAGEs) are not configured, and any 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 4-79.
modelname`_datatype.arxml`	Data types and related elements.
modelname`_implementation.arxml`	Software component implementation.
modelname`_interface.arxml`	Interfaces, including S-R, C-S, M-S, and NV interfaces.
modelname`_behavior.arxml`	Software component internal behavior (generated only for schema 3.x or earlier).

For more information, see "Export AUTOSAR Software Component" on page 5-6.

- 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 single-file or modular `arxml` file partitioning selected with the XML option **Exported XML file packaging**) Inspect and modify the AR-PACKAGE parameters grouped under the headings **Package Paths**, **Qualified Names**, and **Additional Packages**. For more information, see "Configure AUTOSAR Packages" on page 4-70.
- 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, set **ImplementationDataType Reference** to `NotAllowed`. For more information, see "Control Application Data Type Generation" on page 4-237.

- 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 4-191.

- Control the direction of `CompuMethod` conversion for linear-function `CompuMethod`s. 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 4-244.

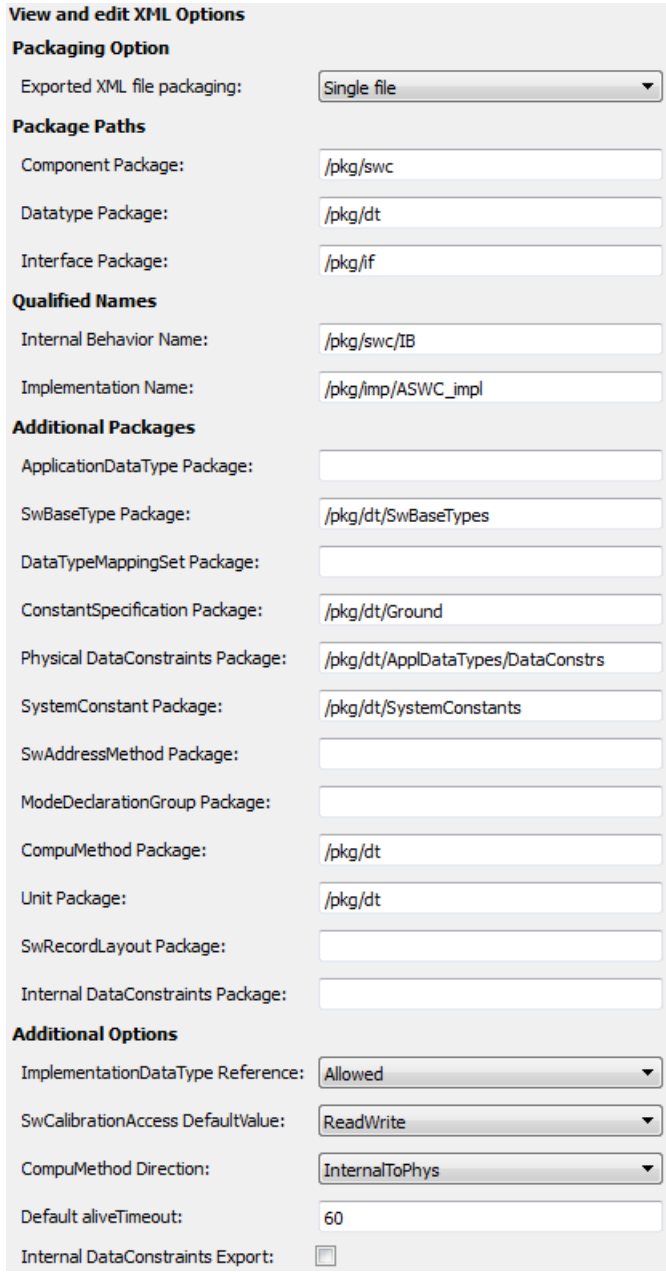
- If AUTOSAR ports in your model do not have an associated `aliveTimeout` value, use **Default aliveTimeout** to control the `aliveTimeout` value generated for the ports in subsequent `arxml` exports.

  In AUTOSAR applications, the `aliveTimeout` value for a port specifies the number of seconds after which the AUTOSAR software component is notified if the port has not received data. If you import `aliveTimeout` values for AUTOSAR ports from `arxml` files, the software preserves the imported values for export.

  If imported `arxml` code did not specify an `aliveTimeout` value for one or more AUTOSAR ports, or if you created and mapped AUTOSAR ports in Simulink, you can use **Default aliveTimeout**. By default, for an AUTOSAR port for which no imported `aliveTimeout` exists, the `arxml` exporter generates an `aliveTimeout` value of 60 seconds. You can specify a different positive quantity of seconds, or 0

not to monitor port timeouts. For more information, see “Configure AUTOSAR Port aliveTimeout Value” on page 4-86.

- 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 4-254.


View and edit XML Options
Packaging Option
Exported XML file packaging:
Single file
Package Paths
Component Package:
/pkg/swc
Datatype Package:
/pkg/dt
Interface Package:
/pkg/if
Qualified Names
Internal Behavior Name:
/pkg/swc/IB
Implementation Name:
/pkg/imp/ASWC_impl
Additional Packages
ApplicationDataType Package:
SwBaseType Package:
/pkg/dt/SwBaseTypes
DataTypeMappingSet Package:
ConstantSpecification Package:
/pkg/dt/Ground
Physical DataConstraints Package:
/pkg/dt/ApplDataTypes/DataConstrs
SystemConstant Package:
/pkg/dt/SystemConstants
SwAddressMethod Package:
ModeDeclarationGroup Package:
CompuMethod Package:
/pkg/dt
Unit Package:
/pkg/dt
SwRecordLayout Package:
Internal DataConstraints Package:
Additional Options
ImplementationDataType Reference:
Allowed
SwCalibrationAccess DefaultValue:
ReadWrite
CompuMethod Direction:
InternalToPhys
Default aliveTimeout:
60
Internal DataConstraints Export:

41 Click the **Validate** icon to validate the AUTOSAR interface configuration. If errors are reported, address them and then retry validation.

Related Examples

- "Map Model Elements Using Simulink-AUTOSAR Mapping Explorer" on page 4-60
- "Configure and Map AUTOSAR Component Programmatically" on page 4-267

More About

- "AUTOSAR Interface Configuration" on page 4-3

Map Model Elements Using Simulink-AUTOSAR Mapping Explorer

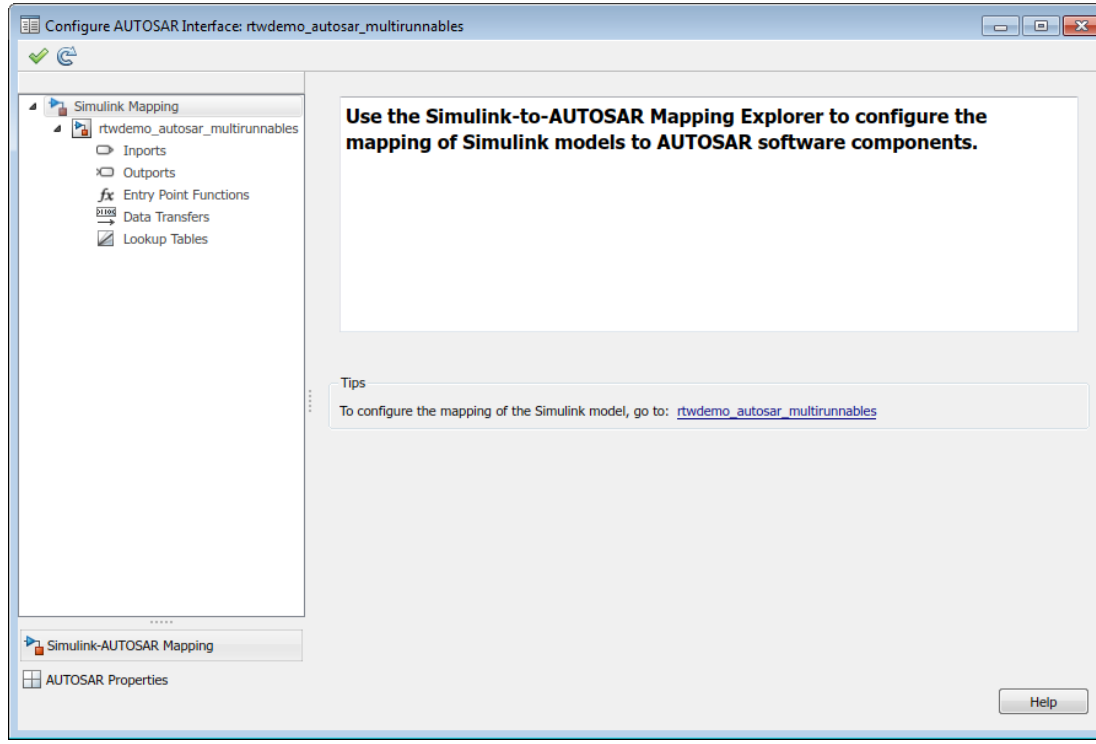
In Simulink, you can use the Configure AUTOSAR Interface dialog box to graphically configure an AUTOSAR software component and map Simulink model elements to AUTOSAR component elements. For more information, see "AUTOSAR Interface Configuration" on page 4-3.

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. Open the Configure AUTOSAR Interface dialog box by selecting **Code > C/C++ Code > Configure Model as AUTOSAR Component**. If the Simulink-AUTOSAR Mapping Explorer is not already selected, click **Simulink-AUTOSAR Mapping**. The model tree in the top-level view shows the types of Simulink elements that can be mapped to AUTOSAR component elements:

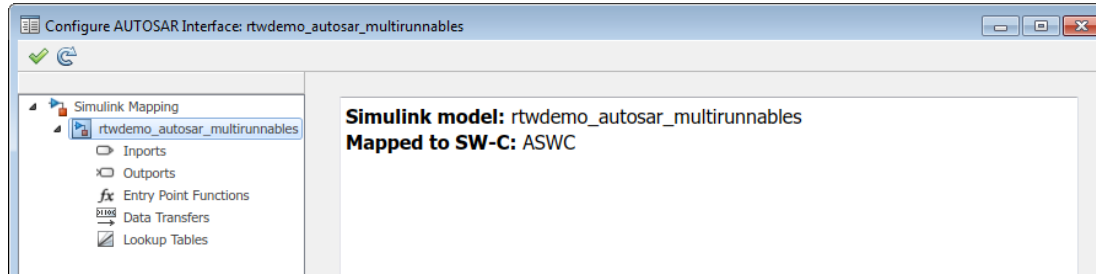
   - A Simulink model can be mapped to an AUTOSAR component.
   - A Simulink inport or outport can be mapped to a data element of an AUTOSAR port, with a specific data access mode.
   - A Simulink entry-point function can be mapped to an AUTOSAR runnable.
   - A Simulink data transfer line can be mapped to an AUTOSAR inter-runnable variable (IRV).
   - A Simulink lookup table can be mapped to an AUTOSAR parameter.

Note: Additionally, if the model contains Function Caller blocks in a client/server interface configuration, **Function Callers** appears in the model tree. A Simulink function caller can be mapped to an AUTOSAR client port and an AUTOSAR operation.



3 In the left-hand pane of the Configure AUTOSAR Interface dialog box, under **Simulink Mapping**, select the model name.

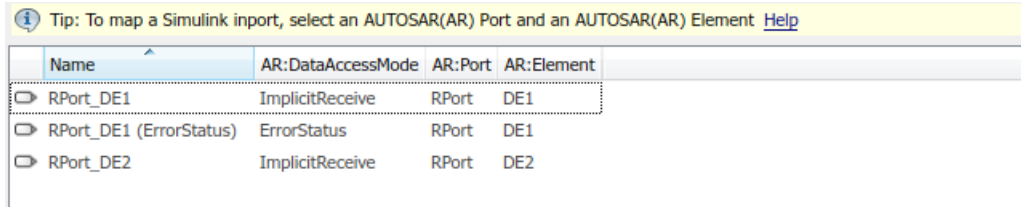
The model view of the Simulink-AUTOSAR Mapping Explorer displays the model name and the name of the AUTOSAR component to which the model is being mapped.



4 In the left-hand pane of the Configure AUTOSAR Interface dialog box, under the model name, select **Inports**.

The Inports view of the Simulink-AUTOSAR Mapping Explorer maps each Simulink inport to a data element of an AUTOSAR port. You can:

- Map a Simulink inport by selecting the inport and then selecting menu values for an AUTOSAR Port and an AUTOSAR Element.
- Select a port and then select a menu value for its AUTOSAR port data access mode: `ImplicitReceive`, `ExplicitReceive`, `QueuedExplicitReceive`, `ErrorStatus`, `ModeReceive`, `IsUpdated`, `EndToEndRead`, or `ExplicitReceiveByVal`.

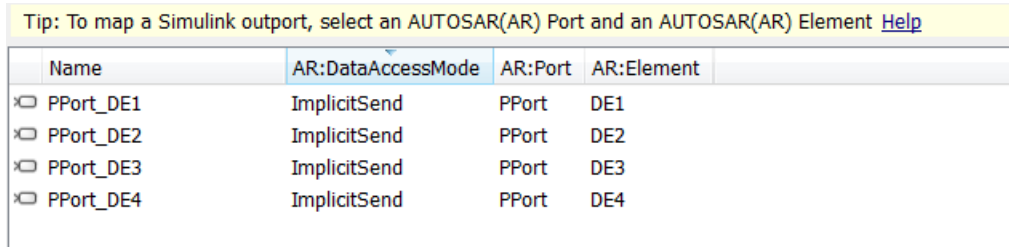


5 In the left-hand pane of the Configure AUTOSAR Interface dialog box, select **Outports**.

The Outports view of the Simulink-AUTOSAR Mapping Explorer maps each Simulink outport to a data element of an AUTOSAR port. You can:

- Map a Simulink outport by selecting the outport and then selecting menu values for an AUTOSAR Port and an AUTOSAR Element.

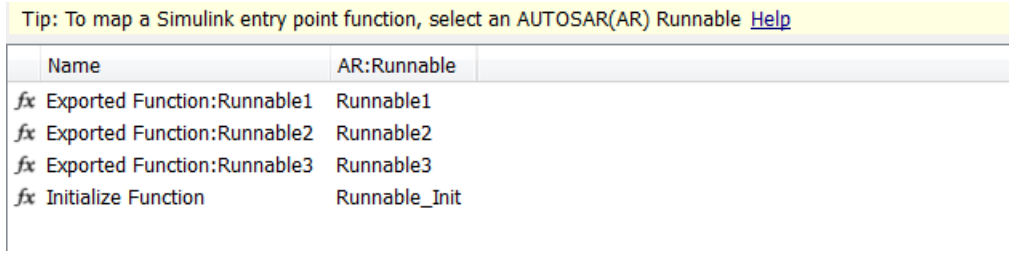
- Select a port and then select a menu value for its AUTOSAR port data access mode: `ImplicitSend`, `ImplicitSendByRef`, `ExplicitSend`, `EndToEndWrite`, or `ModeSend`.



6 In the left-hand pane of the Configure AUTOSAR Interface dialog box, select **Entry Point Functions**.

The Entry Point Functions view of the Simulink-AUTOSAR Mapping Explorer maps each Simulink entry-point function to an AUTOSAR runnable. Click the **Synchronize** icon to load or update Simulink entry-point functions in your model.

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.



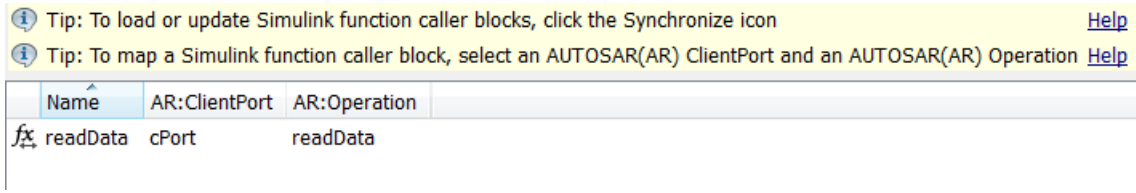
7 In the left-hand pane of the Configure AUTOSAR Interface dialog box, select **Function Callers**.

Note: **Function Callers** is available for selection only if the model contains Function Caller blocks in a client/server interface configuration. If **Function Callers** is absent, skip this step.

The Function Callers view of the Simulink-AUTOSAR Mapping Explorer maps each Simulink function caller to an AUTOSAR client port and an AUTOSAR operation.

Click the **Synchronize** icon to load or update Simulink function callers in your model.

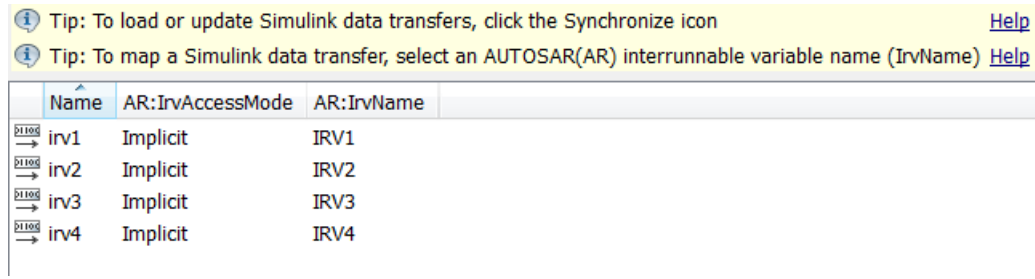
You can map a Simulink function caller by selecting the function name and then selecting menu values for a client port and an operation, among those listed for the AUTOSAR component.



8 In the left-hand pane of the Configure AUTOSAR Interface dialog box, select **Data Transfers**.

The Data Transfers view of the Simulink-AUTOSAR Mapping Explorer maps each Simulink data transfer line to an AUTOSAR inter-runnable variable (IRV). Click the **Synchronize** icon to load or update Simulink data transfers in your model.

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, among those listed for the AUTOSAR component.

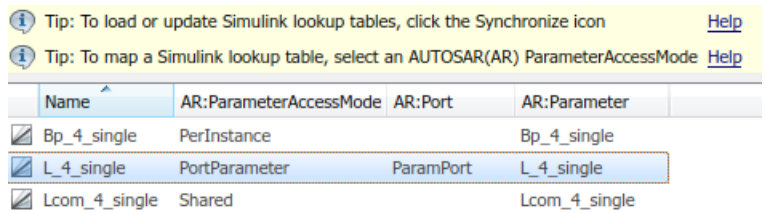


9 In the left-hand pane of the Configure AUTOSAR Interface dialog box, select **Lookup Tables**.

The Lookup Tables view of the Simulink-AUTOSAR Mapping Explorer maps each Simulink lookup table to an AUTOSAR parameter. Click the **Synchronize** icon to load or update Simulink lookup tables in your model.

You can map a Simulink lookup table by selecting a lookup table or breakpoint object name and then selecting menu values for an AUTOSAR parameter access mode (`PortParameter`, `Shared`, `PerInstance`, or `ConstantMemory`) and a parameter name.

If the AUTOSAR parameter is a parameter interface data element (access mode `PortParameter`), select a parameter receiver port and data element, among those listed for the AUTOSAR component. For more information, see "Configure AUTOSAR Port-Based Calibration Parameters" on page 4-169 and "Configure Receiver for AUTOSAR Parameter Communication" on page 4-144.



In this example:

- `Simulink.LookupTable` object `L_4_single` is mapped to an AUTOSAR port-based parameter of the same name.
- `Simulink.Breakpoint` object `Bp_4_single` is mapped to an AUTOSAR internal parameter of the same name. Each instance of the AUTOSAR software component has its own copy of the parameter.
- `Simulink.LookupTable` object `Lcom_4_single` is mapped to an AUTOSAR internal parameter of the same name. The parameter is shared by all instances of the AUTOSAR software component.

10 Click the **Validate** icon to validate the AUTOSAR interface configuration. If errors are reported, address them and then retry validation.

Related Examples

- "Configure AUTOSAR Component Using AUTOSAR Properties Explorer" on page 4-6
- "Configure and Map AUTOSAR Component Programmatically" on page 4-267

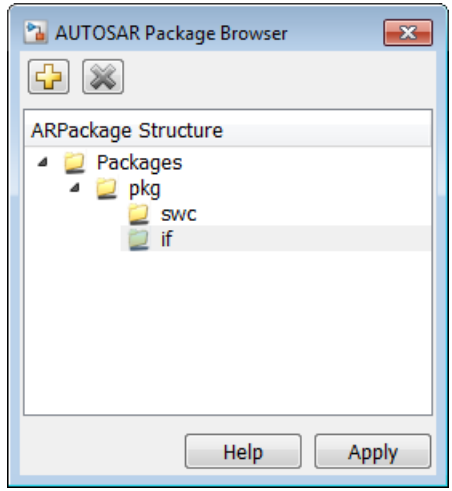
More About

- "AUTOSAR Interface Configuration" on page 4-3

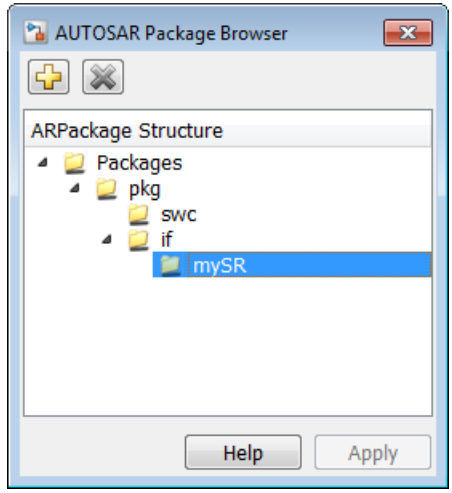
Configure AUTOSAR Package for Interface or CompuMethod

As part of configuring an AUTOSAR interface or an AUTOSAR CompuMethod, you specify the AUTOSAR package (AR-PACKAGE) to be generated for individual communication interfaces or individual CompuMethods in your configuration. For example, here is the AUTOSAR Properties Explorer view for an individual interface.

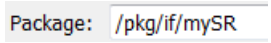
You can enter a package path in the **Package** parameter field, or use the AUTOSAR Package Browser to select a package. To open the browser, click the button to the right of the **Package** field. The AUTOSAR Package Browser opens.



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** icon . For example, to add a new interface package, select the `if` folder and click the **Add** icon. 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.



For more information about AR-PACKAGEs, see “Configure AUTOSAR Packages” on page 4-70.

Related Examples

- “Configure AUTOSAR Component Using AUTOSAR Properties Explorer” on page 4-6
- “Configure and Map AUTOSAR Component Programmatically” on page 4-267

More About

- “AUTOSAR Interface Configuration” on page 4-3

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 4-70
"Configure AUTOSAR Packages and Paths" on page 4-72
"Control AUTOSAR Elements Affected by Package Path Modifications" on page 4-75
"Export AUTOSAR Packages" on page 4-75
"AR-PACKAGE Location in Exported ARXML Files" on page 4-79

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.

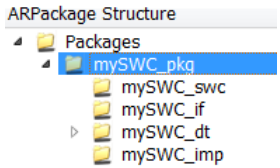
The `arxml` importer imports AR-PACKAGEs, their elements, and their paths into Simulink. The model configuration preserves the packages for subsequent export to `arxml` code. In general, the software preserves AUTOSAR packages across round-trips between an AAT and Simulink.

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`. Select the AUTOSAR system target file. Use default component creation to create an initial AUTOSAR configuration. When you build the model, its initial AR-PACKAGE structure resembles the following figure.



After component creation, you can use the XML Options view in the Configure AUTOSAR Interface dialog box to specify additional AR-PACKAGEs. 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)

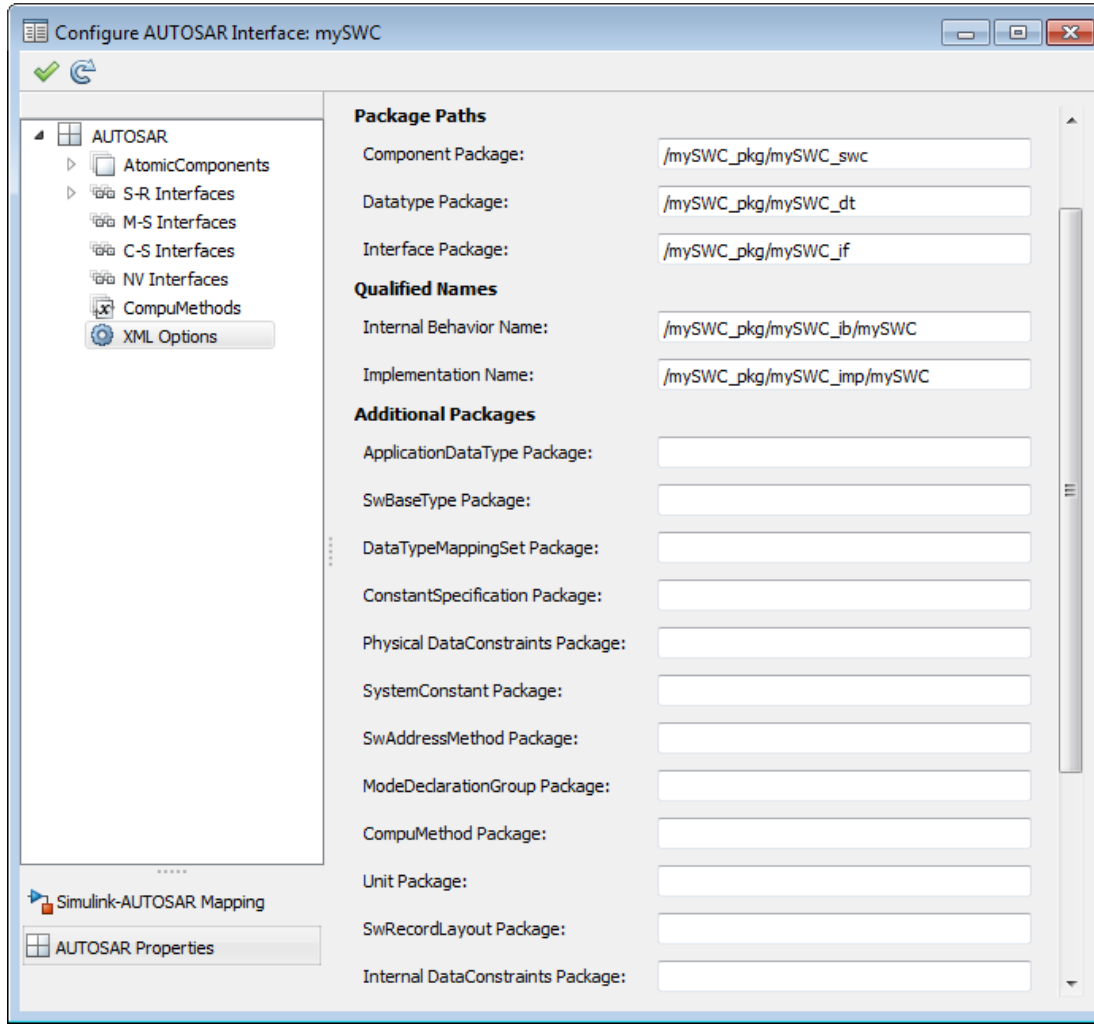
Note: For packages that you define in XML options, the `arxml` exporter generates a package only if the model contains an element of the package category. For example, the exporter generates a software address method package only if the model contains a software address method element.

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 4-79.

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 XML Options view of the Configure AUTOSAR Interface dialog box. 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, select the AUTOSAR system target file, and use default component creation to create an initial AUTOSAR configuration. The AUTOSAR component builder creates an initial default AR-PACKAGE structure. After component creation, the XML Options view in the Configure AUTOSAR Interface dialog box resembles the following figure.



Using the **Additional Packages** subpane, you can populate the additional package fields with paths 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
ComponentQualifiedName	*modelname*_pkg/*modelname*_swc/*modelname* (dialog box displays the component path without the shortname)
DataTypePackage	*modelname*_pkg/*modelname*_dt
InterfacePackage	*modelname*_pkg/*modelname*_if
InternalBehaviorQualifiedName	*modelname*_pkg/*modelname*_ib/*modelname*
ImplementationQualifiedName	*modelname*_pkg/*modelname*_imp/*modelname*
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:

```
open_system('rtwdemo_autosar_counter');
arProps=autosar.api.getAUTOSARProperties('rtwdemo_autosar_counter');
set(arProps,'XmlOptions','ApplicationDataTypePackage','/rtwdemo_autosar_counter_pkg/ADTs');
get(arProps,'XmlOptions','ApplicationDataTypePackage')
```

For a sample script, see “Configure AUTOSAR XML Export” on page 4-293.

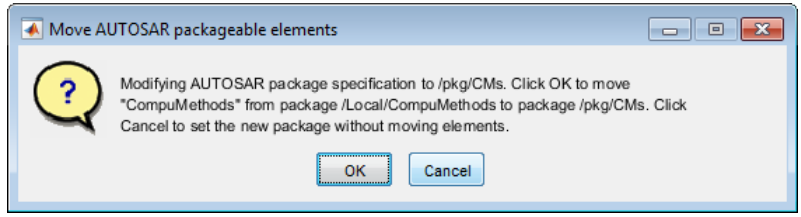
For an example of configuring and exporting AUTOSAR packages, see “Export AUTOSAR Packages” on page 4-75.

Control AUTOSAR Elements Affected by Package Path Modifications

If you modify an AUTOSAR package path, and if packageable elements of that category are affected, you can:

- Move the elements from the existing package to the new package.
- Set the new package path without moving the elements.

If you modify a package path in the Configure AUTOSAR Interface dialog box, 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.

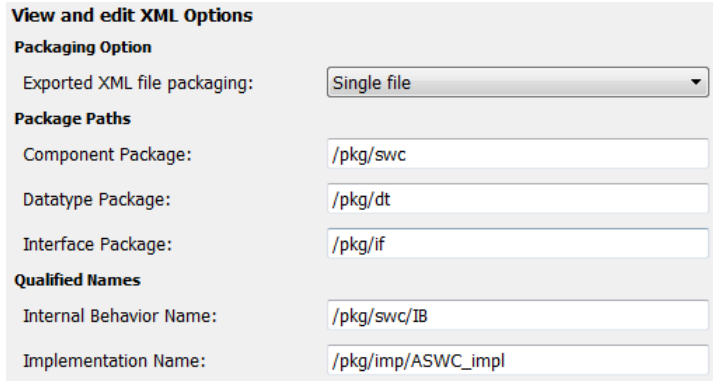
```
arProps = autosar.api.getAUTOSARProperties('cmSpeed');
set(arProps,'XmlOptions','CompuMethodPackage','/pkg/CMs','MoveElements','None');
```

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 `rtwdemo_autosar_multirunnables`.

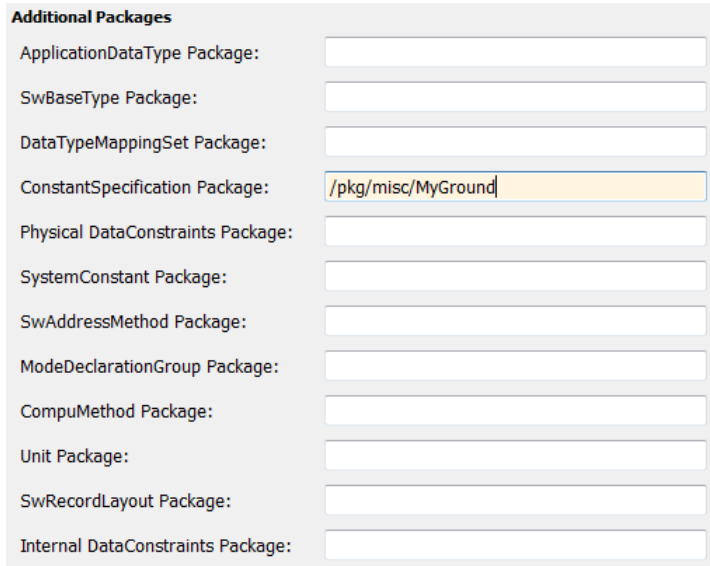
2 Open the Configure AUTOSAR Interface dialog box, select **AUTOSAR Properties**, and select **XML Options**. Initially, only the first five AUTOSAR package parameters are configured.



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**.

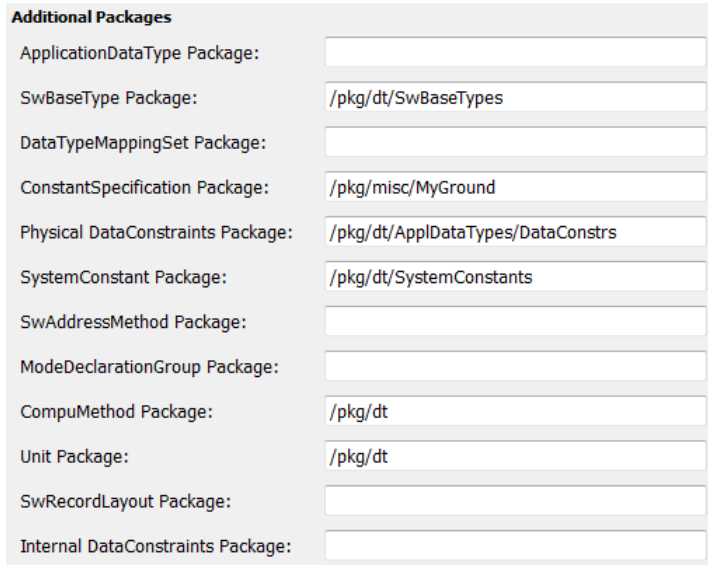
The example model exports multiple AUTOSAR constant specifications. This example sets the **ConstantSpecification Package** parameter to `/pkg/misc/MyGround`. (This value overrides the rule-based default, `/pkg/dt/Ground`.)



4. Validate the AUTOSAR interface configuration. In the upper-left corner of the dialog box, click the **Validate** icon .
5. Generate code for the model.
6. Open the generated file *`modelname`*`.arxml`. (If you set **Exported XML file packaging** to `Modular`, open the generated file *`modelname`*`_component.arxml`.)
7. Search the XML code for the packages that you configured, for example, using the text AR-PACKAGE. For the example model, searching `rtwdemo_autosar_multirunnables.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>
```

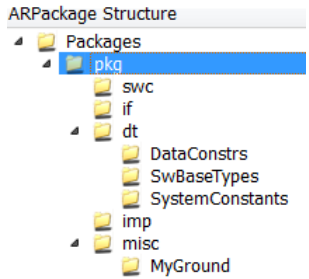
8 During the code generation that you initiated in a previous step, the `arxml` exporter filled in other package paths based on internal rules. Return to the XML Options view in the Configure AUTOSAR Interface dialog box. The display now includes five additional package paths.



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 `rtwdemo_autosar_multirunnables` with the following AR-PACKAGE structure. (See the steps in "Export AUTOSAR Packages" on page 4-75). 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.

`rtwdemo_autosar_multirunnables.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, `rtwdemo_autosar_multirunnables_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/`.

rtwdemo_autosar_multirunnables_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>
```

rtwdemo_autosar_multirunnables_interface.arxml:

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

rtwdemo_autosar_multirunnables_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>
```

rtwdemo_autosar_multirunnables_implementation.arxml:

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

Related Examples

- “Import AUTOSAR Software Component” on page 3-4
- “Configure AUTOSAR Package for Interface or CompuMethod” on page 4-67

- "Configure AUTOSAR XML Export" on page 4-293
- "Export AUTOSAR Component XML and C Code" on page 5-2

More About

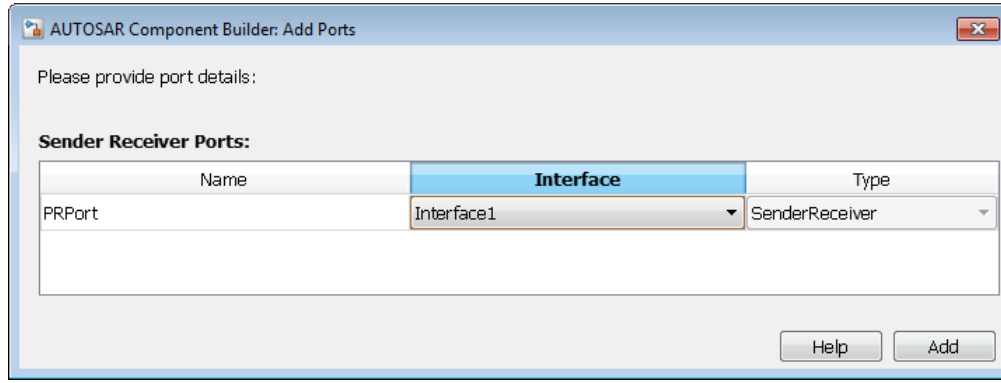
- "AUTOSAR Interface Configuration" on page 4-3

Configure AUTOSAR Provide-Require Port

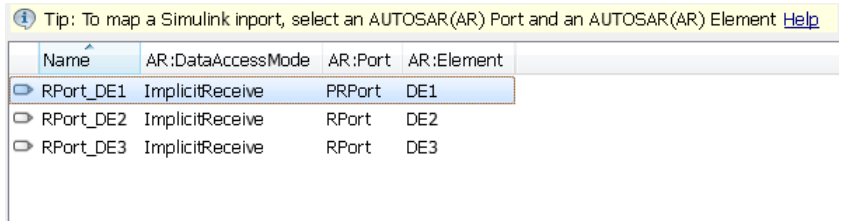
AUTOSAR Release 4.1 introduced the AUTOSAR provide-require port (`PRPort`). Modeling an AUTOSAR `PRPort` involves using a Simulink inport and outport pair with matching data type, dimension, and signal type. You can associate a `PRPort` with a sender-receiver (S-R) interface or a nonvolatile data (NV)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

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

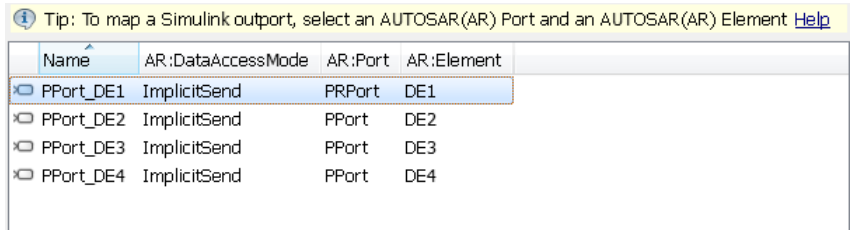
1. Open a model that is configured for AUTOSAR, and in which a runnable has an inport and an outport suitable for pairing into an AUTOSAR `PRPort`. In this example, the `RPort_DE1` inport and `PPort_DE1` outport both use data type `int8`, port dimension 1, and signal type `real`.
2. Open the Configure AUTOSAR Interface dialog box, and navigate to the SenderReceiverPorts view of the AUTOSAR Properties Explorer. (To configure a `PRPort` for NV communication, use the NvSenderReceiverPorts view instead.)
3. To add a sender-receiver port, click the **Add** icon . In the Add Ports dialog box, specify **Name** as `PRPort` and specify **Interface** as `Interface1`. Click **Add**.



4. Select the Simulink-AUTOSAR Mapping Explorer and navigate to the Inports view. To map a Simulink inport to the AUTOSAR sender-receiver port you created, select the inport, set **AR:Port** to the value `PRPort`, and set **AR:Element** to a data element that the import and outport will share.



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



6 Click the **Validate** icon to validate the updated AUTOSAR interface 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.

```
open_system('my_autosar_multirunnables')
arProps = autosar.api.getAUTOSARProperties('my_autosar_multirunnables');
swcPath = find(arProps,[],'AtomicComponent')

swcPath =
    'ASWC

add(arProps,'ASWC','SenderReceiverPorts','PRPort','Interface','Interface1')
prportPath = find(arProps,[],'DataSenderReceiverPort')

prportPath =
    'ASWC/PRPort'

slMap = autosar.api.getSimulinkMapping('my_autosar_multirunnables');
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
```

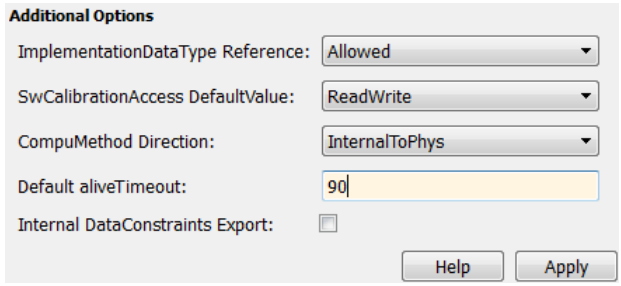
Configure AUTOSAR Port aliveTimeout Value

In AUTOSAR applications, the `aliveTimeout` value for a port specifies the number of seconds after which the AUTOSAR software component is notified if the port has not received data. If you import `aliveTimeout` values for AUTOSAR ports from `arxml` files, the software preserves the imported values for export.

If imported `arxml` code did not specify an `aliveTimeout` value for one or more AUTOSAR ports, or if you created and mapped AUTOSAR ports in Simulink, you can use the XML option **Default aliveTimeout** to control the `aliveTimeout` value generated for the ports in subsequent `arxml` exports.

By default, for an AUTOSAR port for which no imported `aliveTimeout` value exists, `arxml` export generates an `aliveTimeout` value of 60 seconds. You can specify a different positive quantity of seconds, or 0 not to monitor port timeouts.

To configure the `aliveTimeout` value in your model, open the Configure AUTOSAR Interface dialog box, select **AUTOSAR Properties**, and select **XML Options**. The **Default aliveTimeout** parameter is located under **Additional Options** and has an initial default of 60. Specify a positive integer or 0, and click **Apply**.



To set an `aliveTimeout` value programmatically, use commands similar to the following:

```
arProps = autosar.api.getAUTOSARProperties(bdroot);
set(arProps,'XmlOptions','DefaultAliveTimeout',90);
```

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 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.

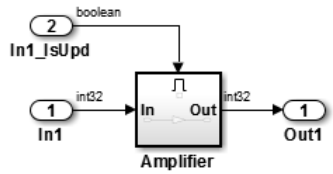
For the sender, the AUTOSAR run-time environment (RTE) sets the status of an update flag, indicating whether the data element has been written. The receiver calls the `Rte_IsUpdated_`*`Port_Element`* API, which reads and then clears the update flag.

In Simulink, you can:

- Import an AUTOSAR receiver port for which `IsUpdated` service is configured.
- Configure an AUTOSAR receiver port for `IsUpdated` service.
- Generate C and `arxml` code for an AUTOSAR receiver port for which `IsUpdated` service is configured.

To model `IsUpdated` service in Simulink, you pair an inport that is configured for `ExplicitReceive` data access with a new inport configured for `IsUpdated` data access. To configure an AUTOSAR receiver port for `IsUpdated` service:

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

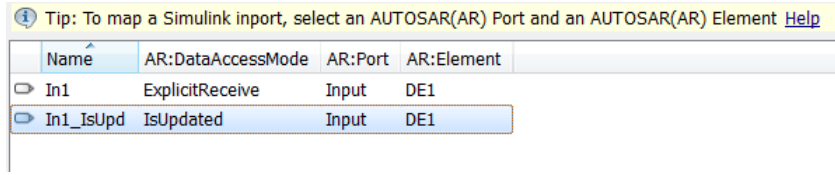


3. Open the Configure AUTOSAR Interface dialog box. Select **Code** > **C/C++ Code** > **Configure Model as AUTOSAR Component**.
4. Select **Simulink-AUTOSAR Mapping** and select **Inports**.
5. In the Inports view, configure the mapping properties for both inports.

    a. If the data inport is not already configured, set **AR:DataAccessMode** to `ExplicitReceive`. Select **AR:Port** and **AR: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 **AR:DataAccessMode** to `IsUpdated`. Select **AR:Port** and **AR:Element** values that exactly match the data inport.



6 Validate the AUTOSAR interface configuration. In the upper-left corner of the dialog box, click the **Validate** icon .

7 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>
```

Related Examples

- “Model AUTOSAR Communication” on page 2-11
- “Import AUTOSAR Software Component” on page 3-4
- “Export AUTOSAR Component XML and C Code” on page 5-2

More About

- "AUTOSAR Interface Configuration" on page 4-3

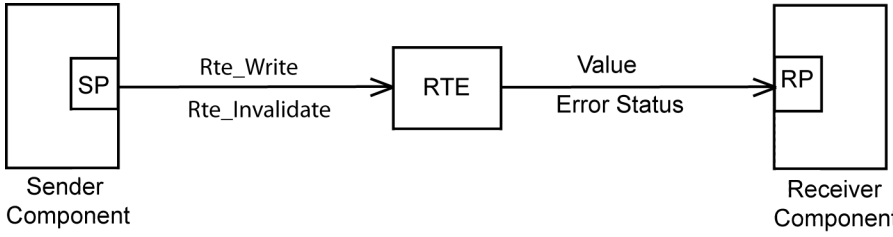
Configure AUTOSAR Sender Port for Data Element Invalidation

AUTOSAR defines an invalidation mechanism for data elements on AUTOSAR sender ports. A sender component can notify a downstream receiver component that data in a sender port is invalid. Each data element in a sender-receiver interface can optionally have an invalidation policy. In Simulink, you can:

- Import AUTOSAR sender port data elements for which an invalidation policy is configured.
- Select AUTOSAR sender port data elements and configure an invalidation policy.
- Generate C and `arxml` code for AUTOSAR sender port data elements for which an invalidation policy is configured.

For each sender port data element, you can set the **InvalidationPolicy** property to `Keep`, `Replace`, `DontInvalidate`, or `None`. When sender port data is invalid, the sender component notifies the AUTOSAR run-time environment (RTE). If sender port data is invalid and the data element **InvalidationPolicy** is `Keep` or `Replace`, the RTE provides a different value and an error status to the receiver port.

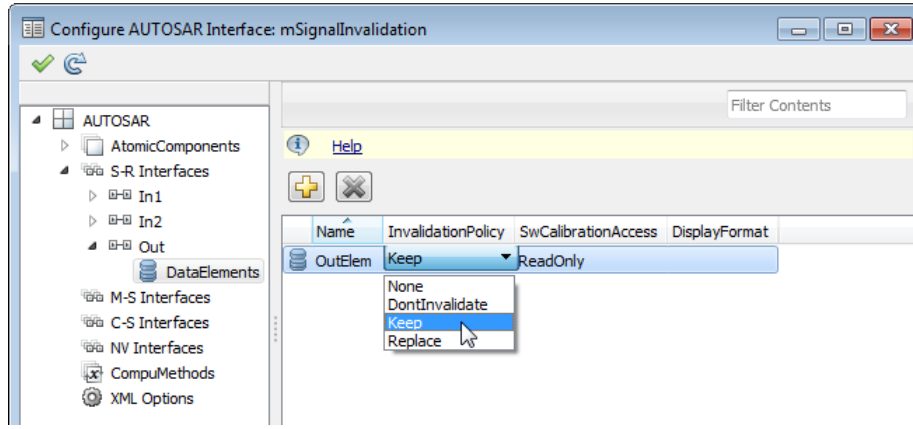
Invalidation Policy	Value	Error Status
`Keep`	Previous good value	RTE_E_INVALID
`Replace`	Initial value	RTE_E_OK



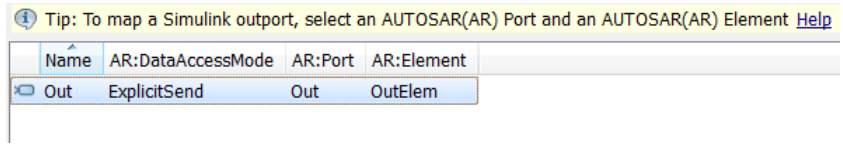
Note: The **InvalidationPolicy** values `DontInvalidate` and `None` indicate no invalidation policy. `DontInvalidate` causes the ARXML exporter to generate `INVALIDATION-POLICY` tagging with the property `HANDLE-INVALID` set to `DONT-INVALIDATE`. `None` does not generate `INVALIDATION-POLICY` tagging.

To configure an AUTOSAR sender port for a data element invalidation policy in Simulink:

1. Open a model for which an AUTOSAR sender-receiver interface is configured.
2. Open the Configure AUTOSAR Interface dialog box. Select **Code** > **C/C++ Code** > **Configure Model as AUTOSAR Component**.
3. Select **AUTOSAR Properties** and expand **S-R Interfaces**.
4. Navigate to the AUTOSAR data element for which you want to configure an invalidation policy. From the **S-R Interfaces** list, select and expand the S-R interface that the AUTOSAR sender port maps. Select **Data Elements**.
5. Select the data element. Click the **InvalidationPolicy** field and select an invalidation policy, either `Keep` or `Replace`.



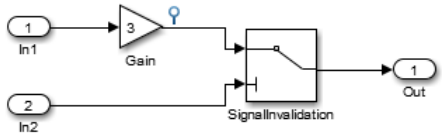
6. Select **Simulink-AUTOSAR Mapping** and select **Outports**.
7. In the Outports view, select the Simulink outport mapped to the AUTOSAR sender port and data element. Verify that its data access mode is set to `ExplicitSend` or `EndToEndWrite`.



8. Add a `Signal Invalidation` block to the model.

a Connect the block to the root outport that corresponds to the AUTOSAR sender data element that you configured.

b Connect the first block input to the outport source data. Connect the second block input to a scalar Boolean indicating the invalidation condition. When the second input, invalidation control, is false, the signal is valid. The Signal Invalidation block passes through the first input, the data input. When the second input is true, the signal is invalid. The block keeps the previous good value.



9 Validate the AUTOSAR interface configuration. In the upper-left corner of the dialog box, click the **Validate** icon .

10 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>/SignalInvalidation' incorporates:
 *  Inport: '<Root>/In2'
 */
if (!Rte_IRead_Runnable_Step_In2_In2()) {
  /* Outport: '<Root>/Out' */
  Rte_Write_Out_OutElem(mSignalInvalidation_B.Gain);
} else {
  Rte_Invalidate_Out_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>
```

See Also

`Signal Invalidation`

Related Examples

- “Model AUTOSAR Communication” on page 2-11

- "Import AUTOSAR Software Component" on page 3-4
- "Export AUTOSAR Component XML and C Code" on page 5-2

More About

- "AUTOSAR Interface Configuration" on page 4-3

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:

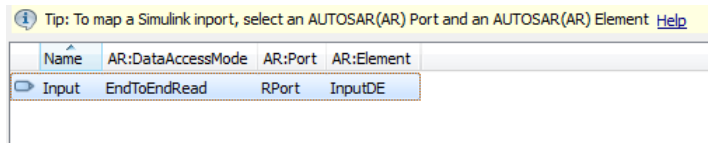
- Simulation behavior is unaffected.
- Code generation is very similar to explicit write and read, with these differences:
  - E2E initialization wrapper API calls appear in C initialization code.
  - E2E protection wrapper API calls appear in C step code.
  - When combined with an error status inport,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

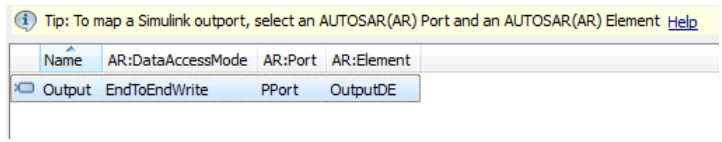
To configure an AUTOSAR sender or receiver port for E2E protection:

1. Open a model for which an AUTOSAR sender-receiver interface is configured.
2. Open the Configure AUTOSAR Interface dialog box. Select **Code** > **C/C++ Code** > **Configure Model as AUTOSAR Component**.
3. Select **Simulink-AUTOSAR Mapping**.

4 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.

5 Set the AUTOSAR data access mode to `EndToEndRead` (inport) or `EndToEndWrite` (outport).





6 Validate the AUTOSAR interface configuration. In the upper-left corner of the dialog box, click the **Validate** icon .

7 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'... */
  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>
…
```

Related Examples

- "Model AUTOSAR Communication" on page 2-11
- "Import AUTOSAR Software Component" on page 3-4
- "Export AUTOSAR Component XML and C Code" on page 5-2

More About

- "AUTOSAR Interface Configuration" on page 4-3

Configure AUTOSAR Receiver Port for DataReceiveErrorEvent

In AUTOSAR sender-receiver communication between software components, the run-time 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.

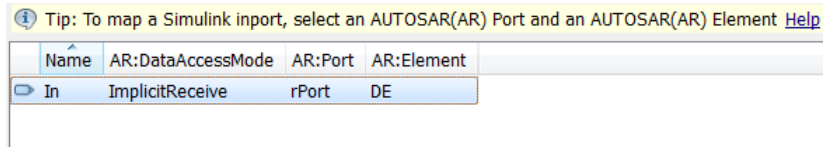
Embedded Coder supports creating `DataReceiveErrorEvent`s in AUTOSAR receiver components. In Simulink, you can:

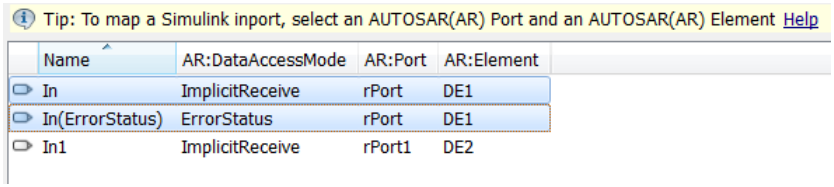
- Import an AUTOSAR `DataReceiveErrorEvent` definition.
- Define a `DataReceiveErrorEvent`.
- Generate `arxml` code for AUTOSAR receiver ports for which a `DataReceiveErrorEvent` is configured.

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 the Configure AUTOSAR Interface dialog box. Select **Code** > **C/C++ Code** > **Configure Model as AUTOSAR Component**.
3. Select **Simulink-AUTOSAR Mapping** and select **Inports**.
4. Select the data inport that is mapped to the AUTOSAR receiver port for which you want to configure a `DataReceiveErrorEvent`. Set its **AR:DataAccessMode** to `ImplicitReceive`, `ExplicitReceive`, or `EndToEndRead`. Here are two examples.





5 Select **AUTOSAR Properties** and expand **AtomicComponents**. Expand the receiver component and select **Runnables**.

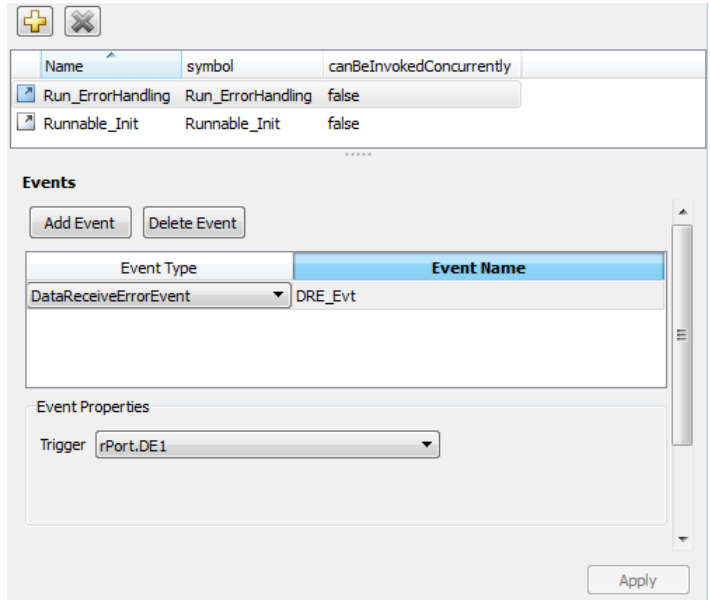
6 In the Runnables view, create a runnable to handle `DataReceiveErrorEvents`.

a Click the **Add** icon 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);
```

7 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>
        <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>
        <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>
```

Related Examples

- “Model AUTOSAR Communication” on page 2-11
- “Import AUTOSAR Software Component” on page 3-4
- “Export AUTOSAR Component XML and C Code” on page 5-2

More About

- “AUTOSAR Interface 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 2-13.

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 4-101
“Configure AUTOSAR Client” on page 4-112
“Configure AUTOSAR Client-Server Error Handling” on page 4-120
“Concurrency Constraints for AUTOSAR Server Runnables” on page 4-126
“Configure and Map AUTOSAR Server and Client Programmatically” on page 4-128

Configure AUTOSAR Server

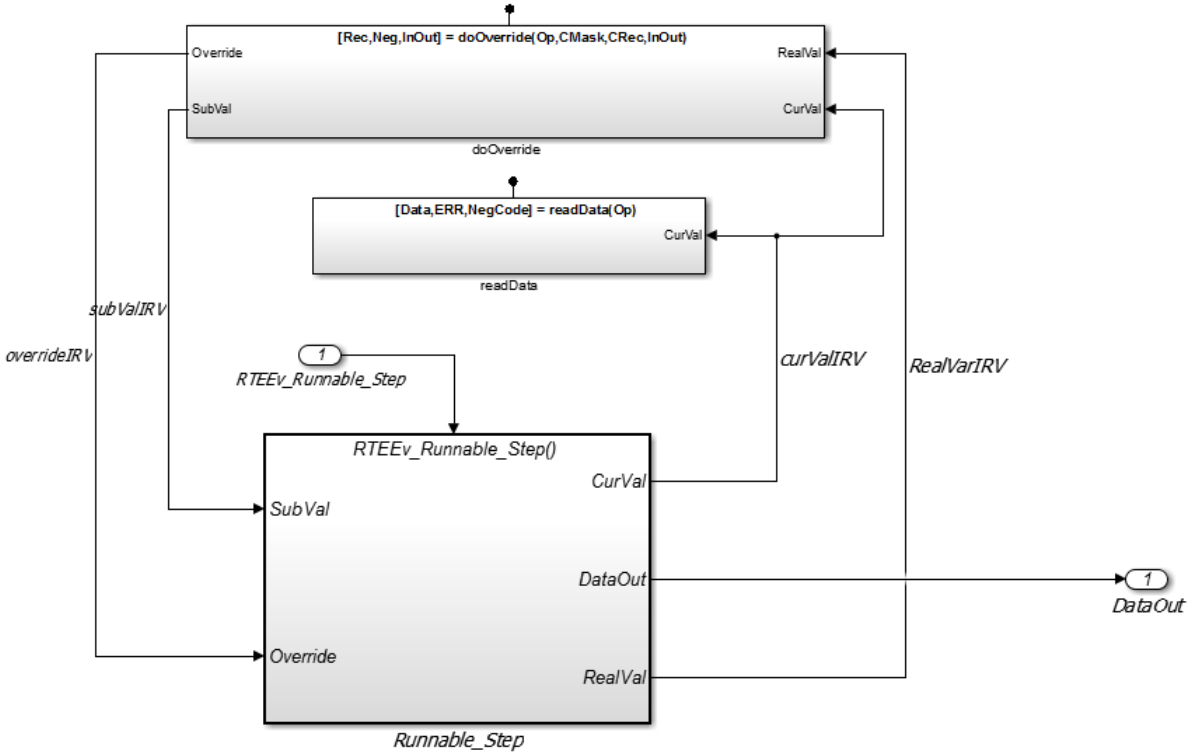
This example shows how to configure a `Simulink Function` block as an AUTOSAR server. The example uses these files in the folder *`matlabroot`*`/help/toolbox/ecoder/examples/autosar` (open):

- `mControllerWithInterface_server.slx`
- `ExampleApplicationErrorType.m`

If you copy the files to a working folder, collocate the MATLAB file with the model file.

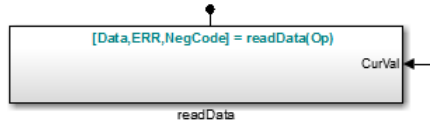
1. Open a model in which you want to create and configure an AUTOSAR server, or open the example model `mControllerWithInterface_server.slx`.
2. Add a Simulink Function block to the model. In the Simulink Library Browser, the Simulink Function block is in **User-Defined Functions**.

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`.

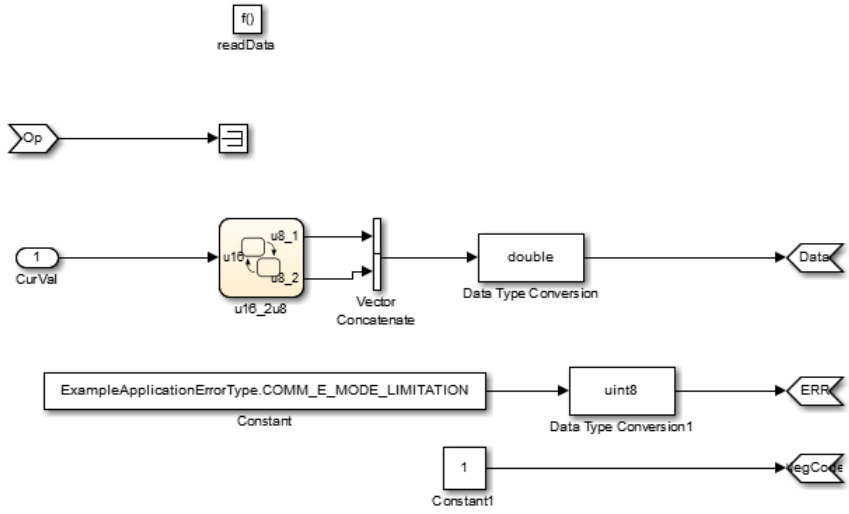


The contents include:

- `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)`.

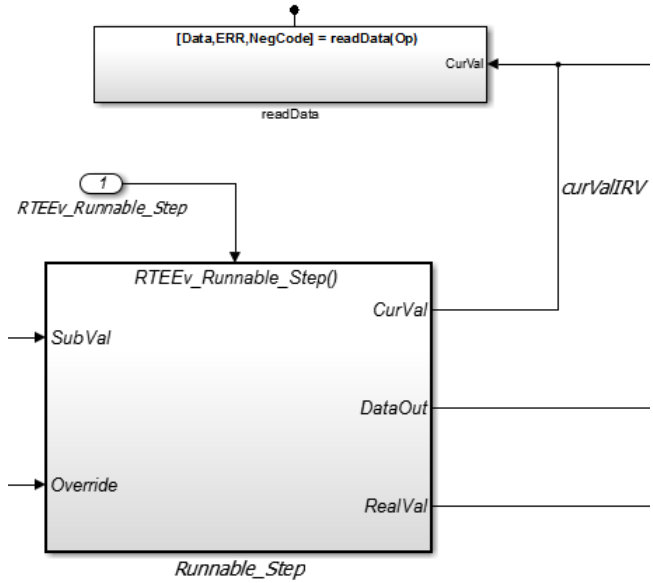
Note: When configuring server function arguments, you must specify signal data type, port dimensions, and signal type on the **Signal Attributes** tab of the inport and outport blocks. The AUTOSAR configuration fails validation if signal attributes are absent for server function arguments.

- 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.

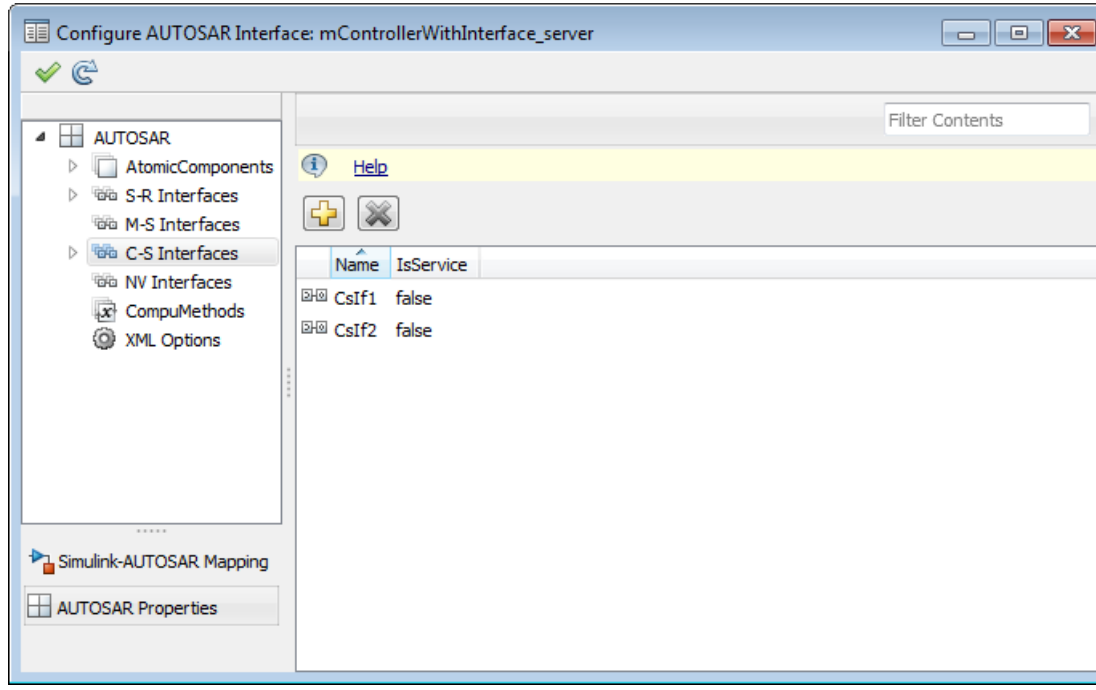


5 The required elements to configure an AUTOSAR server, in the general order they are created, are:

- 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

To view the AUTOSAR configuration for the model, in the model window, select **Code > C/C++ Code > Configure Model as AUTOSAR Component**. This action opens the Configure AUTOSAR Interface dialog box.

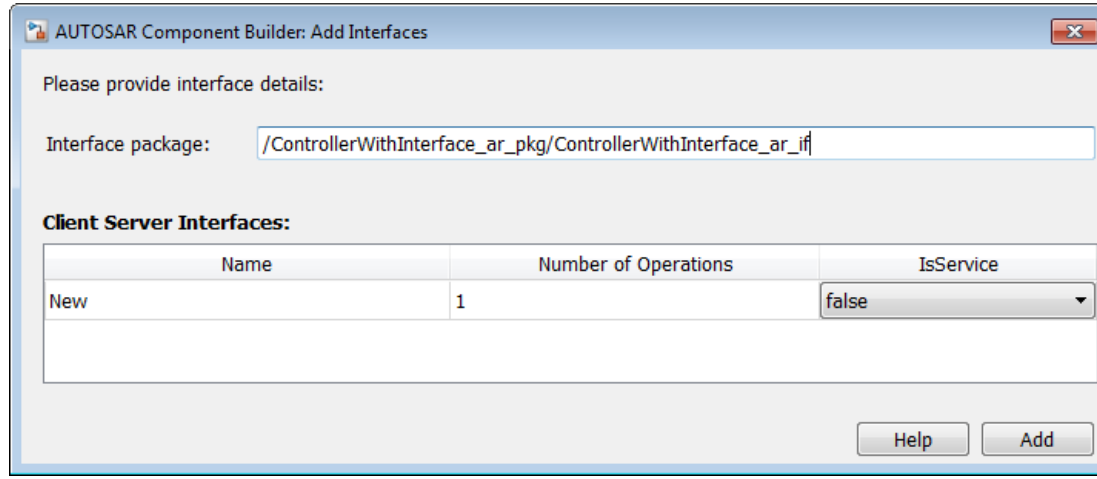
To view AUTOSAR C-S interfaces in the model, go to the C-S Interfaces view. Select **AUTOSAR Properties** and then **C-S Interfaces**. The example model already contains client-server interfaces.



If a C-S interface does not yet exist in your model, create one.

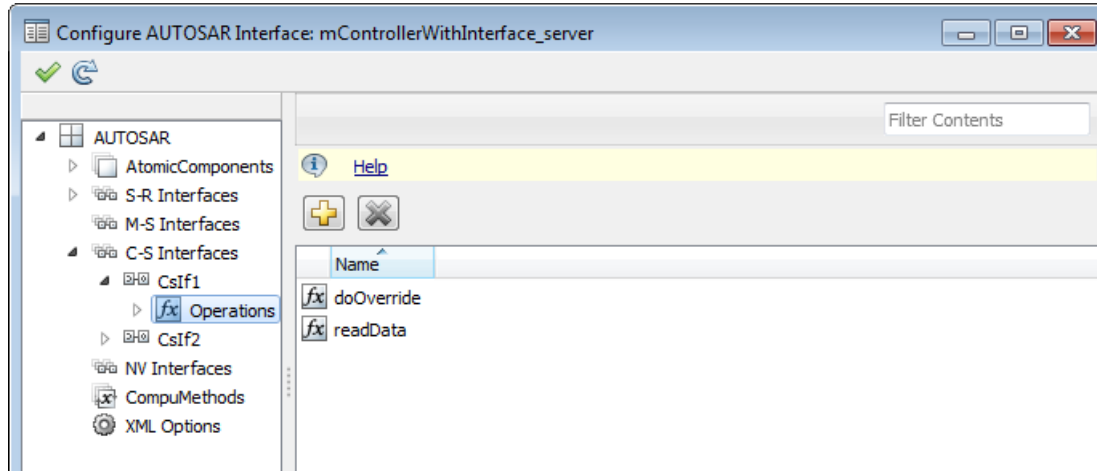
a In the C-S Interfaces view, click the **Add** icon . This action opens the Add Interfaces dialog box.

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`.

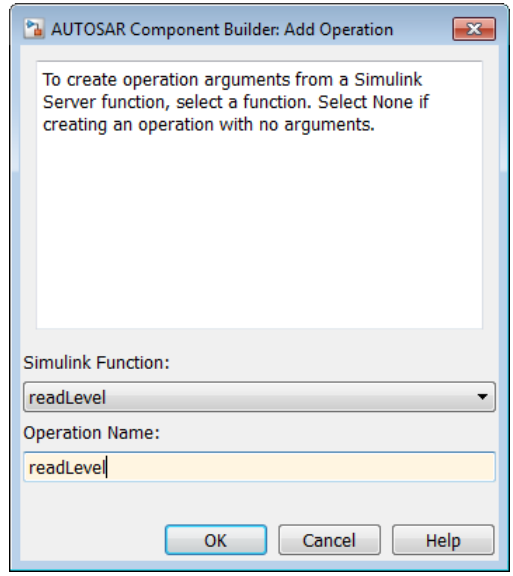


If a server operation does not yet exist in your model, create one. (If there is a placeholder operation named `Operation1` in your C-S interface, you can safely delete it.)

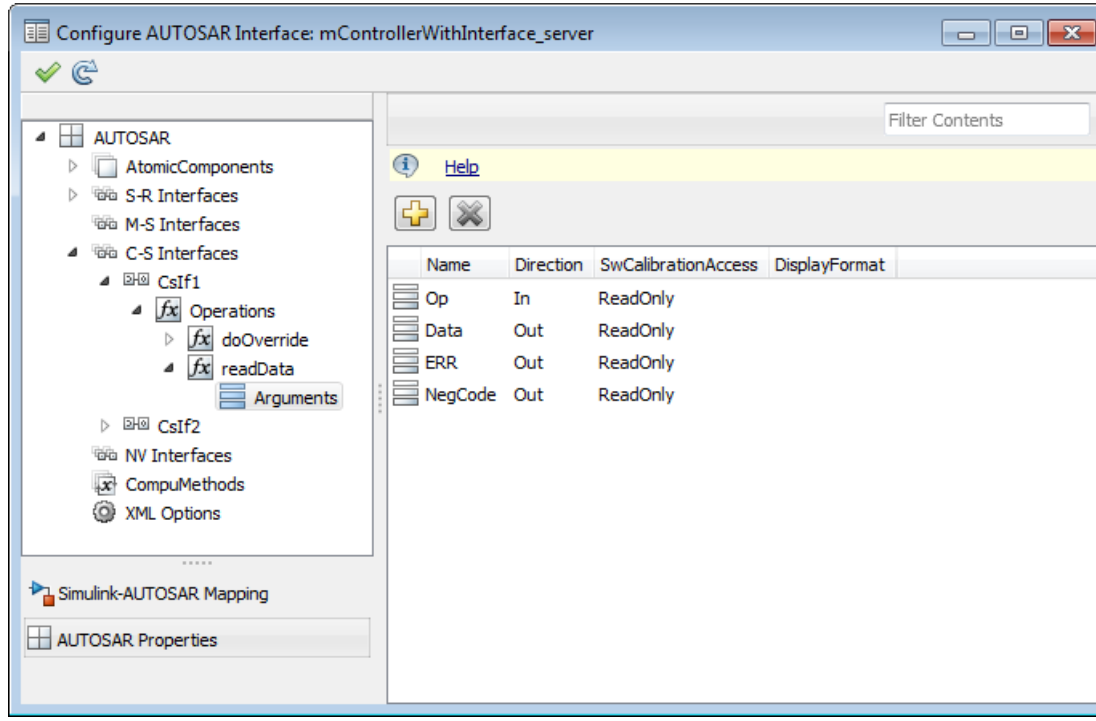
a In the Operations view, click the **Add** icon . This action opens the Add Operation dialog box.

b In the dialog box, enter the **Operation Name**. Specify the name of the corresponding Simulink server function.

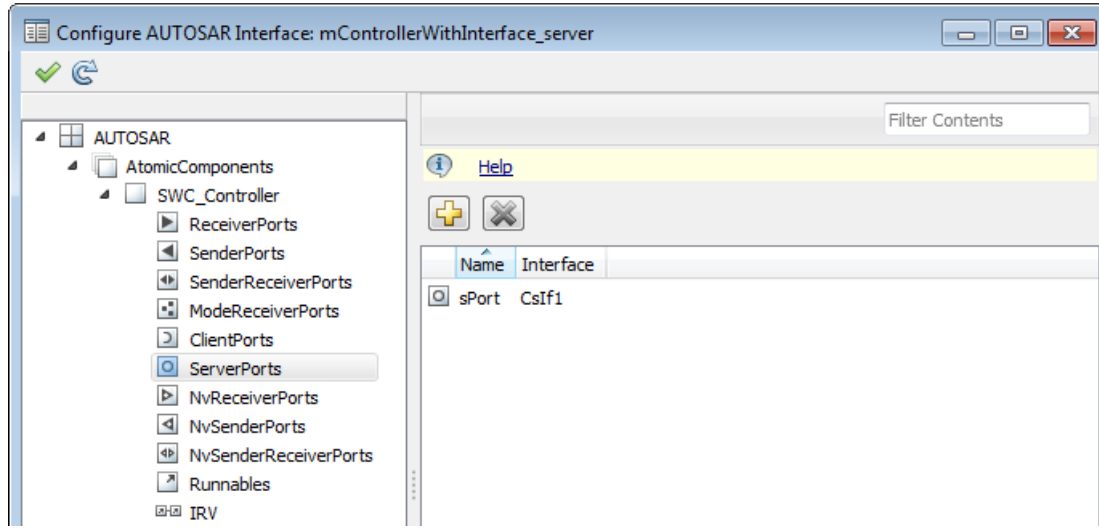
c If the corresponding Simulink server function has arguments, select the function in the **Simulink Function** list. This action causes AUTOSAR operation arguments to be automatically created based on the Simulink server function arguments. Click **OK**. The operation and its arguments appear in the Operations view.



7 Examine the arguments listed for the AUTOSAR server operation. In the Operations view, expand **Operations**, expand the individual operation (for example, `readData`), and select **Arguments**. The listed arguments correspond to the Simulink server function prototype.



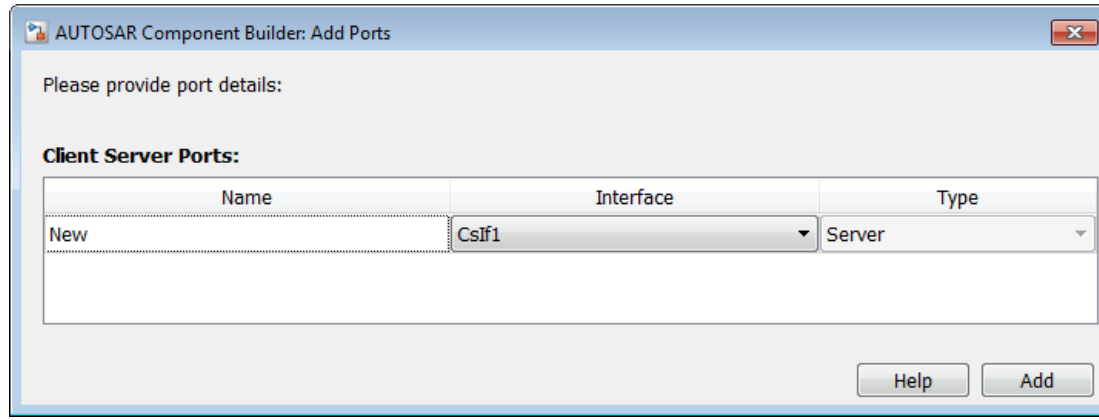
8 To view AUTOSAR server ports in the model, go to the ServerPorts view. Expand **AtomicComponents**, expand the individual component that you are configuring, and select **ServerPorts**. The example model already contains an AUTOSAR server port named `sPort`.



If a server port does not yet exist in your model, create one.

a In the ServerPorts view, click the **Add** icon . 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 ServerPorts 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** icon . 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`.)

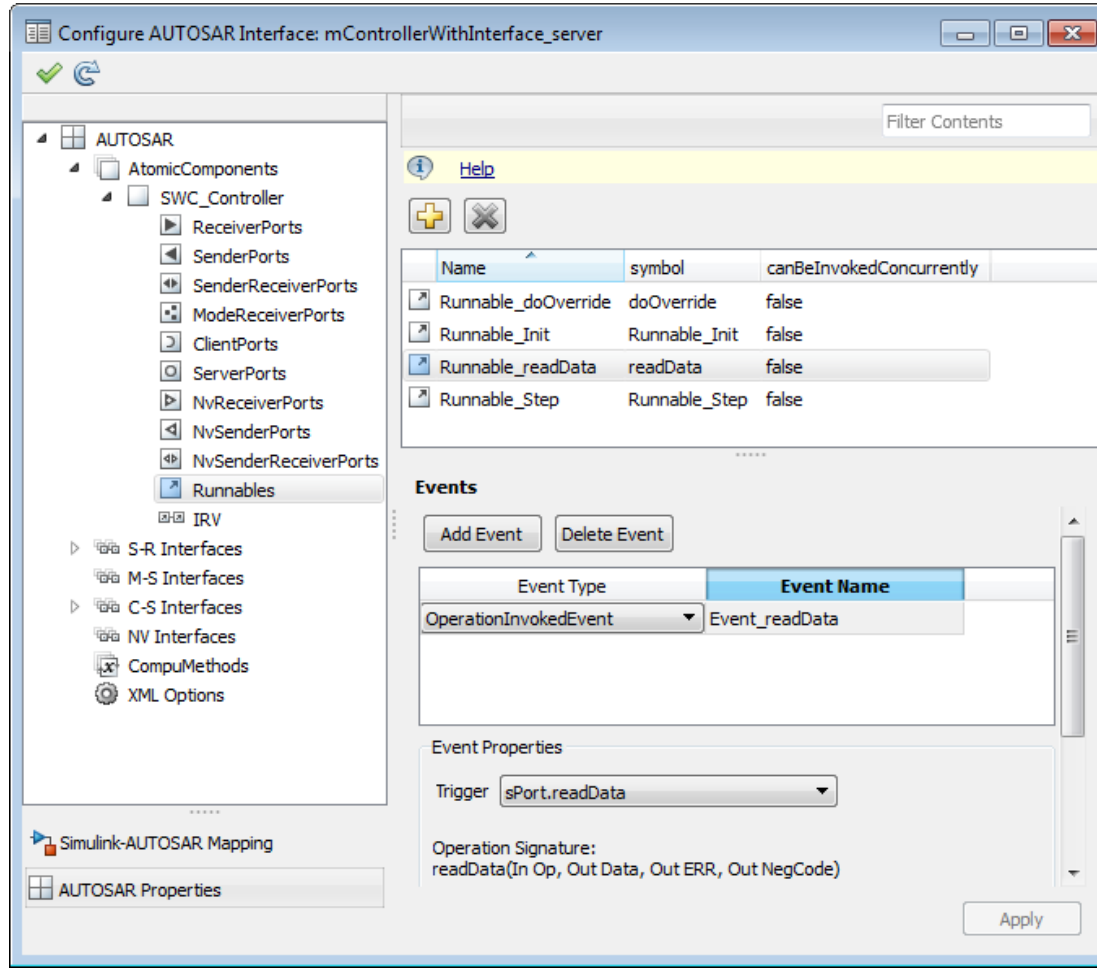
c Create an operation-invoked event to trigger the server runnable. (The example model defines event `event_readData` for server runnable `Runnable_readData`.)

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**.

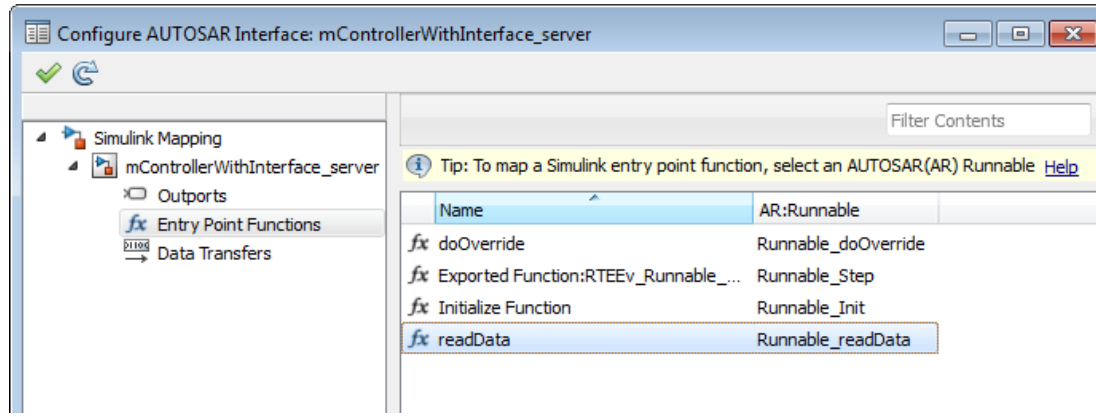


This step completes the configuration of an AUTOSAR server in the AUTOSAR Properties Explorer view of the configuration.

10 Switch to the Simulink-AUTOSAR Mapping Explorer view of the configuration, and map the Simulink server function to the AUTOSAR server runnable.

a In the Configure AUTOSAR Interface dialog box, select **Simulink-AUTOSAR Mapping** and then select **Entry Point Functions**.

b In the Entry Point Functions view, select the Simulink server function. To map the function to an AUTOSAR runnable, click on the **AR:Runnables** 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 interface configuration, click the **Validate** icon in the upper left area of the Configure AUTOSAR Interface dialog box. 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 4-112.

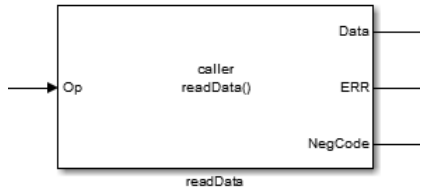
Configure AUTOSAR Client

After you configure an AUTOSAR server, as described in "Configure AUTOSAR Server" on page 4-101, 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/ecoder/examples/autosar/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`.

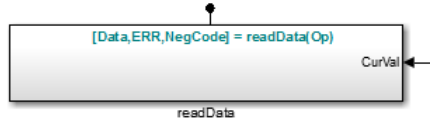
2 Add a Function Caller block to the model. In the Simulink Library Browser, the Function Caller block is in **User-Defined Functions**.

The example model provides a Simulink Function block named `readData`, which is located inside `Runnable3_Subsystem`.

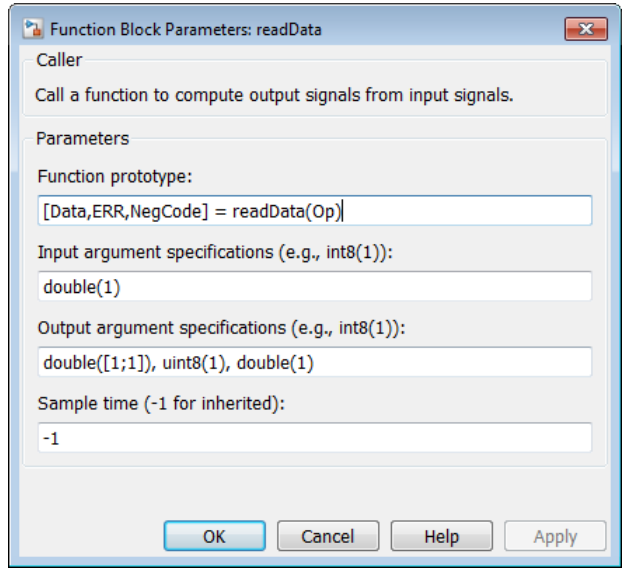


3 Configure the Function Caller block to call a corresponding Simulink Function block. Double-click the block to open it, and edit the block parameters to specify the server function prototype.

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 4-101. 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 4-101. The argument types and dimensions also must match the server function arguments.

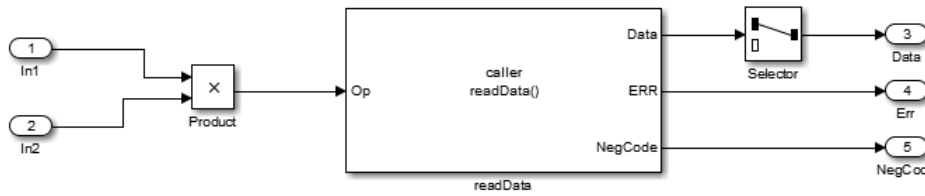


Note: If you want to simulate the function invocation at this point, you must place the Function Caller block in a common model or test harness with the corresponding Simulink Function block. Simulation is not required for this example.

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 configure the AUTOSAR interface, 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 3-15.

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.



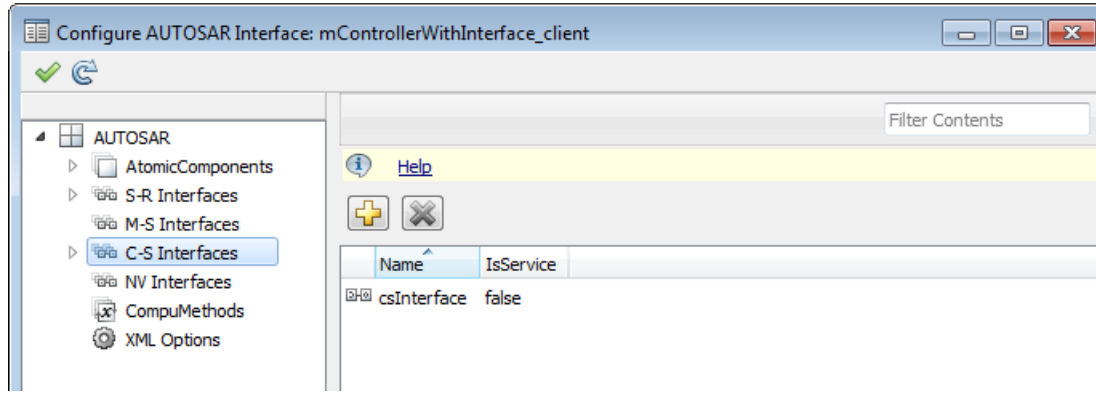
Note: Whenever you add or change a Function Caller block in an AUTOSAR model, synchronize function callers in the AUTOSAR configuration. If the Configure AUTOSAR Interface dialog box is not already open, in the model window, select **Code > C/C++ Code > Configure Model as AUTOSAR Component**. In the dialog box, click the **Synchronize** icon in the upper left area. This action loads or updates Simulink data transfers, function callers, and numeric types in your model. After synchronizing, the function caller you added appears in the Function Callers view of the Simulink-AUTOSAR Mapping Explorer.

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

To view the AUTOSAR configuration for the model, in the model window, select **Code > C/C++ Code > Configure Model as AUTOSAR Component**. This action opens the Configure AUTOSAR Interface dialog box.

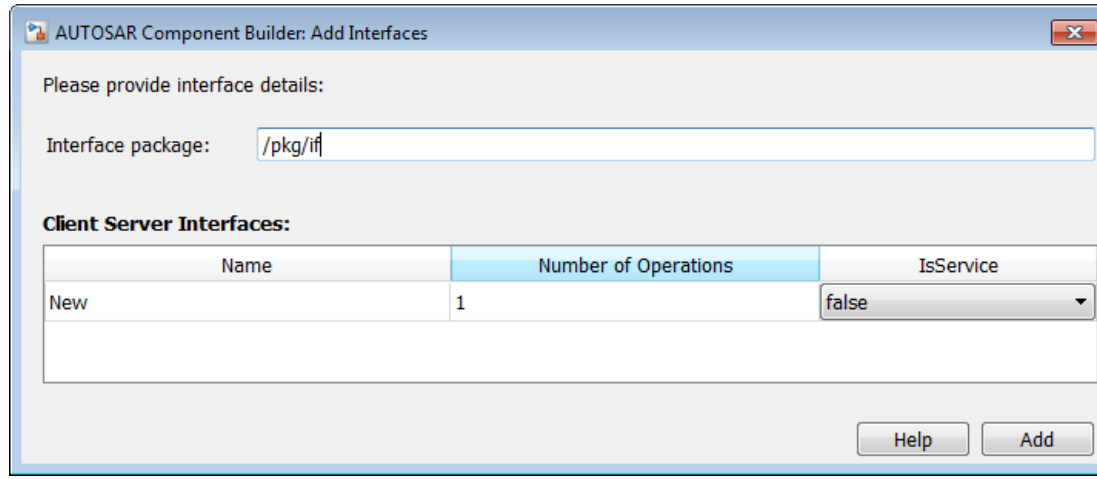
To view AUTOSAR C-S interfaces in the model, go to the C-S Interfaces view. Select **AUTOSAR Properties** and then **C-S Interfaces**. The example model already contains a client-server interface named `csInterface`.



If a C-S interface does not yet exist in the AUTOSAR configuration, create one.

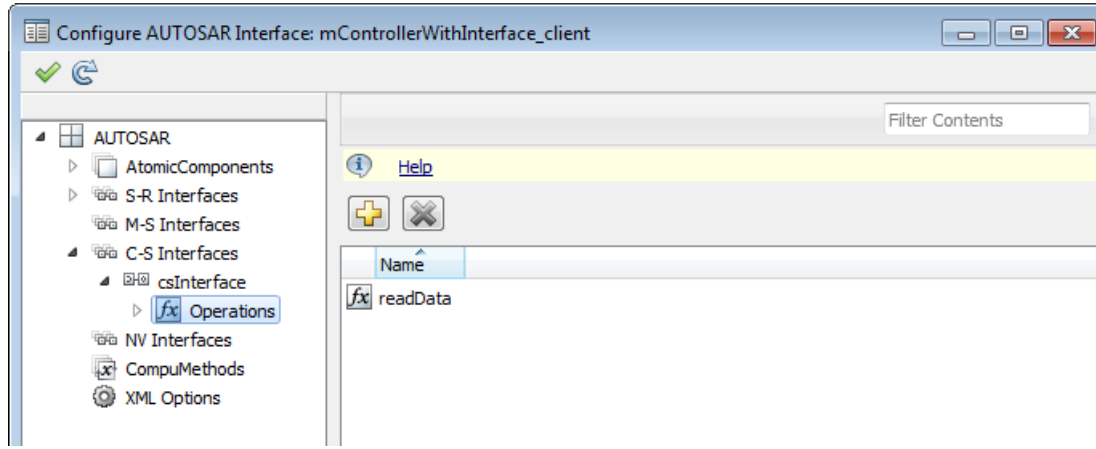
a In the C-S Interfaces view, click the **Add** icon . This action opens the Add Interfaces dialog box.

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 there is a placeholder operation named `Operation1` in your C-S interface, you can safely delete it.)

a In the Operations view, click the **Add** icon . This action opens the Add Operation dialog box.

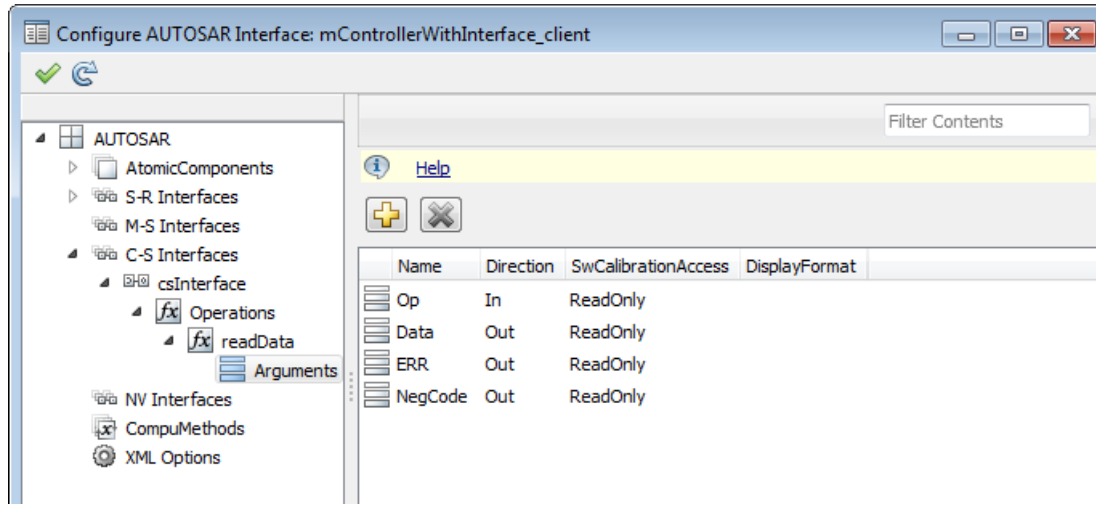
b In the dialog box, enter the **Operation Name**. Specify the name of the corresponding Simulink server function. Click **OK**. The new interface appears in the C-S Interfaces view.

c In the Operations view, expand **Operations**, expand the individual operation (for example, `readData`), and select **Arguments**.

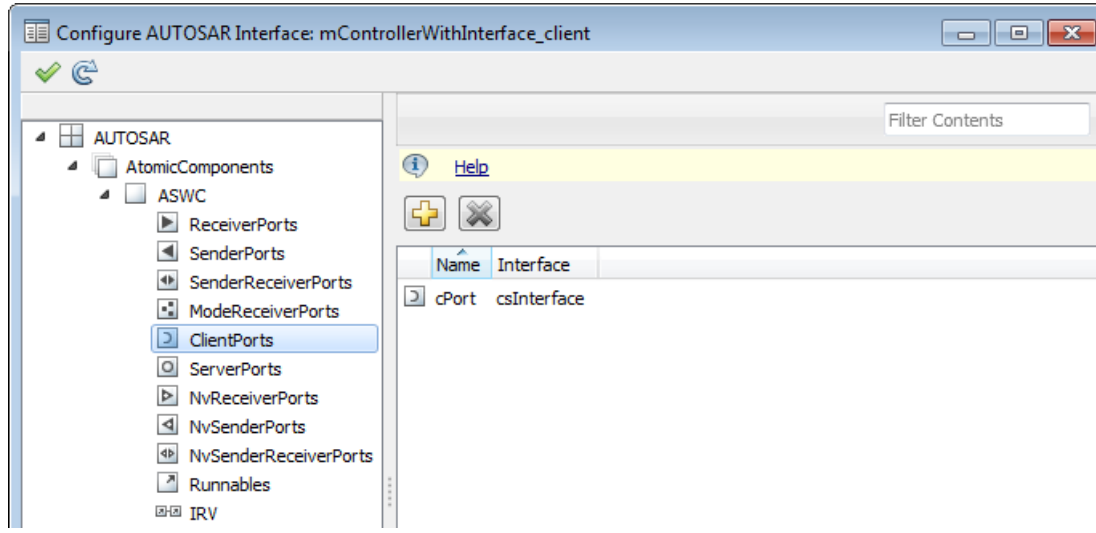
d In the Arguments view, click the **Add** icon one time for each function argument. For example, for `readData`, click the **Add** icon four times, for arguments `Op`, `Data`, `ERR`, and `NegCode`. Each click creates one new argument entry.

e Select each argument entry and set the argument **Name** and **Direction** to match the function prototype.

7 Examine the arguments listed for the AUTOSAR operation. In the Operations view, expand **Operations**, expand the individual operation (for example, `readData`), and select **Arguments**. verify that the listed arguments correspond to the function prototype defined by the Function Caller block in your model.



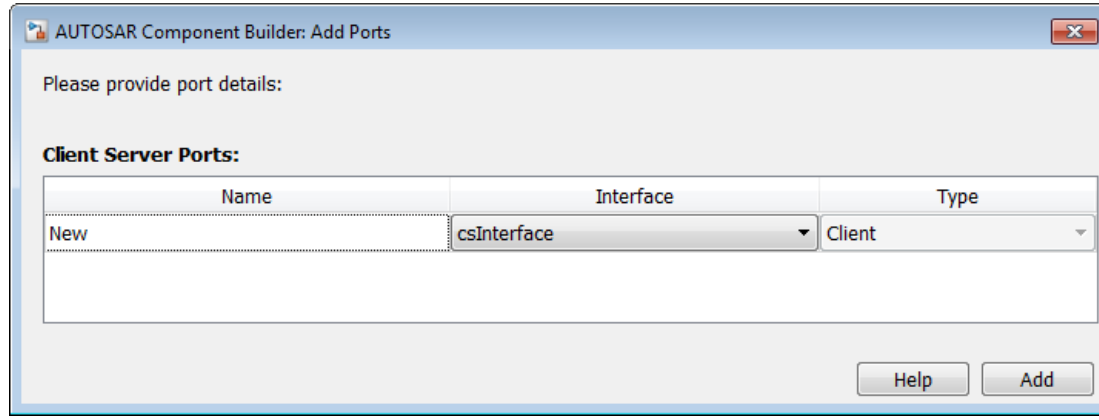
8 To view AUTOSAR client ports in the model, go to the ClientPorts view. Expand **AtomicComponents**, expand the individual component that you are configuring, and select **ClientPorts**. The example model already contains an AUTOSAR client port named `cPort`.



If a client port does not yet exist in your model, create one.

a In the ClientPorts view, click the **Add** icon . 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 ClientPorts view.

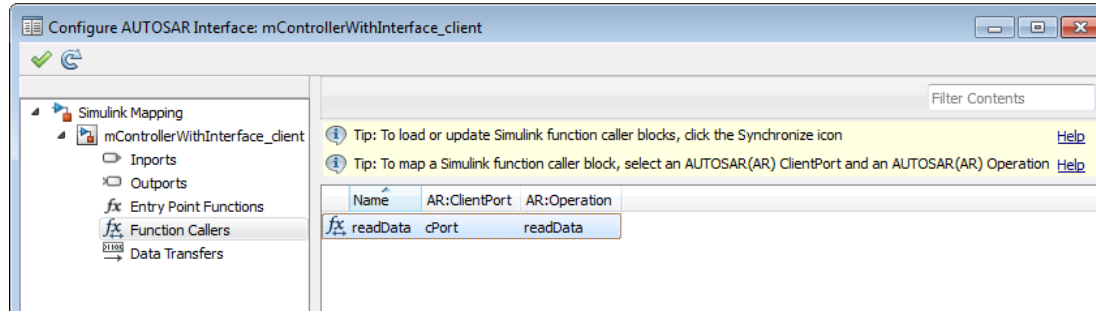


This step completes the configuration of an AUTOSAR client in the AUTOSAR Properties Explorer view of the configuration.

9 Switch to the Simulink-AUTOSAR Mapping Explorer view of the configuration, and map the Simulink function caller to an AUTOSAR client port and C-S operation.

   a In the Configure AUTOSAR Interface dialog box, select **Simulink-AUTOSAR Mapping** and then select **Function Callers**.

   b In the Function Callers view, select the Simulink function caller. Click on the **AR:ClientPort** field and select a port from the list of available AUTOSAR client ports. Click on the **AR: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 interface configuration, click the **Validate** icon in the upper left area of the Configure AUTOSAR Interface dialog box. 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.

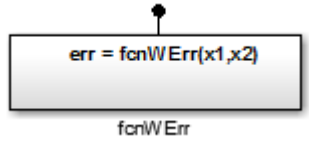
To implement AUTOSAR client-server error handling in Simulink:

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`:

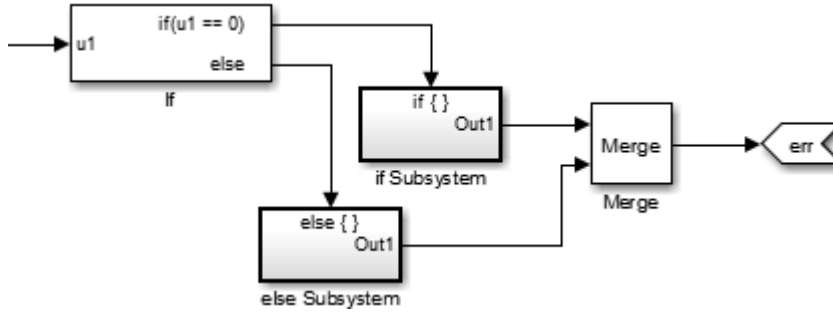
```
classdef(Enumeration) appErrType < uint8

   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 affect 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.

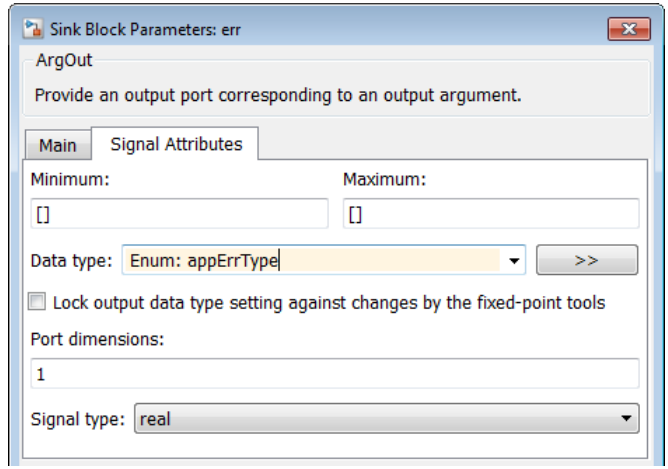
2 Define an error status output argument for the `Simulink Function` block that models the AUTOSAR server. Configure the error status argument as the only function output or add it to other outputs. For example, here is a Simulink Function block that returns an error status value in output `err`.



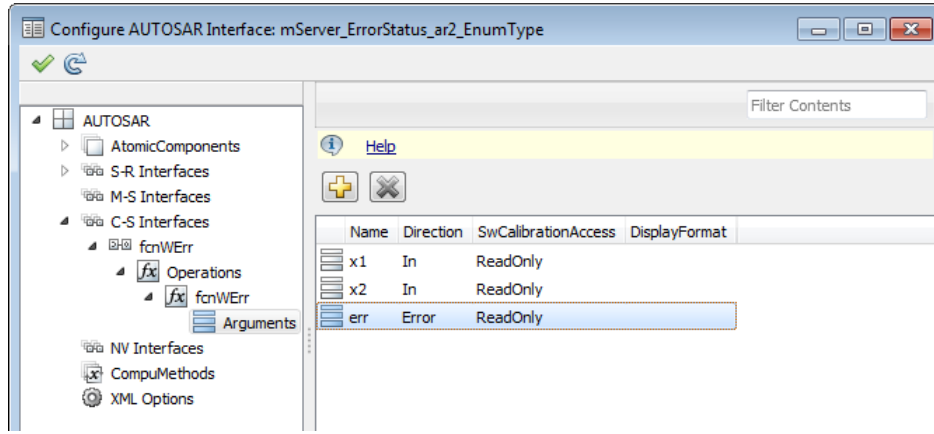
The Simulink Function block implements an algorithm to return error status.



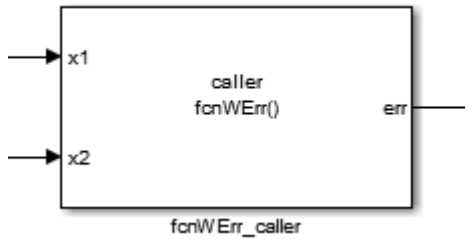
3 Reference the possible error values type in the model. In the `Argument Outport` block parameters for the error outport, specify the error status data type, in this case, `appErrType`. Set **Port dimensions** to 1 and **Signal type** to `real`.



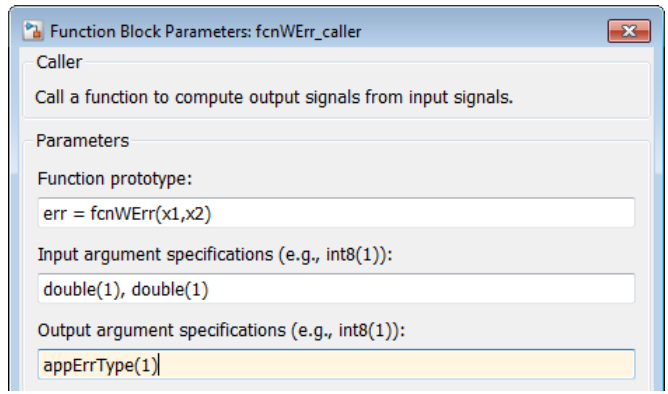
4 Configure the AUTOSAR properties of the error argument in the client-server interface. Open the Configure AUTOSAR Interface dialog box, select **AUTOSAR Properties**, expand **C-S Interfaces**, and navigate to the Arguments view of the AUTOSAR operation. To add an argument, click the **Add** icon . Configure the argument name and set **Direction** to `Error`.



5 Create an error port in each `Function-Caller` block that models an AUTOSAR client invocation. For example, here is a Function-Caller block that models an invocation of `fcnWErr`.



In the Function-Caller block parameters, specify the same error status data type.



Configure the AUTOSAR properties of the error argument to match the information in Configure AUTOSAR Interface dialog box, 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`.

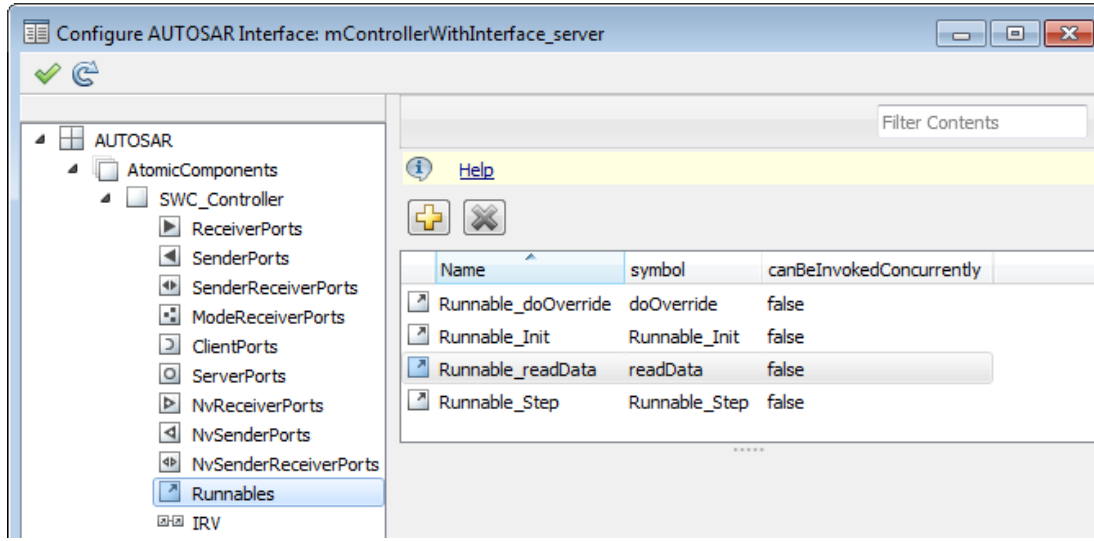
```
>> arProps = autosar.api.getAUTOSARProperties(bdroot)
>> set(arProps,'fcnWErr/fcnWErr/err','Direction','Error')
>> get(arProps,'fcnWErr/fcnWErr/err','Direction')
ans =
    Error
```

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 (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 of the AUTOSAR Properties Explorer.



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 4-289.

See Also

`Argument Inport` | `Argument Outport` | `Function Caller` | `Simulink Function` | `Trigger`

Related Examples

- “Client-Server Interface” on page 2-13
- “Configure AUTOSAR Client-Server Interfaces” on page 4-289
- “Import AUTOSAR Software Component” on page 3-4
- “Export AUTOSAR Component XML and C Code” on page 5-2

More About

- “Model AUTOSAR Communication” on page 2-11

- “AUTOSAR Interface 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 4-130

"Configure Mode Sender Port and Mode Switch Point for Application Mode Manager" on page 4-137

Configure Mode Receiver Port and Mode-Switch Event for Mode User

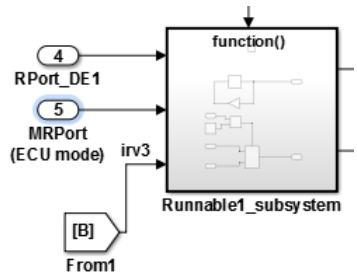
To model a mode user software component, use an AUTOSAR mode receiver port and a mode-switch event. The mode receiver port uses a mode-switch (M-S) interface to connect and communicate with a mode manager, which provides notifications of mode changes. You configure a mode-switch event to respond to a 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.

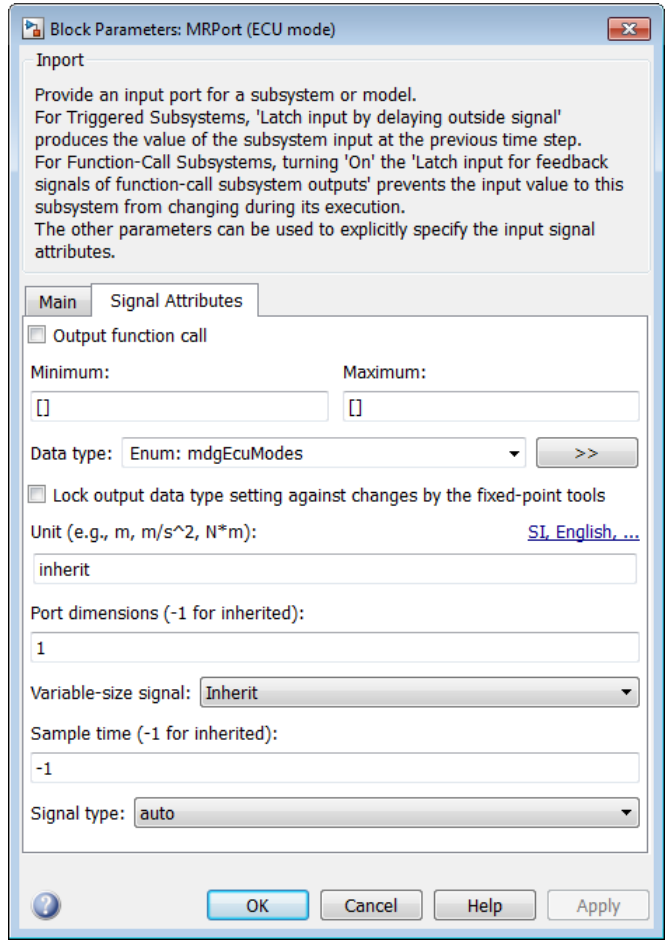
1. Open the example model `rtwdemo_autosar_multirunnables`. Save a copy to a writable work folder.
2. Declare a mode declaration group — a group of mode values — using Simulink enumeration. Enter the following command in the MATLAB Command Window:

```
Simulink.defineIntEnumType('mdgEcuModes', ...
 {'Run', 'Sleep'}, [0;1], ...
 'Description', 'Mode declaration group for ECU modes', ...
 'DefaultValue', 'Run', ...
 'HeaderFile', 'Rte_Type.h', ...
 'AddClassNameToEnumNames', false);
```

3 In the model window, rename the Simulink inport `RPort_DE1 (ErrorStatus)` to `MRPort (ECU mode)`. In a later step, you will map this inport to an AUTOSAR mode-receiver port.



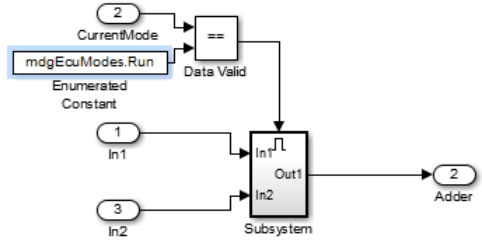
4 Next, apply the mode declaration group `mdgEcuModes` to inport `MRPort (ECU mode)`. Double-click the inport to open the Inport block parameters dialog box. Select the **Signal Attributes** tab. For **Data type**, enter `Enum: mdgEcuModes`. Additionally, set **Signal type** to `auto`. Click **Apply**.



5 In the model window, open the function-call subsystem named `Runnable1_subsystem` and make the following changes:

   a Rename inport `ErrorStatus` to `CurrentMode`.

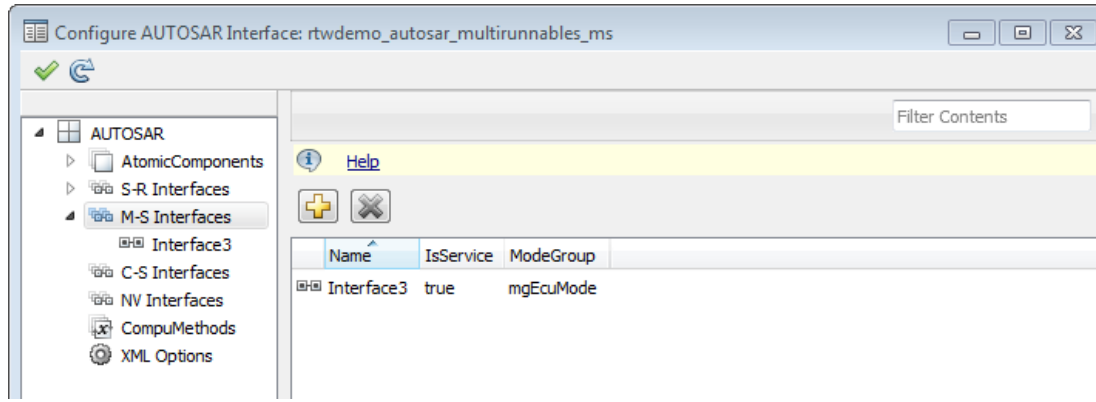
   b Replace Constant block `RTE_E_OK` with an Enumerated Constant block. (The Enumerated Constant block can be found in the Sources block group.) Double-click the block to open its block parameters dialog box. Set **Output data type** to `Enum: mdgEcuModes` and set **Value** to `mdgEcuModes.Run`. Click **OK**.



6 Add an AUTOSAR mode-switch interface to the model. Open the Configure AUTOSAR Interface dialog box using **Code > C/C++ Code > Configure Model as AUTOSAR Component**. Select the AUTOSAR Properties Explorer, and select **M-S Interfaces**. To open the Add Interfaces dialog box, click the **Add** icon . In the Add Interfaces dialog box, specify **Name** as `Interface3` and specify **ModeGroup** as `mgEcuMode`.

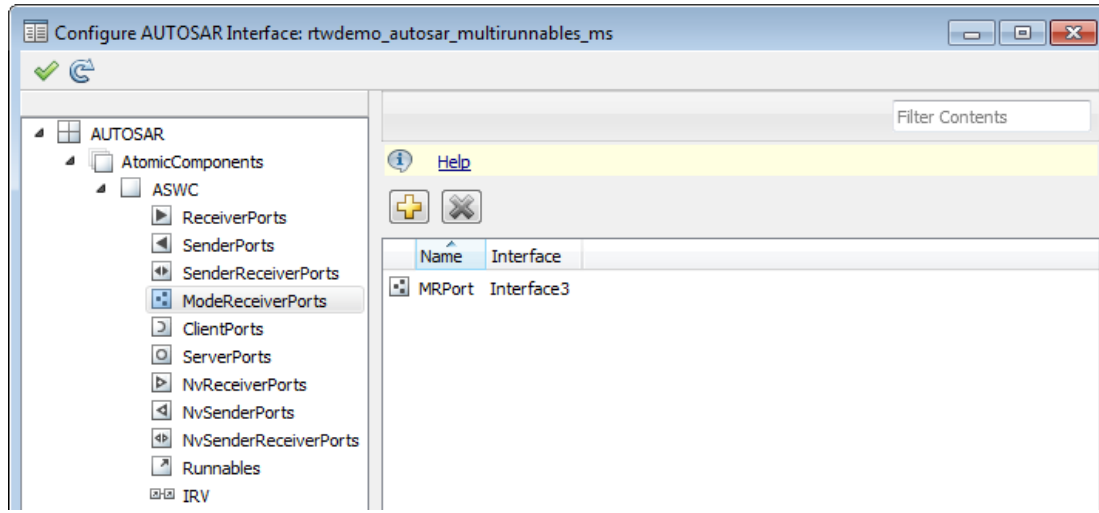
The **IsService** property of an M-S interface defaults to `true`. For the purposes of this example, you can leave **IsService** at its default setting, unless you have a reason to change it. For more information, see M-S Interfaces View of AUTOSAR Properties Explorer.

Click **Add**.

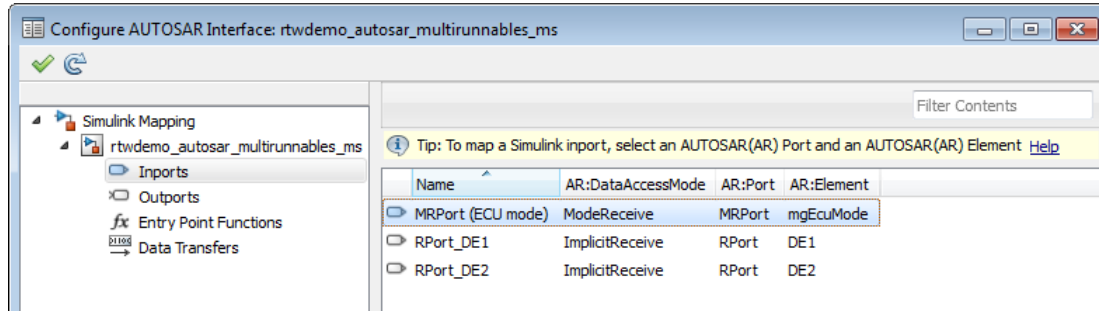


The value you specify for the AUTOSAR mode group is used in a later step, when you map a Simulink inport to an AUTOSAR mode-receiver port and element.

7 Add an AUTOSAR mode-receiver port to the model. In the AUTOSAR Properties Explorer, expand component ASWC and select **ModeReceiverPorts**. To open the Add Ports dialog box, click the **Add** icon . 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 Map the Simulink inport `MRPort (ECU mode)` to the AUTOSAR mode-receiver port and element. Select the Simulink-AUTOSAR Mapping Explorer, and select **Inports**. In the row for inport `MRPort (ECU mode)`, set **AR:DataAccessMode** to `ModeReceive`, set **AR:Port** to `MRPort`, and set **AR: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** icon to validate the AUTOSAR interface configuration. If errors are reported, address them and then retry validation. When the model passes validation, save the model.

Note: The remaining steps create an AUTOSAR mode-switch event and set it up to trigger activation of an AUTOSAR runnable. If you intend to use ECU modes to control program execution, without using an event to activate a runnable, you can skip the remaining steps and implement the required flow-control logic in your design.

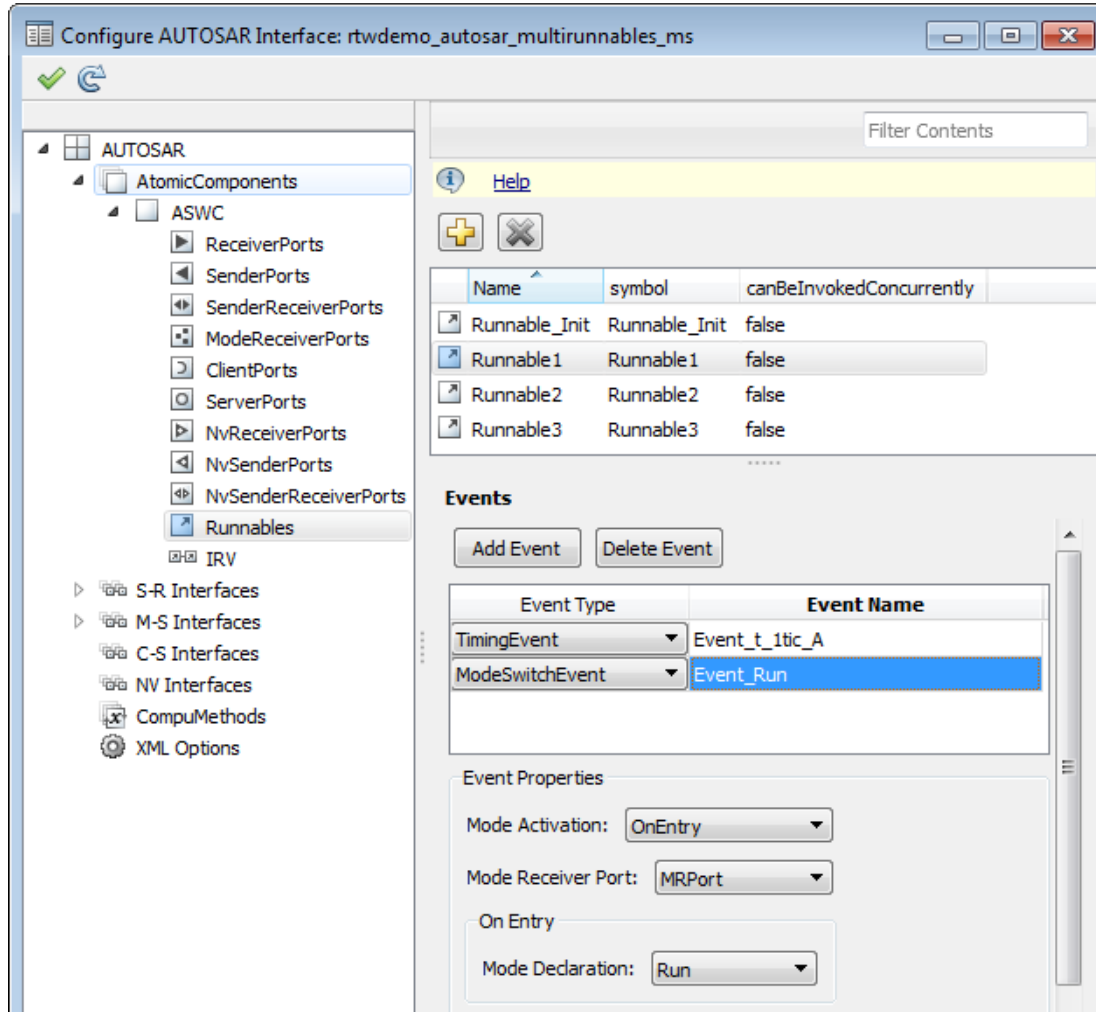
---

9 To add an AUTOSAR mode-switch event for a runnable:

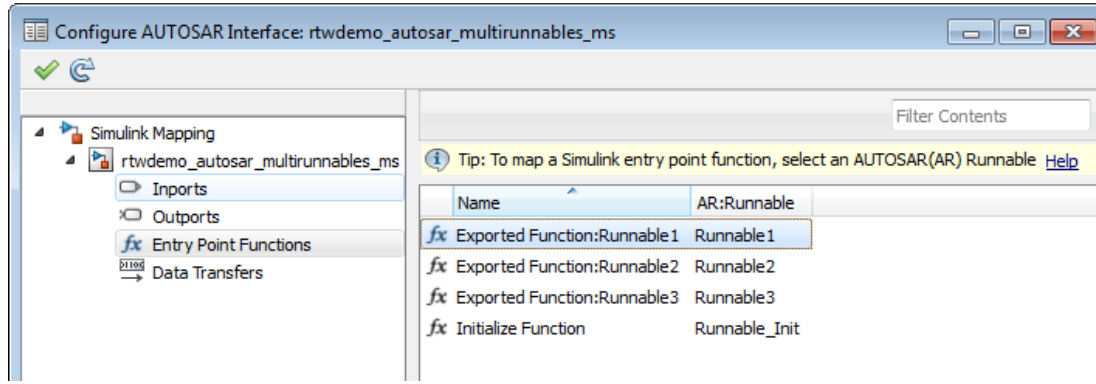
a Open the Configure AUTOSAR Interface dialog box, if it is not already open. Select the AUTOSAR Properties Explorer, expand the `ASWC` component, and select **Runnables**. In the list of runnables, select `Runnable1`. This selection activates an **Events** configuration pane for the runnable.

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 Select the Simulink-AUTOSAR Explorer, and select **Entry Point Functions**. 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** icon to validate the AUTOSAR interface 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.

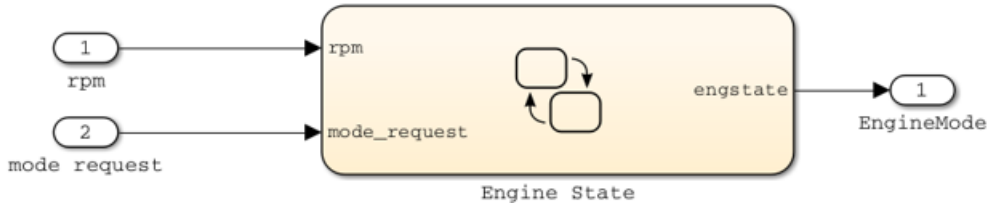
Configure Mode Sender Port and Mode Switch Point for Application Mode Manager

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.

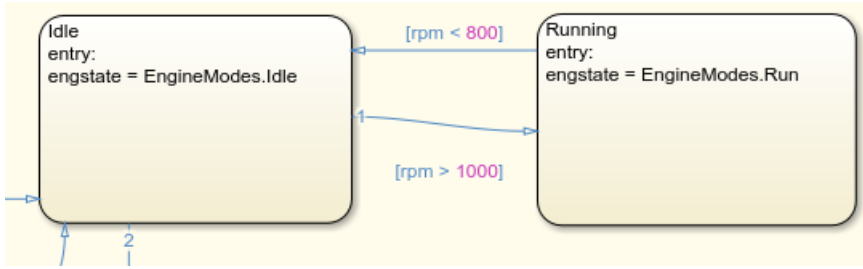
You model the mode sender port as a model root outport, which is mapped to an AUTOSAR mode sender port and a mode-switch (M-S) interface. The outport data type is an enumeration class, representing an AUTOSAR mode declaration group.

This example shows how to configure a mode sender port and related elements for an application mode manager.

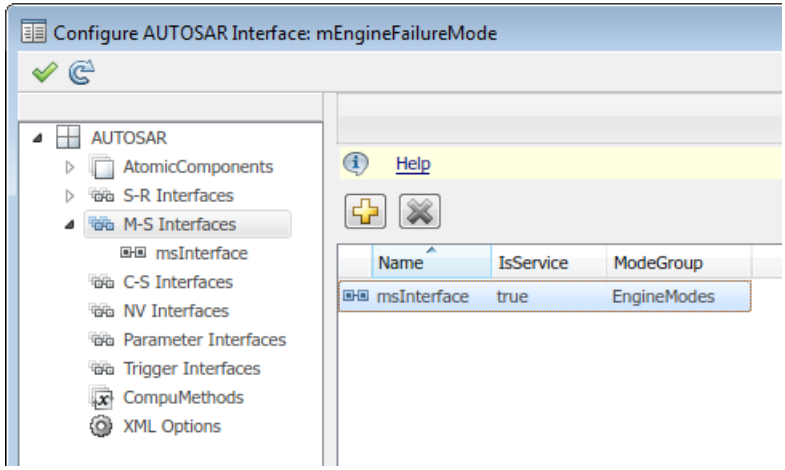
1 Open a model configured for AUTOSAR code generation. This example uses a model that contains Stateflow logic for maintaining engine state. The model outputs the current engine mode value.



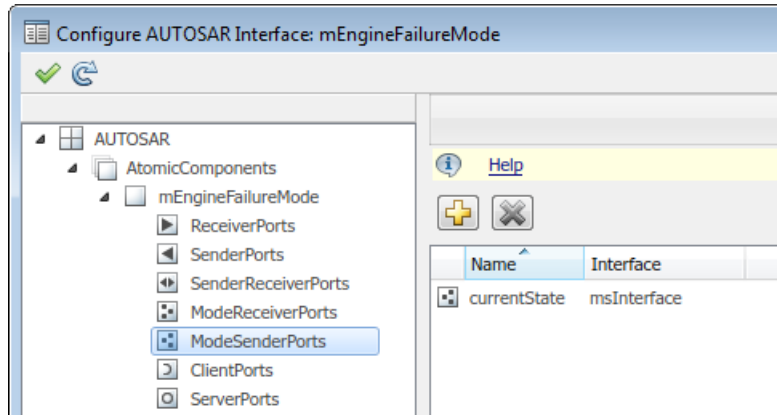
2 Declare a mode declaration group — a group of mode values. You can declare mode values with Simulink enumeration. In this example, the Stateflow logic defines `EngineModes` values `Off`, `Crank`, `Stall`, `Idle`, and `Run`. For example:



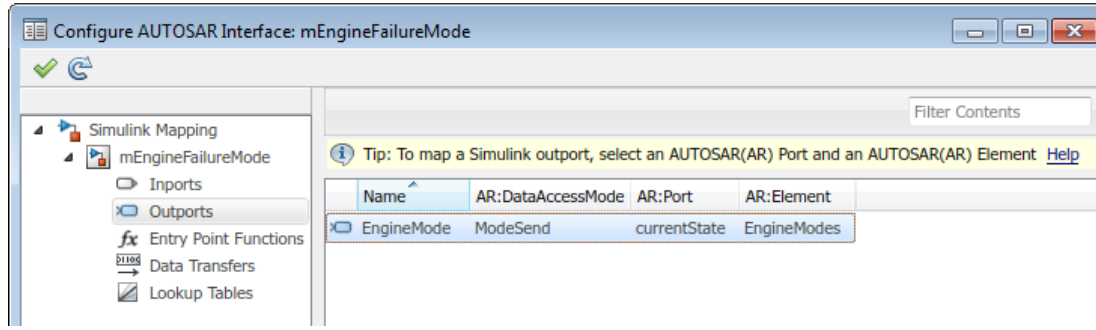
3 Open the Configure AUTOSAR Interface dialog box, select the AUTOSAR Properties Explorer, and add an AUTOSAR M-S interface to the model. Set **isService** to `true` and enter a **ModeGroup** name. In this example, the mode declaration group is `EngineModes`.



4 Go to the Mode Sender Ports view and add an AUTOSAR mode sender port. Set **Interface** to the M-S interface name.



5 Switch to the Simulink-AUTOSAR Mapping Explorer, and go to the Outports view. Select the Simulink outport that outputs the mode value. To map the outport to the AUTOSAR mode sender port, set **AR:DataAccessMode** to `ModeSend`, select the **AR:Port** name, and for **AR: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);
```

Related Examples

- "Mode-Switch Interface" on page 2-15
- "Configure AUTOSAR Mode-Switch Interfaces" on page 4-292
- "Import AUTOSAR Software Component" on page 3-4
- "Export AUTOSAR Component XML and C Code" on page 5-2

More About

- "Model AUTOSAR Communication" on page 2-11
- "AUTOSAR Interface Configuration" on page 4-3

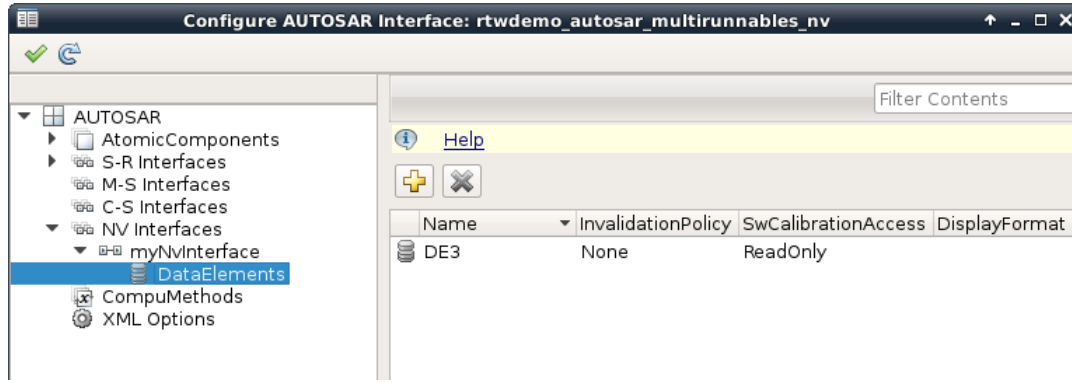
Configure AUTOSAR Nonvolatile Data Communication

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.

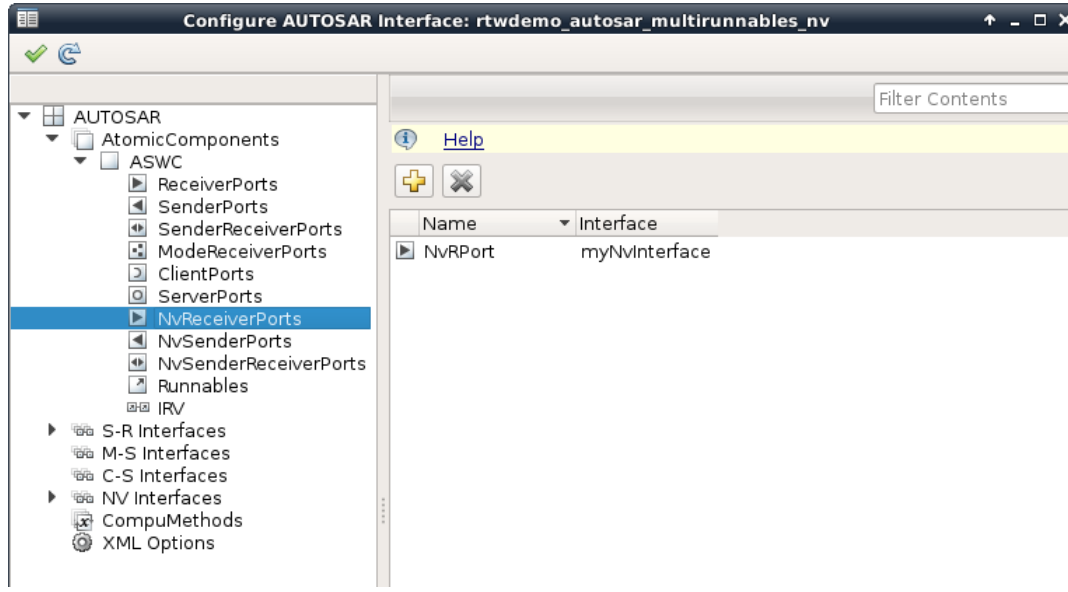
In Simulink, you can create AUTOSAR NV interfaces and ports, and map Simulink inports and outports to AUTOSAR NV ports. You model AUTOSAR NV ports with Simulink inports and outports, in the same manner described in "Sender-Receiver Interface" on page 2-12.

To create an NV data interface and ports in Simulink:

1. Open the Configure AUTOSAR Interface dialog box and select **AUTOSAR Properties**.
2. Select **NV Interfaces**. Click the **Add** icon to create a new NV data interface. Specify its name and the number of associated NV data elements.
3. Select and expand the new NV interface. Select **Data Elements**, and modify the data element attributes.

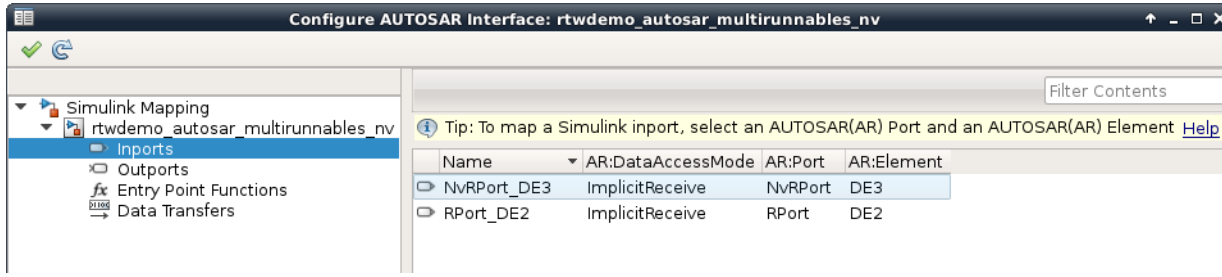


4. In the left-hand pane of the Configure AUTOSAR Interface dialog box, under **AUTOSAR**, select **AtomicComponents**. Expand **AtomicComponents** and select an AUTOSAR component. Expand the component.
5. Select and use the **NvReceiverPorts**, **NvSenderPorts**, and **NvSenderReceiverPorts** views to add the NV ports you require. For each NV port, select the NV interface you created.



6 Switch to the Simulink mapping view. Select **Simulink-AUTOSAR Mapping**.

7 Select and use the **Inports** and **Outports** views to map Simulink inports and outports to AUTOSAR NV 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('rtwdemo_autosar_multirunnables_nv');
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('rtwdemo_autosar_multirunnables_nv');
mapInport(slMap,'NvRPort_DE3','NvRPort','DE3','ImplicitReceive');
```

Related Examples

- “Nonvolatile Data Interface” on page 2-20
- “Configure AUTOSAR Sender-Receiver Interfaces” on page 4-288
- “Import AUTOSAR Software Component” on page 3-4
- “Export AUTOSAR Component XML and C Code” on page 5-2

More About

- “Model AUTOSAR Communication” on page 2-11
- “AUTOSAR Interface Configuration” on page 4-3

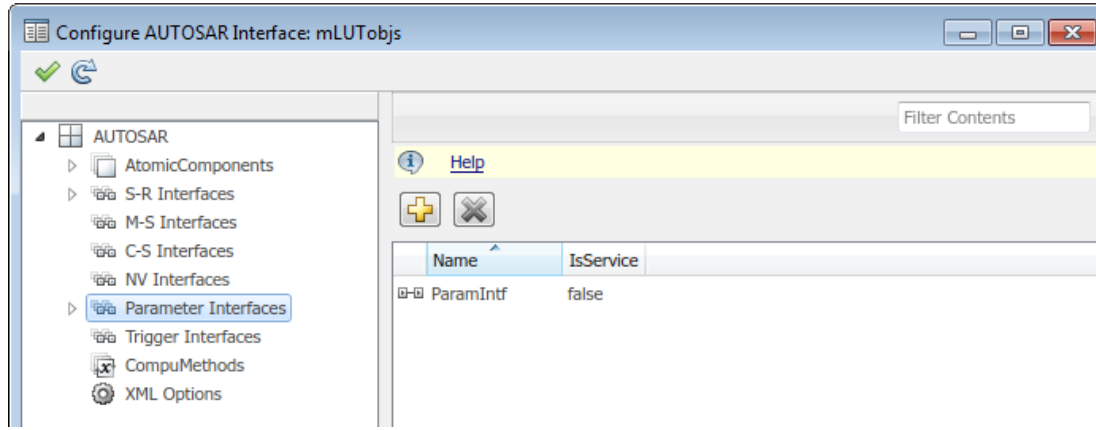
Configure Receiver for AUTOSAR Parameter Communication

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.

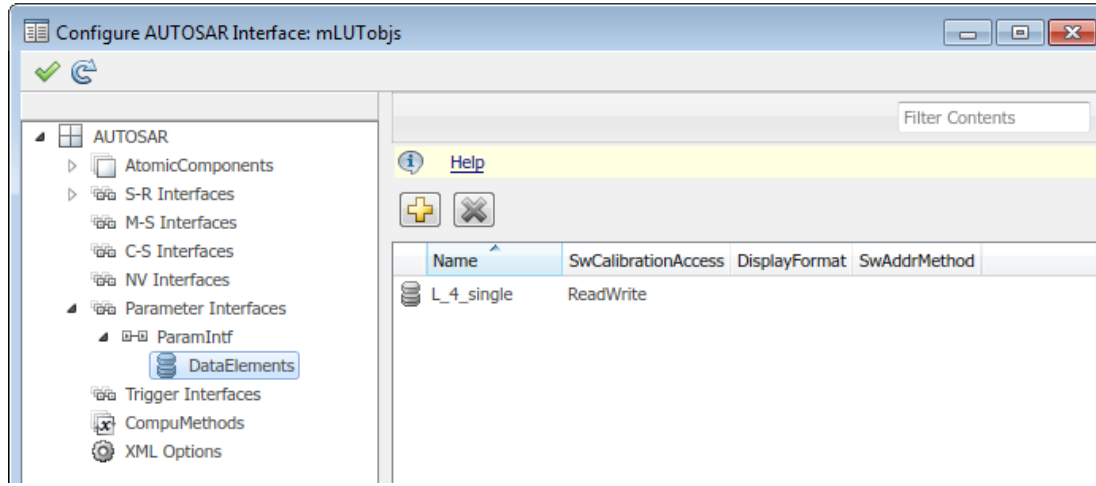
In Simulink, you can model the receiver portion of AUTOSAR port-based parameter communication. In an AUTOSAR atomic software component, you create a parameter interface with data elements and a parameter receiver port. If you model AUTOSAR integrated and distributed lookups as described in "Configure STD_AXIS and COM_AXIS Lookup Tables for AUTOSAR Measurement and Calibration" on page 4-174, you can map Simulink lookup tables to AUTOSAR port-based parameters.

This example shows how to configure an AUTOSAR software component as a receiver for parameter communication.

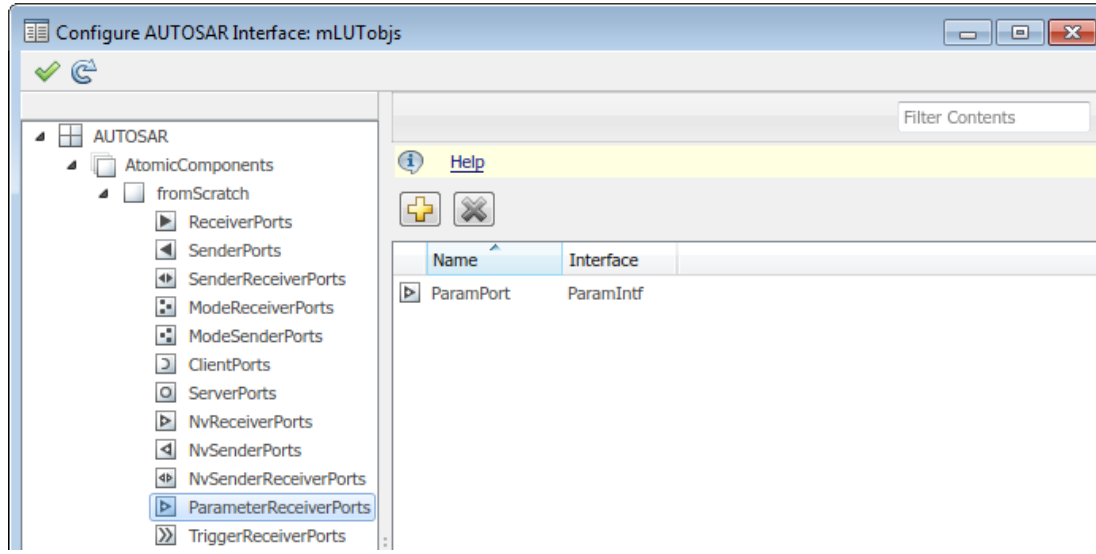
1. Open a model configured for AUTOSAR code generation in which the software component requires port-based access to parameter data.
2. Open the Configure AUTOSAR Interface dialog box, and select the AUTOSAR Properties Explorer. Go to the Parameter Interfaces view, and use the **Add** icon to add a parameter 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. Click the **Add** button to create the parameter interface and associated parameter data element.



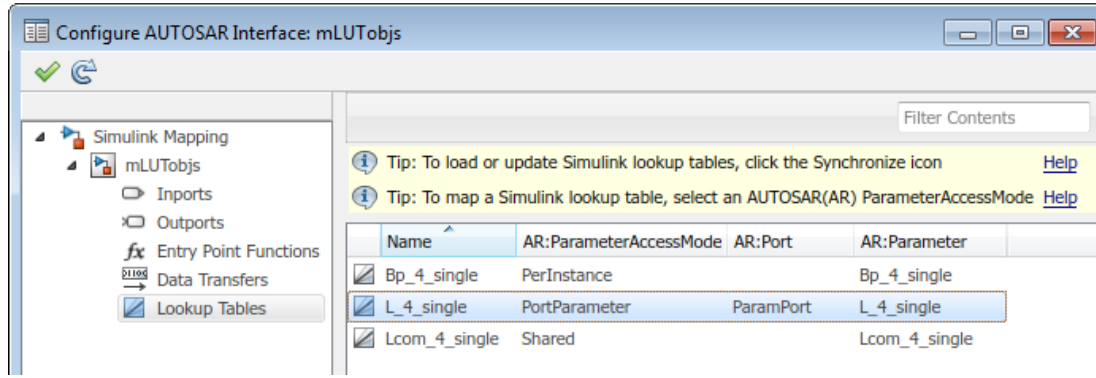
3 Expand the Parameter Interfaces node, and go to the Data Elements view. Examine and modify the properties of the associated data element that you created.



4 Expand the Atomic Components node. Go to the Parameter Receiver Ports view, and use the **Add** icon 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. Click the **Add** button to create the port.



5 The AUTOSAR parameter interface data elements that you create subsequently are available for Simulink lookup table mapping, using the Simulink-AUTOSAR Mapping Explorer or AUTOSAR map functions. Switch to the Simulink-AUTOSAR Mapping Explorer. Go to the Lookup Tables view. Select a Simulink lookup table or breakpoint object. To map the lookup table to a port-based AUTOSAR parameter, select parameter access mode `PortParameter`, and select a parameter receiver port and a parameter interface data element.



Related Examples

- "Configure AUTOSAR Port-Based Calibration Parameters" on page 4-169
- "Configure STD_AXIS and COM_AXIS Lookup Tables for AUTOSAR Measurement and Calibration" on page 4-174
- "Import AUTOSAR Software Component" on page 3-4
- "Export AUTOSAR Component XML and C Code" on page 5-2

More About

- "Model AUTOSAR Communication" on page 2-11
- "AUTOSAR Interface Configuration" on page 4-3

Configure Receiver for AUTOSAR External Trigger Event Communication

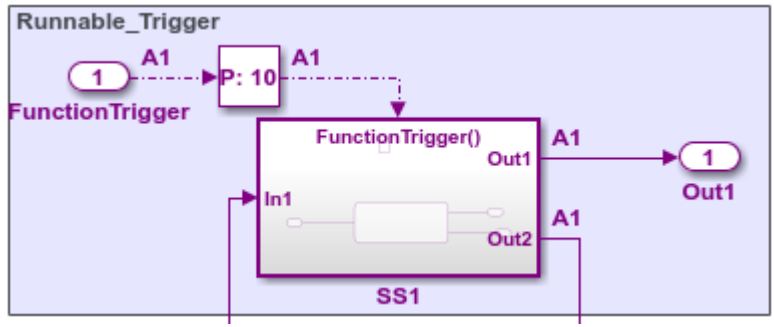
AUTOSAR Release 4.0 introduced external trigger event communication, in which an AUTOSAR software component or service signals an external trigger occurred event (`ExternalTriggerOccurredEvent`) to another component. The receiving component activates a runnable in response to the event.

In Simulink, you can model the receiver portion of AUTOSAR external trigger event communication. 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 is activated by the event.

This example shows how to configure an AUTOSAR software component as a receiver for external trigger event communication.

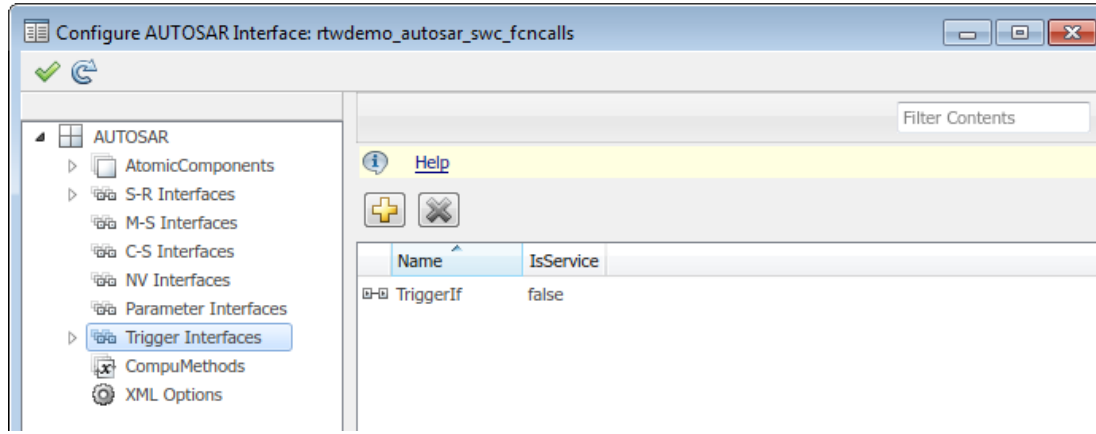
1 Open a model configured for AUTOSAR code generation, in which you want to activate a runnable based on receiving an AUTOSAR `ExternalTriggerOccurredEvent`.

For a sample model that leverages external trigger event communication, see `rtwdemo_autosar_swc_fcncalls`. In `rtwdemo_autosar_swc_fcncalls`, asynchronous function-call subsystem SS1 models an AUTOSAR runnable that is activated by an `ExternalTriggerOccurredEvent`.

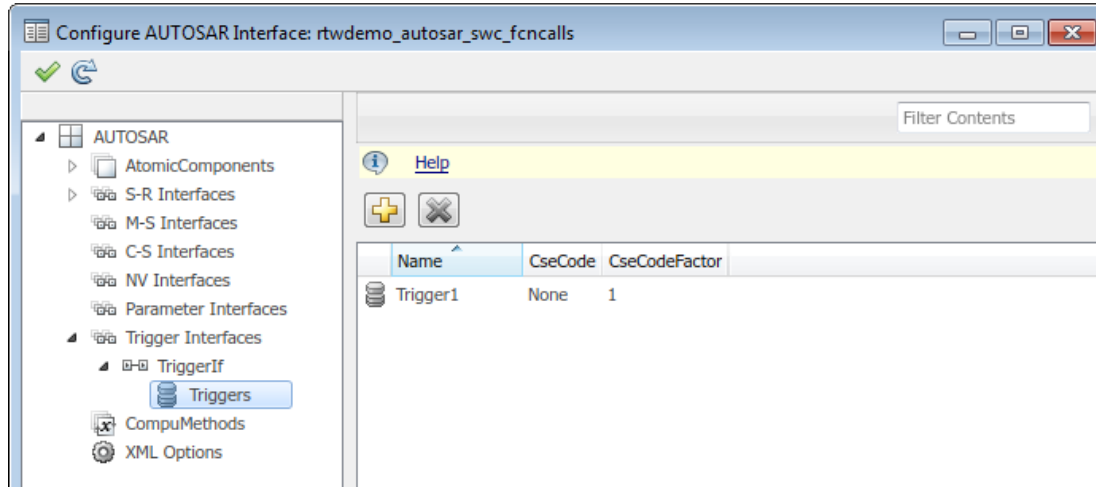


2 Open the Configure AUTOSAR Interface dialog box, and select the AUTOSAR Properties Explorer. Go to the Trigger Interfaces view, and use the **Add** icon to add a trigger interface to the model. In the Add Interfaces dialog box, specify the

name of the new interface and set **Number of Triggers** to 1. Click the **Add** button to create the trigger interface and associated trigger.

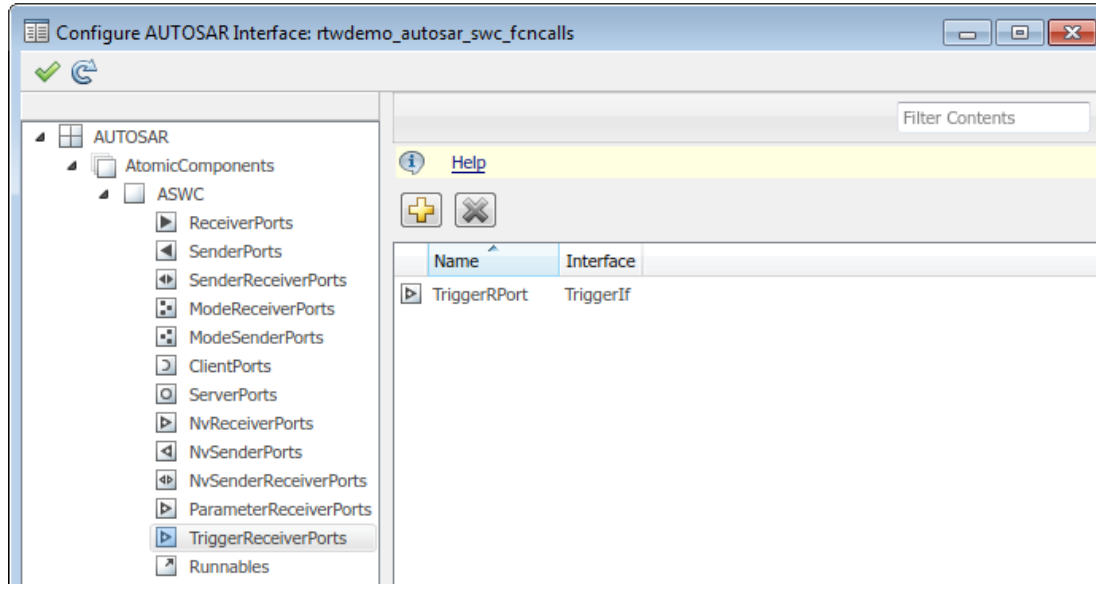


3 Expand the Trigger Interfaces node, and go to the Triggers view. Examine the properties of the associated trigger. For an asynchronous (nonperiodic) trigger, set **CseCode** to `None`, indicating an unspecified trigger period. For more information about specifying trigger periods, click the help link in the Triggers view.

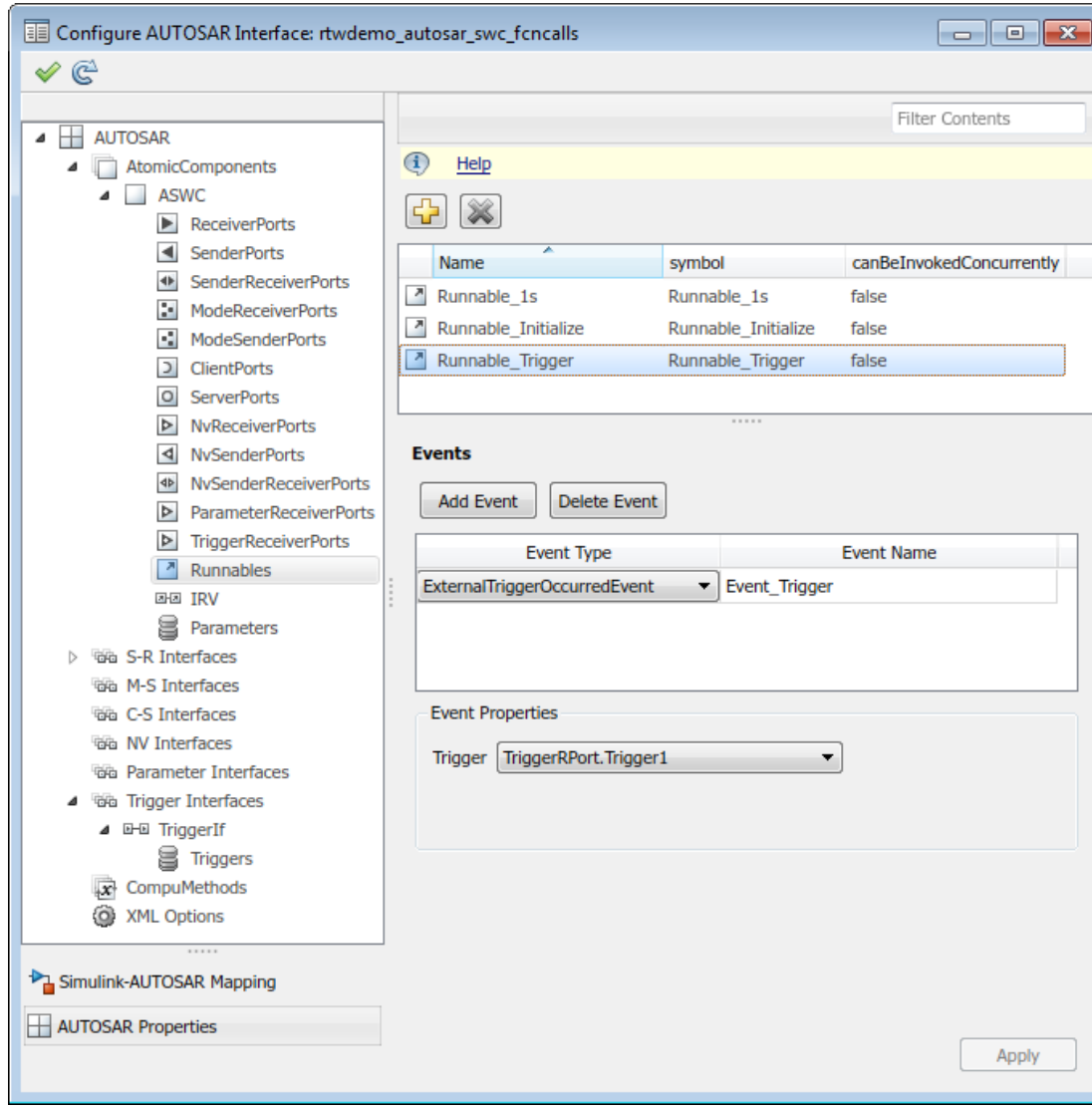


4 Expand the Atomic Components node. Go to the Trigger Receiver Ports view, and use the **Add** icon to add a trigger receiver port to the model. In the Add Ports dialog

box, specify the name of the new port and set **Interface** to the name of the trigger interface you created. Click the **Add** button to create the port.

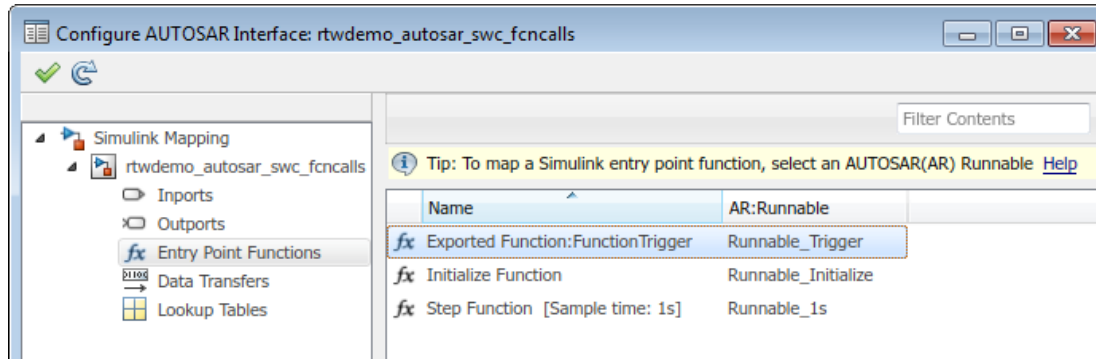


5 Go to the Runnables view and select the runnable that you want to activate based on receiving an AUTOSAR `ExternalTriggerOccurredEvent`. In the **Events** subpane, set **Event Type** to `ExternalTriggerOccurredEvent`. To display event properties, select the event name. For **Trigger**, select the value corresponding to the trigger receiver port and trigger you created.



6 To complete the trigger receiver configuration, switch to the Simulink-AUTOSAR Mapping Explorer. Go to the Entry Point Functions view. Select the Simulink entry-point function for the subsystem that models the AUTOSAR

`ExternalTriggerOccurredEvent` runnable. Map the function by selecting the runnable name.



Related Examples

- "Add Top-Level Asynchronous Trigger to Periodic Rate-Based System" on page 4-222
- "Import AUTOSAR Software Component" on page 3-4
- "Export AUTOSAR Component XML and C Code" on page 5-2

More About

- "Model AUTOSAR Communication" on page 2-11
- "AUTOSAR Interface Configuration" on page 4-3

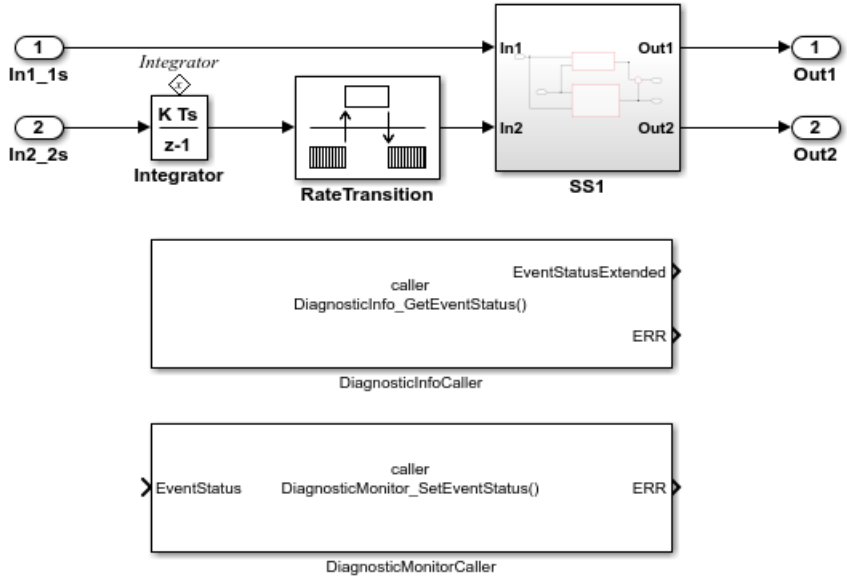
Configure Calls to AUTOSAR Diagnostic Event Manager (Dem) Service

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.

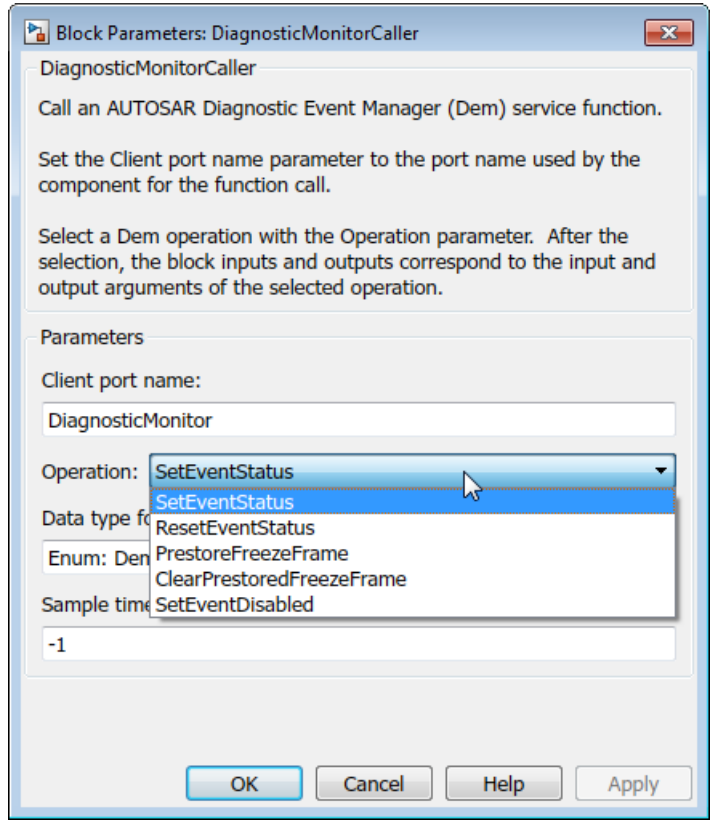
To support system-level modeling of AUTOSAR components and services, Embedded Coder Support Package for AUTOSAR Standard provides an AUTOSAR Basic Software block library. The library contains preconfigured Function Caller blocks for modeling component calls to AUTOSAR BSW services. For information about using the blocks to model client calls to AUTOSAR BSW service interfaces, see "Model AUTOSAR Basic Software (BSW) Service Calls" on page 2-22.

Here is an example of configuring client calls to Dem service interfaces in your AUTOSAR software component.

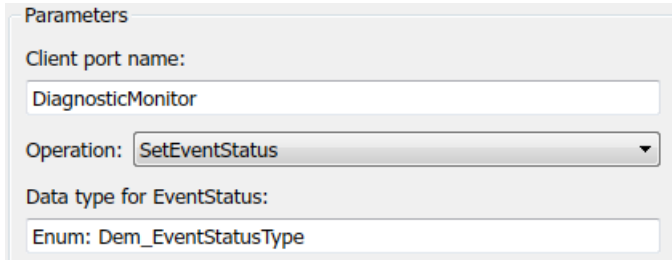
1. Open a model that is configured for AUTOSAR code generation. Using the Library Browser or by typing block names in the model window, add Dem blocks to the model. This example adds the blocks `DiagnosticInfoCaller` and `DiagnosticMonitorCaller` to a writable copy of the example model `rtwdemo_autosar_swc`.



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.



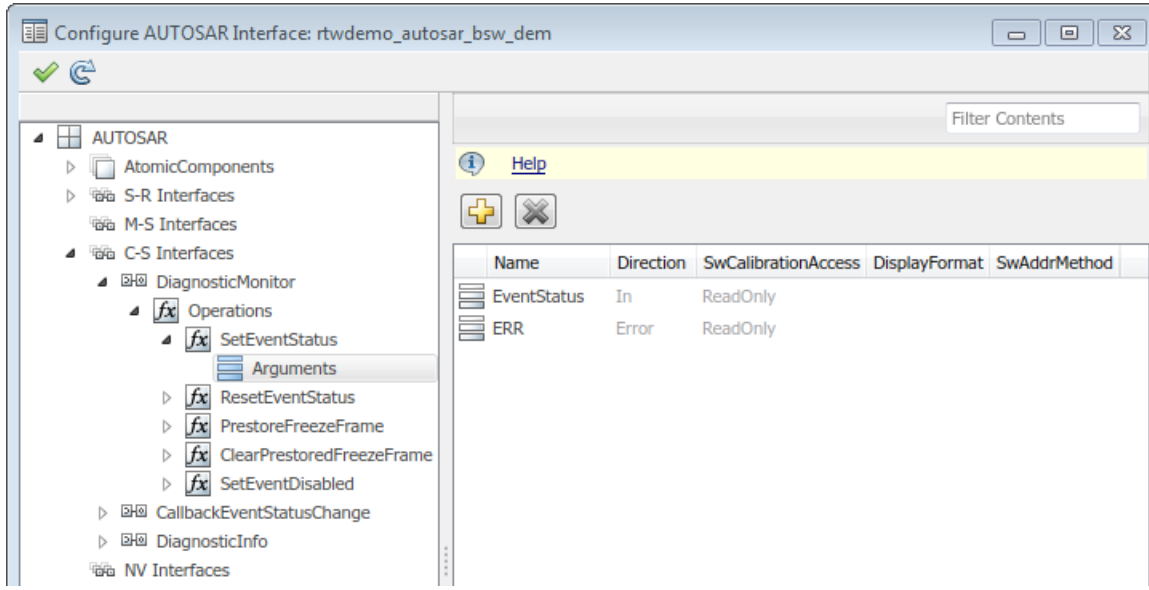
For some Dem operations, such as `GetEventStatus`, `SetEventStatus`, and `GetDTCOfEvent`, the block parameters dialog box displays a data type parameter. The data type parameter specifies an enumerated data type for a function input or output that represents a Dem event status or format type. Default data types are provided, such as `Dem_EventStatusType` or `Dem_DTCFormatType`. For more information about event status or format type values, see the AUTOSAR standard *Specification of Diagnostic Event Manager*.



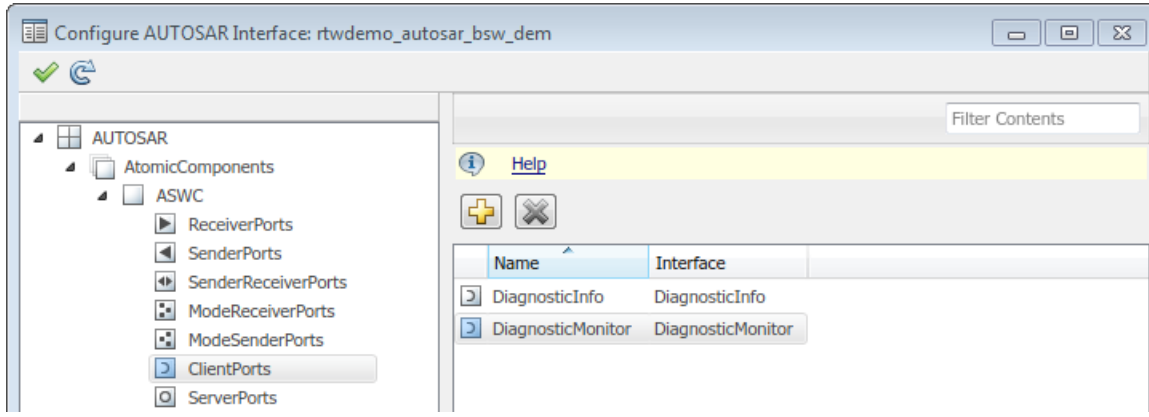
3 Open the Configure AUTOSAR Interface dialog box. Click the **Synchronize** icon to update the Simulink to AUTOSAR mapping of the model with changes to Simulink function callers. The software creates AUTOSAR client-service interfaces, operations, and ports, and maps each Simulink function caller to an AUTOSAR client port and operation.

For example, for the DiagnosticMonitorCaller block in this example, for which the `SetEventStatus` operation is selected:

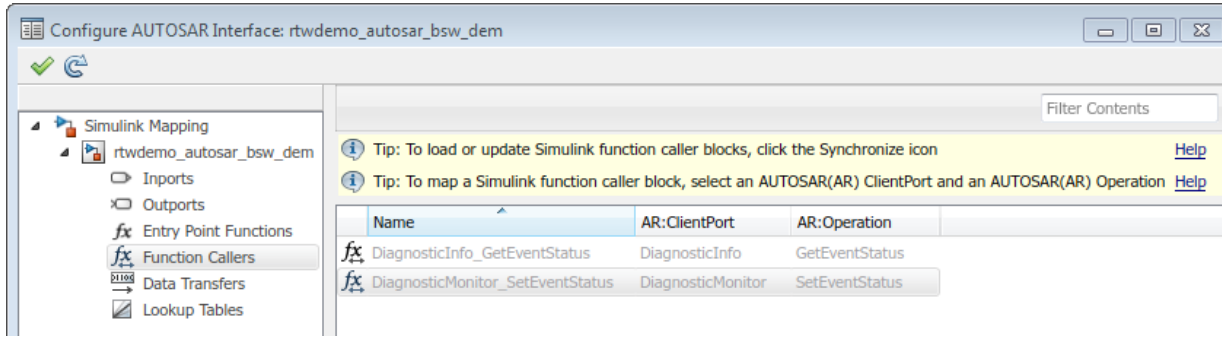
- The software creates C-S interface `DiagnosticMonitor`, and under `DiagnosticMonitor`, all of its supported operations. For each operation, arguments are provided with read-only properties. Here are the arguments for the `DiagnosticMonitor` operation `SetEventStatus`.



- The software creates a client port with the default name `DiagnosticMonitor`. Unlike the C-S-interface, operation, and argument names, the client port name can be customized. The client port is mapped to the `DiagnosticMonitor` interface.



- The software maps the `DiagnosticMonitor` function caller block to AUTOSAR client port `DiagnosticMonitor` and AUTOSAR operation `SetEventStatus`.



4 Optionally, build your 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_GetEventStatus(&rtb_DiagnosticInfoCaller_o1);

/* FunctionCaller: '<Root>/DiagnosticMonitorCaller' */
Rte_Call_DiagnosticMonitor_SetEventStatus(DEM_EVENT_STATUS_PASSED);
```

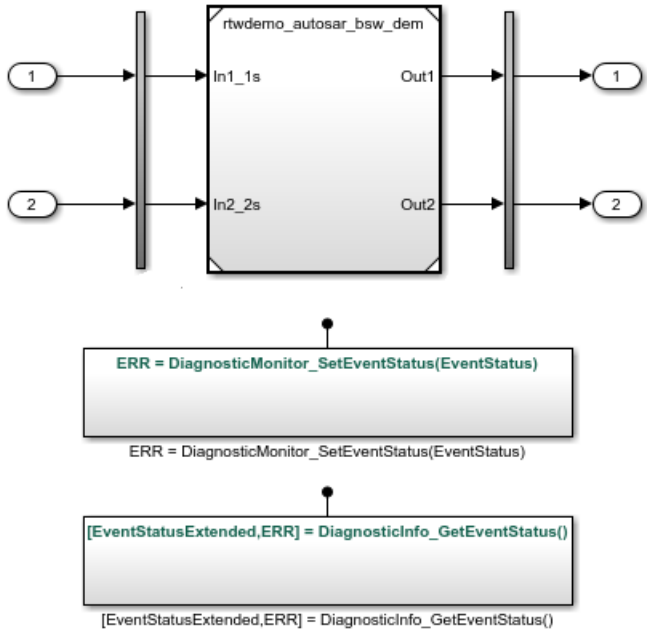
The `EventStatus` argument of operation `SetEventStatus` is of enumerated type `Dem_EventStatusType`, as specified in the DiagnosticMonitorCaller block. To see DEM_EVENT_STATUS_PASSED and the other enumerated values for the argument, search the generated `arxml` code for the string `Dem_EventStatusType`.

Additionally, 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 interfaces called by the component as synchronous 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">/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 If you want to use the model in software-in-the-loop (SIL) simulation, create a test harness. 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 and creates additional ports and stub Simulink Function blocks required for SIL simulation.



See Also

`CallbackEventStatusChangeCaller` | `DiagnosticInfoCaller` | `DiagnosticMonitorCaller`

Related Examples

- "Configure AUTOSAR Client-Server Communication" on page 4-101
- "Verify AUTOSAR C Code with SIL and PIL" on page 5-29

More About

- "Model AUTOSAR Basic Software (BSW) Service Calls" on page 2-22

- "Model AUTOSAR Communication" on page 2-11
- "AUTOSAR Interface Configuration" on page 4-3

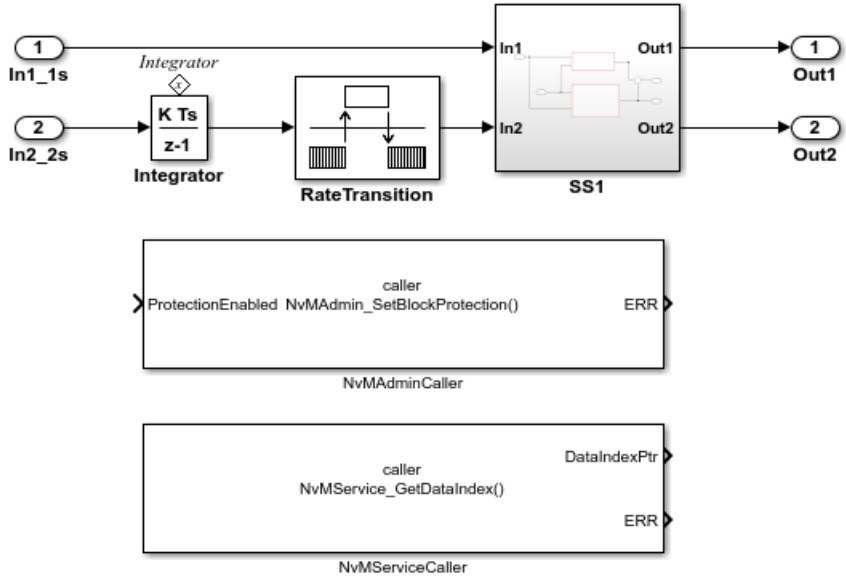
Configure Calls to AUTOSAR NVRAM Manager (NvM) Service

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.

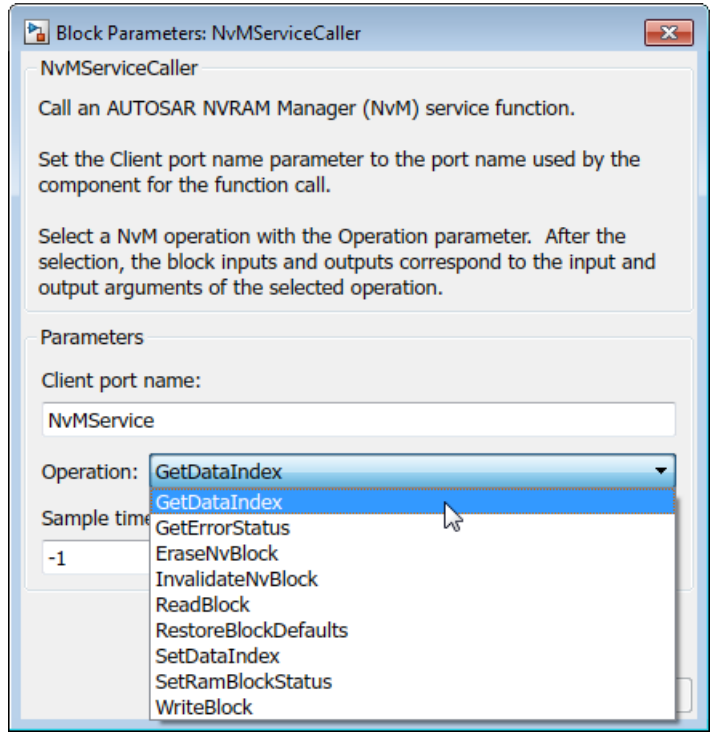
To support system-level modeling of AUTOSAR components and services, Embedded Coder Support Package for AUTOSAR Standard provides an AUTOSAR Basic Software block library. The library contains preconfigured Function Caller blocks for modeling component calls to AUTOSAR BSW services. For information about using the blocks to model client calls to AUTOSAR BSW service interfaces, see “Model AUTOSAR Basic Software (BSW) Service Calls” on page 2-22.

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 `rtwdemo_autosar_swc`.



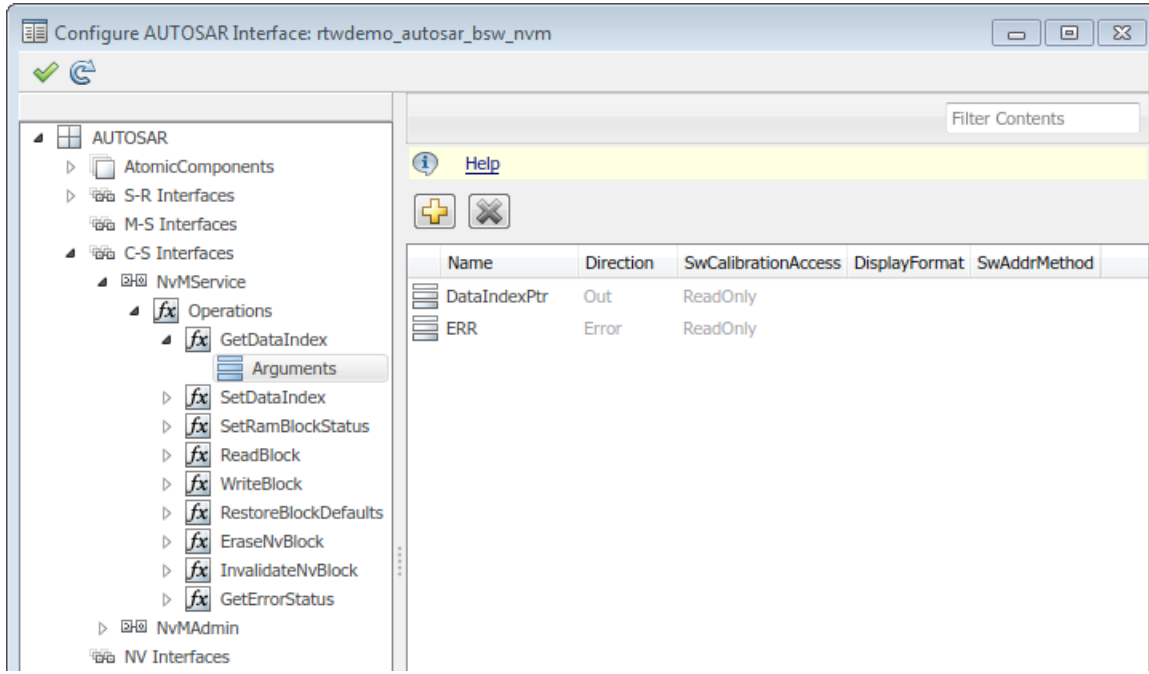
2 Open each block and examine the parameters, especially **Operation**. If you select a different operation and click **Apply**, the software updates the block inputs and outputs to match the arguments of the selected operation.



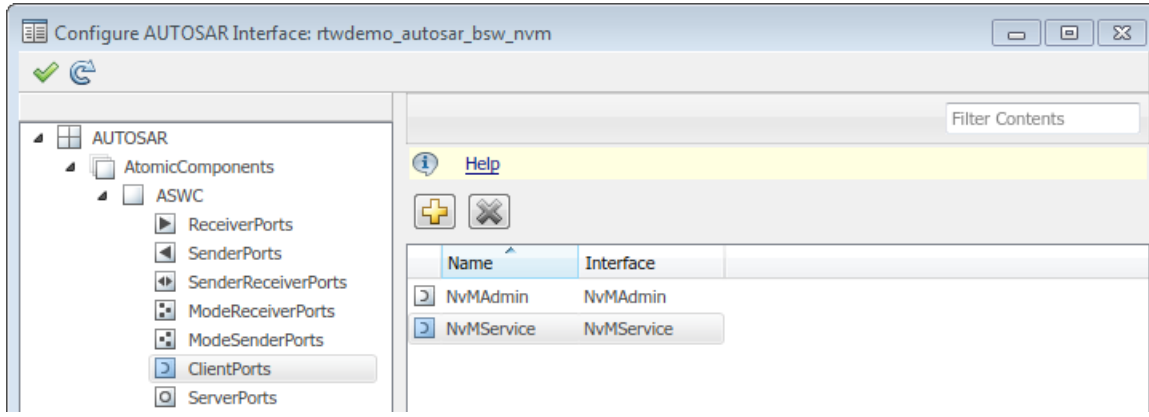
3 Open the Configure AUTOSAR Interface dialog box. Click the **Synchronize** icon to update the Simulink to AUTOSAR mapping of the model with changes to Simulink function callers. The software creates AUTOSAR client-service interfaces, operations, and ports, and maps each Simulink function caller to an AUTOSAR client port and operation.

For example, for the NvMServiceCaller block in this example, for which the `GetDataIndex` operation is selected:

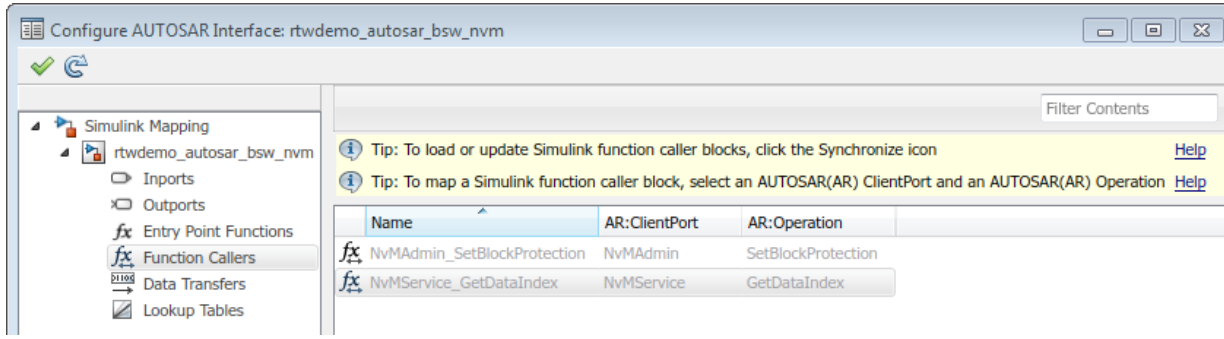
- The software creates C-S interface `NvMService`, and under `NvMService`, all of its supported operations. For each operation, arguments are provided with read-only properties. Here are the arguments for the `NvMService` operation `GetDataIndex`.



- The software creates a client port with the default name `NvMService`. Unlike the C-S-interface, operation, and argument names, the client port name can be customized. The client port is mapped to the `NvMService` interface.



- The software maps the `NvMService` function caller block to AUTOSAR client port `NvMService` and AUTOSAR operation `GetDataIndex`.



4 Optionally, build your model and examine the generated C and `arxml` code. The C code includes the client calls to the BSW services, for example:

```
/* FunctionCaller: '<Root>/NvMAdminCaller' */
Rte_Call_NvMAdmin_SetBlockProtection(false);

/* FunctionCaller: '<Root>/NvMServiceCaller' */
Rte_Call_NvMService_GetDataIndex(&rtb_NvMServiceCaller_o1);
```

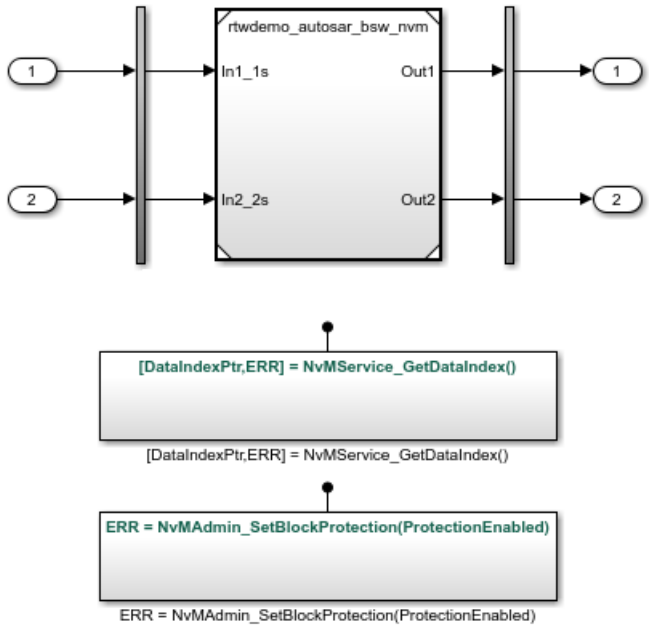
Additionally, generated RTE include files define the server operation call points, such as `Rte_Call_NvMService_GetDataIndex`.

The `arxml` code defines the BSW service interfaces called by the component as synchronous server call points, for example:

```
<SERVER-CALL-POINTS>
...
    <SYNCHRONOUS-SERVER-CALL-POINT UUID="...">
        <SHORT-NAME>SC_NvMService_GetDataIndex</SHORT-NAME>
        <OPERATION-IREF>
            <CONTEXT-R-PORT-REF DEST="R-PORT-PROTOTYPE">/ASWC/NvMService</CONTEXT-R-PORT-REF>
            <TARGET-REQUIRED-OPERATION-REF DEST="CLIENT-SERVER-OPERATION">
              /AUTOSAR/Services/NvM/NvMService/GetDataIndex</TARGET-REQUIRED-OPERATION-REF>
        </OPERATION-IREF>
        <TIMEOUT>1.0E-06</TIMEOUT>
    </SYNCHRONOUS-SERVER-CALL-POINT>
</SERVER-CALL-POINTS>
```

5 If you want to use the model in software-in-the-loop (SIL) simulation, create a test harness. 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 and creates additional ports and stub Simulink Function blocks required for SIL simulation.



See Also

`NvMAdminCaller` | `NvMServiceCaller`

Related Examples

- "Configure AUTOSAR Client-Server Communication" on page 4-101
- "Verify AUTOSAR C Code with SIL and PIL" on page 5-29

More About

- "Model AUTOSAR Basic Software (BSW) Service Calls" on page 2-22
- "Model AUTOSAR Communication" on page 2-11
- "AUTOSAR Interface Configuration" on page 4-3

Configure AUTOSAR Internal Calibration Parameters

You can specify the type of calibration parameter that you export by configuring properties of the corresponding block parameter in the base workspace.

To configure calibration parameters that belong to a calibration component (`ParameterSwComponent`), which can be accessed by multiple AUTOSAR software components, see "Configure AUTOSAR Calibration Component" on page 4-170.

To configure an internal calibration parameter, defined and accessed by only one AUTOSAR software component:

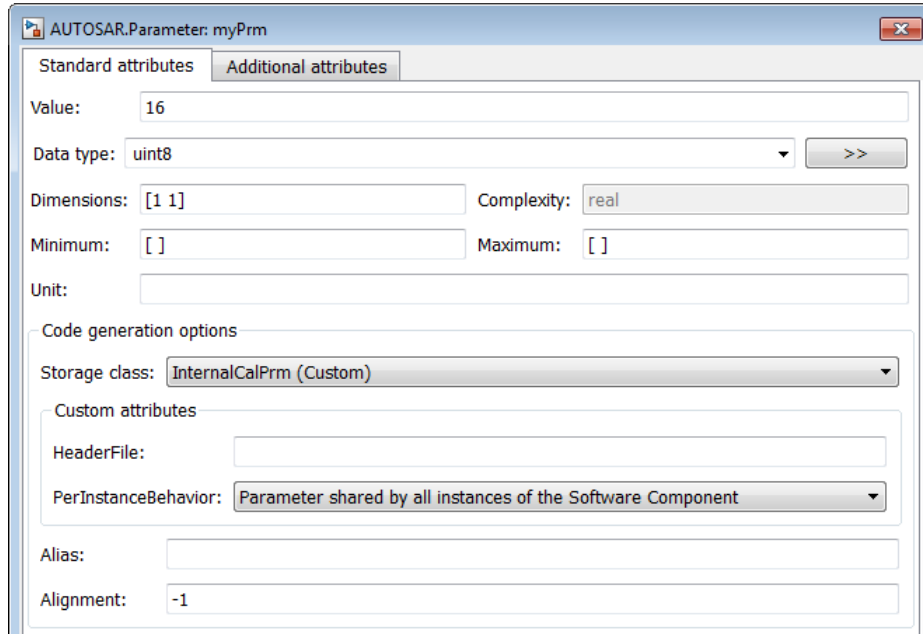
1. In the base workspace, create an `AUTOSAR.Parameter` object. For example:

   ```
   myPrm = AUTOSAR.Parameter;
   ```

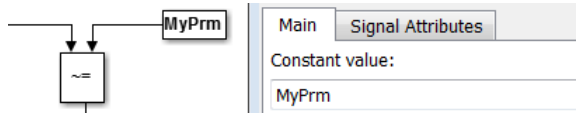
2. In the base workspace, double-click the object to open its dialog box. Configure the following properties of this data object:

   - **Value** — Specify a value for the calibration parameter. For an internal calibration parameter, this value represents the initial value.
   - **Data type** — Specify a data type for the calibration parameter. For more information, see "Specify Data Types Using Data Type Assistant" in the Simulink documentation.
   - **Storage class** — To specify an internal calibration parameter, from the drop-down list, select `InternalCalPrm`. To specify **Per instance behavior**, select one of the following:

     - `Parameter shared by all instances of the Software Component`
     - `Each instance of the Software Component has its own copy of the parameter`



3 Reference the data object in your model. For example, if you open the example model `rtwdemo_autosar_counter`, you can change the value of the Constant block from `LIMIT` to `myPrm`.



4 Open the Configuration Parameters dialog box and clear the option **All Parameters > Ignore custom storage classes**, if it is not already cleared.

5 Generate code.

Note: The software does not support the use of AUTOSAR calibration parameters within Model blocks.

Configure AUTOSAR Port-Based Calibration Parameters

To map a Simulink lookup table to an AUTOSAR parameter, you create AUTOSAR calibration parameters (`ParameterDataPrototype`s) using the AUTOSAR Properties Explorer or AUTOSAR property functions. You can create either internal AUTOSAR parameters, defined and accessed only within your software component, or port-based AUTOSAR parameters, associated with a port-based parameter interface.

- Internal parameters — In the AUTOSAR Properties Explorer, Parameters view, use the **Add** icon to add a new parameter and configure its properties.
- Port-based parameters — In the AUTOSAR Properties Explorer:

  1. In the Parameter Interfaces view, use the **Add** icon to create a new parameter interface. In the Add interface dialog box, specify the number of data elements to create. Configure the properties for each parameter data element.
  2. In the ParameterReceiverPorts view, use the **Add** icon to add a parameter receiver port. In the Add Port dialog box, specify the parameter interface that you created.

The AUTOSAR parameters that you create subsequently are available for Simulink lookup table mapping, using the Simulink-AUTOSAR Mapping Explorer or AUTOSAR map functions.

Related Examples

- “Configure Receiver for AUTOSAR Parameter Communication” on page 4-144
- “Configure STD_AXIS and COM_AXIS Lookup Tables for AUTOSAR Measurement and Calibration” on page 4-174

Configure AUTOSAR Calibration Component

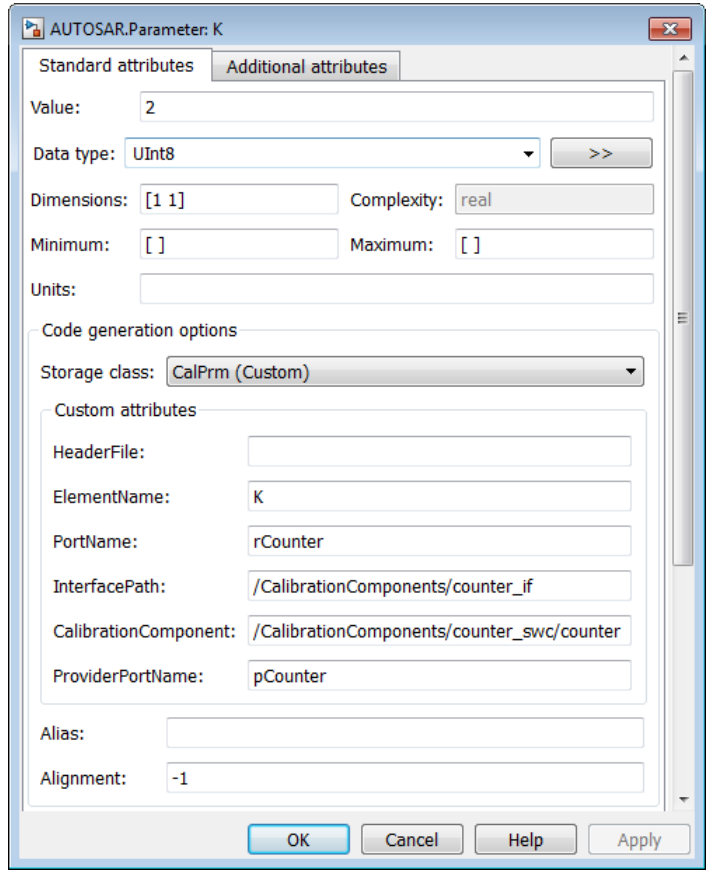
An AUTOSAR calibration parameter component (`ParameterSwComponent`) contains calibration parameters that can be accessed by AUTOSAR software components (SWCs) using an associated provider port. You can import a calibration component from `arxml` code into Simulink or create a calibration component in Simulink.

To create a calibration component in Simulink, open the AUTOSAR parameters in your model and configure them for export in a calibration component. For example:

1. Open a model configured for AUTOSAR that has `AUTOSAR.Parameter` data objects, or to which you can add `AUTOSAR.Parameter` data objects. This procedure uses the example model `rtwdemo_autosar_counter`.
2. Open an AUTOSAR calibration parameter from the workspace or data dictionary. Go to the **Standard attributes** tab of the `AUTOSAR.Parameter` dialog box. Use the following attributes of the `CalPrm` CSC to configure the parameter for export in a calibration component:

   - **CalibrationComponent** — Qualified name of the calibration component to be exported, containing this parameter.
   - **ProviderPortName** — Short name of the provider port associated with the calibration component.

   The following diagram shows the **CalibrationComponent** and **ProviderPortName** values that are specified for the `AUTOSAR.Parameter` data objects used by `rtwdemo_autosar_counter`.



3 Apply any changes and save the model.

When you generate code from the model:

- The software exports the calibration components specified for the AUTOSAR calibration parameters. For example, here is an excerpt of the `ParameterSwComponent` code exported from `rtwdemo_autosar_counter` based on the configuration of the calibration parameter `K`:

```
<AR-PACKAGE UUID="...">
  <SHORT-NAME>counter_swc</SHORT-NAME>
  <ELEMENTS>
    <PARAMETER-SW-COMPONENT-TYPE UUID="...">
      <SHORT-NAME>counter</SHORT-NAME>
```

```
      <PORTS>
        <P-PORT-PROTOTYPE UUID="...">
          <SHORT-NAME>pCounter</SHORT-NAME>
          <PROVIDED-COM-SPECS>
...
            <PARAMETER-PROVIDE-COM-SPEC>
              <INIT-VALUE>
                <CONSTANT-REFERENCE>
                  <SHORT-LABEL>K</SHORT-LABEL>
                  <CONSTANT-REF DEST="CONSTANT-SPECIFICATION">/rtwdemo_autosar_counter_pkg/
                                        rtwdemo_autosar_counter_dt/Ground/K</CONSTANT-REF>
                </CONSTANT-REFERENCE>
              </INIT-VALUE>
              <PARAMETER-REF DEST="PARAMETER-DATA-PROTOTYPE">/CalibrationComponents/
                                                    counter_if/K</PARAMETER-REF>
            </PARAMETER-PROVIDE-COM-SPEC>
...
          </PROVIDED-COM-SPECS>
          <PROVIDED-INTERFACE-TREF DEST="PARAMETER-INTERFACE">/CalibrationComponents/
                                                  counter_if</PROVIDED-INTERFACE-TREF>
        </P-PORT-PROTOTYPE>
      </PORTS>
    </PARAMETER-SW-COMPONENT-TYPE>
  </ELEMENTS>
</AR-PACKAGE>
```

- Parameter initial values are exported on the `ParameterProvideComSpec` in the `ParameterSwComponent` and the `ParameterRequireComSpec` in the `ApplicationSwComponent`. Here is an excerpt of the `ParameterRequireComSpec` code exported from `rtwdemo_autosar_counter`:

```
<R-PORT-PROTOTYPE UUID="...">
  <SHORT-NAME>rCounter</SHORT-NAME>
  <REQUIRED-COM-SPECS>
...
    <PARAMETER-REQUIRE-COM-SPEC>
      <INIT-VALUE>
        <CONSTANT-REFERENCE>
          <SHORT-LABEL>K</SHORT-LABEL>
          <CONSTANT-REF DEST="CONSTANT-SPECIFICATION">/rtwdemo_autosar_counter_pkg/
                                rtwdemo_autosar_counter_dt/Ground/K</CONSTANT-REF>
        </CONSTANT-REFERENCE>
      </INIT-VALUE>
      <PARAMETER-REF DEST="PARAMETER-DATA-PROTOTYPE">/CalibrationComponents/counter_if/
                                                              K</PARAMETER-REF>
    </PARAMETER-REQUIRE-COM-SPEC>
...
  </REQUIRED-COM-SPECS>
  <REQUIRED-INTERFACE-TREF DEST="PARAMETER-INTERFACE">/CalibrationComponents/counter_if
                                                          </REQUIRED-INTERFACE-TREF>
</R-PORT-PROTOTYPE>
```

For calibration component parameters, after you export your AUTOSAR components, you must include your calibration interface definition XML file to import the parameters into an authoring tool.

Note: Use the `CalPrm` CSC attributes **CalibrationComponent** and **ProviderPortName** only to originate a calibration component in Simulink, not for a calibration component originated in an AUTOSAR authoring tool.

Configure STD_AXIS and COM_AXIS Lookup Tables for AUTOSAR Measurement and Calibration

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:

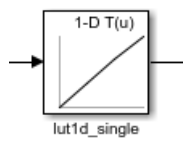
- Implement high-performance search operations.
- Support tuning of the application with measurement and calibration tools.

To model lookup tables for automotive application tuning, use the classes Simulink.LookupTable and Simulink.Breakpoint. With `Simulink.LookupTable` and `Simulink.Breakpoint` objects, you can store and share lookup table and breakpoint data and configure the data for AUTOSAR code generation.

This example shows how to create STD_AXIS and COM_AXIS lookup tables in Simulink, using `Simulink.LookupTable` and `Simulink.Breakpoint` objects, and configure the lookup tables for AUTOSAR code generation.

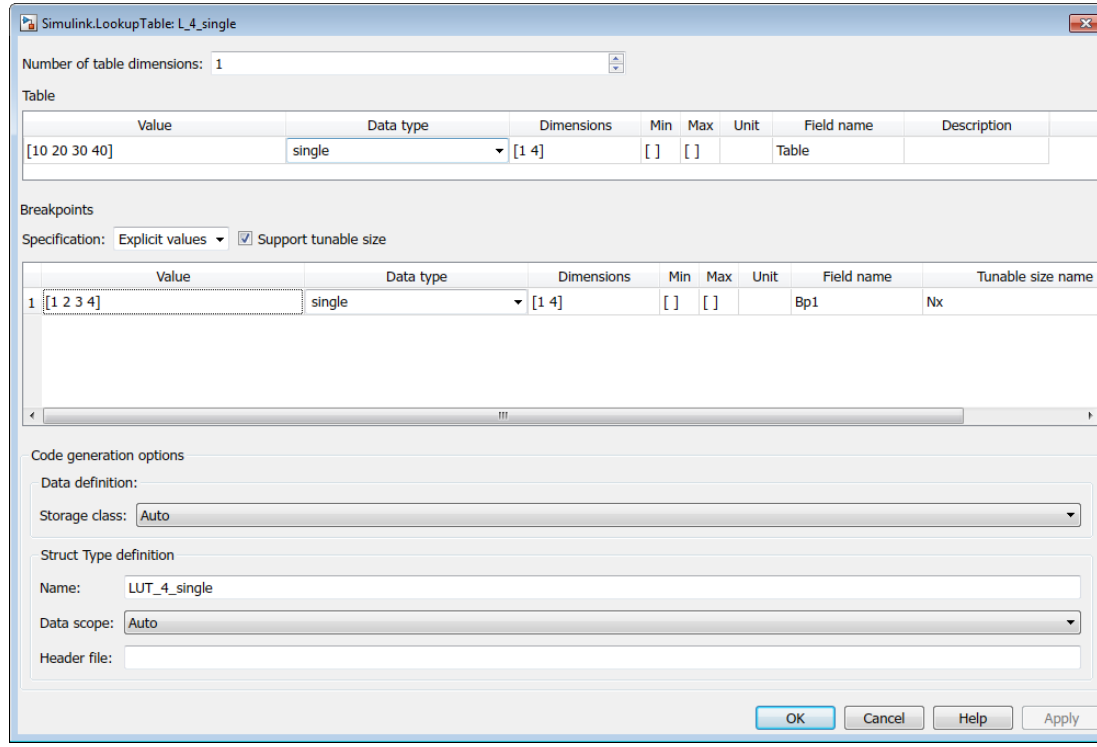
1 Model an AUTOSAR lookup table in a STD_AXIS configuration.

   a Create an n-D Lookup Table block.

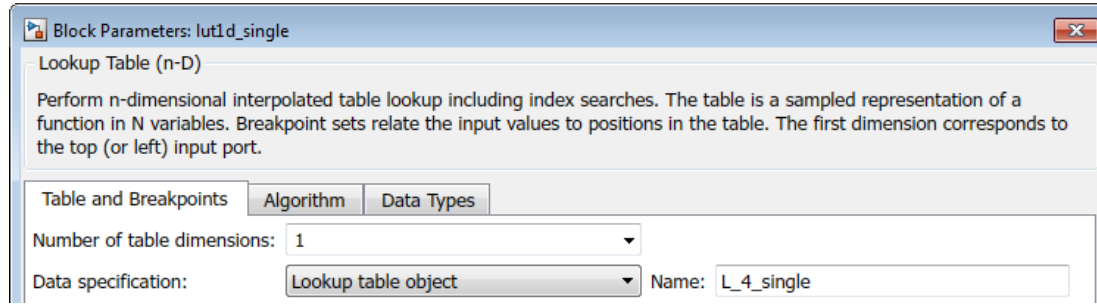


   Open the block and set the **Data specification parameter** parameter to `Lookup table object`.

   b Create a `Simulink.LookupTable` object and configure it to store all of the lookup table data.



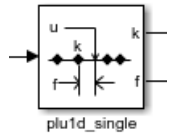
c Use the `Simulink.LookupTable` object name in the n-D Lookup Table block.



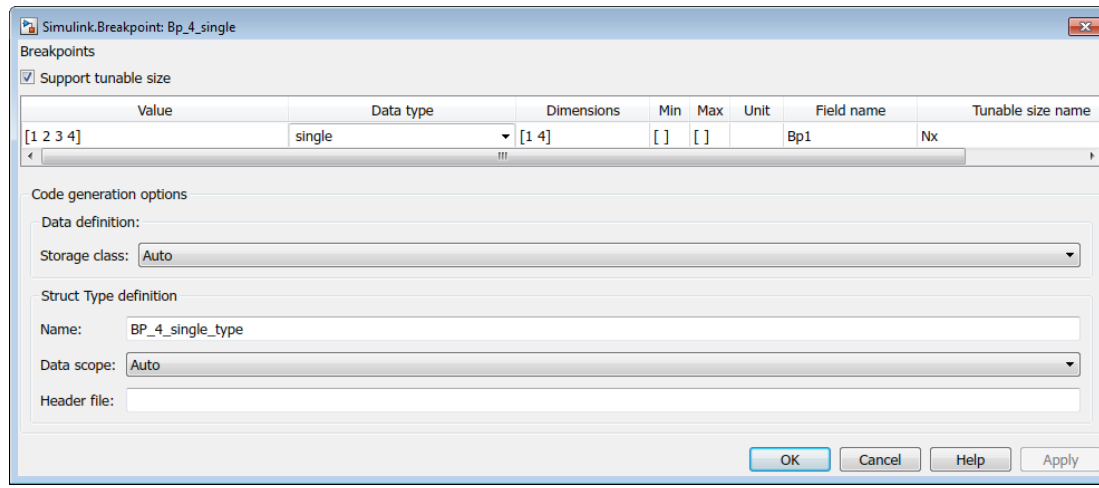
Data appears in the generated C code as fields of a single structure. To control the characteristics of the structure type, such as its name, use the properties of the object.

2 Model an AUTOSAR lookup table in a COM_AXIS configuration.

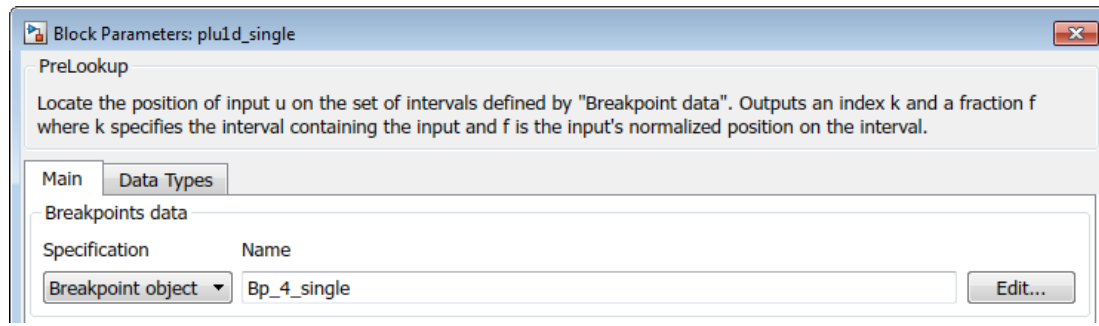
a Create one or more Prelookup blocks.



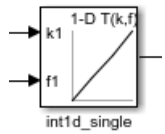
b For each breakpoint vector, create and configure a `Simulink.Breakpoint` object.



c Use each `Simulink.Breakpoint` object name in a `Prelookup` block. You can reduce memory consumption by sharing breakpoint data between lookup tables.

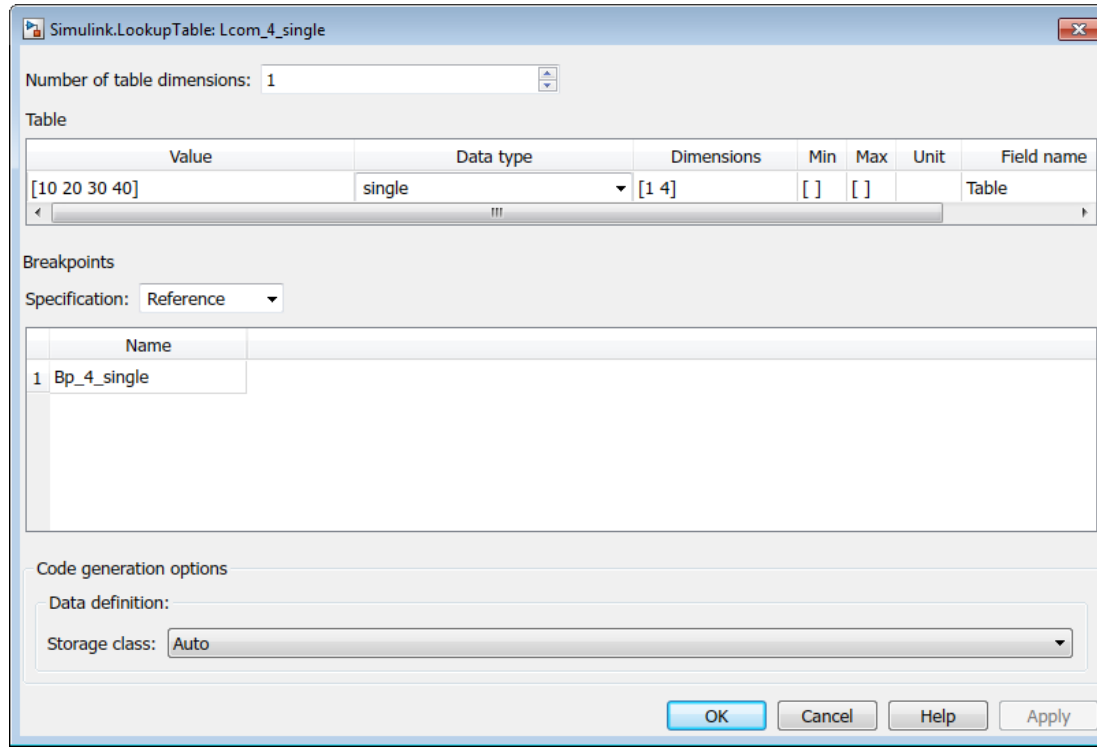


d Create one or more Interpolation Using Prelookup blocks.

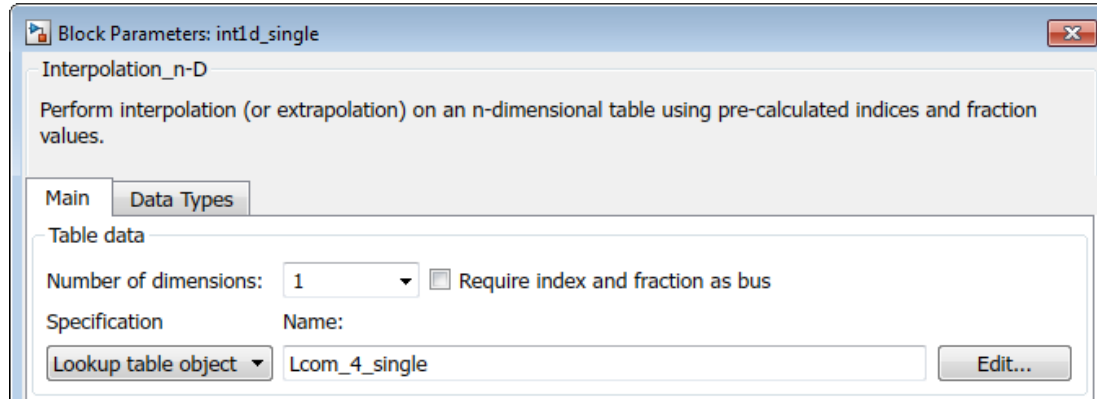


Open each block and set the **Specification** parameter to `Lookup table object`.

e For each set of table data, create and configure a `Simulink.LookupTable` object.

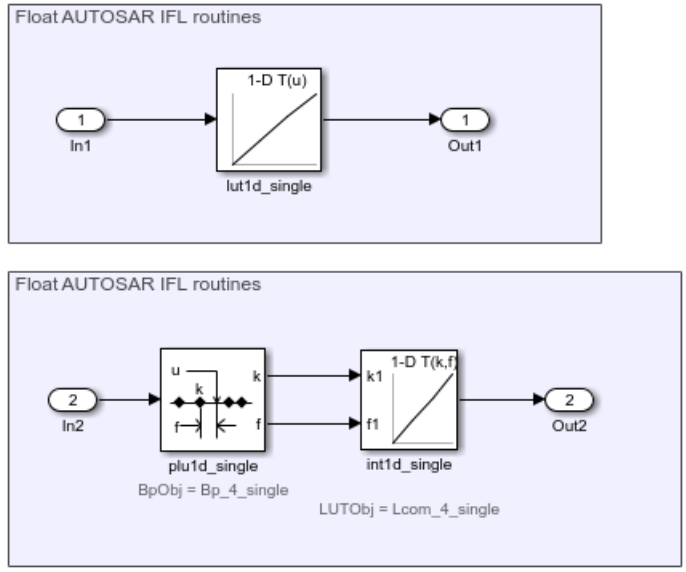


f Use each `Simulink.LookupTable` object name in an Interpolation Using Prelookup block.



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 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.

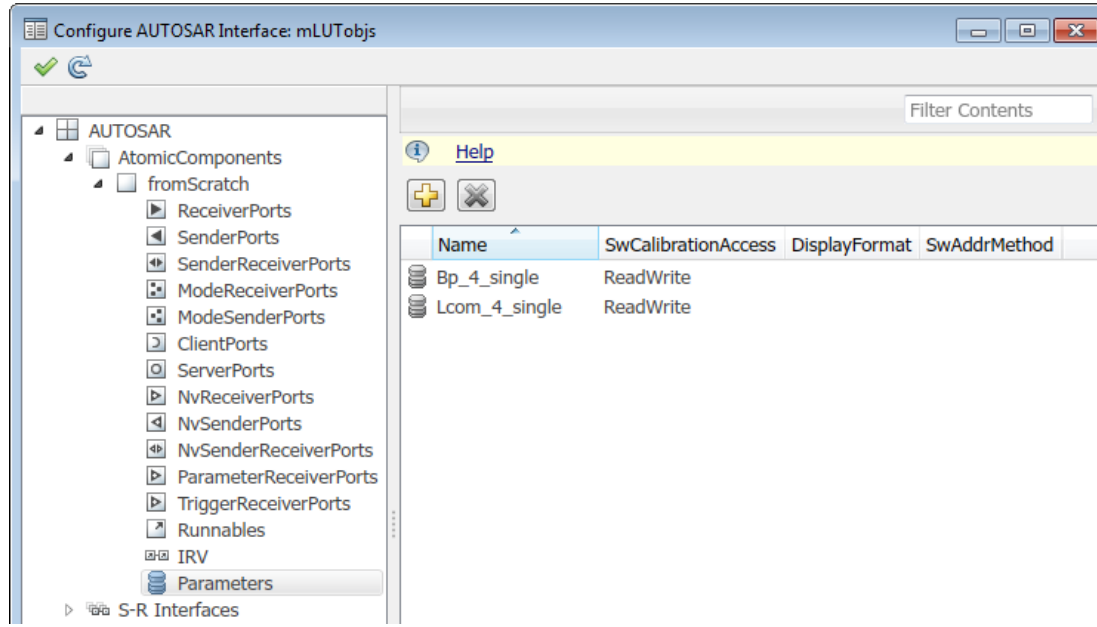
3 Add AUTOSAR operating points to the lookup tables. Connect root level inports to n-D Lookup Table or Prelookup blocks. Alternatively, configure input signals to n-D Lookup Table or Prelookup blocks with static global memory.



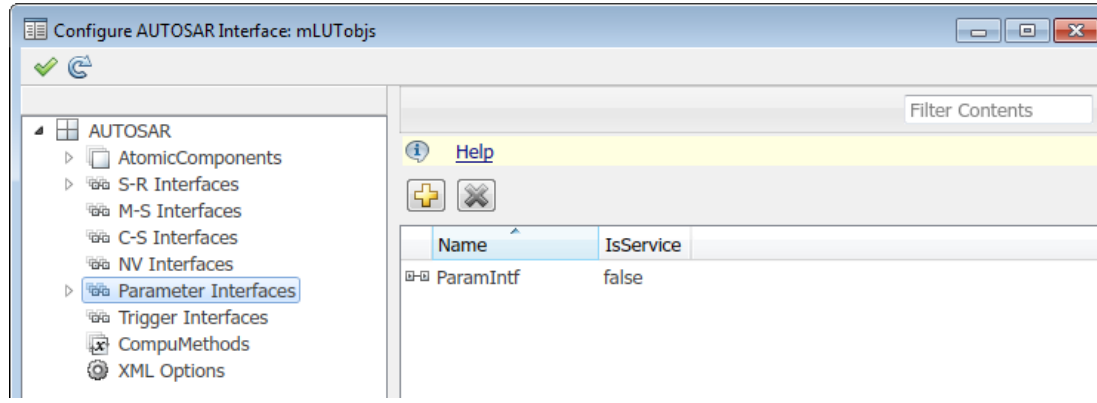
4 Create AUTOSAR parameters corresponding to the `Simulink.LookupTable` and `Simulink.Breakpoint` objects you created. The AUTOSAR parameters can be internal parameters, used only by your AUTOSAR software component, or port-based parameters, maintained and provided by an AUTOSAR parameter software component (`ParameterSwComponent`).

Open the Configure AUTOSAR Interface dialog box and select the AUTOSAR Properties Explorer.

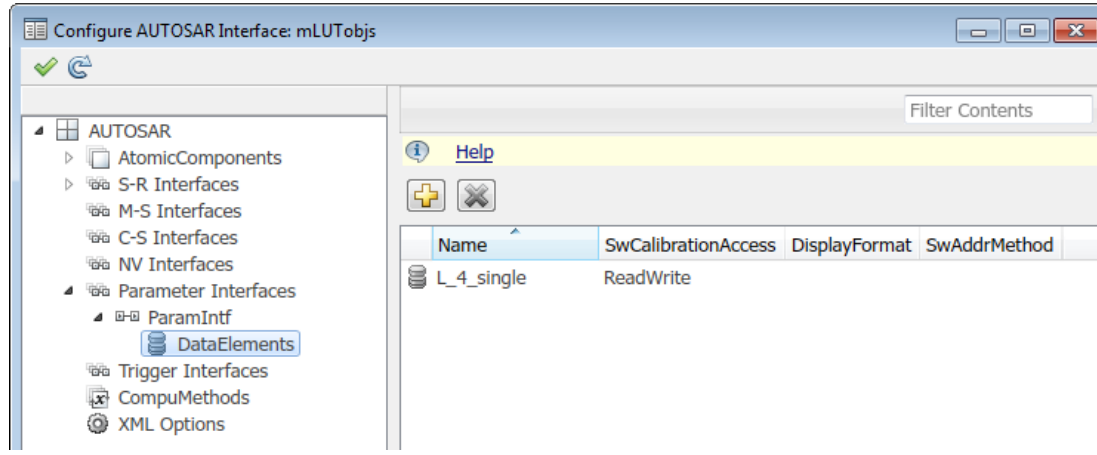
a To create AUTOSAR internal parameters, expand the AtomicComponents node and go to the Parameters view. Use the **Add** icon to add internal parameters to the model. Select each parameter and configure its name and calibration properties.



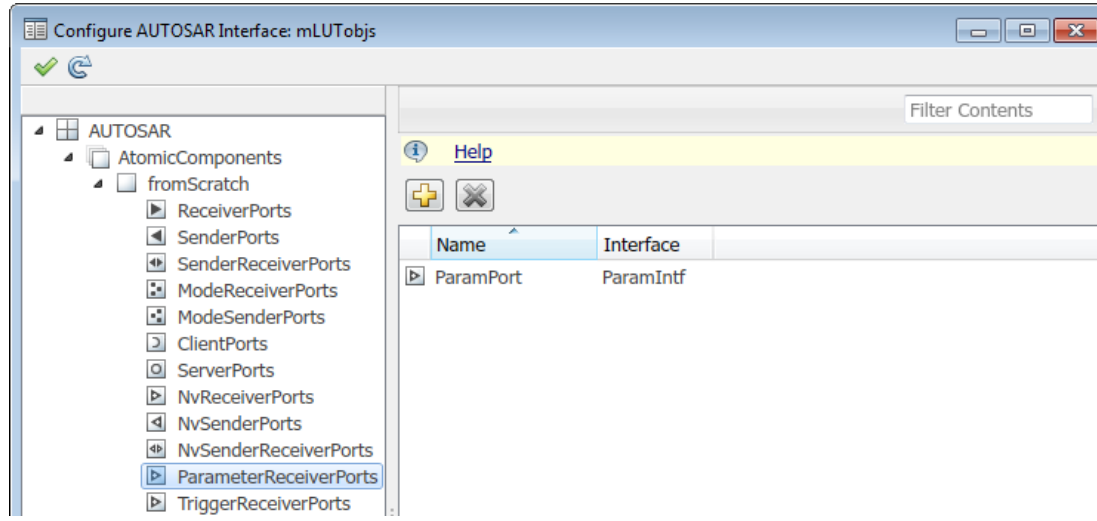
b To create AUTOSAR port-based parameters, create a parameter interface and associated data elements. Go to the Parameter Interfaces view and use the **Add** icon to add a parameter interface to the model. In the Add Interfaces dialog box, specify the number of parameter data elements to create. Click the **Add** button to create the interface.



Open the Parameter Interfaces node and go to the Data Elements view. Select each port-based parameter you created and configure its name and calibration properties.

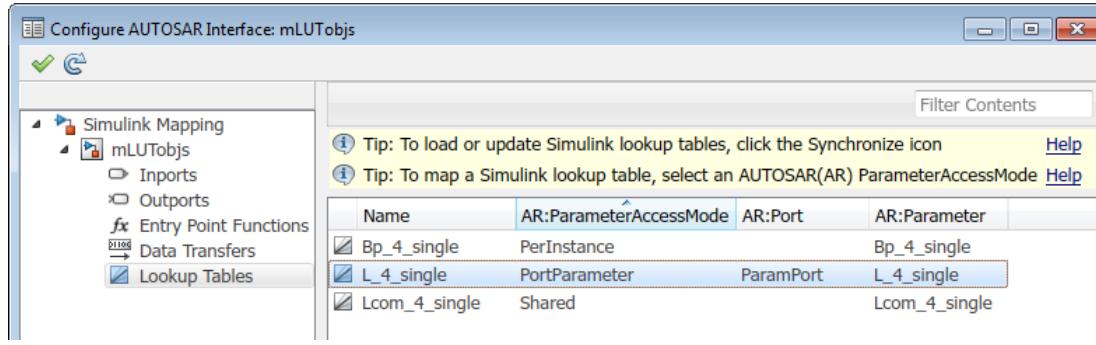


Create a parameter receiver port that references the parameter interface. Expand the AtomicComponents node and go to the Parameter Receiver Ports view. Use the **Add** icon to add a parameter receiver port to the model. In the Add Ports dialog box, select the name of the parameter interface that you created. Click the **Add** button to create the port.



For more information, see “Configure Receiver for AUTOSAR Parameter Communication” on page 4-144.

5 Switch to the Simulink-AUTOSAR Mapping Explorer. In the Lookup Tables view, select each `Simulink.LookupTable` and `Simulink.Breakpoint` object that you created and map it to an AUTOSAR parameter. If the parameter is port-based, specify parameter access mode `PortParameter` and select a parameter interface data element.



In this example:

- The STD_AXIS `Simulink.LookupTable` object `L_4_single` is mapped to an AUTOSAR port-based parameter of the same name.
- The COM_AXIS `Simulink.Breakpoint` object `Bp_4_single` is mapped to an AUTOSAR internal parameter of the same name. Each instance of the AUTOSAR software component has its own copy of the parameter.
- The COM_AXIS `Simulink.LookupTable` object `Lcom_4_single` is mapped to an AUTOSAR internal parameter of the same name. The parameter is shared by all instances of the AUTOSAR software component.

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 Library" on page 5-8.

7 Build the model. The generated C code contains appropriate `Ifl` and `Ifx` lookup function calls and `Rte` data access function calls. The generated `arxml` files contain data types of category CURVE (1-D table data), MAP (2-D table data), and COM_AXIS (axis data), with the data calibration properties that you configured.

See Also

Simulink.LookupTable | Simulink.Breakpoint

Related Examples

- "Configure Receiver for AUTOSAR Parameter Communication" on page 4-144
- "Configure AUTOSAR Port-Based Calibration Parameters" on page 4-169
- "Import AUTOSAR Software Component" on page 3-4
- "Export AUTOSAR Component XML and C Code" on page 5-2

More About

- "Model AUTOSAR Calibration Parameters and Lookup Tables" on page 2-24
- "AUTOSAR Interface Configuration" on page 4-3

Configure COM_AXIS Lookup Table Using `AUTOSAR.Parameter` Objects

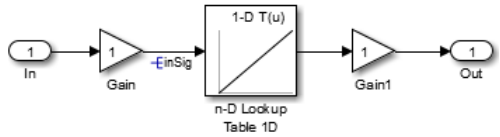
You can model common axis (COM_AXIS) lookup tables for AUTOSAR applications. To model a COM_AXIS lookup table, import COM_AXIS lookup table content from `arxml` files or create a COM_AXIS lookup table in Simulink. For information about the Simulink blocks you use and the high-level workflow, see "Calibration Parameters for STD_AXIS and COM_AXIS Lookup Tables" on page 2-25.

To create and configure a COM_AXIS lookup table:

1. Add an `n-D Lookup Table` block to your model.
2. If AUTOSAR calibration parameters for the lookup table are not already present in your model, create them. They can include:
   - Internal calibration parameters, modeled as described in "Configure AUTOSAR Internal Calibration Parameters" on page 4-167.
   - Calibration parameters in a calibration component, modeled as described in "Configure AUTOSAR Calibration Component" on page 4-170.
   - Constant memory parameters, modeled as described in "Configure AUTOSAR Static or Constant Memory" on page 4-234.
3. Configure the lookup table block for COM_AXIS data. For table data and axis data that you want to tune or manipulate at run time, reference AUTOSAR calibration parameters.
4. To model an AUTOSAR input variable that measures lookup table inputs, do either of the following:
   - Define a static global signal and use it on an input line that connects to a lookup table block input port. Model the signal parameter using an `AUTOSAR4.Signal` data object, as described in "Configure AUTOSAR Static or Constant Memory" on page 4-234.
   - Connect a root level inport to a lookup table block input port.

Note: Consider selecting the AUTOSAR 4.0 code replacement library (CRL) for the model. Selecting the AUTOSAR CRL customizes the C/C++ code generator to produce code that more closely aligns with the AUTOSAR standard. For more information, see "Code Generation with AUTOSAR Library" on page 5-8.

For example, here is a simple model that contains a 1-D lookup table block.



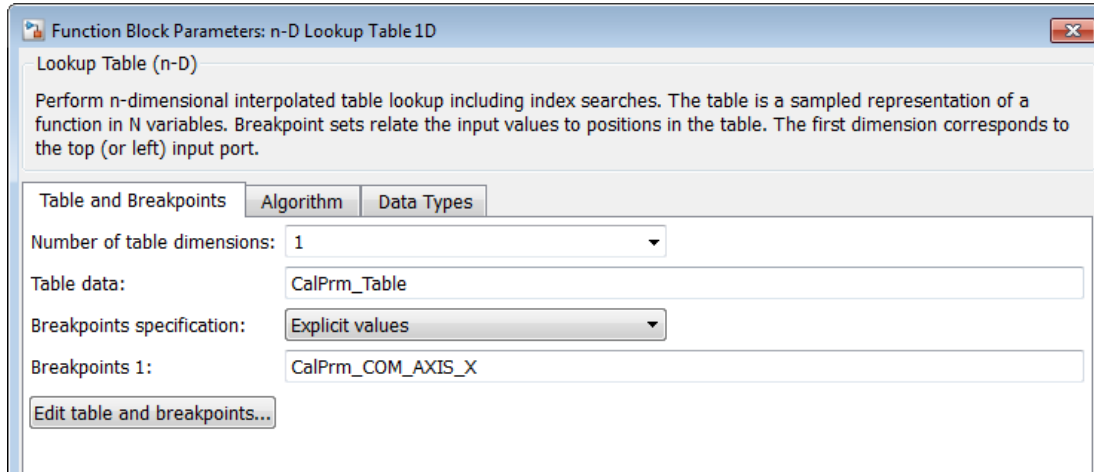
For this model, `AUTOSAR.Parameter` data objects in the workspace define two calibration parameters. `CalPrm_Table` represents table data.

```
CalPrm_Table =

  Parameter with properties:

    SwCalibrationAccess: 'ReadWrite'
          DisplayFormat: ''
                  Value: [4 8 12 16 20]
              CoderInfo: [1x1 Simulink.CoderInfo]
            Description: ''
               DataType: 'int16'
                    Min: 0
                    Max: 100
                   Unit: ''
             Complexity: 'real'
             Dimensions: [1 5]
```

`CalPrm_COM_AXIS_X` represents axis data.

```
CalPrm_COM_AXIS_X =

  Parameter with properties:

    SwCalibrationAccess: 'ReadWrite'
          DisplayFormat: ''
                  Value: [-2 -1 0 1 2]
              CoderInfo: [1x1 Simulink.CoderInfo]
            Description: ''
               DataType: 'fixdt(1,16,0.1,0)'
                    Min: -100
                    Max: 100
                   Unit: ''
             Complexity: 'real'
             Dimensions: [1 5]
```

The lookup table block parameters specify calibration parameter `CalPrm_Table` for table data and calibration parameter `CalPrm_COM_AXIS_X` for axis data.



When you generate code, the `arxml` code for calibration parameter `CalPrm_Table` references application data type `Appl_CalPrm_Table`.

```
<PARAMETER-DATA-PROTOTYPE UUID="...">
    <SHORT-NAME>CalPrm_Table</SHORT-NAME>
    <CATEGORY>CURVE</CATEGORY>
    ...
    <TYPE-TREF DEST="APPLICATION-PRIMITIVE-DATA-TYPE">/MW/ApplDataTypes/Appl_CalPrm_Table</TYPE-TREF>
    ...
</PARAMETER-DATA-PROTOTYPE>
```

The category generated for application data type `Appl_CalPrm_Table` is CURVE, reflecting a 1-D table. `Appl_CalPrm_Table` in turns references application data type `Appl_CalPrm_COM_AXIS_X` (axis data) and a software record layout, `RL_Appl_CalPrm_Table`.

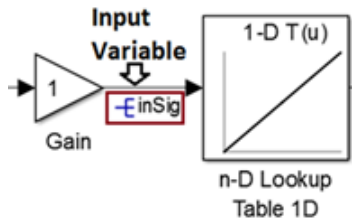
```
<APPLICATION-PRIMITIVE-DATA-TYPE UUID="...">
    <SHORT-NAME>Appl_CalPrm_Table</SHORT-NAME>
    <CATEGORY>CURVE</CATEGORY>
    <SW-DATA-DEF-PROPS>
        <SW-DATA-DEF-PROPS-VARIANTS>
            <SW-DATA-DEF-PROPS-CONDITIONAL>
                <SW-CALIBRATION-ACCESS>READ-WRITE</SW-CALIBRATION-ACCESS>
                <SW-CALPRM-AXIS-SET>
                    <SW-CALPRM-AXIS>
                        <SW-AXIS-INDEX>1</SW-AXIS-INDEX>
                        <CATEGORY>COM_AXIS</CATEGORY>
                        <SW-AXIS-GROUPED>
                            <SHARED-AXIS-TYPE-REF DEST="APPLICATION-PRIMITIVE-DATA-TYPE">
                              /MW/ApplDataTypes/Appl_CalPrm_COM_AXIS_X</SHARED-AXIS-TYPE-REF>
                        </SW-AXIS-GROUPED>
```

```
                </SW-CALPRM-AXIS>
              </SW-CALPRM-AXIS-SET>
              ...
              <SW-RECORD-LAYOUT-REF DEST="SW-RECORD-LAYOUT">/MW/RecordLayouts/
                                    RL_Appl_CalPrm_Table</SW-RECORD-LAYOUT-REF>
          </SW-DATA-DEF-PROPS-CONDITIONAL>
      </SW-DATA-DEF-PROPS-VARIANTS>
  </SW-DATA-DEF-PROPS>
</APPLICATION-PRIMITIVE-DATA-TYPE>
```

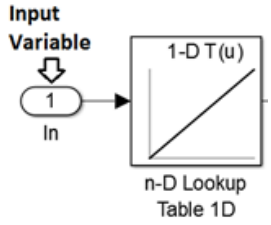
To configure an AUTOSAR input variable that measures a lookup table input in your model, do either of the following:

- Define a static global signal. Use the static global signal on an input line that connects to a lookup table block input port. Model the signal parameter using an `AUTOSAR4.Signal` data object, as described in "Configure AUTOSAR Static or Constant Memory" on page 4-234. For example:

  ```
  inSig =

    Signal with properties:

      SwCalibrationAccess: 'ReadWrite'
            DisplayFormat: ''
                CoderInfo: [1x1 Simulink.CoderInfo]
              Description: ''
                 DataType: 'fixdt(1,16,0.1,0)'
                      Min: []
                      Max: []
                     Unit: ''
               Dimensions: -1
          DimensionsMode: 'auto'
              Complexity: 'auto'
              SampleTime: -1
            InitialValue: ''
  ```

- Connect a root level inport directly to a lookup table block input port. For example:



When you export `arxml` code for a model with an input variable, the parameter access for the corresponding calibration parameter makes the corresponding input variable reference. That is, `SwDataDefProps` contains a reference to one of the following:

- A static memory variable corresponding to a static global signal in the model.
- An S-R interface data element corresponding to a root-level inport in the model.

For example, if you generate code for the static global signal example, the parameter access for calibration parameter `CalPrm_COM_AXIS_X` describes an input variable corresponding to global signal `inSig`.

```
<PARAMETER-ACCESS UUID="...">
    <SHORT-NAME>PICALPRM_CalPrm_COM_AXIS_X</SHORT-NAME>
    <ACCESSED-PARAMETER>
        <LOCAL-PARAMETER-REF DEST="PARAMETER-DATA-PROTOTYPE">/MW/SwCompTypes/ASWC/
                                  IB_CompA/CalPrm_COM_AXIS_X</LOCAL-PARAMETER-REF>
    </ACCESSED-PARAMETER>
    <SW-DATA-DEF-PROPS>
        ...
                <AUTOSAR-VARIABLE>
                    <LOCAL-VARIABLE-REF DEST="VARIABLE-DATA-PROTOTYPE">
                      /MW/SwCompTypes/ASWC/IB_CompA/inSig</LOCAL-VARIABLE-REF>
                </AUTOSAR-VARIABLE>
        ...
    </SW-DATA-DEF-PROPS>
</PARAMETER-ACCESS>
...
<STATIC-MEMORYS>
    <VARIABLE-DATA-PROTOTYPE UUID="...">
        <SHORT-NAME>inSig</SHORT-NAME>
        <CATEGORY>VALUE</CATEGORY>
        ...
        <TYPE-TREF DEST="APPLICATION-PRIMITIVE-DATA-TYPE">/MW/ApplDataTypes/SFIX16_SP1</TYPE-TREF>
    </VARIABLE-DATA-PROTOTYPE>
    ...
</STATIC-MEMORYS>
```

See Also

`AUTOSAR.Parameter` | `n-D Lookup Table`

Related Examples

- "Import AUTOSAR Software Component" on page 3-4
- "Model AUTOSAR Calibration Parameters and Lookup Tables" on page 2-24
- "Configure AUTOSAR Internal Calibration Parameters" on page 4-167
- "Configure AUTOSAR Calibration Component" on page 4-170
- "Configure AUTOSAR Static or Constant Memory" on page 4-234
- "Code Generation with AUTOSAR Library" on page 5-8
- "Export AUTOSAR Component XML and C Code" on page 5-2

More About

- "Calibration Parameters for STD_AXIS and COM_AXIS Lookup Tables" on page 2-25
- "Static and Constant Memory" on page 2-32
- "AUTOSAR Interface Configuration" on page 4-3

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 4-190
"Configure SwCalibrationAccess" on page 4-191
"Configure DisplayFormat" on page 4-194
"Configure SwAddrMethod" on page 4-198
"Configure SwAlignment" on page 4-202
"Export SwImplPolicy" on page 4-203
"Export SwRecordLayout for Lookup Table Data" on page 4-203

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.
- 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 4-191, “Configure DisplayFormat” on page 4-194, “Configure SwAddrMethod” on page 4-198, “Configure SwAlignment” on page 4-202, “Export SwImplPolicy” on page 4-203, and “Export SwRecordLayout for Lookup Table Data” on page 4-203.

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 4-191
- “Specify SwCalibrationAccess for Signal and Parameter Data Objects” on page 4-193
- “Specify Default SwCalibrationAccess for Application Data Types” on page 4-194

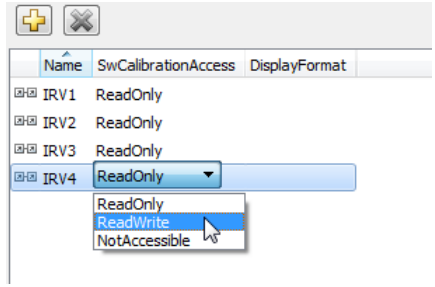
Specify `SwCalibrationAccess` for AUTOSAR Data Elements

You can use either the Configure AUTOSAR Interface dialog box or MATLAB function calls to specify the `SwCalibrationAccess` property for the following AUTOSAR data elements:

- Sender-receiver interface data elements
- Nonvolatile interface data elements
- Client-server arguments
- Inter-runnable variables

For example:

1. Open a model that is configured for AUTOSAR.
2. Open the Configure AUTOSAR Interface dialog box. For example, in the model window, select **Code** > **C/C++ Code** > **Configure Model as AUTOSAR Component**.
3. Select the AUTOSAR Properties Explorer. Navigate to one of the following views:
   - S-R or NV interface, **Data Elements** view
   - C-S interface, **Arguments** view
   - Atomic component, **IRV** view
4. 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 `rtwdemo_autosar_multirunnables` example model and sets measurement and calibration access to inter-runnable variable IRV4 to `ReadWrite`.

```
>> open_system('rtwdemo_autosar_multirunnables')
>> dataobj = autosar.api.getAUTOSARProperties('rtwdemo_autosar_multirunnables');
>> get(dataobj,'/pkg/swc/ASWC/Behavior/IRV4','SwCalibrationAccess')

ans =
ReadOnly

>> set(dataobj,'/pkg/swc/ASWC/Behavior/IRV4','SwCalibrationAccess','ReadWrite');
>> get(dataobj,'/pkg/swc/ASWC/Behavior/IRV4','SwCalibrationAccess')

ans =
ReadWrite

>>
```

Here is a sample call to the AUTOSAR properties `set` function to set `SwCalibrationAccess` for an S-R interface data element:

```
set(dataobj,'/rtwdemo_autosar_counter_pkg/rtwdemo_autosar_counter_if/Input/Input',...
  'SwCalibrationAccess','ReadWrite');
```

Specify `SwCalibrationAccess` for Signal and Parameter Data Objects

You can specify the `SwCalibrationAccess` property for the following AUTOSAR signal and parameter data objects in your model, using either the Property dialog box or MATLAB commands:

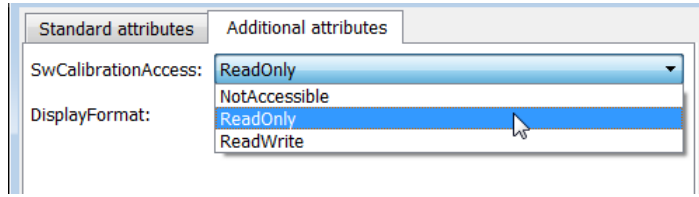
- `AUTOSAR.Signal`
- `AUTOSAR4.Signal`
- `AUTOSAR.Parameter`
- `AUTOSAR4.Parameter`
- AUTOSAR.DualScaledParameter

For example:

1. Open MATLAB.
2. Create a signal or parameter data object with a command like the following:

```
>> arSig=AUTOSAR.Signal

arSig =

  Signal with properties:

    SwCalibrationAccess: 'ReadOnly'
              CoderInfo: [1x1 Simulink.CoderInfo]
            Description: ''
               DataType: 'auto'
                    Min: []
                    Max: []
                   Unit: ''
             Dimensions: -1
         DimensionsMode: 'auto'
             Complexity: 'auto'
             SampleTime: -1
           InitialValue: ''

>>
```

3. Open the data object, for example, by double-clicking the object in the workspace.
4. Select the **Additional attributes** tab, or for `AUTOSAR.DualScaledParameter`, the **Calibration Attributes** tab. Use the **SwCalibrationAccess** drop-down list to select the level of measurement and calibration tool access to allow for the data object.



Alternatively, you can access and modify the `SwCalibrationAccess` property for AUTOSAR signal or parameter data objects using MATLAB commands. For example:

```
O2Sensor = AUTOSAR.Signal;
O2Sensor.SwCalibrationAccess = 'ReadOnly'
```

Specify Default `SwCalibrationAccess` for Application Data Types

The AUTOSAR XML options include the property `SwCalibrationAccess`, 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 `rtwdemo_autosar_multirunnables` example model and changes the default measurement and calibration access for AUTOSAR application data types from `ReadWrite` to `ReadOnly`.

```
>> open_system('rtwdemo_autosar_multirunnables')
>> dataObj = autosar.api.getAUTOSARProperties('rtwdemo_autosar_multirunnables');
>> get(dataObj,'XmlOptions','SwCalibrationAccessDefault')

ans =
ReadWrite

>> set(dataObj,'XmlOptions','SwCalibrationAccessDefault','ReadOnly');
>> get(dataObj,'XmlOptions','SwCalibrationAccessDefault')

ans =
ReadOnly

>>
```

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 data objects and elements:

- Signal and parameter data objects (AUTOSAR and AUTOSAR4 classes)

- 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.

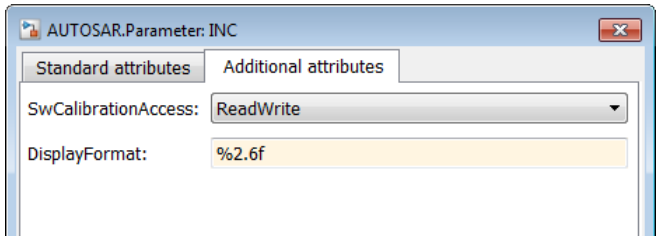
Field	Description

`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 appropriate).

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 the AUTOSAR data objects and elements to which it applies. You can specify display format in a dialog box, or with a data object or element API that can modify attributes.

```
INC = AUTOSAR.Parameter;
INC.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 (`SwAddrMethod`s) to group data in memory for access by measurement and calibration tools. In an AUTOSAR software component configuration, you assign common memory sections to data. When the runtime environment instantiates calibration parameters, calibration parameters that reference the same `SwAddrMethod` are placed within the same calibration parameter group.

To support the round-trip workflow, the `arxml` importer imports and preserves the `SwAddrMethod` property for AUTOSAR data.

In Simulink, you can specify or modify AUTOSAR software address methods for measurement and calibration tools in the following ways:

- "Configure SwAddrMethod for Data Accessed by RTE Function Calls" on page 4-199
- "Edit SwAddrMethod for AUTOSAR Static or Constant Memory" on page 4-200
- "Add, Find, or Set SwAddrMethod for Data" on page 4-201

Configure `SwAddrMethod` for Data Accessed by RTE Function Calls

You can import `SwAddrMethod`s from `arxml` files and apply them to data accessed by RTE function calls. For example, you can apply them to sender-receiver (S-R) interface data elements or inter-runnable variables (IRVs). Follow this general procedure.

1. Import `SwAddrMethod`s from `arxml` files.

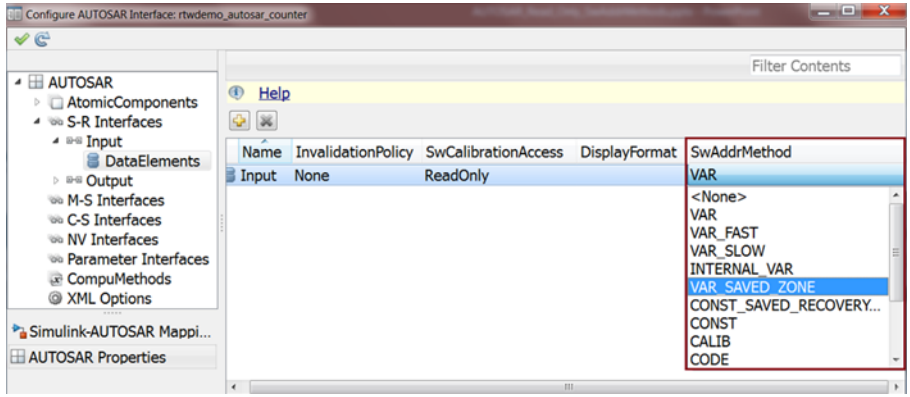
   - If you create your AUTOSAR model from `arxml` files using importer method `createComponentAsModel`, include an `arxml` file that contains `SwAddrMethod` definitions in the import. The imported `SwAddrMethod`s are preserved and later exported in `arxml` code.
   - If you create your AUTOSAR model in Simulink, you can import reference definitions of `SwAddrMethod`s from `arxml` files. You can apply the definitions, on a read-only basis, to data in your AUTOSAR software component. Use importer method `updateReferences`. For example:

     ```
     importerObj = arxml.importer(arxmlFileName);
     updateReferences(importerObj,modelName);
     ```

     When you generate model code, the exported `arxml` code contains references to the imported read-only `SwAddrMethod` elements, but not their definitions.

     For more information, see "Import or Update Shared AUTOSAR Reference Element Definitions" on page 3-22.

2. After importing `SwAddrMethod`s into your model, you can graphically or programmatically select `SwAddrMethod` values for AUTOSAR data accessed by RTE function calls. For example, open the Configure AUTOSAR Interface dialog box and select **AUTOSAR Properties**. Navigate to the graphical view of a data item, such as an IRV, an S-R interface data element, or a client-server (C-S) interface operation argument. Select a data item, and select an available value for the **SwAddrMethod** property.



3 Build the model and search the generated `arxml` code for the value you selected.

```
<SENDER-RECEIVER-INTERFACE>
    <SHORT-NAME>Input</SHORT-NAME>
    <IS-SERVICE>false</IS-SERVICE>
    <DATA-ELEMENTS>
        <VARIABLE-DATA-PROTOTYPE>
            <SHORT-NAME>Input</SHORT-NAME>
            <CATEGORY>VALUE</CATEGORY>
            ...
            <SW-DATA-DEF-PROPS>
                <SW-ADDR-METHOD-REF DEST="SW-ADDR-METHOD">
                    /AUTOSAR/MemMap/SwAddrMethods_Blueprint/VAR_SAVED_ZONE</SW-ADDR-METHOD-REF>
                ...
            </SW-DATA-DEF-PROPS>
            ...
        </VARIABLE-DATA-PROTOTYPE>
    </DATA-ELEMENTS>
</SENDER-RECEIVER-INTERFACE>
```

Edit SwAddrMethod for AUTOSAR Static or Constant Memory

Modeling of AUTOSAR R4.x software address methods for AUTOSAR Static or Constant memory is based on the Embedded Coder memory section mechanism for data objects. To support AUTOSAR needs, you can use the following additional tokens in AUTOSAR memory sections:

- `%<AUTOSAR_COMPONENT>` and `%<MemorySectionName>`

  When specified in the Pre/Post Pragma fields within a Memory-Section definition, C code generation expands these tokens.

- `MySwc_START_SEC_FLASHMEMORY` and `MySwc_STOP_SEC_FLASHMEMORY`

When specified with an include file within an AUTOSAR memory section, C preprocessor stage expands the include file to the corresponding memory section. These tokens allow you to separate platform-independent variable declarations from platform-dependent `#pragma` statements in an include file.

These code examples compare an Embedded Coder memory section with an AUTOSAR memory section. For more information, see "Control Data and Function Placement in Memory by Inserting Pragmas" in the Embedded Coder documentation.

ERT Memory Section

```
#pragma begin_flashsec.bss

const real_T KaGain_HVAC = 4.0;

#pragma end_flashsec.bss
```

AUTOSAR Memory Section

```
#define MySwc_START_SEC_FLASHMEMORY
#include "MySwc_MemMap.h"

const real_T KaGain_HVAC = 4.0;

#define MySwc_STOP_SEC_FLASHMEMORY
#include "MySwc_MemMap.h"
```

Add, Find, or Set `SwAddrMethod` for Data

You can use the following AUTOSAR property functions to add, find, or set `SwAddrMethod` for S-R interface data elements, inter-runnable variables, and runnables.

- `addSwAddrMethod`
- `find`
- `set`

The function `addSwAddrMethod(arProps,qName,property,value)` adds a `SwAddrMethod` with the qualified name `qName` to the AUTOSAR configuration for a model. If property-value pairs are specified, the function sets properties for the `SwAddrMethod`.

The following example adds a `SwAddrMethod` to the AUTOSAR configuration for example model `rtwdemo_autosar_counter`. The `addSwAddrMethod` call specifies qualified name `'/A/B/C/SwAddressMethods/sw1'`, sets

`MemoryAllocationKeywordPolicy` to `'ADDR-METHOD-SHORT-NAME'`, and sets `SectionType` to `'VAR'`:

```
dataObj = autosar.api.getAUTOSARProperties('rtwdemo_autosar_counter');
addSwAddrMethod(dataObj,'/A/B/C/SwAddressMethods/sw1',...
  'MemoryAllocationKeywordPolicy','ADDR-METHOD-SHORT-NAME',...
  'SectionType','VAR');
```

The following example returns the path of a `SwAddrMethod`:

```
swAddrPaths = find(dataObj,[],'SwAddrMethod','PathType',...
  'FullyQualified','SectionType','VAR')
```

The following example sets the path of an existing `SwAddrMethod`:

```
sw1Path = '/A/B/C/SwAddressMethods/sw1';
interfacePath = '/A/B/C/Interfaces/If1/';
dataElementName = 'El1';
set(dataObj,[interfacePath dataElementName],'SwAddrMethod',sw1Path);
swPathGet = get(dataObj,[interfacePath dataElementName],'SwAddrMethod',...
  'PathType','FullyQualified');
```

Configure `SwAlignment`

The `SwAlignment` property describes the intended alignment of AUTOSAR data objects 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 object, 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 method `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 reference definitions of `SwRecordLayouts` from `arxml` files. Use importer method `updateReferences`. For example:

```
importerObj = arxml.importer(arxmlFileName);
updateReferences(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 “Import or Update Shared AUTOSAR Reference Element Definitions” on page 3-22.

See Also

AUTOSAR.DualScaledParameter | `AUTOSAR.Parameter` | `AUTOSAR.Signal` | `AUTOSAR4.Parameter` | `AUTOSAR4.Signal`

Related Examples

- “Import AUTOSAR Software Component” on page 3-4
- “Model AUTOSAR Calibration Parameters and Lookup Tables” on page 2-24
- “Configure AUTOSAR CompuMethods” on page 4-241
- “Export AUTOSAR Component XML and C Code” on page 5-2

More About

- “AUTOSAR Interface Configuration” on page 4-3

Configure AUTOSAR Multiple Runnables

Simulink supports the following modeling styles for implementing an AUTOSAR software component with multiple runnables:

- Rate-based multitasking modeling style — Model multiple runnables by creating or importing model content with multiple periodic rates. Select this modeling style if you want multiple runnable entities with rate-based multitasking execution semantics.
- Function-call-based modeling style — Model multiple runnables using function-call subsystems — or (for client-server modeling) `Simulink Function` blocks — at the top level of a model. Select this modeling style if you want multiple runnable entities with function-call execution semantics.

For more information about the modeling styles, see “Multi-Runnable Software Components” on page 2-5 and “Model for AUTOSAR Platform”.

This topic provides examples of configuring AUTOSAR multi-runnables in each modeling style.

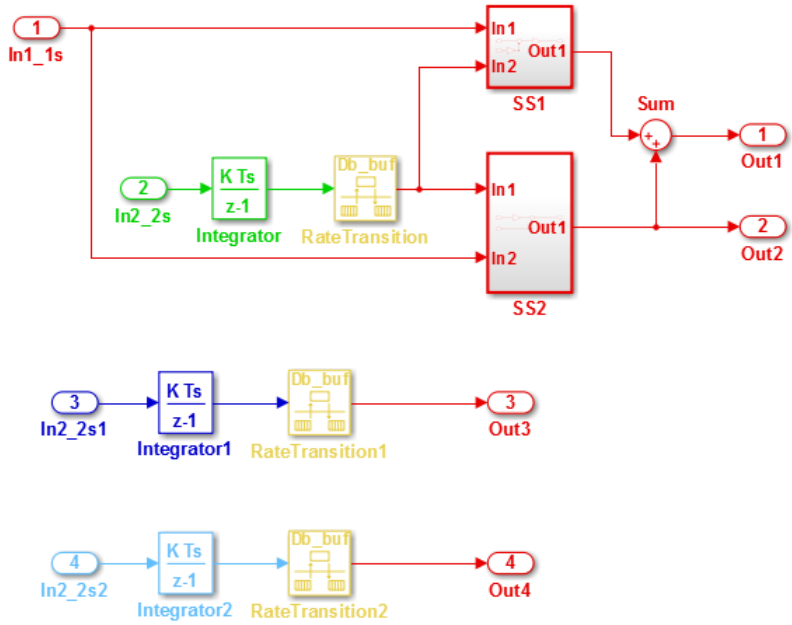
In this section...

“Configure Multiple Runnables Using Rate-Based Multitasking” on page 4-205
“Configure Multiple Runnables Using Function Calls” on page 4-211

Configure Multiple Runnables Using Rate-Based Multitasking

You can model AUTOSAR multi-runnables using Simulink rate-based, multitasking. You can create multiple periodic runnables in Simulink, import multiple periodic runnables from `arxml` files into Simulink, or migrate an existing rate-based, multitasking model to the AUTOSAR system target file.

Here is an example of a rate-based, multitasking model. The colors displayed when you update the model (if **Display > Sample Time > Colors** is selected) represent the different periodic rates present.



From an AUTOSAR perspective, the model represents an AUTOSAR software component. Root model inports and outports represent AUTOSAR ports, and `Rate Transition` blocks that handle data transfers between blocks operating at different rates represent AUTOSAR inter-runnable variables (IRVs).

When you generate code from the model, the exported `arxml` code describes the periodic runnables in the AUTOSAR software component. The generated model C code contains rate-grouped model step functions corresponding to AUTOSAR runnables, one for each discrete rate in the model. The C code also contains corresponding AUTOSAR RTE API calls. (The periodic step functions must be called in the manner of a rate-monotonic scheduler.)

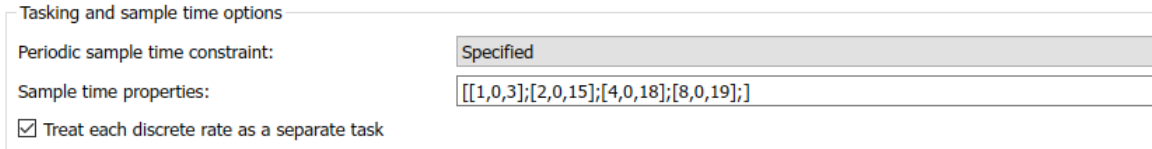
If you configure the component to be multi-instantiable, the generated step functions and RTE API calls have an instance handle as an argument. For more information, see “Multi-Instance Software Components” on page 2-4.

This example shows the general workflow for configuring a rate-based, multitasking Simulink model as an AUTOSAR software component.

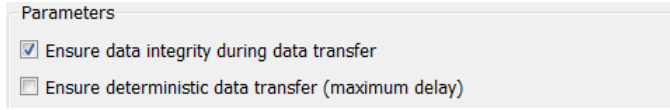
Note: The illustrations and code excerpts in this example use the preconfigured model *matlabroot*`/help/toolbox/ecoder/examples/autosar/mMultitasking_4rates.slx`. If you prefer to follow the example with a model not already configured for the AUTOSAR system target file, you can use the example model `rtwdemo_mrmtbb`.

1. Configure a Simulink model to generate multitasking code. The minimum requirements are:

   - The model must use multiple rates.
   - In the Solver pane of the Configuration Parameters dialog box, select the option to **Treat each discrete rate as a separate task** to enable the multitasking mode.



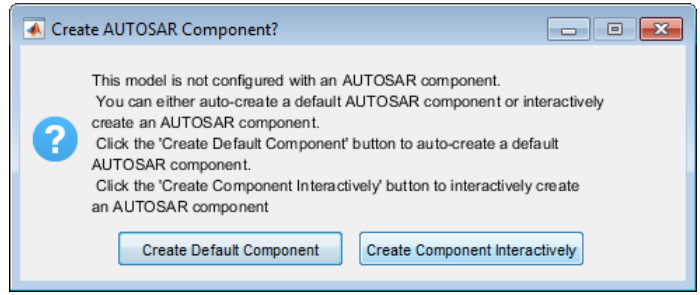
   - To allow mapping of Rate Transition blocks to AUTOSAR IRVs, Rate Transition blocks in the model must have block option **Ensure data integrity during data transfer** selected and block option **Ensure deterministic data transfer** cleared.



2. If the model is not already configured for the AUTOSAR system target file, in the Configuration Parameters dialog box, **Code Generation** pane, change the model **System target file** to `autosar.tlc`.
3. Create AUTOSAR elements and map them to Simulink elements.

   a. In the model window, select **Code > C/C++ Code > Configure Model as AUTOSAR Component**.

   b. If the model is not already configured with an AUTOSAR interface, your selection opens the **Create AUTOSAR Component?** dialog box.



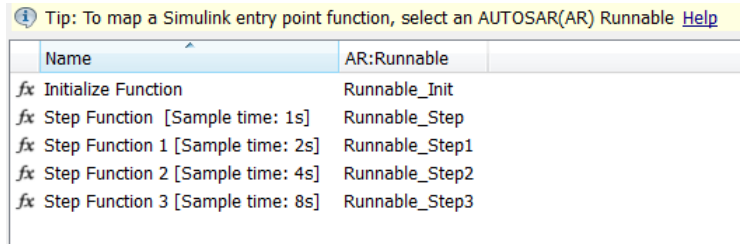
The dialog box offers two paths for creating an AUTOSAR software component. Choose **Create Default Component**. As part of creating a default AUTOSAR component, the software creates corresponding AUTOSAR elements, such as AUTOSAR ports, runnables, and inter-runnable variables (IRVs). The software maps the AUTOSAR elements to Simulink elements, such as inports, outports, subsystems, data transfer lines, and Rate Transition blocks.

After you click **Create Default Component**, a new AUTOSAR component opens in the Configure AUTOSAR Interface dialog box.

Note: If you manually create and map elements, click the **Synchronize** icon in the Configure AUTOSAR Interface dialog box. This action triggers a Simulink update diagram, which allows the mapping of AUTOSAR elements to Simulink elements, such as IRVs to Rate Transition blocks, to be resolved. As previously noted, to allow mapping of IRVs to Rate Transition blocks, Rate Transition blocks in the model must have block option **Ensure data integrity during data transfer** selected and block option **Ensure deterministic data transfer** cleared.

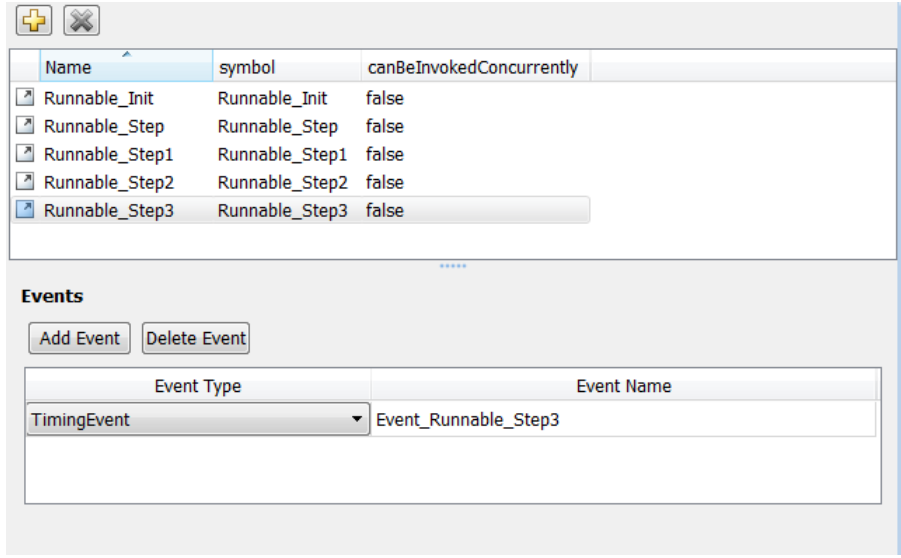
---

4 In the Configure AUTOSAR Interface dialog box, verify the mapping of AUTOSAR periodic runnables to Simulink periodic step functions. Select the Simulink-AUTOSAR Mapping Explorer, select the **Entry Point Functions** view, and examine the periodic step functions mapped by the component builder software.



If a step function is not yet mapped to an AUTOSAR runnable, the **AR:Runnable** property displays `Select`.

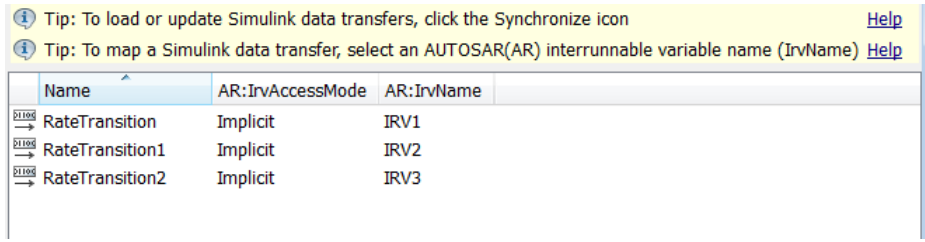
Select the AUTOSAR Properties Explorer. In the **Runnables** view, examine the AUTOSAR periodic runnables created and mapped by the component builder software. Each runnable must have a timing event.



If you previously found, in the **Entry Point Functions** view, that a Simulink periodic step function was not mapped to an AUTOSAR runnable, use the **Runnables** view to add a runnable. Also, create a timing event for the runnable. Then return to the Simulink-AUTOSAR Mapping Explorer, **Entry Point**

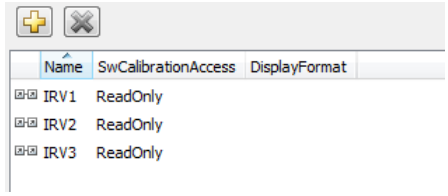
Functions view, and use **AR:Runnable** to map the step function to the corresponding runnable.

5 Verify the mapping of AUTOSAR IRVs to Simulink rate transitions. Select the Simulink-AUTOSAR Mapping Explorer, select the **Data Transfers** view, and examine the rate transitions mapped by the component builder software.



If a rate transition is not yet mapped to an AUTOSAR IRV, the **AR:IrvName** property displays `Select`.

Select the AUTOSAR Properties Explorer. In the **IRV** view, examine the AUTOSAR IRVs created and mapped by the component builder software.



If you previously found, in the **Data Transfers** view, that a Simulink rate transition was not mapped to an AUTOSAR IRV, use the **IRV** view to add an IRV. Then return to the Simulink-AUTOSAR Mapping Explorer, **Data Transfers** view, and use **AR:IrvName** to map the rate transition to the corresponding IRV.

6 Click the **Validate** icon to validate the AUTOSAR interface configuration. Fix any errors and repeat until validation succeeds.

7 Generate code from the model and examine the generated C and `arxml` code. The model C code contains rate-grouped model step functions corresponding to AUTOSAR runnables, one for each discrete rate in the model, with corresponding RTE API calls. If the software component is multi-instantiable, the step functions and the RTE API calls have an instance handle as an argument.

```
/* Model step function for TID0 */
void Runnable_Step(void)              /* Sample time: [1.0s, 0.0s] */
{
  ...
  /* RateTransition: '<Root>/RateTransition' */
  rtb_RateTransition = Rte_IrvIRead_Runnable_Step_IRV1();
  ...
  /* Outport: '<Root>/Out2' */
  Rte_IWrite_Runnable_Step_Out2_Out2(rtb_Sum_o);
  ...
}

/* Model step function for TID1 */
void Runnable_Step1(void)              /* Sample time: [2.0s, 0.0s] */
{
  ...
}

/* Model step function for TID2 */
void Runnable_Step2(void)              /* Sample time: [4.0s, 0.0s] */
{
  ...
}

/* Model step function for TID3 */
void Runnable_Step3(void)              /* Sample time: [8.0s, 0.0s] */
{
  ...
}
```

Configure Multiple Runnables Using Function Calls

You can model AUTOSAR multi-runnables as function-call subsystems or functions at the top level of a model. Root model inports and outports represent AUTOSAR ports, and signal lines connecting function-call subsystems represent AUTOSAR inter-runnable variables (IRVs).

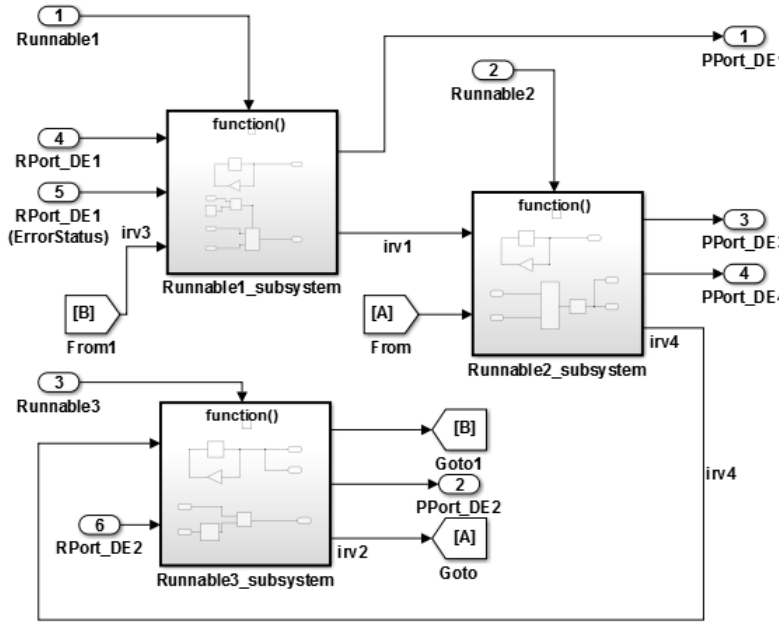
When you generate code for the model, each function-call subsystem representing a runnable appears in the model C code as a callable model entry-point function. For example, here is an excerpt of the generated C code for function-call subsystem `Runnable3_subsystem` in the example model `rtwdemo_autosar_multirunnables`:

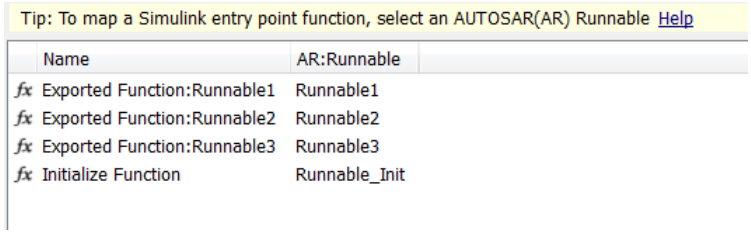
```
void Runnable3(void)
{
  ...
  /* RootInportFunctionCallGenerator:
   * '<Root>/RootFcnCall_InsertedFor_Runnable3_at_outport_1'
   * incorporates:  SubSystem: '<Root>/Runnable3_subsystem'
   */
  /* SignalConversion: '<S3>/OutportBufferForTicToc' incorporates:
   *  UnitDelay: '<S3>/Delay'
```

```
   */
  OutportBufferForTicToc = rtDWork.Delay_DSTATE;
  ...
  /* Update for UnitDelay: '<S3>/Delay' incorporates:
   *  Gain: '<S3>/Gain'
   *  UnitDelay: '<S3>/Delay'
   */
  rtDWork.Delay_DSTATE = (int8_T)-rtDWork.Delay_DSTATE;

  /* End of Outputs for RootInportFunctionCallGenerator:
   * '<Root>/RootFcnCall_InsertedFor_Runnable3_at_outport_1' */

  /* Outport: '<Root>/PPort_DE2' */
  Rte_IWrite_Runnable3_PPort_DE2(OutportBufferForTicToc);
}
```

Note: In previous releases, AUTOSAR multi-runnables were modeled as function-call subsystems within a wrapper subsystem in a Simulink model. To generate code, you right-clicked the wrapper subsystem and exported functions. To convert subsystem multi-runnables to top model multi-runnables, you can use the subsystem-to-model conversion techniques described in "Convert a Subsystem to a Referenced Model" in the Simulink documentation. After the basic conversion, some AUTOSAR configuration information from the subsystem configuration will need to be manually re-established in the new configuration.

You can use the Configure AUTOSAR Interface dialog box to map and validate your AUTOSAR multi-runnable configuration. If you open the example model `rtwdemo_autosar_multirunnables` and select **Code > C/C++ Code > Configure Model as AUTOSAR Component**, you can map and configure the entry-point functions for AUTOSAR multi-runnables in the model. The following figures show the top level of the model and the Simulink mapping dialog box view of the multi-runnable entry points.





You can simulate multiple runnables in an AUTOSAR software component in multiple simulation modes. For example:

- For a periodic runnable, you can edit the properties of the function-call subsystem inport to set the sample time for a periodic event simulation.
- For a non-periodic runnable, you can edit the **Data Import/Export** pane of the Configuration Parameters dialog box to set up data loading for an asynchronous event simulation.

To verify the generated code for AUTOSAR multi-runnables, you can use SIL or PIL verification. Optionally, you can set up a harness model to exercise the function entry-points for the AUTOSAR multi-runnables.

See Also

`Rate Transition` | `Simulink Function`

Related Examples

- “Import AUTOSAR Software Component” on page 3-4
- “Model for AUTOSAR Platform”
- “Model AUTOSAR Software Components” on page 2-3
- “Export AUTOSAR Component XML and C Code” on page 5-2

More About

- “AUTOSAR Interface 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`.

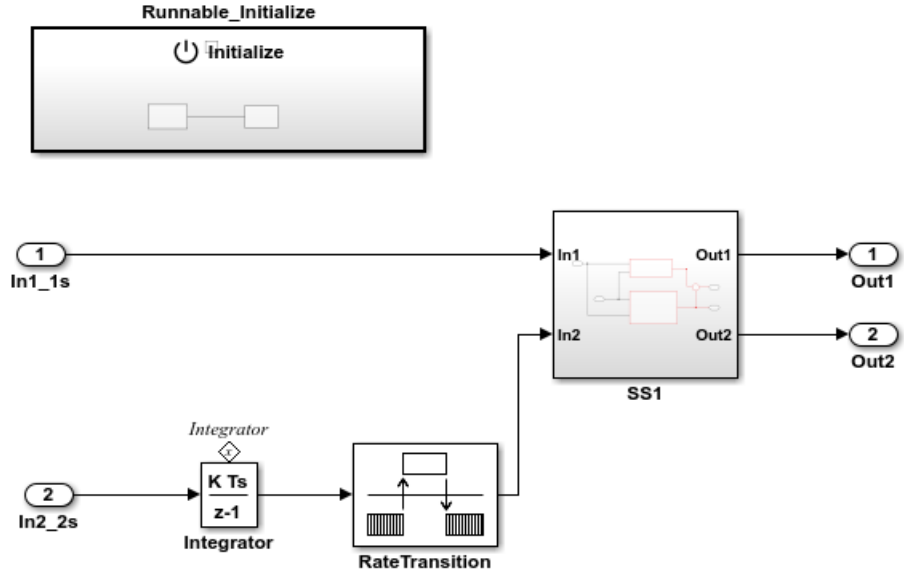
The Initialize Function and Terminate Function blocks can control execution of a component in response to initialize, reset, or terminate events. For more information, see “Create a Model to Initialize and Terminate State”, “Generate Code That Responds to Initialize, Reset, and Terminate Events”, and AUTOSAR topic “Startup, Reset, and Shutdown” on page 2-9.

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.)

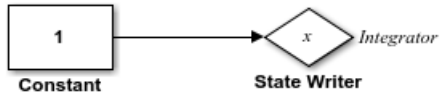
This example shows how to configure an AUTOSAR software component for simple startup and termination processing, using the Initialize Function and Terminate Function blocks.

1 Open a model that is configured for AUTOSAR code generation. This example uses a writable copy of the example model `rtwdemo_autosar_swc`.

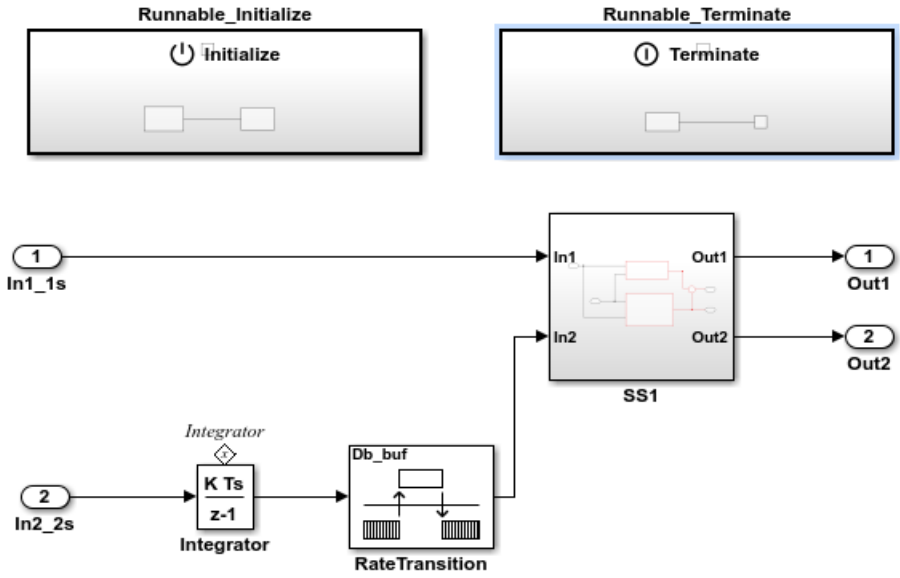
  Add an Initialize Function block to the model.



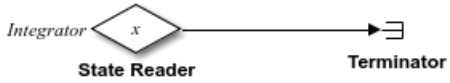
2 In the Initialize Function block, develop the logic that is required to execute during component initialization, using the techniques described in "Create a Model to Initialize and Terminate State".



3 Add a Terminate Function block to the model. For example, you can copy and paste the Initialize Function block, open the Event Listener block in the copy, and set **Event type** to `Terminate`.



4 In the Terminate Function block, develop the logic that is required to execute during component initialization, using the techniques described in “Create a Model to Initialize and Terminate State”.



In this example, the Terminator block is a placeholder for saving the state value.

5 Add a terminate entry-point function to the model. Open the Configuration Parameters dialog box, and select the **All Parameters** tab. Navigate to the **Terminate function required** option and select it. Click **Apply**.

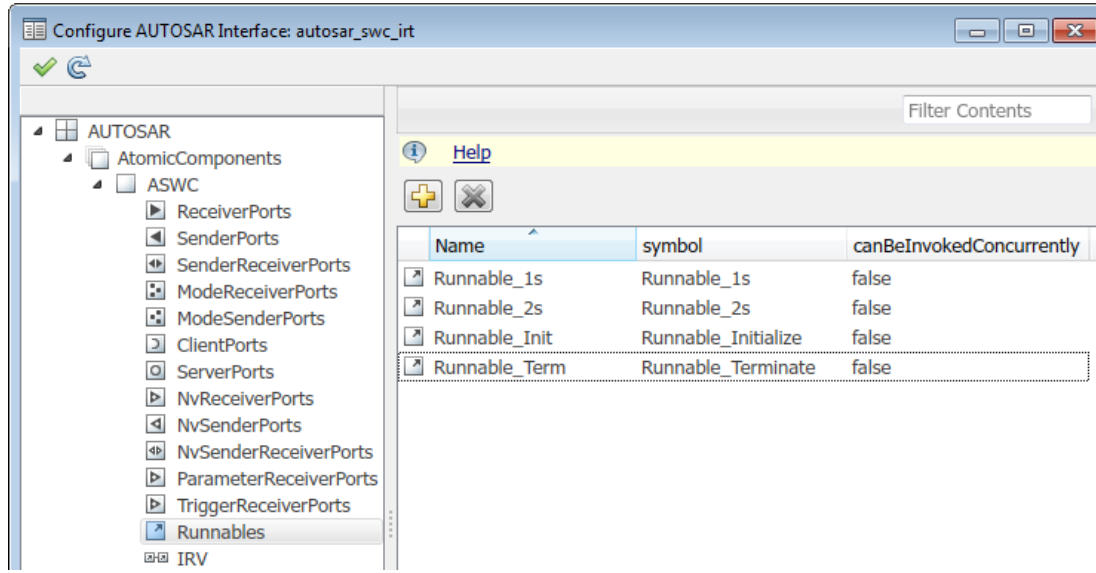
6 Open the Configure AUTOSAR Interface dialog box. Click the **Synchronize** icon to update the Simulink to AUTOSAR mapping of the model. The mapping now

reflects the addition of the Initialize Function and Terminate Function blocks and enabling of a terminate entry-point function.

7 Select the AUTOSAR Properties Explorer. Expand the AtomicComponents node, and go to the Runnables view.

The Runnables list should already contain an initialization runnable, created as part of the initial Simulink representation of the AUTOSAR software component. Use the **Add** icon to add a terminate runnable to the component. Select each runnable and configure its name and properties.

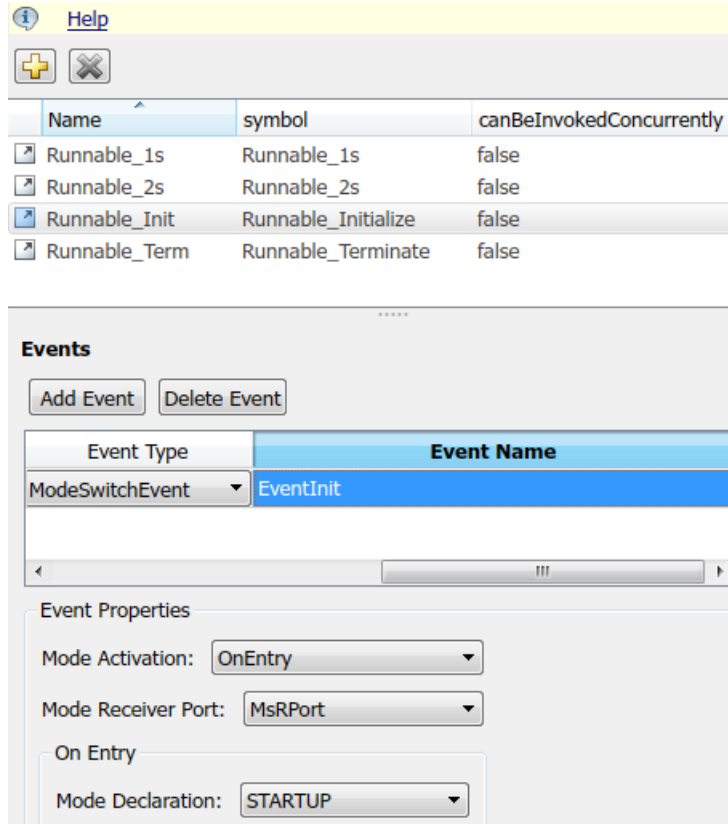
The initialize, reset, or terminate runnable symbol name shown in the Runnables view becomes the runnable function name, and is used in names of RTE access methods generated for parameters, states, function callers, and side I/O.



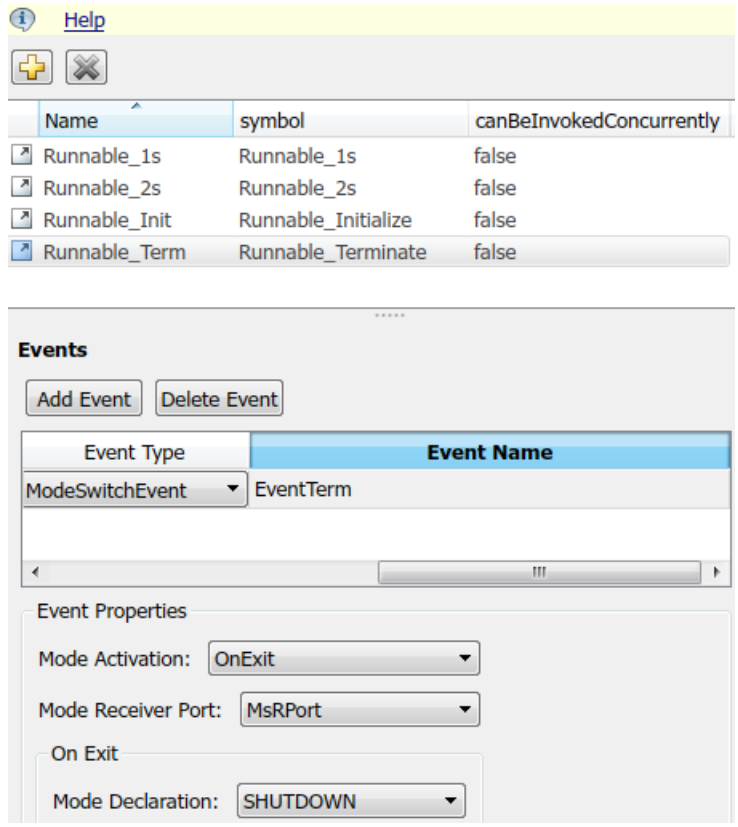
8 For both the initialize and terminate runnables, configure an AUTOSAR event that activates the runnable.

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 4-130.

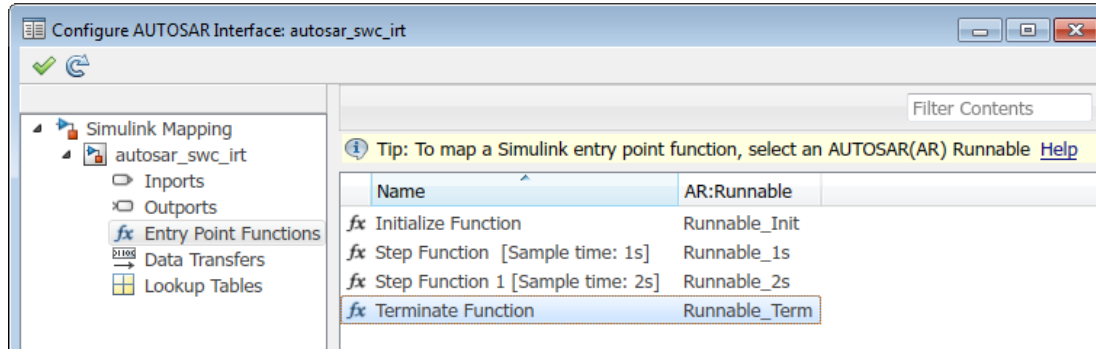
In the Runnables view, click the initialize runnable name to display and modify its associated event properties. Add and configure an event.



In the Runnables view, click the terminate runnable name to display and modify its associated event properties. Add and configure an event.



9 Switch to the Simulink-AUTOSAR Mapping Explorer, and go to the Entry Point Functions view. 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 side I/O associated with the runnable.

See Also

`Event Listener` | `Initialize Function` | `State Reader` | `State Writer` | `Terminate Function`

Related Examples

- "Create a Model to Initialize and Terminate State"
- "Generate Code That Responds to Initialize, Reset, and Terminate Events"
- "Create a Test Harness to Generate Events"
- "Configure AUTOSAR Mode-Switch Communication" on page 4-130

More About

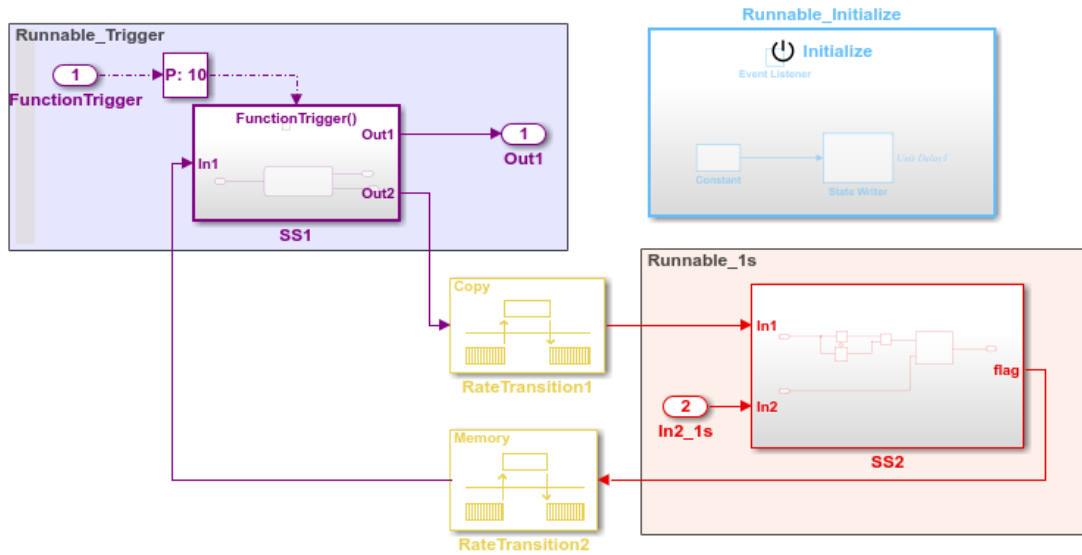
- "Startup, Reset, and Shutdown" on page 2-9
- "Entry-Point Functions and Scheduling"
- "AUTOSAR Interface Configuration" on page 4-3

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 leverages periodic and asynchronous rates (sample times).

The approach can be used to model the JMAAB complex control model type beta (ß) architecture 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 `http://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, `rtwdemo_autosar_swc_fcncalls`, in which an asynchronous function-call runnable at the top level of the model interacts with a periodic rate-based runnable.

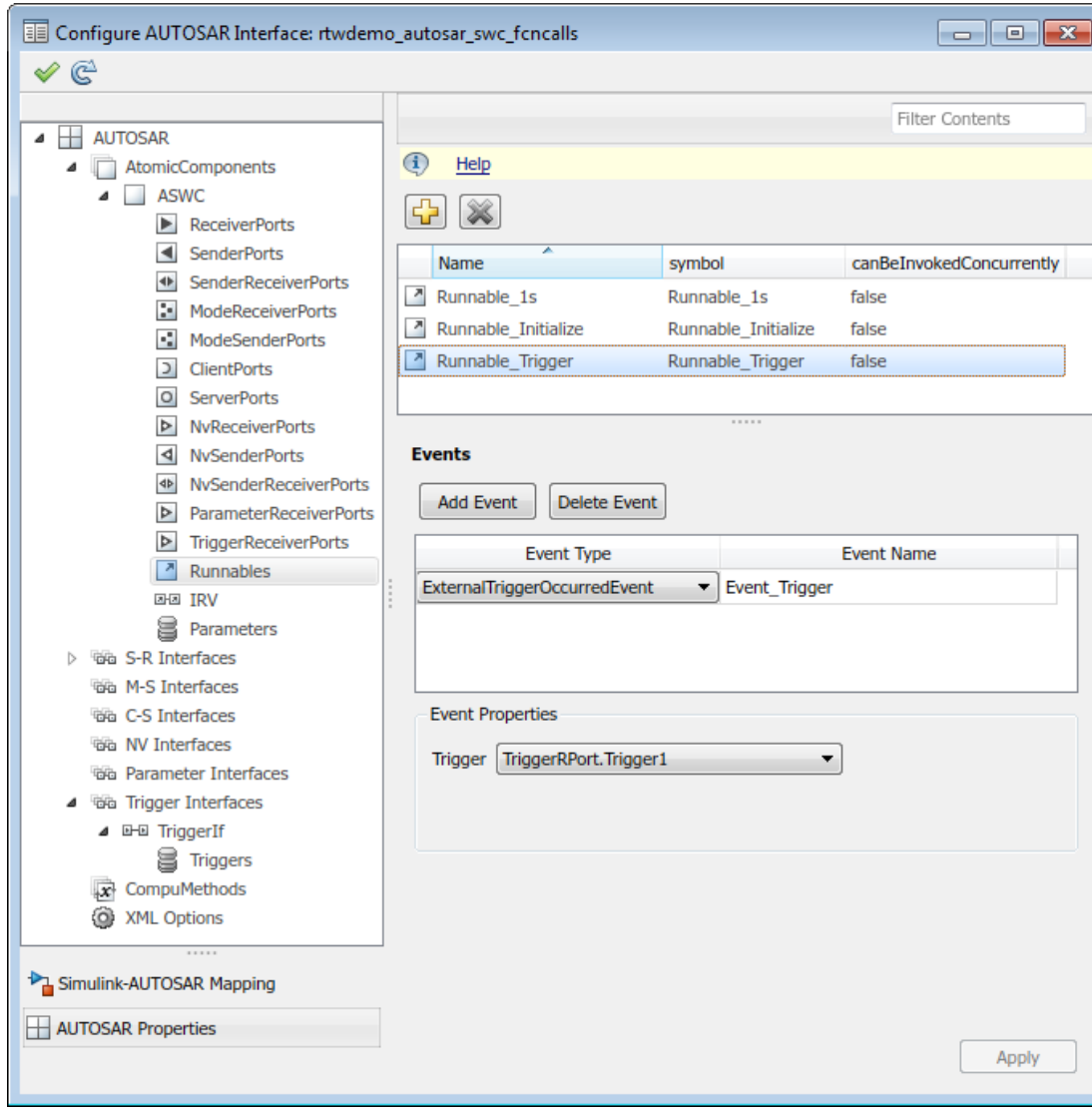


Some guidelines apply to AUTOSAR modeling of the JMAAB type beta controller layout:

- 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.

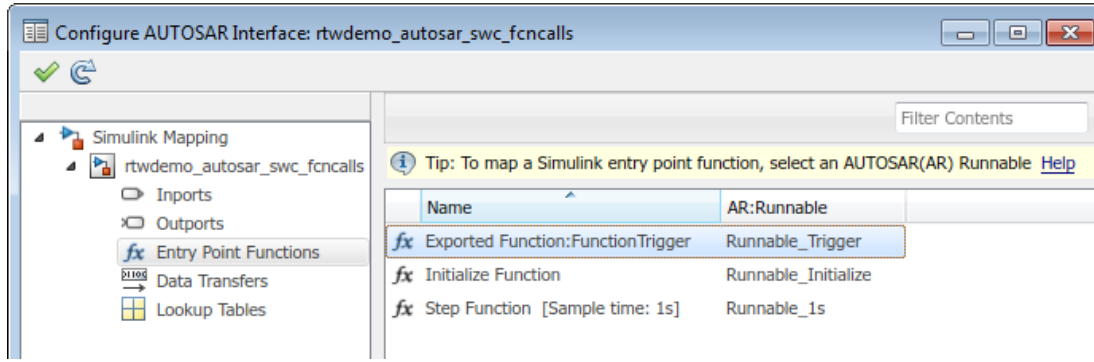
Here is the AUTOSAR Properties Explorer view of the runnables. The asynchronous function-call runnable must be triggered by an event of type `DataReceivedEvent`, `DataReceiveErrorEvent`, `ModeSwitchEvent`, `InitEvent`, or `ExternalTriggerOccurredEvent`.



In this example, the AUTOSAR runnable is activated by an `ExternalTriggerOccurredEvent`, which is delivered to a trigger receiver port via a trigger interface. For more information about `ExternalTriggerOccurredEvents`, see

“Configure Receiver for AUTOSAR External Trigger Event Communication” on page 4-148.

Here is the Simulink-AUTOSAR Mapping Explorer 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 4-148
- “Model for AUTOSAR Platform”
- “Export AUTOSAR Component XML and C Code” on page 5-2

More About

- “Model AUTOSAR Software Components” on page 2-3
- “AUTOSAR Interface Configuration” on page 4-3

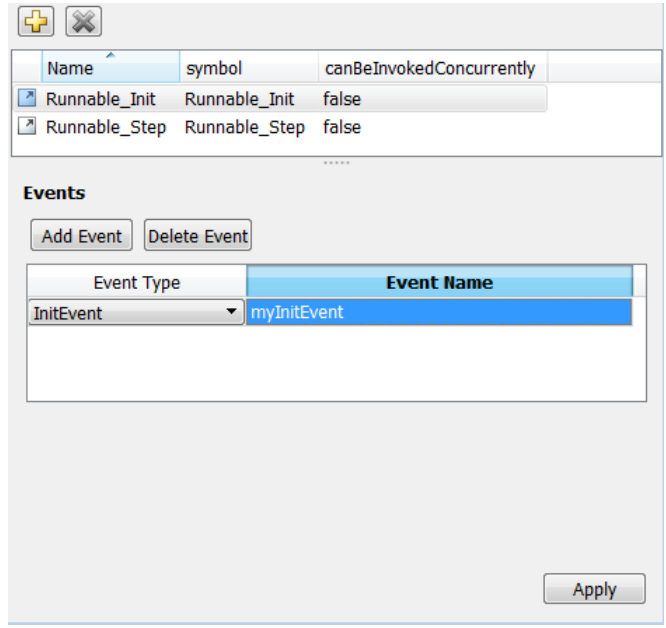
Configure AUTOSAR Initialization Runnable (R4.1)

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 4-130.)

If you import `arxml` code that describes a runnable with an `InitEvent`, the `arxml` importer configures the runnable in Simulink as an initialization runnable.

Alternatively, you can configure a runnable to be the initialization runnable in Simulink. For example,

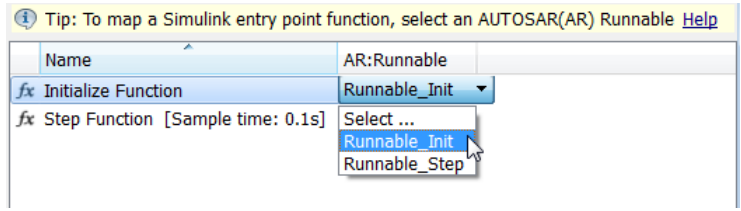
1. Open a model configured for AUTOSAR.
2. Open the Configuration Parameters dialog box, go to **Code Generation** > **AUTOSAR Code Generation Options**, and verify that the selected AUTOSAR schema version is `4.1` or higher.
3. Open the Configure AUTOSAR Interface dialog box, and select the AUTOSAR Properties Explorer.
4. Navigate to a software component, and select the **Runnables** view.
5. 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.

6 Select the Simulink-AUTOSAR Mapping Explorer, and select the **Entry Point Functions** view.

7 To map an initialization function to the initialization runnable, select the function. From the **AR: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 of an event is `'mrPortName.modeName'` or `''`. 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 mode-switch event named `Event_Run`. The call potentially disables the event for mode `Sleep`, which is defined on mode-receiver port `MRPort`.

```
open_system('rtwdemo_autosar_multirunnables_ms')
arProps = autosar.api.getAUTOSARProperties('rtwdemo_autosar_multirunnables_ms');
eventPaths = find(arProps,[],'ModeSwitchEvent')
```

```
eventPaths =
    'ASWC/Behavior/Event_Run'
```

```
dsblMode = get(arProps,'ASWC/Behavior/Event_Run','DisabledMode')
```

```
dsblMode =
   Empty cell array: 1-by-0
```

```
set(arProps,'ASWC/Behavior/Event_Run','DisabledMode','MRPort.Sleep')
dsblMode = get(arProps,'ASWC/Behavior/Event_Run','DisabledMode')
```

```
dsblMode =
    'MRPort.Sleep'
```

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 2-31.

When you import `arxml` code that contains an AUTOSAR PIM definition into Simulink, the importer:

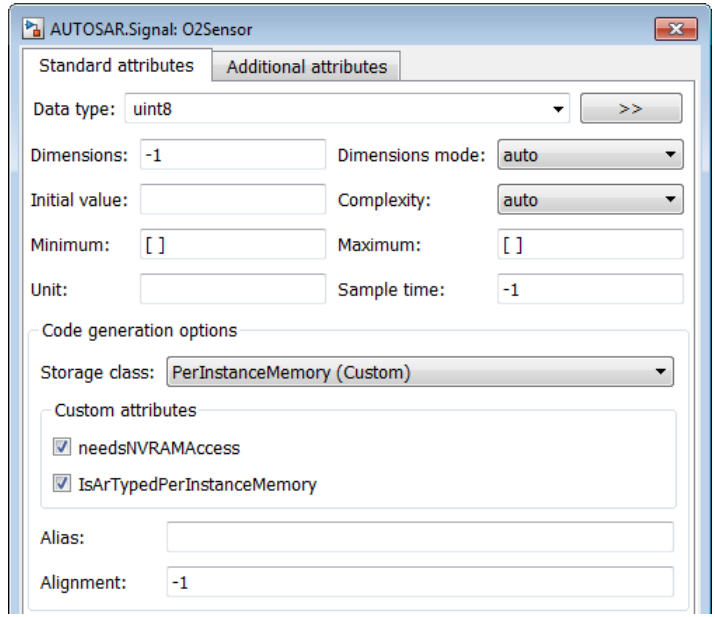
- Creates an `AUTOSAR.Signal` data object, sets its **Storage class** to `PerInstanceMemory`, and configures per-instance memory attributes.
- Creates a Data Store Memory block that references the `AUTOSAR.Signal` object.

You can use the same modeling pattern to create per-instance memory in your model. For example:

1. In the base workspace, create an `AUTOSAR.Signal` object. For example:

   ```
   O2Sensor = AUTOSAR.Signal;
   ```

2. In the base workspace, double-click the object to open its dialog box. Set the **Storage class** to `PerInstanceMemory`. This selection enables custom attributes **needsNVRAMAccess** and **IsArTypedPerInstanceMemory**.

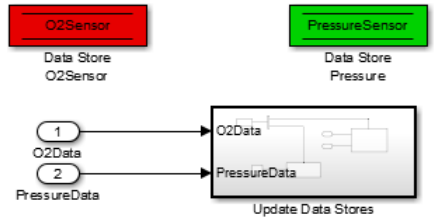


- **needsNVRAMAccess** — Select to configure this per-instance memory to be a mirror block for a specific NVRAM block.
- **IsArTypedPerInstanceMemory** — Select for AUTOSAR-typed per-instance memory (`arTypedPerInstanceMemory`); clear for C-typed per-instance memory. `arTypedPerInstanceMemory` requires AUTOSAR schema version 4.0 or later

3 Optionally, you can specify an **Initial value** for the global variable corresponding to per-instance memory.

4 Go to the **Additional attributes** tab, and configure software calibration access to the data. Set the **SwCalibrationAccess** attribute to `NotAccessible`, `ReadOnly`, or `ReadWrite`. Optionally, you can specify the **DisplayFormat** of the data for measurement and calibration tools. Close the dialog box.

5 Create a Data Store Memory block that references the `AUTOSAR.Signal` object. For example, in the model *`matlabroot`*`/help/toolbox/ecoder/examples/autosar/mAutosarPIM.slx`, Data Store Memory blocks reference signal objects `O2Sensor` and `PressureSensor` in the base workspace.



For more information, see the `Data Store Memory` block reference page.

Note: The software does not support per-instance memory modeling in referenced models.

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.

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

`AUTOSAR.Signal` | `Data Store Memory`

Related Examples

- “Import AUTOSAR Software Component” on page 3-4
- “Model AUTOSAR Component Behavior” on page 2-28
- “Export AUTOSAR Component XML and C Code” on page 5-2

More About

- “Per-Instance Memory” on page 2-31

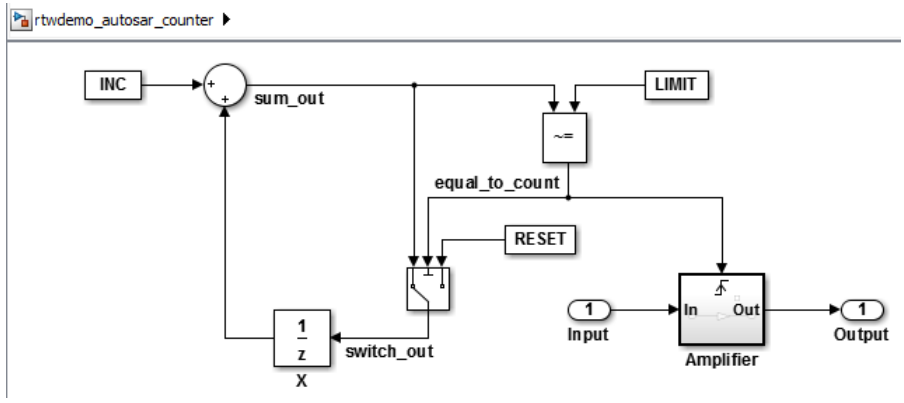
Configure AUTOSAR Static or Constant Memory

When you import `arxml` files containing AUTOSAR R4.x Static Memory and Constant Memory data into Simulink, the importer creates `AUTOSAR4.Signal` and `AUTOSAR4.Parameter` data objects to represent the data and assigns them a global variable storage class. You can open the data objects and examine the attributes, including **SwCalibrationAccess**, which controls software calibration tool access to the data.

To configure new AUTOSAR Static and Constant memory data in a Simulink model, the general procedure is to select or create a signal or parameter, configure it to resolve to a data object in the workspace, and set its **Storage class** property to `ExportedGlobal`, `Global (Custom)`, or another custom storage class that generates a global variable in the model code.

For example:

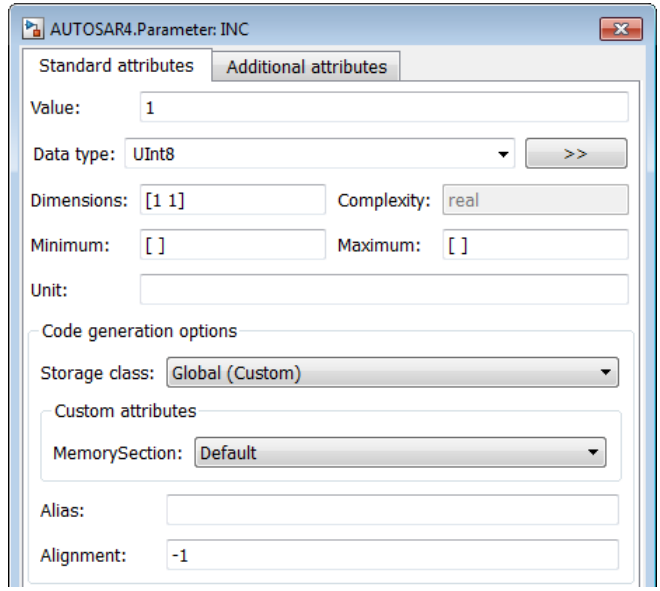
1. Open the example model `rtwdemo_autosar_counter`.



2. The parameter `INC` initially is configured as an AUTOSAR calibration parameter, of type `UInt8` (`uint8` alias), using an `AUTOSAR.Parameter` data object in the base workspace. Issue the following MATLAB commands to reconfigure it as AUTOSAR Constant Memory data.

```
INC = AUTOSAR4.Parameter;
INC.DataType = 'UInt8';
INC.Value = 1;
```

3 After issuing the MATLAB commands, open the `INC` data object, for example, by double-clicking it in the workspace. Examine the settings in the Properties dialog box.



Go to the **Additional attributes** tab, and configure software calibration access to the data. Set the **SwCalibrationAccess** attribute to `NotAccessible`, `ReadOnly`, or `ReadWrite`. Optionally, you can specify the **DisplayFormat** of the data for measurement and calibration tools. Close the dialog box.

4 The signal `sum_out` initially does not resolve to a data object in the workspace. Issue the following MATLAB command to create a `sum_out` signal data object.

```
sum_out = AUTOSAR4.Signal;
```

In the model window, right-click the `sum_out` signal, select **Properties**, and select the option **Signal name must resolve to Simulink signal object**. Close the dialog box.

As with the `INC` parameter, you can double-click the `sum_out` data object and examine the settings in the Properties dialog box, including **SwCalibrationAccess**. Close the dialog box.

5 Generate code, and verify that `INC` and `sum_out` appear in the generated code as Constant Memory and Static Memory respectively. In the generated file `rtwdemo_autosar_counter_component.arxml`:

- `INC` now appears in a constant memory specification, rather than a calibration parameter.
- `sum_out` now appears in a static memory specification (previously was absent).

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 2-40.

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 4-237

"Configure DataTypeMappingSet Package and Name" on page 4-238

"Initialize Data with ApplicationValueSpecification" on page 4-240

Control Application Data Type Generation

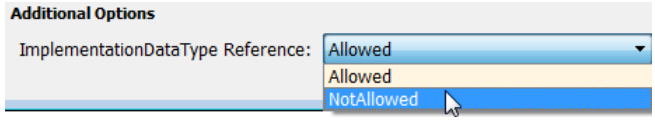
For AUTOSAR data types created in Simulink, by default, the software generates application base types only for fixed-point data types and enumerated date types with storage types. If you want to override the default behavior for generating application types, you can configure the `arxml` exporter to 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:

```
open_system('rtwdemo_autosar_multirunnables');
arProps=autosar.api.getAUTOSARProperties('rtwdemo_autosar_multirunnables');
set(arProps,'XmlOptions','ImplementationTypeReference','NotAllowed');
get(arProps,'XmlOptions','ImplementationTypeReference')
```

To set the `ImplementationDataTypeReference` property in the Configure AUTOSAR Interface dialog box, select **AUTOSAR Properties** and select **XML Options**. Set the parameter **ImplementationDataType Reference** to the value that you want. Click **Apply**.



Configure DataTypeMappingSet Package and Name

For AUTOSAR software components created in Simulink, you can control the AUTOSAR package and short name exported for AUTOSAR data type mapping sets. To configure the data type mapping set package for export, set the `XMLOptions` property `DataTypeMappingPackage` using the Configure AUTOSAR Interface dialog box or the AUTOSAR property `set` function.

```
open_system('rtwdemo_autosar_multirunnables');
arProps=autosar.api.getAUTOSARProperties('rtwdemo_autosar_multirunnables');
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 = 'rtwdemo_autosar_multirunnables';
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 (`ApplicationValueSpecification`s). Embedded Coder provides the following support:

- The `arxml` importer uses `ApplicationValueSpecification`s 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 `ApplicationValueSpecification`s to specify initial values for AUTOSAR data.

For AUTOSAR parameters typed by implementation data type, code generation exports `arxml` code that uses `NumericalValueSpecification`s and (for enumerated types) `TextValueSpecification`s to specify initial values. If initial values for parameters specify multiple values, generated code uses `ArrayValueSpecification`s.

Configure AUTOSAR CompuMethods

Embedded Coder software imports AUTOSAR computation methods (`CompuMethod`s) described in `arxml` code and preserves them across round-trips between an AUTOSAR authoring tool (AAT) and Simulink. You can modify the properties of imported CompuMethods in Simulink.

You also can create CompuMethods in Simulink, initialize their properties, and control the CompuMethod direction exported for linear functions.

This topic provides examples of configuring AUTOSAR CompuMethods in Simulink.

In this section...

"Configure AUTOSAR CompuMethod Properties" on page 4-241

"Create AUTOSAR CompuMethods" on page 4-243

"Configure CompuMethod Direction for Linear Functions" on page 4-244

"Export CompuMethod Unit References" on page 4-246

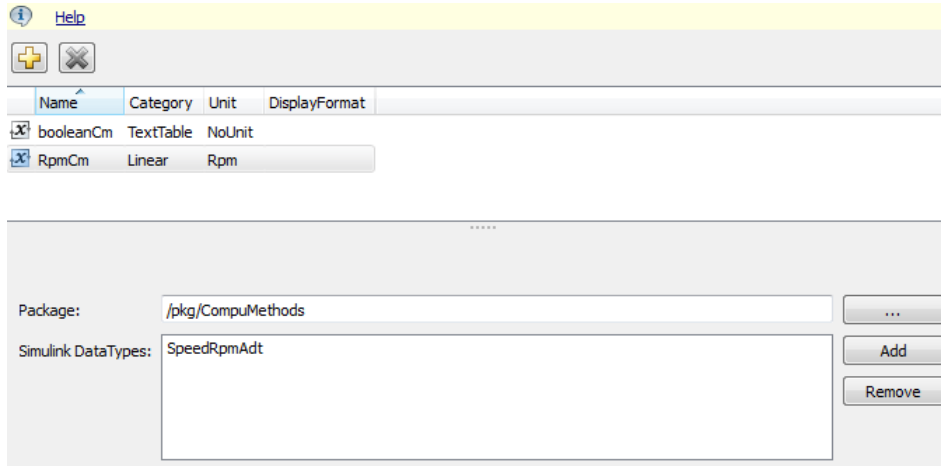
"Modify Linear Scaling for SCALE_LINEAR_AND_TEXTTABLE CompuMethod" on page 4-246

"Configure Rational Function CompuMethod for Dual-Scaled Parameter" on page 4-248

Configure AUTOSAR CompuMethod Properties

You can configure AUTOSAR CompuMethod properties in your model, either graphically or programmatically. The CompuMethod properties you can modify include name, category, unit, display format, AUTOSAR package, and Simulink data types.

To configure a CompuMethod using the graphical interface, open the Configure AUTOSAR Interface dialog box, select AUTOSAR Properties Explorer, 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.

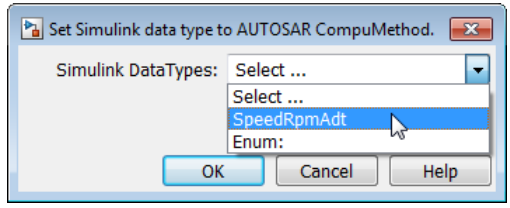
- **Name** — Specify name text
- **Category** — Select `Identical`, `Linear`, `RatFunc`, `TextTable`, or `LinearAndTextTable` (see "CompuMethod Categories for Data Types" on page 2-43)
- **Unit** — Select from units available in the model
- **DisplayFormat** — Optionally specify format to be used by 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 4-194.
- **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 interface **Package** parameter value is updated with your selection. For more information about

the AUTOSAR Package Browser, see “Configure AUTOSAR Package for Interface or CompuMethod” on page 4-67.

To associate a CompuMethod with a Simulink data type used in the model, select a CompuMethod and click the **Add** button to the right of **Simulink DataTypes**. This action opens a dialog box with a list of available data types. In the list of values, select a `Simulink.NumericType` or `Simulink.AliasType`, or enter the name of a Simulink enumerated type. To add the type to the **Simulink DataTypes** list, click **OK**.

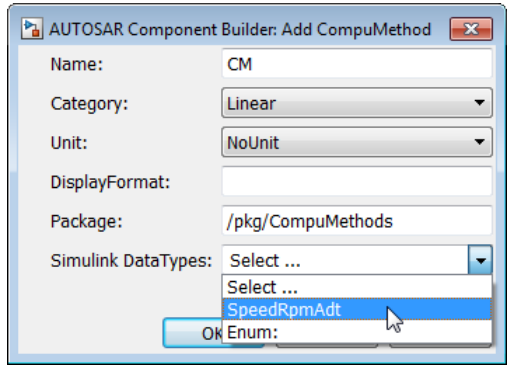


To set the **Simulink DataTypes** property programmatically, open the model and use an AUTOSAR property `set` function call similar to the following:

```
arProps=autosar.api.getAUTOSARProperties('cmSpeed');
set(arProps,'/pkg/CompuMethods/RpmCm','SlDataTypes',{'SpeedRpmAdt'})
sltypes=get(arProps,'/pkg/CompuMethods/RpmCm', 'SlDataTypes')

sltypes =
    'SpeedRpmAdt'
```

Create AUTOSAR CompuMethods

You can create AUTOSAR CompuMethods in your model, either graphically or programmatically. To create an AUTOSAR CompuMethod using the graphical interface, open the Configure AUTOSAR Interface dialog box, select AUTOSAR Properties Explorer, and select the CompuMethods view. To open the Add CompuMethod dialog box, click the **Add** icon . 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 of the Configure AUTOSAR Interface dialog box is updated with the new CompuMethod.



When you generate code, the exported `arxml` code contains the CompuMethod definition and references to it.

Configure CompuMethod Direction for Linear Functions

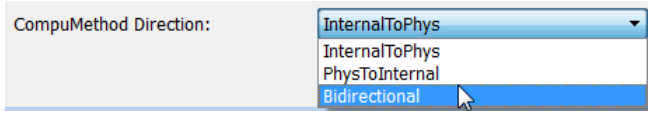
For designs originated in Simulink, you can control properties for an exported `CompuMethod`, including the direction of `CompuMethod` conversion between internal and physical representations of a value. Using either the Configure AUTOSAR Interface dialog box or the AUTOSAR property `set` function, you 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:

```
open_system('rtwdemo_autosar_multirunnables');
arProps=autosar.api.getAUTOSARProperties('rtwdemo_autosar_multirunnables');
set(arProps,'XmlOptions','CompuMethodDirection','Bidirectional');
get(arProps,'XmlOptions','CompuMethodDirection')
```

To specify `CompuMethod` direction in the Configure AUTOSAR Interface dialog box, select **AUTOSAR Properties** and select **XML Options**. Set the parameter **CompuMethod Direction** to the value that you want. Click **Apply**.



When you generate code for your model, the `CompuMethod`s 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: `CompuMethod`s 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 `CompuMethod`s. 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.

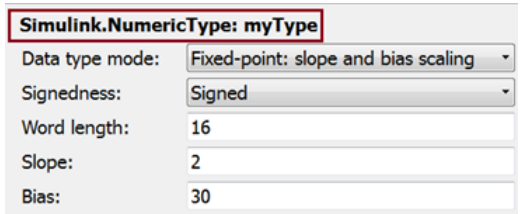
Modify Linear Scaling for SCALE_LINEAR_AND_TEXTTABLE CompuMethod

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

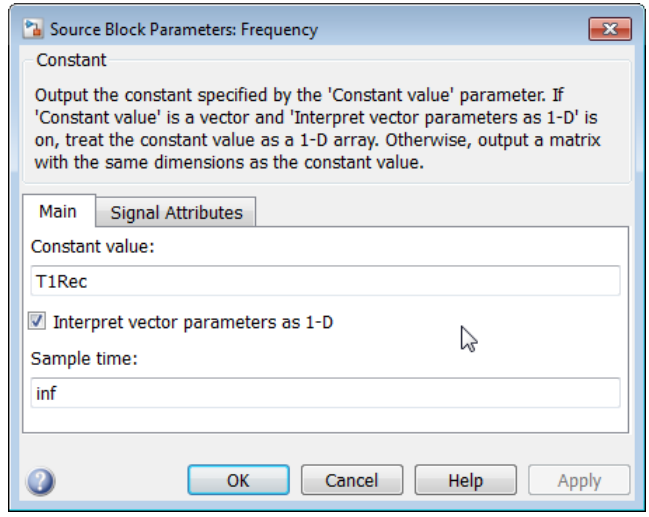
For an AUTOSAR dual-scaled parameter, which stores two scaled values of the same physical value, the software generates the `CompuMethod` category `RAT_FUNC`. The computation method can be a first-order rational function.

To configure and generate a dual-scaled parameter:

1. In the workspace, create an AUTOSAR.DualScaledParameter data object. For example:

   ```
   T1Rec = AUTOSAR.DualScaledParameter;
   ```

2. Open a model that is configured for AUTOSAR. Set up a Constant block to reference the dual-scaled parameter.

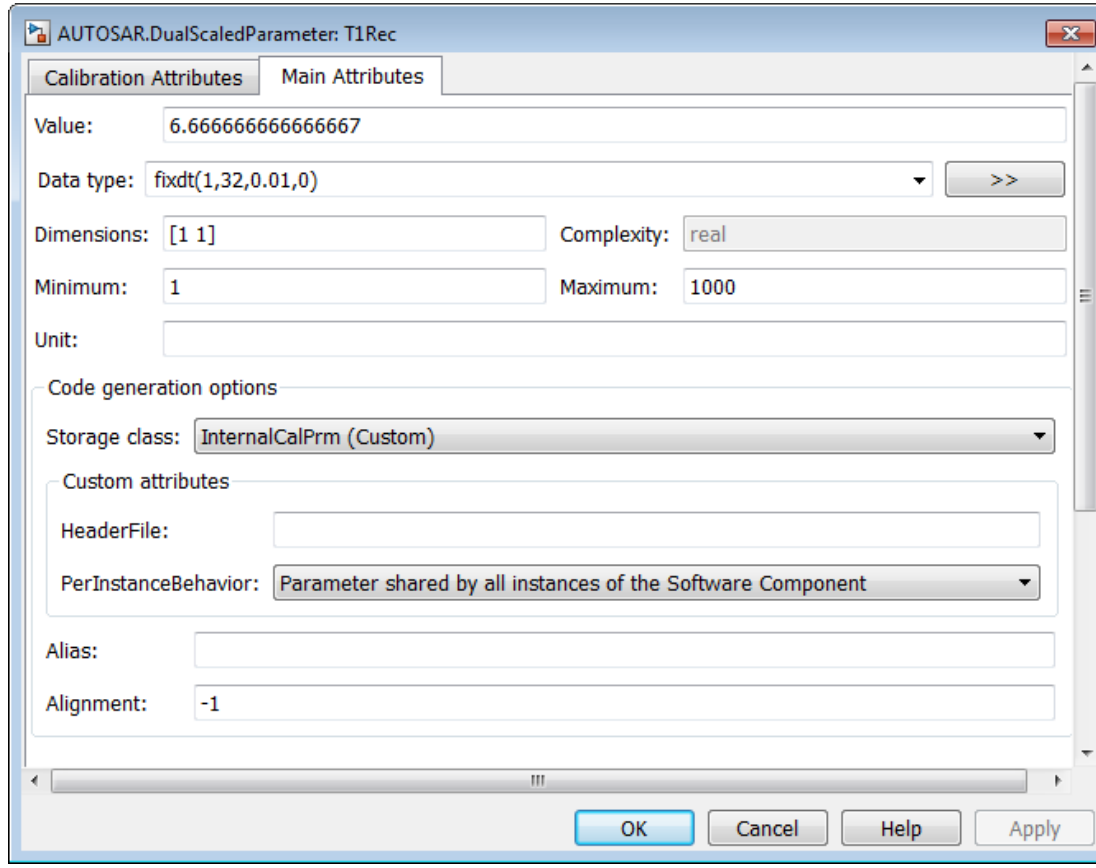
   Connect the Constant block to a Simulink outport.

3. Configure the attributes of the dual-scaled parameter. To configure the parameter attributes for `T1Rec`, created in Step 1, this example uses the following MATLAB

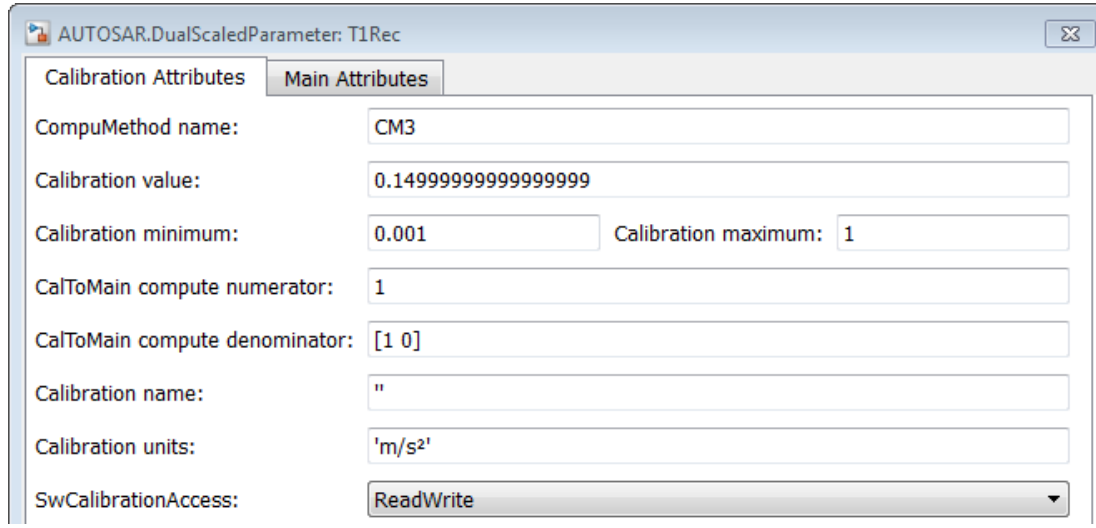
code. The code sets up a conversion from an internal calibration time value to a physical frequency (time reciprocal) value.

```
% 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';
```

4 To open the dual-scaled parameter dialog box, double-click the parameter in the workspace. Here are the main attributes set by the MATLAB code.



5 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.



6 If `CompuMethod` direction is not already set to bidirectional in the AUTOSAR properties, use the Configure AUTOSAR Interface dialog box, XML Options view, to set it.

7 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

AUTOSAR.DualScaledParameter

Related Examples

- "Import AUTOSAR Software Component" on page 3-4
- "Configure AUTOSAR Package for Interface or CompuMethod" on page 4-67
- "Export AUTOSAR Component XML and C Code" on page 5-2

More About

- "CompuMethod Categories for Data Types" on page 2-43
- "AUTOSAR Interface Configuration" on page 4-3

Configure AUTOSAR Internal Data Constraints Export

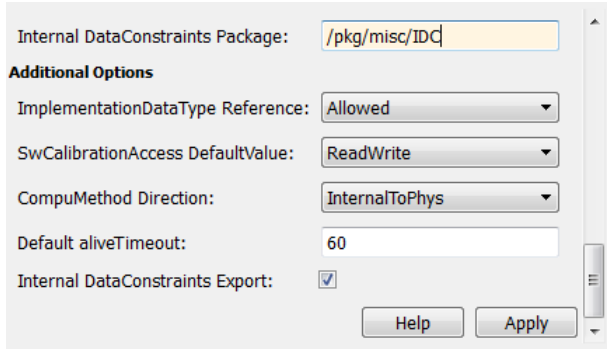
AUTOSAR applications use data constraints to implement limits on data types and provide a controlled range of possible values. Internal data constraints represent minimum and maximum values for implementation data types, reflecting the internal or machine view of the data.

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 the Configure AUTOSAR Interface dialog box. Select **Code** > **C/C++ Code** > **Configure Model as AUTOSAR Component**.
2. Select **AUTOSAR Properties** and select **XML Options**.
3. In the XML Options view, under **Additional Options**, select **Internal DataConstraints Export**.
4. Optionally, under **Additional Packages**, enter a package path for **Internal DataConstraints Package**.



5 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>
```

.

Related Examples

- “Configure AUTOSAR Release 4.x Data Types” on page 4-237
- “Model AUTOSAR Data Types” on page 2-34
- “Export AUTOSAR Component XML and C Code” on page 5-2

More About

- “Release 4.x Data Types” on page 2-40
- “AUTOSAR Interface Configuration” on page 4-3

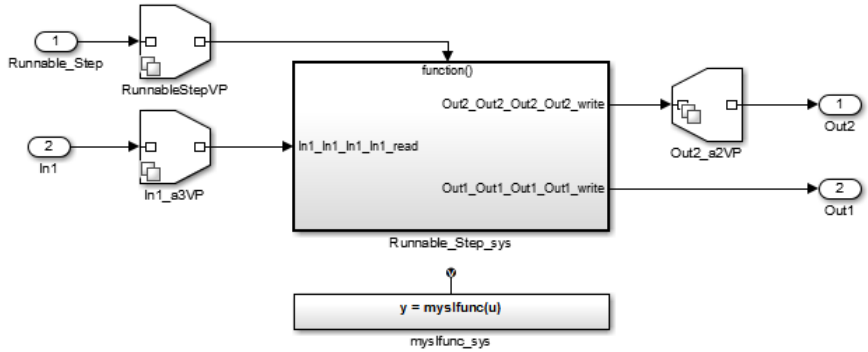
Configure AUTOSAR Variants in Ports and Runnables

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. Simulink represents the variation points with elements and data objects including:

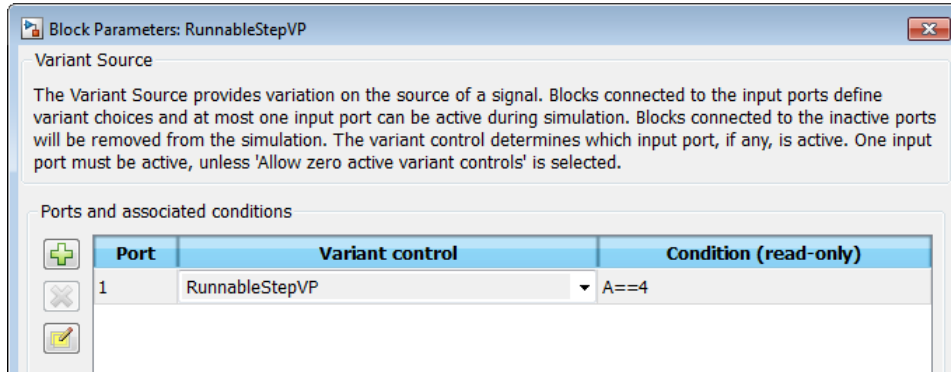
- `Simulink.Variant` data objects for defining variant condition logic
- `AUTOSAR.Parameter` data objects with storage class `SystemConstant` for representing AUTOSAR system constants
- `Variant Sink` and `Variant Source` blocks for propagating variant conditions for the AUTOSAR elements

The imported variation points are preserved for export. Before exporting, you can 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.

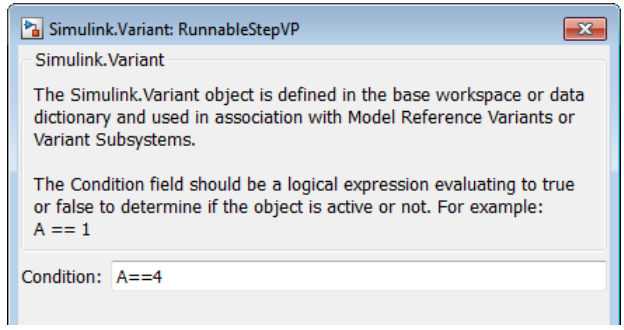
For example, here is a model created by importing an AUTOSAR runnable with variation points from an `arxml` file, and further refined in Simulink.



The model contains two Variant Source blocks and one Variant Sink block, which reference `Simulink.Variant` data objects `RunnableStepVP`, `In1_a3VP`, and `Out2_a2VP` in the base workspace. The `Simulink.Variant` data objects reference `AUTOSAR.Parameter` data object `A`, with storage class `SystemConstant`, representing an AUTOSAR system constant. Here is some of the variant control information for Variant Source block `RunnableStepVP`.



Here is the variant condition expression for `Simulink.Variant` data object `RunnableStepVP`. The expression references AUTOSAR system constant A.



When you build the model, the software exports variation point definitions in `arxml` code.

```
<RUNNABLE-ENTITY ...>
    <SHORT-NAME>Runnable_Step</SHORT-NAME>
...
    <VARIATION-POINT>
        <SHORT-LABEL>RunnableStepVP</SHORT-LABEL>
        <SW-SYSCOND BINDING-TIME="PRE-COMPILE-TIME">
          <SYSC-REF DEST="SW-SYSTEMCONST">/pkg/dt/SystemConstants/A</SYSC-REF>==4
        </SW-SYSCOND>
    </VARIATION-POINT>
</RUNNABLE-ENTITY>
```

For more information, see “Variant Systems” (Embedded Coder) and “Variant Systems” (Simulink).

See Also

`Simulink.Variant` | `AUTOSAR.Parameter` | `Variant Source` | `Variant Sink`

Related Examples

- “Model AUTOSAR Variants” on page 2-46

More About

- “Variant Systems”
- “System Constants” on page 2-30

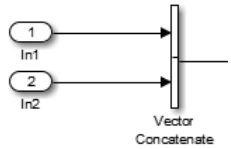
Configure AUTOSAR Variants in Array Sizes

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.

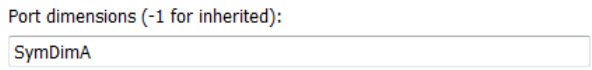
Suppose that you create a Simulink inport `In1` to represent an AUTOSAR receiver port with a variant array size.



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.



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.

Generated C Code

```
/* SignalConversion: '<Root>/ConcatBufferAtVector ConcatenateIn1' */
  for (i = 0; i < Rte_SysCon_SymDimA; i++) {
    rtb_VectorConcatenate[i] = tmp[i];
```

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”
- “Model AUTOSAR Variants” on page 2-46

More About

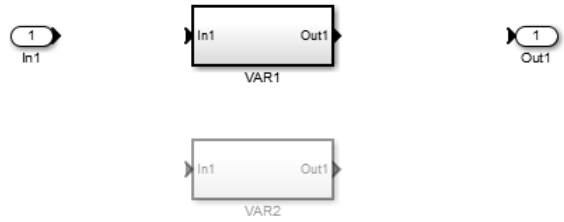
- “System Constants” on page 2-30

Configure AUTOSAR Variants in Runnable Condition Logic

AUTOSAR software components can specify variant condition logic inside a runnable. To model variant condition logic inside an AUTOSAR runnable:

- Use a `Variant Subsystem` or `Model Variants` block to represent variant implementations of a subsystem or model.
- Use `AUTOSAR.Parameter` data objects in the base workspace to model AUTOSAR system constants. The system constants represent the condition values that determine the active subsystem or model implementation.
- Use `Simulink.Variant` data objects in the base workspace to define the variant condition logic.

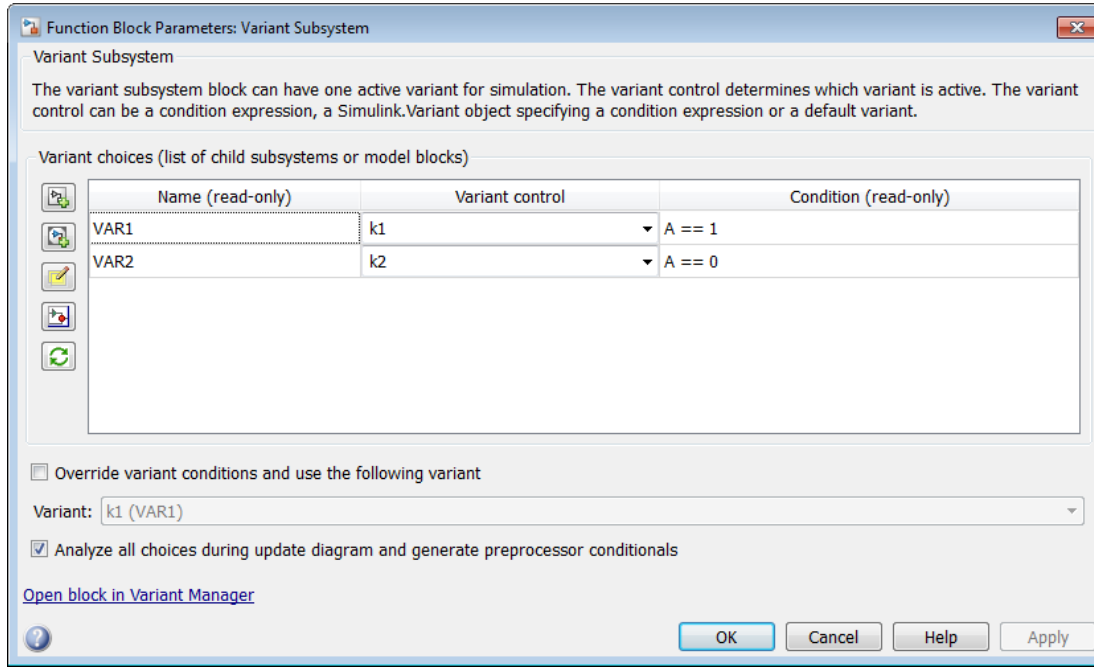
For example, suppose that you implement a Variant Subsystem block. The variant choices are subsystems VAR1 and VAR2. The blocks are not connected because connectivity is determined during simulation, based on the active variant.



In this model:

- `AUTOSAR.Parameter` data object `A` models an AUTOSAR system constant. In the parameter dialog box, **Data type** is set to `uint8`, **Storage class** is set to `SystemConstant`, and **Value** is set to 1.
- `Simulink.Variant` data objects `k1` and `k2` define the variant condition logic, by using system constant `A`. For example, in the `k1` variant dialog box, **Condition** is set to `A == 1`.

You can view their relationship in the Variant Manager or in the Variant Subsystem block properties dialog box.



When you export `arxml` and C code:

- In the `arxml` code, the variant choices appear as `VARIATION-POINT-PROXY` entries with short-names `k1` and `k2`. `A` appears as a system constant representing the associated conditional value. For example:

```
<VARIATION-POINT-PROXYS>
    <VARIATION-POINT-PROXY UUID="uuidstring">
        <SHORT-NAME>k1</SHORT-NAME>
        <CONDITION-ACCESS BINDING-TIME="PRE-COMPILE-TIME">
          <SYSC-REF DEST="SW-SYSTEMCONST">/basic_pkg/SystemConstants/A</SYSC-REF>
          == 1</CONDITION-ACCESS>
    </VARIATION-POINT-PROXY>
    <VARIATION-POINT-PROXY UUID="uuidstring">
        <SHORT-NAME>k2</SHORT-NAME>
        <CONDITION-ACCESS BINDING-TIME="PRE-COMPILE-TIME">
           <SYSC-REF DEST="SW-SYSTEMCONST">/basic_pkg/SystemConstants/A</SYSC-REF>
           == 0</CONDITION-ACCESS>
    </VARIATION-POINT-PROXY>
</VARIATION-POINT-PROXYS>
```

- In the RTE compatible C code, short-names `k1` and `k2` are encoded in the names of preprocessor symbols used in the variant condition logic. For example:

```
#if Rte_SysCon_k1
...
#elif Rte_SysCon_k2
...
#endif
```

See Also

`Simulink.Variant` | `AUTOSAR.Parameter` | `Model Variants` | `Variant Subsystem`

Related Examples

- “Model AUTOSAR Variants” on page 2-46

More About

- “System Constants” on page 2-30

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

For example, in `arxml` code, you can define `SwSystemconst`s for automobile features, such as `Transmission`, `Headlight`, `Sunroof`, and `Turbocharge`. Then a `PredefinedVariant` can map feature combinations to automobile model variants, such as `Basic`, `Economy`, `Senior`, `Sportive`, and `Junior`.

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

Typical steps include:

1. Get a list of the `PredefinedVariant`s or `SwSystemconstantValueSet`s defined in the `arxml` file.

```
>> obj = arxml.importer('mySWC.arxml');
>> find(obj,'/','PredefinedVariant','PathType','FullyQualified');
ans =
   '/pkg/body/Variants/Basic'
   '/pkg/body/Variants/Economy'
   '/pkg/body/Variants/Senior'
   '/pkg/body/Variants/Sportive'
   '/pkg/body/Variants/Junior'

>> obj = arxml.importer('mySWC.arxml');
>> find(obj,'/','SystemConstValueSet','PathType','FullyQualified')
ans =
   '/pkg/body/SystemConstantValues/A'
   '/pkg/body/SystemConstantValues/B'
   '/pkg/body/SystemConstantValues/C'
   '/pkg/body/SystemConstantValues/D'
```

2. Create a model from the `arxml` file, and specify a `PredefinedVariant` or one or more `SwSystemconstantValueSet`s.

   This example specifies `PredefinedVariant` Senior, which describes a combination of values for `Transmission`, `Headlight`, `Sunroof`, and `Turbocharge`.

   ```
   >> createComponentAsModel(obj,compNames{1},'CreateInternalBehavior',true,...
       'PredefinedVariant','/pkg/body/Variants/Senior');
   ```

   This example specifies `SwSystemconstantValueSet`s A and B, which together provide values for `SwSystemconst`s in the AUTOSAR software component.

   ```
   >> createComponentAsModel(obj,compNames{1},'CreateInternalBehavior',true,...
       '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 `SwSystemconstantValueSet`s.

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

In Simulink, you can redefine the `SwSystemconst` data that controls variation points without recreating the model. Call the AUTOSAR property function `createSystemConstants`, and specify a different imported `PredefinedVariant` or a different cell array of `SwSystemconstantValueSet`s. 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 `PredefinedVariant`s and `SwSystemconstantValueSet`s to `arxml` code.

Related Examples

- “Model AUTOSAR Variants” on page 2-46
- “Configure AUTOSAR Variants in Ports and Runnables” on page 4-256

More About

- “System Constants” on page 2-30

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 Properties** and **Simulink-AUTOSAR Mapping** views of the Configure AUTOSAR Interface dialog box.

In this section...
"AUTOSAR Property and Map Functions" on page 4-267
"Tree View of AUTOSAR Configuration" on page 4-268
"Properties of AUTOSAR Elements" on page 4-269
"Specify AUTOSAR Element Location" on page 4-272

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 "AUTOSAR Component Development".

For example scripts, see "AUTOSAR Property and Map Function Examples" on page 4-274.

Note: For information about functions for creating or importing an AUTOSAR software component, see "AUTOSAR 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 Properties Explorer** view in the Configure AUTOSAR Interface dialog box, 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
      - ClientPorts
      - ServerPorts
      - NvReceiverPorts
      - NvSenderPorts
      - NvSenderReceiverPorts
      - Runnables
      - IRV
  - S-R Interfaces
    - *`SRInterface1`*
      - DataElements
  - M-S Interfaces
    - *`MSInterface1`*
  - C-S Interfaces
    - *`CSInterface1`*
      - Operations

        - *operation1*
            - Arguments
- NV Interfaces
    - *NVInterface1*
        - DataElements
- 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) • `Behavior` (add/delete) • `Kind` • `Name`
`ApplicationComponentBehavior`	• `Runnables` (add/delete) • `Events` (add/delete) • `PIM` (add/delete) • `IRV` (add/delete) • `Parameters` (add/delete)

AUTOSAR Element Class	Properties
	• DataTypeMapping • Name
DataReceiverPort DataSenderPort DataSenderReceiverPort ClientPort ServerPort ModeReceiverPort NvDataReceiverPort NvDataSenderPort NvDataSenderReceiverPort	• Interface • Name
Runnable	• symbol • canBeInvokedConcurrently • SwAddrMethod • Name
TimingEvent	• Period • StartOnEvent • DisabledMode • Name
DataReceivedEvent	• Trigger • StartOnEvent • DisabledMode • Name
DataReceiveErrorEvent	• Trigger • StartOnEvent • DisabledMode • Name

AUTOSAR Element Class	Properties
`ModeSwitchEvent`	• `Trigger` • `Activation` • `StartOnEvent` • `DisabledMode` • `Name`
`OperationInvokedEvent`	• `Trigger` • `StartOnEvent` • `DisabledMode` • `Name`
`InitEvent`	• `StartOnEvent` • `Name`
`IrvData`	• `Type` • `SwAddrMethod` • `SwCalibrationAccess` • `DisplayFormat` • `SwAlignment` • `Name`
`SenderReceiverInterface`	• `DataElements` (add/delete) • `IsService` • `Name`
`FlowData`	• `Type` • `SwAddrMethod` • `InvalidationPolicy` • `SwCalibrationAccess` • `DisplayFormat` • `SwAlignment` • `Name`

AUTOSAR Element Class	Properties
`ModeSwitchInterface`	• `ModeGroup` (add/delete) • `IsService` • `Name`
`ModeDeclarationGroupElement`	• `ModeGroup` • `SwCalibrationAccess` • `DisplayFormat` • `Name`
`ClientServerInterface`	• `Operations` (add/delete) • `IsService` • `Name`
`NvDataInterface`	• `DataElements` (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
```

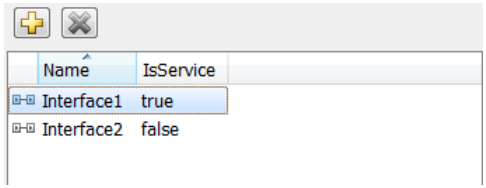
For AUTOSAR property functions other than `addPackageableElement`, you can specify a partially-qualified path that does not include the package hierarchy, for example:

```
grandParentName/parentName/childName
```

The following code sets the `IsService` property for the Sender-Receiver Interface located at path `Interface1` in the example model `rtwdemo_autosar_multirunnables` to `true`. In this case, specifying the name `Interface1` is enough to locate the element.

```
>> arProps = autosar.api.getAUTOSARProperties('rtwdemo_autosar_multirunnables');
>> set(arProps,'Interface1','IsService',true);
```

Here is the resulting display in the S-R Interfaces view of the Configure AUTOSAR Interface dialog box.



If you added a Sender-Receiver Interface to a component package, you would specify a fully qualified path, for example:

```
>> arProps = autosar.api.getAUTOSARProperties('rtwdemo_autosar_multirunnables');
>> 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 Configure AUTOSAR Interface dialog box view of the AUTOSAR configuration. A potential disadvantage is that a partially qualified path could refer to more than one element in the AUTOSAR configuration. For example, the path `s/r` conceivably might designate both a data element of a Sender-Receiver Interface and a runnable of a component. When a conflict occurs, the software displays an error and lists the fully-qualified paths.

Most AUTOSAR elements have properties that are made up of multiple parts (composite). For example, an atomic software component has composite properties such as `ReceiverPorts`, `SenderPorts`, and `InternalBehavior`. For elements that have composite properties that you can manipulate, such as property `ReceiverPorts` of a component, child elements are named and are uniquely defined within the parent element. To locate a child element within a composite property, use the parent element path and the child name, without the property name. For example, if the qualified path of a parent atomic software component is `/A/B/SWC`, and a child receiver port is named `RPort1`, the location of the receiver port is `/A/B/SWC/RPort1`.

AUTOSAR Property and Map Function Examples

After creating a Simulink model representation of an AUTOSAR software component, you refine the AUTOSAR configuration. You can refine the AUTOSAR configuration graphically, using the Configure AUTOSAR Interface dialog box, 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 "AUTOSAR Component Creation". (To graphically refine an AUTOSAR configuration, see "AUTOSAR Interface Configuration" on page 4-3.)

Here is representative ordering of programmatic configuration tasks.

1. "Configure AUTOSAR Software Component" on page 4-275
   a. "Configure AUTOSAR Software Component Name and Type" on page 4-275
   b. "Configure AUTOSAR Ports" on page 4-276
   c. "Configure AUTOSAR Runnables and Events" on page 4-280
   d. "Configure AUTOSAR Inter-Runnable Variables" on page 4-285
2. "Configure AUTOSAR Interfaces" on page 4-287
   a. "Configure AUTOSAR Sender-Receiver Interfaces" on page 4-288
   b. "Configure AUTOSAR Client-Server Interfaces" on page 4-289
   c. "Configure AUTOSAR Mode-Switch Interfaces" on page 4-292
3. "Configure AUTOSAR XML Export" on page 4-293

For a list of AUTOSAR property and map functions, see the **Functions** list on the "AUTOSAR Component Development" 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 synchronize and validate AUTOSAR model configurations:

- `autosar.api.syncModel` — Synchronize 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 **Synchronize** and **Validate** icons in the graphical views of an AUTOSAR configuration.

Configure AUTOSAR Software Component

- "Configure AUTOSAR Software Component Name and Type" on page 4-275
- "Configure AUTOSAR Ports" on page 4-276
- "Configure AUTOSAR Runnables and Events" on page 4-280
- "Configure AUTOSAR Inter-Runnable Variables" on page 4-285

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 = 'rtwdemo_autosar_multirunnables';
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 or SensorActuator
    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 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 4-276
- “Configure Client-Server Ports” on page 4-278
- “Configure and Map Mode Receiver Ports” on page 4-279

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 4-288.

```
% Open model
hModel = 'rtwdemo_autosar_multirunnables';
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 =
    ''


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 4-289.

```
% Open model
hModel = 'mControllerWithInterface_server';
addpath (fullfile(matlabroot,'/help/toolbox/ecoder/examples/autosar'));
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 4-292.

```
% Open model
hModel = 'rtwdemo_autosar_multirunnables_msconfig_after';
% Add path to model and mode definition files
addpath (fullfile(matlabroot,'/help/toolbox/ecoder/examples/autosar'));
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 =
     ''

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 communcation.
- `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.

- “Configure AUTOSAR TimingEvent for Periodic Runnable” on page 4-281
- “Configure and Map Runnables” on page 4-282
- “Configure Events for Runnable Activation” on page 4-283
- “Gather Information for AUTOSAR Custom Scheduler Script” on page 4-284

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 = 'rtwdemo_autosar_multirunnables';
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/Behavior/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/Behavior/Event_t_1tic_A triggers ASWC/Behavior/myRunnable
AUTOSAR event /pkg/swc/ASWC/Behavior/Event_t_1tic_B triggers ASWC/Behavior/Runnable2
AUTOSAR event /pkg/swc/ASWC/Behavior/Event_t_10tic triggers ASWC/Behavior/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/Behavior/myEvent triggers ASWC/Behavior/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 4-283.

```
% Open model
hModel = 'rtwdemo_autosar_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')
```

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 4-289.

For `ModeSwitchEvent` syntax, see “Configure AUTOSAR Mode-Switch Interfaces” on page 4-292.

```
% Open model
hModel = 'rtwdemo_autosar_multirunnables';
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')
```

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 rtwdemo_autosar_multirunnables 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 1
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/ecoder/examples/autosar/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 4-286
- "Configure Inter-Runnable Variable for Data Transfer with Rate Transition" on page 4-286

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 = 'rtwdemo_autosar_multirunnables';
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')
```

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/ecoder/examples/autosar'));
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')
```

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 4-288
- “Configure AUTOSAR Client-Server Interfaces” on page 4-289

- "Configure AUTOSAR Mode-Switch Interfaces" on page 4-292

Configure AUTOSAR Sender-Receiver Interfaces

- "Configure and Map Sender-Receiver Interface" on page 4-288
- "Configure Sender-Receiver Data Element Properties" on page 4-289

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 4-280.

```
% Open model
hModel = 'rtwdemo_autosar_multirunnables';
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')
```

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 = 'rtwdemo_autosar_multirunnables';
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
```

Configure AUTOSAR Client-Server Interfaces

- “Configure Server Properties” on page 4-289
- “Configure Client Properties” on page 4-291

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/ecoder/examples/autosar'));
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')
```

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/ecoder/examples/autosar'));
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')
```

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/ecoder/examples/autosar'));
hModel = 'rtwdemo_autosar_multirunnables_msconfig';
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/ecoder/examples/autosar/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')
```

Configure AUTOSAR XML Export

- "Configure XML Export Options" on page 4-294
- "Configure AUTOSAR Package Paths" on page 4-294

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 4-294.

```
% Open model
hModel = 'rtwdemo_autosar_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')
```

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 4-72.

To configure other XML export options, see “Configure XML Export Options” on page 4-294.

```
% Open model
hModel = 'rtwdemo_autosar_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','/rtwdemo_autosar_counter_pkg/ADTs');
get(arProps,'XmlOptions','ApplicationDataTypePackage')
```

See Also

`get` | `set`

Related Examples

- “Configure and Map AUTOSAR Component Programmatically” on page 4-267

More About

- "AUTOSAR Interface 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 4-296
“Use the Merge Block for Inter-Runnable Variables” on page 4-296

AUTOSAR Client Block in Referenced Model

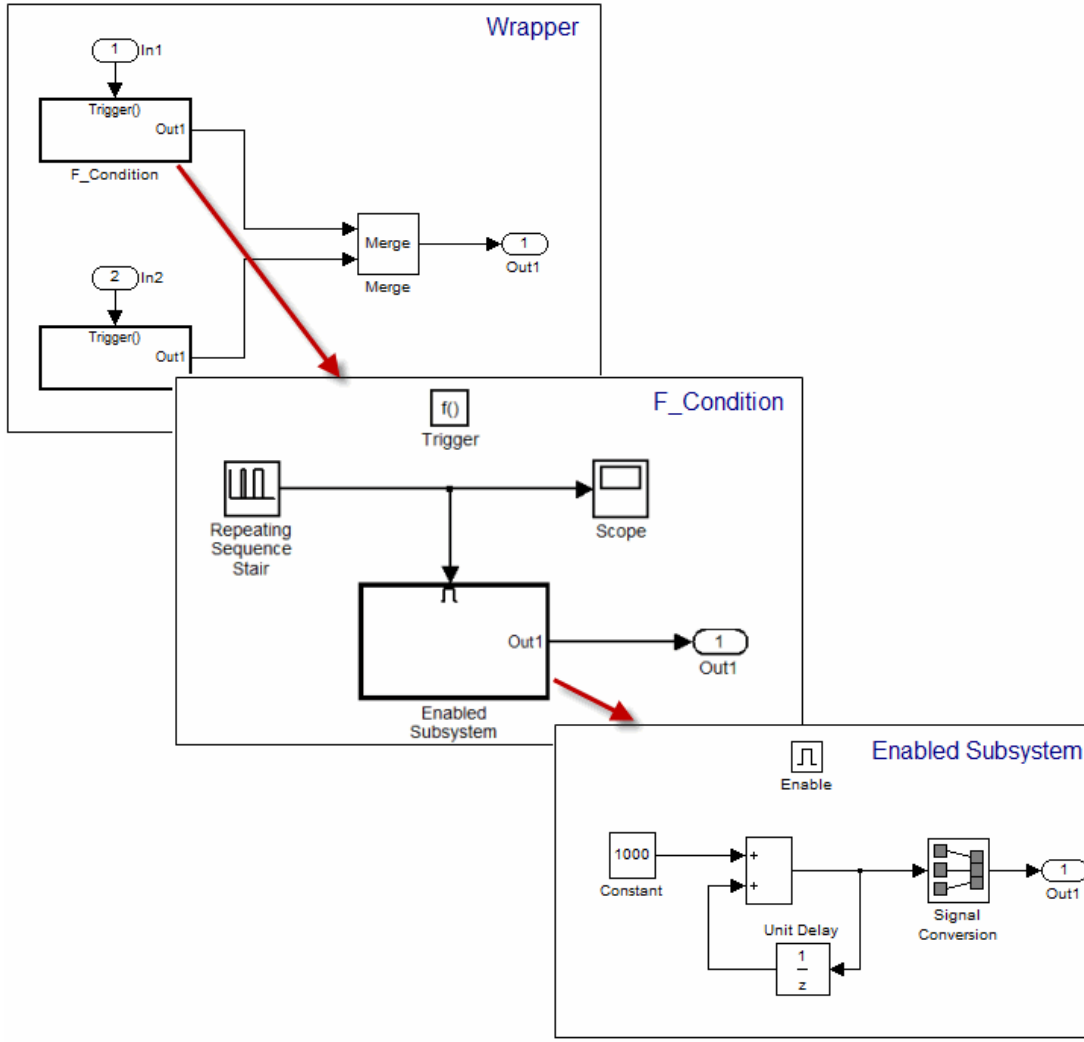
The software does not support the use of an AUTOSAR client block, such as Function Caller or Invoke AUTOSAR Server Operation, in a referenced model.

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.

5

AUTOSAR Code Generation

- "Export AUTOSAR Component XML and C Code" on page 5-2
- "Code Generation with AUTOSAR Library" on page 5-8
- "Verify AUTOSAR C Code with SIL and PIL" on page 5-29
- "Limitations and Tips" on page 5-30

Export AUTOSAR Component XML and C Code

To generate AUTOSAR component XML code and AUTOSAR-compliant C code from a model configured for AUTOSAR:

1. Examine the AUTOSAR XML options that you configured using the AUTOSAR Properties Explorer or AUTOSAR property functions.
2. Examine the AUTOSAR code generation parameters on the **Code Generation** > **AUTOSAR Code Generation Options** pane of the Configuration Parameters dialog box.
3. Build the model.

This topic explains the AUTOSAR code generation options and their effects on generated `arxml` and C code.

In this section...
"Inspect XML Options" on page 5-2
"Select an AUTOSAR Schema" on page 5-2
"Specify Maximum SHORT-NAME Length" on page 5-4
"Configure AUTOSAR Compiler Abstraction Macros" on page 5-4
"Root-Level Matrix I/O" on page 5-6
"Export AUTOSAR Software Component" on page 5-6

Inspect XML Options

Examine the XML options that you configured using the AUTOSAR Properties Explorer. If you have not yet configured them, see XML Options View of AUTOSAR Properties Explorer.

Select an AUTOSAR Schema

The software supports the following AUTOSAR 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
`4.2`	4.2.1, 4.2.2	4.2.2

Schema Version Value	Schema Revisions Supported for Import	Export Schema Revision
`4.1`	4.1.1, 4.1.2, 4.1.3	4.1.3
`4.0` (default)	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
`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, `4.0`.

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

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**.

Export AUTOSAR Software Component

After configuring your AUTOSAR export options, generate code to export your AUTOSAR software component. To generate C and XML code files, build the model (**Ctrl +B**).

The build process generates software component XML and C files to the build folder. The following table shows which XML files are generated, based on the value of the **Exported XML file packaging** option you configured using the AUTOSAR Properties Explorer. (For more information about configuring XML properties in AUTOSAR Properties Explorer, see XML Options View of AUTOSAR Properties Explorer.)

Exported XML File Packaging Value	Exported File Name	By Default Contains...
`Single file`	*`modelname`*`.arxml`	All AUTOSAR elements.
`Modular`	*`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 any elements for which AUTOSAR packages (AR-PACKAGEs) are not configured, and any 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 4-70.
	`modelname``_datatype.arxml`	Data types and related elements.

Exported XML File Packaging Value	Exported File Name	By Default Contains...
	modelname-_implementation.arxml	Software component implementation.
	*modelname*_interface.arxml	Interfaces, including S-R, C-S, M-S, and NV interfaces.
	*modelname*_behavior.arxml	Software component internal behavior (generated only for schema 3.x or earlier).

You can merge the software component information back into an AUTOSAR authoring tool. This software 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 Elements and UUIDs” on page 3-14.

For examples of how to generate AUTOSAR-compliant code and export AUTOSAR software component description XML files from a Simulink model, see:

- “Generate Code That Is Compliant with the AUTOSAR Standard”.
- “Generate AUTOSAR-Compliant Code for Multiple Runnable Entities”.

Code Generation with AUTOSAR Library

If your model is configured for AUTOSAR code generation, you can use the AUTOSAR 4.0 code replacement library to produce code that more closely aligns with the AUTOSAR standard. The AUTOSAR 4.0 code replacement library is intended for use with AUTOSAR schema version 4.0 or later.

In this section...

"AUTOSAR Code Replacement Library" on page 5-8

"Supported AUTOSAR Library Routines" on page 5-9

"Configure Code Generator to Use AUTOSAR Code Replacement Library" on page 5-9

"Replace Code with Functions Compatible with AUTOSAR IFL and IFX Library Routines" on page 5-9

"Required Algorithm Property Settings for IFL/IFX Function and Block Mappings" on page 5-10

"Code Replacement Checks for AUTOSAR Lookup Table Functions" on page 5-27

AUTOSAR Code Replacement Library

The AUTOSAR 4.0 code replacement library provides a way for you to customize the C/C++ code generator to produce code that more closely aligns with the AUTOSAR standard. Considering using the 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 routines for a 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 "AUTOSAR 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?" and "Code Replacement Libraries".

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, enter `crviewer` at the command prompt.

For more information, see "Choose a Code Replacement Library".

Configure Code Generator to Use AUTOSAR Code Replacement Library

To configure the code generator to apply the AUTOSAR code replacement library, select `AUTOSAR 4.0` for the **Code replacement library** model configuration parameter.

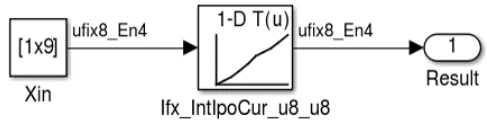
For more information on code replacement and code replacement libraries, see "What Is Code Replacement?" and "Code Replacement Libraries".

Replace Code with Functions Compatible with AUTOSAR IFL and IFX Library Routines

To replace code generated for Simulink lookup table blocks with functions that are compatible with AUTOSAR floating-point interpolation (IFL) and fixed-point interpolation (IFX) library routines:

1. In your Simulink model, use the Simulink lookup table blocks Interpolation Using Prelookup, Prelookup, and n-D Lookup Table.

   For example:



2. For each lookup table block in the model, use information in "Required Algorithm Property Settings for IFL/IFX Function and Block Mappings" on page 5-10 to configure the block algorithm parameters. Given those parameter settings, the code generator produces code that is compatible with a corresponding AUTOSAR IFX or IFL routine.
3. Configure the model for the code generator 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**. From the command line or programmatically, use `set_param` to set the `CodeReplacementLibrary` parameter to `'AUTOSAR 4.0'`.
4. Optionally, 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, on the **All Parameters** tab, select **Summarize which blocks triggered code replacements**. In the command line or programmatically, use `set_param` to set the `GenerateCodeReplacementReport` parameter to `'on'`.
5. Generate code.
6. Review the generated code for expected code replacements. For example:

```
for (iU = 0; iU < 9; iU++) {
  /* Lookup_n-D: '<Root>/Ifx_IntIpoCur_u8_u8' incorporates:
   *  Constant: '<Root>/Xin'
   */
  rtb_Ifx_IntIpoCur_u8_u8 = Ifx_IntIpoCur_u8_u8(look01_ConstP.Xin_Value[iU],
    10U, (*Rte_CData_X_array_u8()), (*Rte_CData_Val_array_u8()));

  /* Outport: '<Root>/Result' */
  look01_Y.Result[iU] = rtb_Ifx_IntIpoCur_u8_u8;
}
```

Required Algorithm Property Settings for IFL/IFX Function and Block Mappings

IFL/IFX Function and Block Mapping	Algorithm Property Parameters	Value
`Ifl_DPSearch` Prelookup	**Extrapolation method** `ExtrapMethod`	`Clip`
	Index search method `IndexSearchMethod`	`Linear search` or `Binary search`

IFL/IFX Function and Block Mapping	Algorithm Property Parameters	Value
	Use last breakpoint for input at or above upper limit `UseLastBreakPoint`	On
	Remove protection against out-of-range input in generated code `RemoveProtectionInput`	Off
	Integer rounding mode `RndMeth`	`Round` or `Zero`
`Ifl_IpoCur` Interpolation Using Prelookup	**Interpolation method** `InterpMethod`	`Linear`
	Extrapolation method `ExtrapMethod`	`Clip`
	Valid index input may reach last index `ValidIndexMayReachLast`	On
	Remove protection against out-of-range input in generated code `RemoveProtectionInput`	Off
	Integer rounding mode `RndMeth`	`Round` or `Zero`
`Ifl_IpoMap` Interpolation Using Prelookup	**Interpolation method** `InterpMethod`	`Linear`

IFL/IFX Function and Block Mapping	Algorithm Property Parameters	Value
	Extrapolation method `ExtrapMethod`	`Clip`
	Valid index input may reach last index `ValidIndexMayReachLast`	On
	Remove protection against out-of-range input in generated code `RemoveProtectionInput`	Off
	Integer rounding mode `RndMeth`	`Round` or `Zero`
`Ifl_IntIpoCur` n-D Lookup Table	**Interpolation method** `InterpMethod`	`Linear`
	Extrapolation method `ExtrapMethod`	`Clip`
	Index search method `IndexSearchMethod`	`Linear search` or `Binary search`
	Use last table value for inputs at or above last breakpoint `UseLastTableValue`	On
	Remove protection against out-of-range input in generated code `RemoveProtectionInput`	Off

IFL/IFX Function and Block Mapping	Algorithm Property Parameters	Value
	Integer rounding mode `RndMeth`	`Round` or `Zero`
`Ifl_IntIpoMap` n-D Lookup Table	**Interpolation method** `InterpMethod`	`Linear`
	Extrapolation method `ExtrapMethod`	`Clip`
	Index search method `IndexSearchMethod`	`Linear search` or `Binary search`
	Use last table value for inputs at or above last breakpoint `UseLastTableValue`	On
	Remove protection against out-of-range input in generated code `RemoveProtectionInput`	Off
	Integer rounding mode `RndMeth`	`Round` or `Zero`
`Ifx_DPSearch` Prelookup	**Extrapolation method** `ExtrapMethod`	`Clip`
	Index search method `IndexSearchMethod`	`Linear search` or `Binary search`
	Use last breakpoint for input at or above upper limit `UseLastBreakPoint`	On

IFL/IFX Function and Block Mapping	Algorithm Property Parameters	Value
	Remove protection against out-of-range input in generated code `RemoveProtectionInput`	Off
	Integer rounding mode `RndMeth`	`Round` or `Zero`
`Ifx_IpoCur` Interpolation Using Prelookup	**Interpolation method** `InterpMethod`	`Linear`
	Extrapolation method `ExtrapMethod`	`Clip`
	Valid index input may reach last index `ValidIndexMayReachLast`	On
	Remove protection against out-of-range input in generated code `RemoveProtectionInput`	Off
	Integer rounding mode `RndMeth`	`Round` or `Zero`
`Ifx_LkUpCur` Interpolation Using Prelookup	**Interpolation method** `InterpMethod`	`Flat`
	Extrapolation method `ExtrapMethod`	`Clip`

IFL/IFX Function and Block Mapping	Algorithm Property Parameters	Value
	Remove protection against out-of-range input in generated code `RemoveProtectionInput`	Off
	Integer rounding mode `RndMeth`	`Round` or `Zero`
	Valid index input may reach last index `ValidIndexMayReachLast`	On
`Ifx_IpoMap` Interpolation Using Prelookup	**Interpolation method** `InterpMethod`	`Linear`
	Extrapolation method `ExtrapMethod`	`Clip`
	Valid index input may reach last index `ValidIndexMayReachLast`	On
	Remove protection against out-of-range input in generated code `RemoveProtectionInput`	Off
	Integer rounding mode `RndMeth`	`Round` or `Zero`
`Ifx_LkUpMap` Interpolation Using Prelookup	**Interpolation method** `InterpMethod`	`Nearest`

IFL/IFX Function and Block Mapping	Algorithm Property Parameters	Value
	Extrapolation method `ExtrapMethod`	`Clip`
	Remove protection against out-of-range input in generated code `RemoveProtectionInput`	Off
	Integer rounding mode `RndMeth`	`Round` or `Zero`
	Valid index input may reach last index `ValidIndexMayReachLast`	On
`Ifx_LkUpBaseMap` Interpolation Using Prelookup	**Interpolation method** `InterpMethod`	`Flat`
	Extrapolation method `ExtrapMethod`	`Clip`
	Remove protection against out-of-range input in generated code `RemoveProtectionInput`	Off
	Integer rounding mode `RndMeth`	`Round` or `Zero`
	Valid index input may reach last index `ValidIndexMayReachLast`	On

IFL/IFX Function and Block Mapping	Algorithm Property Parameters	Value
`Ifx_IntIpoCur` n-D Lookup Table	**Interpolation method** `InterpMethod`	`Linear`
	Extrapolation method `ExtrapMethod`	`Clip`
	Index search method `IndexSearchMethod`	`Linear search` or `Binary search`
	Use last table value for inputs at or above last breakpoint `UseLastTableValue`	On
	Remove protection against out-of-range input in generated code `RemoveProtectionInput`	Off
	Integer rounding mode `RndMeth`	`Round` or `Zero`
`Ifx_IntLkUpCur` n-D Lookup Table	**Interpolation method** `InterpMethod`	`Flat`
	Extrapolation method `ExtrapMethod`	`Clip`
	Index search method `IndexSearchMethod`	`Linear search` or `Binary search`
	Remove protection against out-of-range input in generated code `RemoveProtectionInput`	Off

IFL/IFX Function and Block Mapping	Algorithm Property Parameters	Value
	Integer rounding mode `RndMeth`	`Round` or `Zero`
	Use last table value for inputs at or above last breakpoint `UseLastTableValue`	On
`Ifx_IntIpoFixCur` n-D Lookup Table	**Interpolation method** `InterpMethod`	`Linear`
	Extrapolation method `ExtrapMethod`	`Clip`
	Index search method `IndexSearchMethod`	`Evenly spaced points`
	Use last table value for inputs at or above last breakpoint `UseLastTableValue`	On
	Remove protection against out-of-range input in generated code `RemoveProtectionInput`	Off
	Integer rounding mode `RndMeth`	`Round` or `Zero`
	Use last table value for inputs at or above last breakpoint `UseLastTableValue`	On

IFL/IFX Function and Block Mapping	Algorithm Property Parameters	Value
	Model configuration parameter **Optimization > Signals and Parameters > Default parameter behavior** `DefaultParameterBehavior`	`Inlined`
	Breakpoint data should match power 2 spacing.	
`Ifx_IntLkUpFixCur` n-D Lookup Table	**Interpolation method** `InterpMethod`	`Flat`
	Extrapolation method `ExtrapMethod`	`Clip`
	Index search method `IndexSearchMethod`	`Evenly spaced points`
	Remove protection against out-of-range input in generated code `RemoveProtectionInput`	Off
	Integer rounding mode `RndMeth`	`Round` or `Zero`
	Model configuration parameter **Optimization > Signals and Parameters > Default parameter behavior** `DefaultParameterBehavior`	`Inlined`
	Breakpoint data must match power 2 spacing.	

IFL/IFX Function and Block Mapping	Algorithm Property Parameters	Value
`Ifx_IntIpoFixICur` n-D Lookup Table	**Interpolation method** `InterpMethod`	`Linear`
	Extrapolation method `ExtrapMethod`	`Clip`
	Index search method `IndexSearchMethod`	`Evenly spaced points`
	Use last table value for inputs at or above last breakpoint `UseLastTableValue`	On
	Remove protection against out-of-range input in generated code `RemoveProtectionInput`	Off
	Integer rounding mode `RndMeth`	`Round` or `Zero`
	Breakpoint data must not match power 2 spacing.	
`Ifx_IntLkUpFixICur` n-D Lookup Table	**Interpolation method** `InterpMethod`	`Flat`
	Extrapolation method `ExtrapMethod`	`Clip`
	Index search method `IndexSearchMethod`	`Evenly spaced points`

IFL/IFX Function and Block Mapping	Algorithm Property Parameters	Value
	Remove protection against out-of-range input in generated code `RemoveProtectionInput`	Off
	Integer rounding mode `RndMeth`	`Round` or `Zero`
	Use last table value for inputs at or above last breakpoint `UseLastTableValue`	On
	Breakpoint data must not match power 2 spacing.	
`Ifx_IntIpoMap` n-D Lookup Table	**Interpolation method** `InterpMethod`	`Linear`
	Extrapolation method `ExtrapMethod`	`Clip`
	Index search method `IndexSearchMethod`	`Linear search` or `Binary search`
	Use last table value for inputs at or above last breakpoint `UseLastTableValue`	On
	Remove protection against out-of-range input in generated code `RemoveProtectionInput`	Off
	Integer rounding mode `RndMeth`	`Round` or `Zero`

IFL/IFX Function and Block Mapping	Algorithm Property Parameters	Value
`Ifx_IntLkUpMap` n-D Lookup Table	**Interpolation method** `InterpMethod`	`Nearest`
	Extrapolation method `ExtrapMethod`	`Clip`
	Index search method `IndexSearchMethod`	`Linear search` or `Binary search`
	Remove protection against out-of-range input in generated code `RemoveProtectionInput`	Off
	Integer rounding mode `RndMeth`	`Round` or `Zero`
	Use last table value for inputs at or above last breakpoint `UseLastTableValue`	On
`Ifx_IntLkUpBaseMap` n-D Lookup Table	**Interpolation method** `InterpMethod`	`Flat`
	Extrapolation method `ExtrapMethod`	`Clip`
	Index search method `IndexSearchMethod`	`Linear search` or `Binary search`
	Remove protection against out-of-range input in generated code `RemoveProtectionInput`	Off

IFL/IFX Function and Block Mapping	Algorithm Property Parameters	Value
	Integer rounding mode `RndMeth`	`Round` or `Zero`
	Use last table value for inputs at or above last breakpoint `UseLastTableValue`	On
`Ifx_IntIpoFixMap` n-D Lookup Table	**Interpolation method** `InterpMethod`	`Linear`
	Extrapolation method `ExtrapMethod`	`Clip`
	Index search method `IndexSearchMethod`	`Evenly spaced points`
	Use last table value for inputs at or above last breakpoint `UseLastTableValue`	On
	Remove protection against out-of-range input in generated code `RemoveProtectionInput`	Off
	Integer rounding mode `RndMeth`	`Round` or `Zero`
	Model configuration parameter **Optimization > Signals and Parameters > Default parameter behavior** `DefaultParameterBehavior`	`Inlined`

IFL/IFX Function and Block Mapping	Algorithm Property Parameters	Value
	Breakpoint data must match power 2 spacing.	
`Ifx_IntLkUpFixMap` n-D Lookup Table	**Interpolation method** `InterpMethod`	`Nearest`
	Extrapolation method `ExtrapMethod`	`Clip`
	Index search method `IndexSearchMethod`	`Evenly spaced points`
	Remove protection against out-of-range input in generated code `RemoveProtectionInput`	Off
	Integer rounding mode `RndMeth`	`Round` or `Zero`
	Use last table value for inputs at or above last breakpoint `UseLastTableValue`	On
	Model configuration parameter **Optimization > Signals and Parameters > Default parameter behavior** `DefaultParameterBehavior`	`Inlined`
	Breakpoint data must match power 2 spacing.	
`Ifx_IntLkUpFixBaseMap` n-D Lookup Table	**Interpolation method** `InterpMethod`	`Flat`

IFL/IFX Function and Block Mapping	Algorithm Property Parameters	Value
	Extrapolation method `ExtrapMethod`	`Clip`
	Index search method `IndexSearchMethod`	`Evenly spaced points`
	Remove protection against out-of-range input in generated code `RemoveProtectionInput`	Off
	Integer rounding mode `RndMeth`	`Round` or `Zero`
	Use last table value for inputs at or above last breakpoint `UseLastTableValue`	On
	Model configuration parameter **Optimization > Signals and Parameters > Default parameter behavior** `DefaultParameterBehavior`	`Inlined`
	Breakpoint data must match power 2 spacing.	
`Ifx_IntIpoFixIMap` n-D Lookup Table	**Interpolation method** `InterpMethod`	`Linear`
	Extrapolation method `ExtrapMethod`	`Linear`
	Index search method `IndexSearchMethod`	`Evenly spaced points`

IFL/IFX Function and Block Mapping	Algorithm Property Parameters	Value
	Use last table value for inputs at or above last breakpoint `UseLastTableValue`	On
	Remove protection against out-of-range input in generated code `RemoveProtectionInput`	Off
	Integer rounding mode `RndMeth`	`Round` or `Zero`
	Breakpoint data must not match power 2 spacing.	
`Ifx_IntLkUpFixIMap` n-D Lookup Table	**Interpolation method** `InterpMethod`	`Nearest`
	Extrapolation method `ExtrapMethod`	`Clip`
	Index search method `IndexSearchMethod`	`Evenly spaced points`
	Remove protection against out-of-range input in generated code `RemoveProtectionInput`	Off
	Integer rounding mode `RndMeth`	`Round` or `Zero`
	Use last table value for inputs at or above last breakpoint `UseLastTableValue`	On

IFL/IFX Function and Block Mapping	Algorithm Property Parameters	Value
	Breakpoint data must not match power 2 spacing.	
`Ifx_IntLkUpFixIBaseMap` n-D Lookup Table	**Interpolation method** `InterpMethod`	`Flat`
	Extrapolation method `ExtrapMethod`	`Clip`
	Index search method `IndexSearchMethod`	`Evenly spaced points`
	Remove protection against out-of-range input in generated code `RemoveProtectionInput`	Off
	Integer rounding mode `RndMeth`	`Round` or `Zero`
	Use last table value for inputs at or above last breakpoint `UseLastTableValue`	On
	Breakpoint data must not match power 2 spacing.	

Code Replacement Checks for AUTOSAR Lookup Table Functions

The following checks occur during the code replacement match process for AUTOSAR lookup table functions:

Function Type	Match Process Checks Whether
n-D lookup	• Input and corresponding breakpoint arguments have the same data type.

Function Type	Match Process Checks Whether
	• Output and the table data argument have the same data type.
Interpolation using prelookup	Output and the table data argument have the same data type.
Prelookup	Input and break point arguments have the same data type.

Verify AUTOSAR 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 the AUTOSAR system target file (`autosar.tlc`) using the `Software-in-the-Loop (SIL)` and `Processor-in-the-Loop (PIL)` simulation modes.
- You can use Model block SIL/PIL to test AUTOSAR top-model code or referenced model code. (See Model block parameter **Code interface**, which can be set to `Top model` or `Model reference`.)

For more information, see “Simulation with Top Model” and “Simulation with Model Blocks”.

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”. 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.

Limitations and Tips

The following limitations apply to AUTOSAR code generation.

In this section...
"Generate Code Only Check Box" on page 5-30
"AUTOSAR Compiler Abstraction Macros" on page 5-30
"Relative File Paths in AUTOSAR Code Descriptors (Schema Versions 3.x and Earlier)" on page 5-31

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 `rtwdemo_autosar_counter`, the generated file `rtwdemo_autosar_counter_implementation.arxml` has the following `XFILE` element:

```
<XFILE>
  <SHORT-NAME>rtwdemo_autosar_counter_c</SHORT-NAME>
  <CATEGORY>GeneratedFile</CATEGORY>
  <URL>rtwdemo_autosar_counter_autosar_rtw\rtwdemo_autosar_counter.c</URL>
  <TOOL>Embedded Coder</TOOL>
  <TOOL-VERSION>5.6</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.

Functions — Alphabetical List

add
addPackageableElement
arxml.importer
arxml.importer
autosar.api.create
autosar.api.syncModel
autosar.api.validateModel
AUTOSAR.DualScaledParameter
AUTOSAR.Parameter
AUTOSAR.Signal
autosar_ui_close
autosar_ui_launch
AUTOSAR4.Parameter
AUTOSAR4.Signal
createCalibrationComponentObjects
createComponentAsModel
delete
deleteUnmappedComponents
find
get
getApplicationComponentNames
getCalibrationComponentNames
getClientServerInterfaceNames
getComponentNames
getDataTransfer
getDependencies
getFile
getFunction
getFunctionCaller
getInport
getLookupTable
getOutport

getSensorActuatorComponentNames
mapDataTransfer
mapFunction
mapFunctionCaller
mapInport
mapLookupTable
mapOutport
set
setDependencies
setFile
updateModel
updateReferences

add

Add property to AUTOSAR element

Syntax

```
add(arProps,parentPath,property,name)
add(arProps,parentPath,property,name,childproperty,value)
```

Description

`add(arProps,parentPath,property,name)` adds a composite child element with the specified `name` to the AUTOSAR element at `parentPath`, under the specified `property`.

`add(arProps,parentPath,property,name,childproperty,value)` sets the value of a specified property of the added child property element.

Note: This function requires the Embedded Coder Support Package for AUTOSAR Standard. You install support packages using the MATLAB **Add-Ons** menu. On the MATLAB **Home** tab, in the **Environment** section, click **Add-Ons > Get Hardware Support Packages**. For more information, see "Support Package for AUTOSAR Standard" on page 1-3.

Examples

Add Data Element to Sender Interface

Add data element `DE3` to sender interface `Interface1`.

```
rtwdemo_autosar_multirunnables
arProps=autosar.api.getAUTOSARProperties('rtwdemo_autosar_multirunnables');
add(arProps,'Interface1','DataElements','DE3');
get(arProps,'Interface1','DataElements')

ans =
```

```
    'Interface1/DE1'    'Interface1/DE2'    'Interface1/DE3'
```

Add Mode Group to Mode-Switch Interface

Using a fully qualified path, add a mode-switch interface and set the `IsService` property to `true`. Add mode group `mgModes` to the mode-switch interface using the composite property `ModeGroup`.

```
rtwdemo_autosar_multirunnables
arProps=autosar.api.getAUTOSARProperties('rtwdemo_autosar_multirunnables');
addPackageableElement(arProps,'ModeSwitchInterface','/pkg/if','Interface3',...
  'IsService',true);
ifPaths=find(arProps,[],'ModeSwitchInterface','PathType','FullyQualified')
```

```
ifPaths =
    '/pkg/if/Interface3'
```

```
add(arProps,'/pkg/if/Interface3','ModeGroup','mgModes');
```

- “Configure and Map AUTOSAR Component Programmatically” on page 4-267
- “AUTOSAR Interface Configuration” on page 4-3

Input Arguments

`arProps` — AUTOSAR properties information for a model

handle

AUTOSAR properties information for a model, previously returned by *`arProps`* `= autosar.api.getAUTOSARProperties(`*`model`*`)`. *`model`* is a handle or character vector representing the model name.

Example: `arProps`

`parentPath` — Path to a parent AUTOSAR element

character vector

Path to a parent AUTOSAR element to which to add a specified child property element.

Example: `'Input'`

`property` — Type of property

character vector

Type of property to add, among valid properties for the AUTOSAR element.

Example: `'DataElements'`

`name` — Name of child property element

character vector

Name of the child property element to add.

Example: `'DE1'`

`childproperty,value` — Child property and value

name (character vector), value

Child property to set, and its value. Table "Properties of AUTOSAR Elements" on page 4-269 lists properties that are associated with AUTOSAR elements.

Example: `'Name','event1'`

See Also

`delete`

Introduced in R2013b

addPackageableElement

Add element to AUTOSAR package in model

Syntax

```
addPackageableElement(arProps,category,package,name)
addPackageableElement(arProps,category,package,name,property,value)
```

Description

`addPackageableElement(arProps,category,package,name)` adds element `name` of the specified `category` to the specified AUTOSAR `package` in a model configured for AUTOSAR.

`addPackageableElement(arProps,category,package,name,property,value)` sets the value of a specified property of the added element.

Note: This function requires the Embedded Coder Support Package for AUTOSAR Standard. You install support packages using the MATLAB **Add-Ons** menu. On the MATLAB **Home** tab, in the **Environment** section, click **Add-Ons > Get Hardware Support Packages**. For more information, see "Support Package for AUTOSAR Standard" on page 1-3.

Examples

Add Sender-Receiver Interface to Package and Set IsService Property

Using a fully qualified path, add a sender-receiver interface to an interface package and set the `IsService` property to `true`.

```
rtwdemo_autosar_multirunnables
arProps=autosar.api.getAUTOSARProperties('rtwdemo_autosar_multirunnables');
addPackageableElement(arProps,'SenderReceiverInterface','/pkg/if','Interface3',...
  'IsService',true);
ifPaths=find(arProps,[],'SenderReceiverInterface',...
  'IsService',true,'PathType','FullyQualified')
```

```
ifPaths =
    '/pkg/if/Interface3'
```

- "Configure and Map AUTOSAR Component Programmatically" on page 4-267
- "AUTOSAR Interface Configuration" on page 4-3

Input Arguments

`arProps` — AUTOSAR properties information for a model
handle

AUTOSAR properties information for a model, previously returned by *`arProps`* `= autosar.api.getAUTOSARProperties(`*`model`*`)`. *`model`* is a handle or character vector representing the model name.

Example: `arProps`

`category` — Element category
character vector

Category of element to add. Valid category values are `'ClientServerInterface'`, `'DataTypeMappingSet'`, `'ModeDeclarationGroup'`, `'ModeSwitchInterface'`, `'Package'`, `'ParameterComponent'`, `'ParameterInterface'`, `'SenderReceiverInterface'`, `'SwAddrMethod'`, and `'SystemConst'`.

Example: `'SenderReceiverInterface'`

`package` — Package path
character vector

Fully-qualified path to the element package.

Example: `'/pkg/if'`

`name` — Element name
character vector

Name of the element to add.

Example: `'Interface3'`

`property,value` — Element property and value
name (character vector), value

Property/value pairs for setting values of element properties. Table “Properties of AUTOSAR Elements” on page 4-269 lists properties that are associated with AUTOSAR elements.

Example: `'IsService',true`

See Also

`delete`

Introduced in R2014b

arxml.importer class

Package: arxml

Import AUTOSAR component XML

Description

You can use methods of the `arxml.importer` class to import AUTOSAR components in a controlled manner. For example, you can parse an AUTOSAR software component description file exported by DaVinci System Architect (from Vector Informatik Gmbh), and import the component into a Simulink model for subsequent configuration, code generation, and export to XML.

Note: This class requires the Embedded Coder Support Package for AUTOSAR Standard. You install support packages using the MATLAB **Add-Ons** menu. On the MATLAB **Home** tab, in the **Environment** section, click **Add-Ons > Get Hardware Support Packages**. For more information, see "Support Package for AUTOSAR Standard" on page 1-3.

Construction

arxml.importer	Construct `arxml.importer` object

Methods

createCalibrationComponentObjects	Create Simulink calibration objects from AUTOSAR calibration component
createComponentAsModel	Create AUTOSAR atomic software component as Simulink model
getApplicationComponentNames	Get list of application software component names

getCalibrationComponentNames	Get calibration component names
getClientServerInterfaceNames	Get list of client-server interfaces
getComponentNames	Get application and sensor/actuator software component names
getDependencies	Get list of XML dependency files
getFile	Return software component XML file name
getSensorActuatorComponentNames	Get list of sensor/actuator software component names
setDependencies	Set XML file dependencies
setFile	Set software component XML file name
updateModel	Update AUTOSAR model with `arxml` changes
updateReferences	Update model with `arxml` definitions of AUTOSAR reference elements

Copy Semantics

Handle. To learn how this affects your use of the class, see Copying Objects in the MATLAB Programming Fundamentals documentation.

Introduced in R2008a

arxml.importer

Class: arxml.importer
Package: arxml

Construct `arxml.importer` object

Syntax

```
importerObj = arxml.importer(filename)
importerObj = arxml.importer({filename1,filename2,...,filenameN})
```

Description

`importerObj = arxml.importer(filename)` constructs an `arxml.importer` object and parses the AUTOSAR information contained in the XML file specified by *`filename`*.

`importerObj = arxml.importer({filename1,filename2,...,filenameN})` constructs an `arxml.importer` object and parses the AUTOSAR information contained in the XML files that are specified in the cell array. The cell array format allows you to specify multiple XML files that are required for an AUTOSAR import operation in one function call.

Note: This method requires the Embedded Coder Support Package for AUTOSAR Standard. You install support packages using the MATLAB **Add-Ons** menu. On the MATLAB **Home** tab, in the **Environment** section, click **Add-Ons** > **Get Hardware Support Packages**. For more information, see "Support Package for AUTOSAR Standard" on page 1-3.

Tip

If you enter the `arxml.importer` constructor 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 can specify in calls to `createComponentAsModel` and `createCalibrationComponentObjects`.

Input Arguments

`filename`	Name of XML file containing AUTOSAR information.
{*`filename1`*,*`filename2`*, ...,*`filenameN`*}	Cell array of names of XML files containing AUTOSAR information.

Output Arguments

`importerObj`	Handle to newly created `arxml.importer` object.

Examples

Specify the set of XML files required for an AUTOSAR import operation in one function call:

```
x = arxml.importer({'AtomicSensorComponentTypes.arxml', ...
                    'DataTypes.arxml', 'MiscDefs.arxml'})
```

Specify a primary XML file containing AUTOSAR information. Use the arxml.importer.getDependencies method to specify other required XML files:

```
x = arxml.importer('AtomicSensorComponentTypes.arxml')
setDependencies(x,{'DataTypes.arxml', 'MiscDefs.arxml'});
```

See Also

arxml.importer.getDependencies

How To

- “Import AUTOSAR Software Component” on page 3-4

Introduced in R2008a

autosar.api.create

Create AUTOSAR component for Simulink model

Syntax

```
autosar.api.create(model)
autosar.api.create(model,mode)
```

Description

`autosar.api.create(model)` creates AUTOSAR properties and Simulink to AUTOSAR mapping for `model`.

`autosar.api.create(model,mode)` additionally specifies whether to map model inports and outports with default settings for corresponding AUTOSAR properties.

Note: This function requires the Embedded Coder Support Package for AUTOSAR Standard. You install support packages using the MATLAB **Add-Ons** menu. On the MATLAB **Home** tab, in the **Environment** section, click **Add-Ons > Get Hardware Support Packages**. For more information, see "Support Package for AUTOSAR Standard" on page 1-3.

Examples

Create Default AUTOSAR Properties and Mapping

Create AUTOSAR properties and Simulink to AUTOSAR mapping for a model. Map the model inports and outports with default settings for corresponding AUTOSAR properties.

```
rtwdemo_autosar_multirunnables
autosar.api.create('rtwdemo_autosar_multirunnables','default');
```

- "Configure and Map AUTOSAR Component Programmatically" on page 4-267
- "AUTOSAR Interface Configuration" on page 4-3

Input Arguments

`model` — Model for which to create AUTOSAR properties and mapping

handle | character vector

Model for which to create AUTOSAR properties and Simulink to AUTOSAR mapping, specified as a handle or character vector representing the model name.

Example: `'my_model'`

`mode` — Whether to map model inports and outports with default settings

`init` (default) | `default`

Specify `default` to map model inports and outports with default settings for corresponding AUTOSAR properties.

Example: `'default'`

See Also

`autosar_ui_launch`

Introduced in R2013b

autosar.api.syncModel

Synchronize Simulink to AUTOSAR mapping of model with Simulink modifications

Syntax

```
autosar.api.syncModel(model)
```

Description

`autosar.api.syncModel(model)` synchronizes the Simulink to AUTOSAR mapping of `model` with modifications made to Simulink data transfers, entry-point functions, and function callers.

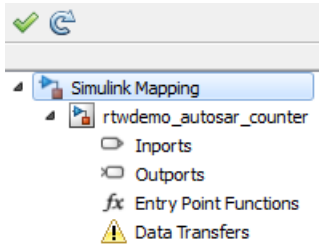
This function is equivalent to using the **Synchronize** icon in the graphical views of an AUTOSAR configuration.

Note: This function requires the Embedded Coder Support Package for AUTOSAR Standard. You install support packages using the MATLAB **Add-Ons** menu. On the MATLAB **Home** tab, in the **Environment** section, click **Add-Ons > Get Hardware Support Packages**. For more information, see "Support Package for AUTOSAR Standard" on page 1-3.

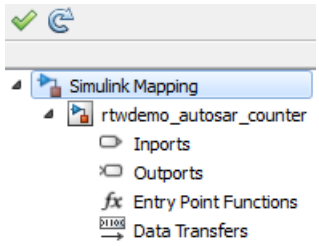
Examples

Synchronize Simulink to AUTOSAR Mapping of Model

When you create or modify an AUTOSAR model, Simulink to AUTOSAR mapping potentially is not current with the model content. For example, this display indicates that Simulink data transfers need loading or updating.



This example opens a model, opens the Configure AUTOSAR Interface dialog box, and synchronizes the model. After calling `autosar.api.syncModel`, the Simulink to AUTOSAR mapping accurately reflects model content.




```
hModel = 'rtwdemo_autosar_counter';
open_system(hModel);
autosar.ui.launch(hModel);

autosar.api.syncModel(hModel)
```

Input Arguments

`model` — Model to synchronize

handle | character vector

Loaded or open model for which to synchronize Simulink to AUTOSAR mapping with model changes, specified as a handle or character vector representing the model name.

Example: `'my_model'`

See Also

`autosar.api.validateModel`

Introduced in R2016a

autosar.api.validateModel

Validate AUTOSAR properties and mapping of Simulink model

Syntax

```
autosar.api.validateModel(model)
```

Description

`autosar.api.validateModel(model)` validates the AUTOSAR properties and Simulink to AUTOSAR mapping of `model`.

This function is equivalent to using the **Validate** icon in the graphical views of an AUTOSAR configuration.

Note: This function requires the Embedded Coder Support Package for AUTOSAR Standard. You install support packages using the MATLAB **Add-Ons** menu. On the MATLAB **Home** tab, in the **Environment** section, click **Add-Ons > Get Hardware Support Packages**. For more information, see "Support Package for AUTOSAR Standard" on page 1-3.

Examples

Validate AUTOSAR Properties and Mapping of Model

This example opens a model in which a Simulink inport is not mapped to an AUTOSAR port and data element. Initial validation reports the error and fails. After the inport is mapped, validation succeeds.

```
hModel = 'rtwdemo_autosar_counter_unmapped_port';
load_system(hModel);

% Initial validation fails
try
    autosar.api.validateModel(hModel)
```

```
catch validationErr
    throw(validationErr)
end
```

```
Block 'rtwdemo_autosar_counter_unmapped_port/Input' is not mapped to an AUTOSAR port element.
```

```
% Map the unmapped port
slMap=autosar.api.getSimulinkMapping(hModel);
mapInport(slMap,'Input','Input','Input','ImplicitReceive');

% Second validation succeeds
autosar.api.validateModel(hModel)
```

Input Arguments

`model` — Model to validate

handle | character vector

Loaded or open model for which to validate AUTOSAR properties and Simulink to AUTOSAR mapping, specified as a handle or character vector representing the model name.

Example: `'my_model'`

See Also

`autosar.api.syncModel`

Introduced in R2016a

AUTOSAR.DualScaledParameter class

Package: AUTOSAR

Specify computation method, calibration value, data type, and other properties of AUTOSAR dual-scaled parameter

Description

This class extends the `AUTOSAR.Parameter` class so that you can define an object that stores two scaled values of the same physical value. For example, for temperature measurement, you can store a Fahrenheit scale and a Celsius scale with conversion defined by a computation method that you provide. Given one scaled value, the `AUTOSAR.DualScaledParameter` can compute the other scaled value using the computation method.

A dual-scaled parameter has:

- A calibration value. The value that you prefer to use.
- A main value. The real-world value that Simulink uses.
- An internal stored integer value. The value that is used in the embedded code.

You can use `AUTOSAR.DualScaledParameter` objects in your model for both simulation and code generation. The parameter computes the internal value before code generation via the computation method. This offline computation results in leaner generated code.

If you provide the calibration value, the parameter computes the main value using the computation method. This method can be a first-order rational function.

$$y = \frac{ax + b}{cx + d}$$

- `x` is the calibration value.
- `y` is the main value.
- `a` and `b` are the coefficients of the CalToMain compute numerator.

- `c` and `d` are the coefficients of the CalToMain compute denominator.

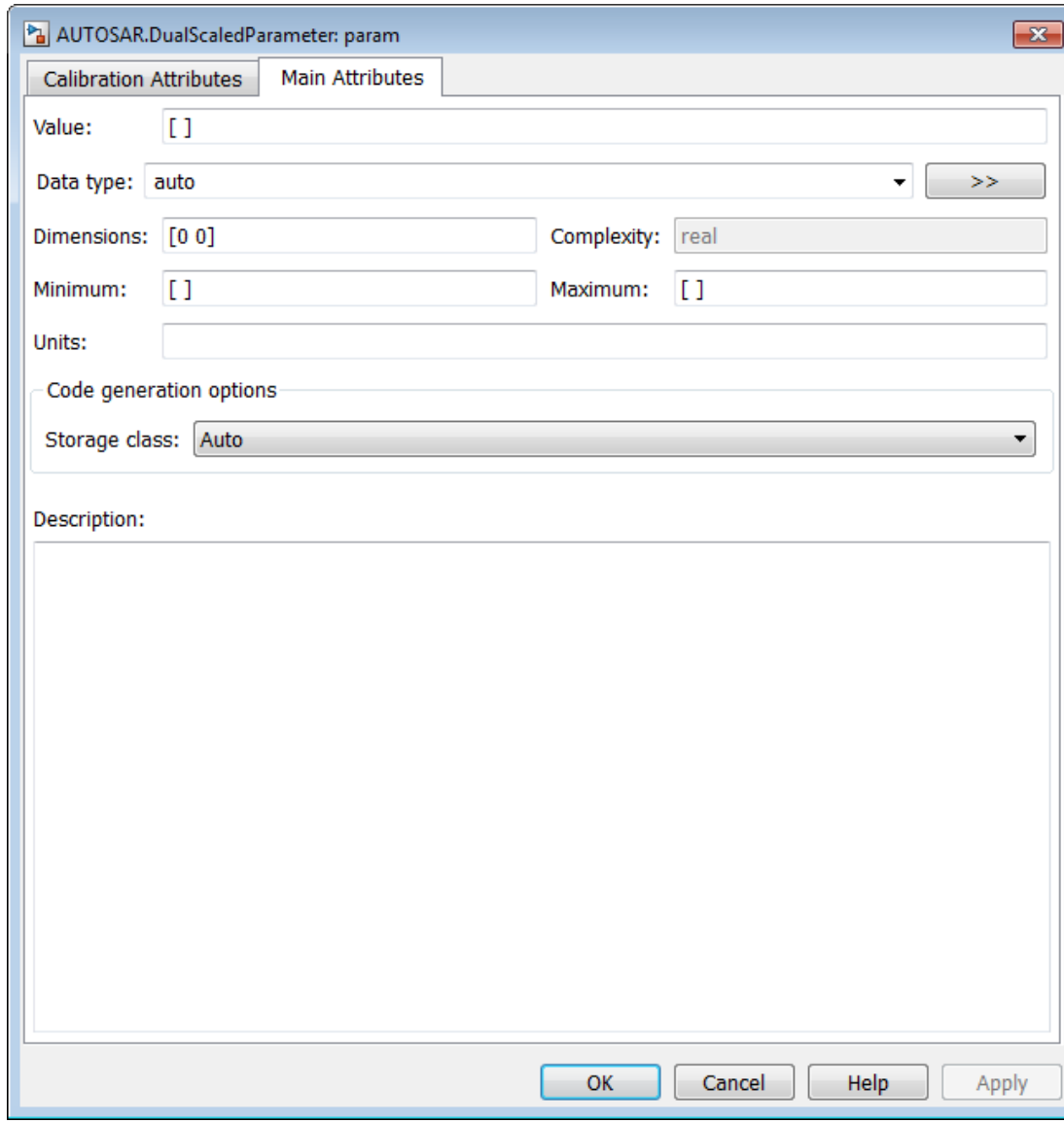
If you provide the calibration minimum and maximum values, the parameter computes minimum and maximum values of the main value. Simulink performs range checking of parameter values. The software alerts you when the parameter object value lies outside a range that corresponds to its specified minimum and maximum values and data type.

You can use the AUTOSAR.DualScaledParameter dialog box to define an `AUTOSAR.DualScaledParameter` object. To open the dialog box:

1. In the Model Explorer, select the base workspace or a model workspace and select **Add** > **Add Custom**.
2. In the Model Explorer — Select Object dialog box, set **Object class** to `AUTOSAR.DualScaledParameter`.

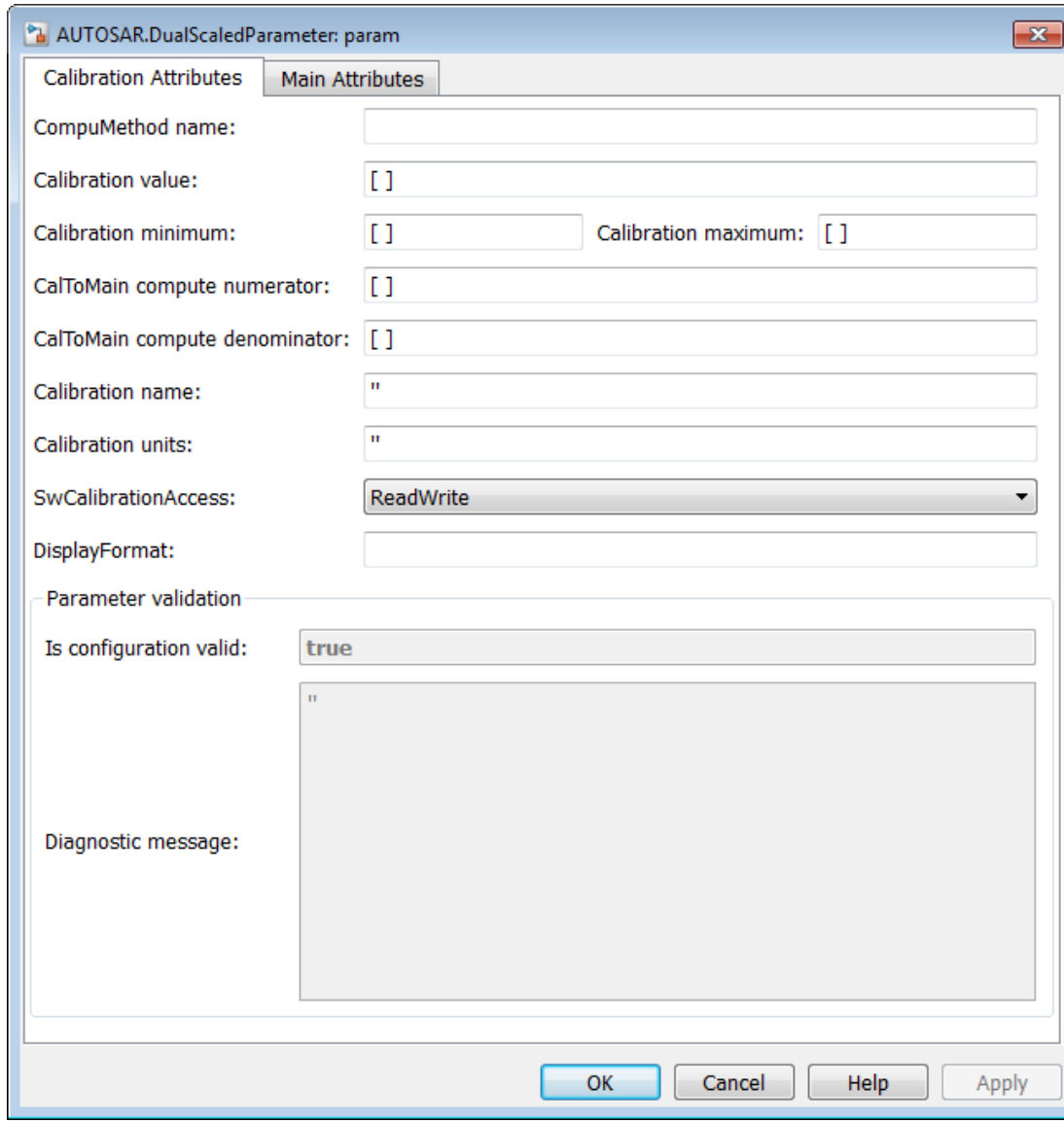
Property Dialog Box

Main Attributes Tab



This tab shows the properties inherited from the `AUTOSAR.Parameter` class. For more information, see `AUTOSAR.Parameter`.

Calibration Attributes Tab



CompuMethod name

Name of the AUTOSAR computation method (`CompuMethod`) to generate for the parameter in `arxml` code. For an AUTOSAR dual-scaled parameter, the code generator produces the `CompuMethod` category RAT_FUNC. For an example, see "Configure Rational Function CompuMethod for Dual-Scaled Parameter" on page 4-248.

Calibration value

Calibration value of the parameter. The value that you prefer to use. The default value is `[ ]` (unspecified). Specify a finite, real, double value.

Before specifying **Calibration value**, you must specify **CalToMain numerator** and **CalToMain denominator** to define the computation method. The parameter uses the computation method and the calibration value to calculate the real-world value that Simulink uses.

Calibration minimum

Minimum value for the calibration parameter. The default value is `[ ]` (unspecified). Specify a finite, real, double scalar value.

Before specifying **Calibration minimum**, you must specify **CalToMain numerator** and **CalToMain denominator** to define the computation method. The parameter uses the computation method and the calibration minimum value to calculate the minimum or maximum value that Simulink uses. A first order rational function is strictly monotonic, either increasing or decreasing. If it is increasing, setting the calibration minimum sets the main minimum value. If it is decreasing, setting the calibration minimum sets the main maximum.

If the parameter value is less than the minimum value or if the minimum value is outside the range of the parameter data type, Simulink generates a warning. In these cases, when updating the diagram or starting a simulation, Simulink generates an error.

Calibration maximum

Maximum value for the calibration parameter can have. The default value is `[ ]` (unspecified). Specify a finite, real double scalar value.

Before specifying **Calibration maximum**, you must specify **CalToMain numerator** and **CalToMain denominator** to define the computation method. The parameter uses the computation method and the calibration maximum value to calculate the corresponding maximum or minimum value that Simulink uses. A first order rational function is strictly monotonic, either increasing or decreasing. If it is

increasing, setting the calibration maximum sets the main maximum value. If it is decreasing, setting the calibration maximum sets the main minimum.

If the parameter value is less than the minimum value or if the minimum value is outside the range of the parameter data type, Simulink generates a warning. In these cases, when updating the diagram or starting a simulation, Simulink generates an error.

CalToMain compute numerator

Specify the numerator coefficients `a` and `b` of the first-order linear equation:

$$y = \frac{ax + b}{cx + d}$$

The default value is `[]` (unspecified). Specify finite, real double scalar values for `a` and `b`. For example, `[1 1]` or, for reciprocal scaling, `1`.

Once you have applied **CalToMain compute numerator**, you cannot change it.

CalToMain compute denominator

Specify the denominator coefficients `c` and `c` of the first-order linear equation:

$$y = \frac{ax + b}{cx + d}$$

The default value is `[]` (unspecified). Specify finite, real, double scalar values for `c` and `d`. For example, `[1 1]`.

Once you have applied **CalToMain compute denominator**, you cannot change it.

Calibration name

Specify the name of the calibration parameter. The default value is `''`. Specify a text value, for example, `'T1'`.

Calibration units

Specify the measurement units for this calibration value. This field is intended for use in documenting this parameter. The default value is `''`. Specify a text value, for example, `'Seconds'`.

SwCalibrationAccess

Specify measurement and calibration tool access to the calibration parameter. 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.

The default value is `ReadWrite`.

DisplayFormat

Optionally specify the format to be used by measurement and calibration tools to display the data. If you specify a display format, exporting `arxml` code generates a corresponding DISPLAY-FORMAT specification.

Use an ANSI C `printf` format specifier string, which has the following general form:

```
%[flags][width][.precision]type
```

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 4-194.

Is configuration valid

Simulink indicates whether the configuration is valid. The default value is `true`. If Simulink detects an issue with the configuration, it sets this field to `false` and provides information in the **Diagnostic message** field. You cannot set this field.

Diagnostic message

If you specify invalid parameter settings, Simulink displays a message in this field. Use the diagnostic information to help you fix an invalid configuration issue. You cannot set this field.

Properties

Name	Access	Description
`CompuMethodName`	`RW`	Name of AUTOSAR `CompuMethod` to generate for this parameter in `arxml` code. (CompuMethod name)
`CalibrationValue`	`RW`	Calibration value of this parameter. (Calibration value)

Name	Access	Description
`CalibrationMin`	`RW`	Calibration minimum value of this parameter. (Calibration minimum)
`CalibrationMax`	`RW`	Calibration maximum value of this parameter. (Calibration maximum)
`CalToMainCompuNumerator`	`RW`	Numerator coefficients of the computation method. (CalToMain compute numerator) Once you have applied `CalToMainCompuNumerator`, you cannot change it.
`CalToMainCompuDenominator`	`RW`	Denominator coefficients of the computation method. (CalToMain compute denominator) Once you have applied `CalToMainCompuDenominator`, you cannot change it.
`CalibrationName`	`RW`	Name of the calibration parameter. (Calibration name)
`CalibrationDocUnits`	`RW`	Measurement units for this calibration parameter's value. (Calibration units)
`SwCalibrationAccess`	`RW`	Measurement and calibration tool access to the calibration parameter — `ReadOnly`, `ReadWrite`, or `NotAccessible`. (SwCalibrationAccess)
`DisplayFormat`	`RW`	Format to be used by measurement and calibration tools to display the data. Format specifier uses the general form `%[flags] [width][.precision]type` — for example, `%2.1d` to produce 12.2. (DisplayFormat)
`IsConfigurationValid`	`RO`	Information about validity of configuration. (Is configuration valid)
`DiagnosticMessage`	`RO`	If the configuration is invalid, diagnostic information to help you fix the issue. (Diagnostic message)

Examples

Create and Update a Dual-Scaled Parameter

Create an `AUTOSAR.DualScaledParameter` object that stores a value as both time and frequency.

```
T1Rec = AUTOSAR.DualScaledParameter;
```

Set the computation method.

```
T1Rec.CalToMainCompuNumerator = [1];
T1Rec.CalToMainCompuDenominator = [1 0];
```

This computation method specifies that the value used by Simulink is the reciprocal of the value that you want to use.

Set the value that you want to see.

```
T1Rec.CalibrationValue = 1/7
```

```
T1Rec =

  DualScaledParameter with properties:

             CompuMethodName: ''
            CalibrationValue: 0.1429
              CalibrationMin: []
              CalibrationMax: []
     CalToMainCompuNumerator: 1
   CalToMainCompuDenominator: [1 0]
             CalibrationName: ''
         CalibrationDocUnits: ''
        IsConfigurationValid: 1
           DiagnosticMessage: ''
         SwCalibrationAccess: 'ReadWrite'
               DisplayFormat: ''
                       Value: 7
                  CoderInfo: [1x1 Simulink.CoderInfo]
                Description: ''
                   DataType: 'auto'
                        Min: []
                        Max: []
```

```
          Unit: ''
    Complexity: 'real'
    Dimensions: [1 1]
```

The `AUTOSAR.DualScaledParameter` calculates `T1Rec.Value` which is the value that Simulink uses. `T1Rec.CalibrationValue` is `1/7`, so `T1Rec.Value` is `7`.

Name this value and specify the units.

```
T1Rec.CalibrationName = 'T1';
T1Rec.CalibrationDocUnits = 'Seconds';
```

Set calibration minimum and maximum values.

```
T1Rec.CalibrationMin = 0.001;
T1Rec.CalibrationMax = 1;
```

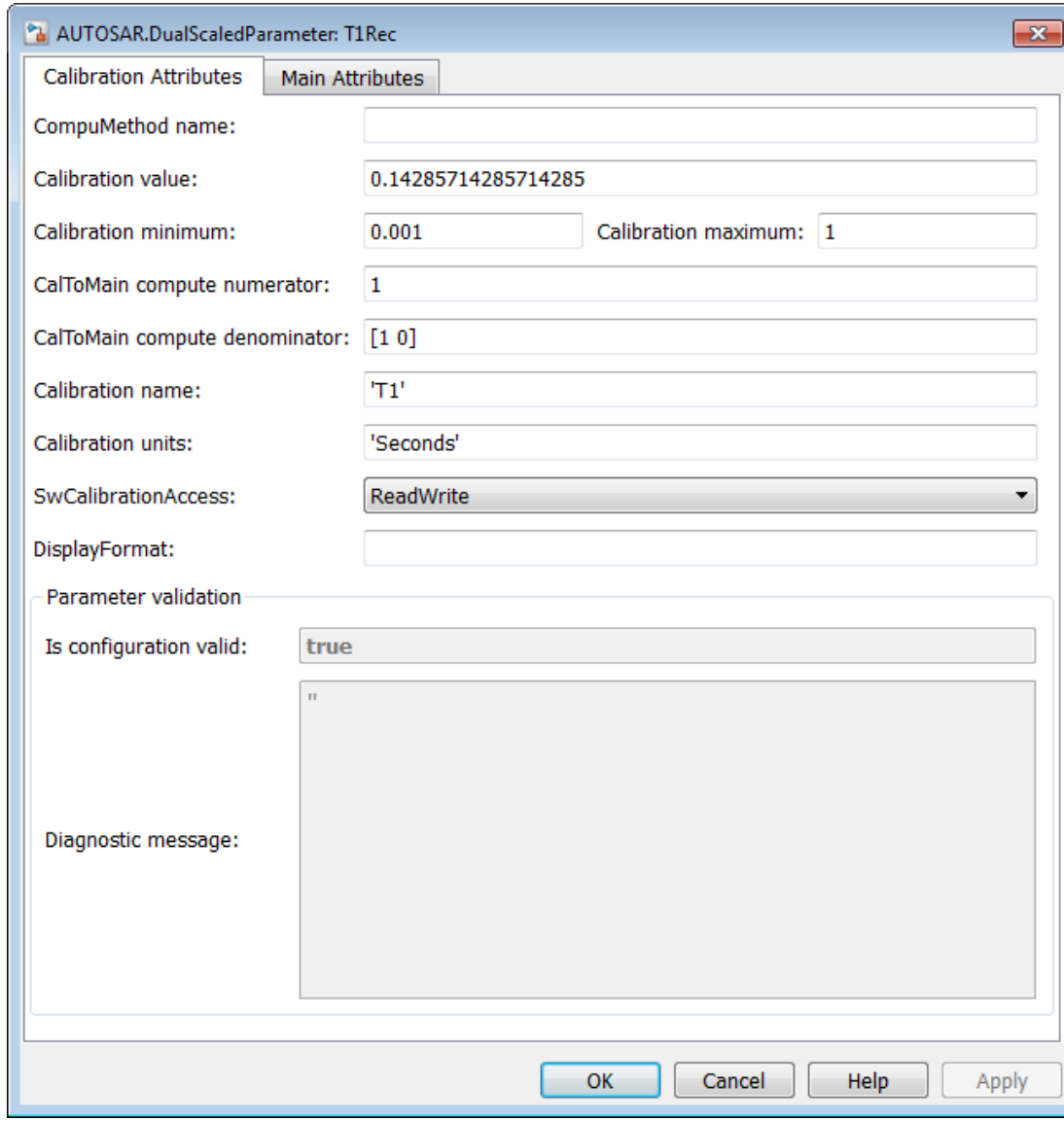
If you specify a value outside this allowable range, Simulink generates a warning.

Specify the units that Simulink uses.

```
T1Rec.Unit = 'Hz';
```

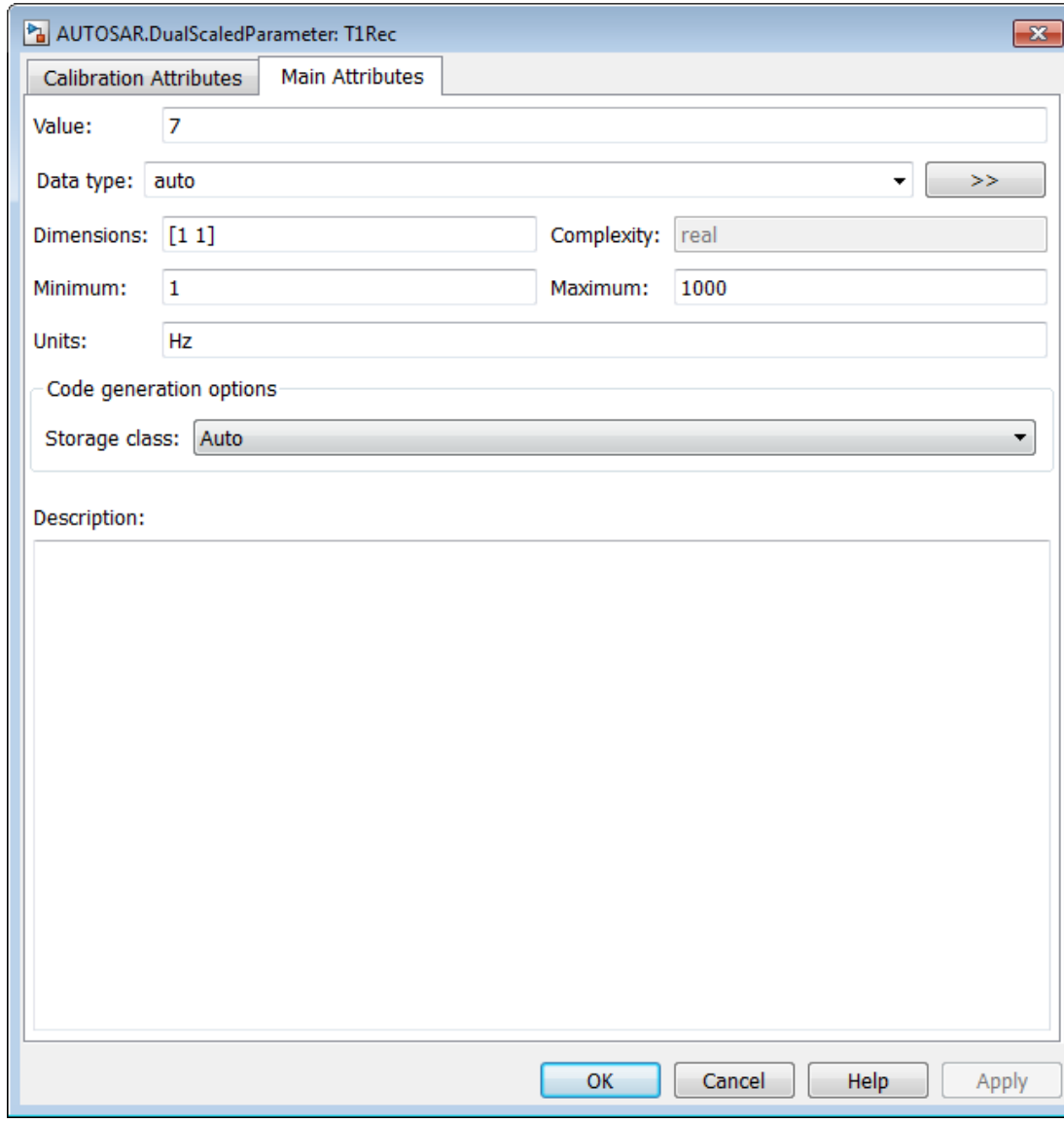
Open the AUTOSAR.DualScaledParameter dialog box.

```
open T1Rec
```



The **Calibration Attributes** tab displays the calibration value and the computation method that you specified.

In the dialog box, click the **Main Attributes** tab.



This tab displays information about the value used by Simulink.

Configure arxml Settings

Create a dual-scaled parameter. Configure its storage class so that when you generate code, the exported `arxml` code includes the dual-scaled parameter.

Create a dual-scaled parameter.

```
T1Rec = AUTOSAR.DualScaledParameter;
T1Rec.CalToMainCompuNumerator = [1];
T1Rec.CalToMainCompuDenominator = [1 0];
T1Rec.CalibrationValue = 1/7;
T1Rec.CalibrationName = 'T1';
T1Rec.CalibrationDocUnits = 'Seconds';
T1Rec.CalibrationMin = 0.001;
T1Rec.CalibrationMax = 1
```

```
T1Rec =

  DualScaledParameter with properties:

             CompuMethodName: ''
            CalibrationValue: 0.1429
              CalibrationMin: 1.0000e-03
              CalibrationMax: 1
      CalToMainCompuNumerator: 1
    CalToMainCompuDenominator: [1 0]
             CalibrationName: 'T1'
         CalibrationDocUnits: 'Seconds'
        IsConfigurationValid: 1
           DiagnosticMessage: ''
         SwCalibrationAccess: 'ReadWrite'
               DisplayFormat: ''
                       Value: 7
                   CoderInfo: [1x1 Simulink.CoderInfo]
                 Description: ''
                    DataType: 'auto'
                         Min: 1
                         Max: 1000
                        Unit: ''
                  Complexity: 'real'
                  Dimensions: [1 1]
```

Set the storage class of the parameter so that the generated `arxml` code includes the parameter.

```
T1Rec.CoderInfo.StorageClass = 'Custom';
T1Rec.CoderInfo.CustomStorageClass = 'InternalCalPrm';
```

You can now use the parameter in a Simulink model. If you configure the model for AUTOSAR, when you generate code for the model, the code generator produces `arxml` code that contains information about the dual-scaled parameter.

See Also

Classes

`AUTOSAR.Parameter`

More About

- Class Attributes
- Property Attributes

AUTOSAR.Parameter

Specify value, data type, code generation options, other properties of parameter

Description

With this class, you can create workspace objects for modeling AUTOSAR calibration parameters. You can create an `AUTOSAR.Parameter` object in the base MATLAB workspace.

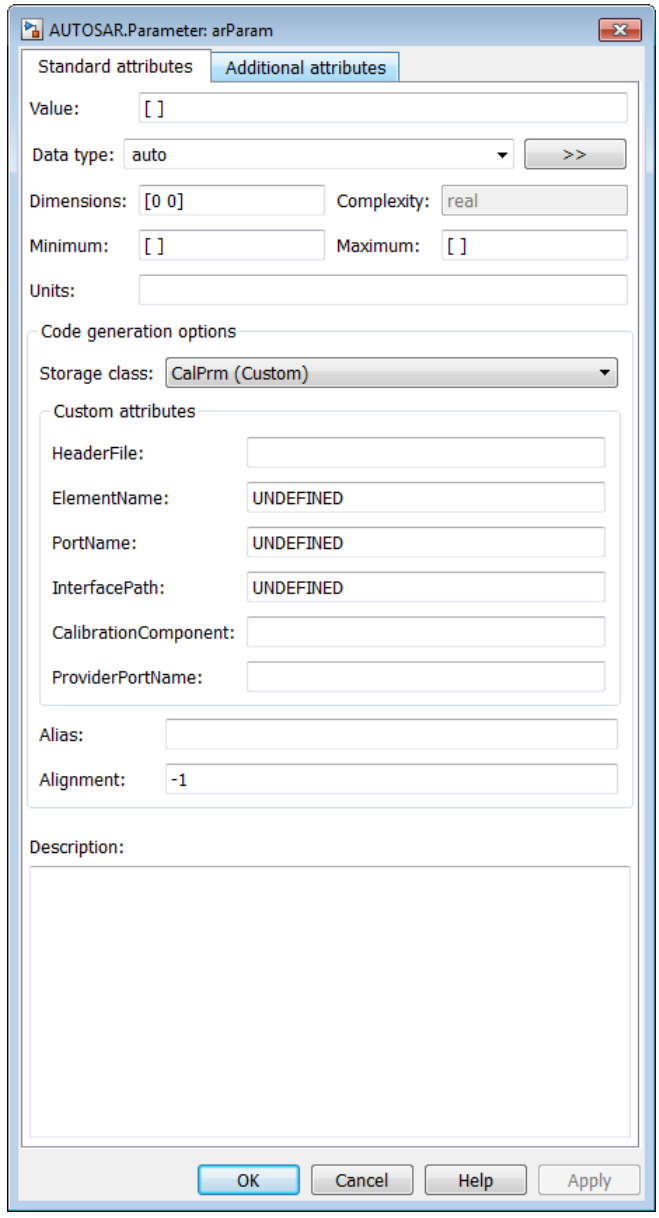
This class extends the `Simulink.Parameter` class. With parameter objects, you can specify the value of a parameter and other information about the parameter, such as its purpose, its dimensions, or its minimum and maximum values. Some Simulink products use this information, for example, to determine whether the parameter is tunable (see "Tune and Experiment with Block Parameter Values").

Simulink performs range checking of parameter values. The software alerts you when the parameter object value lies outside a range that corresponds to its specified minimum and maximum values and data type. For more information, see the Simulink.Parameter reference page.

You can use the AUTOSAR.Parameter dialog box to define an `AUTOSAR.Parameter` object. To open the dialog box:

1. In Model Explorer, select the base workspace. Select **Add > Add Custom**.
2. In the Model Explorer - Select Object dialog box, set **Object class** to `AUTOSAR.Parameter`. Optionally, you can modify the default object name. Click **OK**.

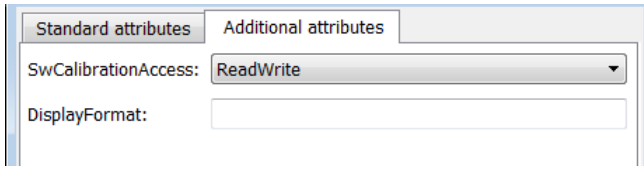
Property Dialog Box



The Simulink.Parameter reference page describes the parameter attributes in detail. The `AUTOSAR.Parameter` class extends the `Simulink.Parameter` class with the following additional selections for the **Storage class** attribute:

- `CalPrm (Custom)` — Calibration parameters belong to a calibration component, which can be accessed by multiple AUTOSAR software components. Selecting this storage class enables the custom attributes **HeaderFile**, **ElementName**, **PortName**, **InterfacePath**, **CalibrationComponent**, and **ProviderPortName**.
  - **HeaderFile** allows you to optionally specify the name of the AUTOSAR software component header file that declares the calibration parameter.
  - **ElementName**, **PortName**, and **InterfacePath** allow you to associate the calibration parameter with a specific AUTOSAR element, AUTOSAR port, and AUTOSAR interface. Specify an element name, a port name, and an interface path. For example, element `K`, port `rCounter`, and interface `rCounter/CalibrationComponents/counter_if`.
  - **CalibrationComponent** and **ProviderPortName** allow you to configure the calibration parameter to be exported in an AUTOSAR calibration component (`ParameterSwComponent`). Calibration parameters exported in a calibration component can be accessed by multiple AUTOSAR software components, using the calibration component name and associated provider port name. **CalibrationComponent** specifies the qualified name of the calibration component to be exported, and **ProviderPortName** specifies the short name of the associated provider port. For example, calibration component `/CalibrationComponents/counter_swc/counter` and provider port `pCounter`.
- `InternalCalPrm (Custom)` — Internal calibration parameters are defined and accessed by only one AUTOSAR software component. Selecting this storage class enables the custom attributes **HeaderFile** and **PerInstanceBehavior**.
  - **HeaderFile** allows you to optionally specify the name of the AUTOSAR software component header file that declares the calibration parameter.
  - **PerInstanceBehavior** allows you to specify `Parameter shared by all instances of the Software Component` or `Each instance of the Software Component has its own copy of the parameter`.
- `SystemConstant (Custom)` — Allows you to control the storage of a systemwide constant in generated code.

The `AUTOSAR.Parameter` class also provides the following attributes, which are independent of storage class, on the **Additional attributes** tab:



- **SwCalibrationAccess** allows you to specify measurement and calibration software access to the data as `NotAccessible`, `ReadOnly`, or `ReadWrite`.
- **DisplayFormat** allows you to specify the format to be used by measurement and calibration tools to display the data. If you specify a display format, exporting `arxml` code generates a corresponding DISPLAY-FORMAT specification. Use 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 4-194.

For more information, see:

- "Override Default Parameter Behavior by Creating Global Variables in the Generated Code"
- "Model AUTOSAR Calibration Parameters and Lookup Tables" on page 2-24
- "Configure AUTOSAR Internal Calibration Parameters" on page 4-167
- "Configure AUTOSAR Calibration Component" on page 4-170
- "Variants in Runnable Condition Logic" on page 2-48
- "Configure AUTOSAR Variants in Runnable Condition Logic" on page 4-261
- "Configure AUTOSAR Data for Measurement and Calibration" on page 4-190

See Also

- Simulink.Parameter
- AUTOSAR.DualScaledParameter
- `AUTOSAR4.Parameter`

AUTOSAR.Signal

Specify data type, code generation options, other attributes of signal

Description

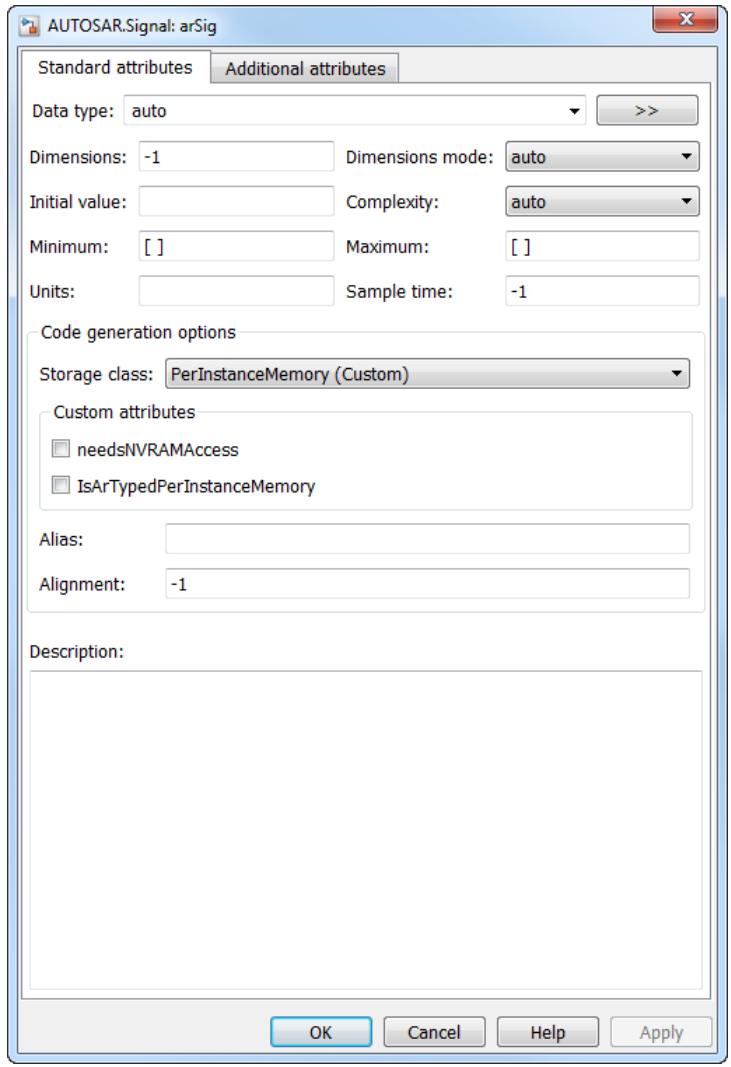
With this class, you can create workspace objects for modeling per-instance memory for AUTOSAR runnables. You can create an `AUTOSAR.Signal` object in the base MATLAB workspace.

This class extends the `Simulink.Signal` class. With signal objects, you can assign or validate the attributes of a signal or discrete state, such as its data type, numeric type, dimensions, and so on. For more information, see the `Simulink.Signal` reference page.

You can use the AUTOSAR.Signal dialog box to define an `AUTOSAR.Signal` object. To open the dialog box:

1. In Model Explorer, select the base workspace. Select **Add > Add Custom**.
2. In the Model Explorer - Select Object dialog box, set **Object class** to `AUTOSAR.Signal`. Optionally, you can modify the default object name. Click **OK**.

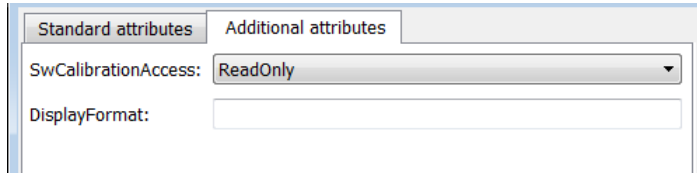
Property Dialog Box



The `Simulink.Signal` reference page describes the signal attributes in detail. The `AUTOSAR.Signal` class extends the `Simulink.Signal` class with the following additional selection for the **Storage class** attribute:

- `PerInstanceMemory (Custom)` — AUTOSAR per-instance memory is instance-specific global memory within an AUTOSAR software component. An AUTOSAR run-time environment generator allocates this memory and provides an API through which you access this memory. Selecting this storage class enables the custom attributes **needsNVRAMAccess** and **IsArTypedPerInstanceMemory**.
  - **needsNVRAMAccess** allows you to specify whether the AUTOSAR signal needs access to nonvolatile RAM on a processor. Depending on the AUTOSAR schema selected for your model, this setting potentially impacts the XML output for your model.
  - **IsArTypedPerInstanceMemory** allows you to specify whether to use AUTOSAR-typed per-instance memory (introduced in AUTOSAR schema version 4.0), rather than C-typed per-instance memory.

The `AUTOSAR.Signal` class also provides the following attributes, which are independent of storage class, on the **Additional attributes** tab:



- **SwCalibrationAccess** allows you to specify measurement and calibration software access to the data as `NotAccessible`, `ReadOnly`, or `ReadWrite`.
- **DisplayFormat** allows you to specify the format to be used by measurement and calibration tools to display the data. If you specify a display format, exporting `arxml` code generates a corresponding DISPLAY-FORMAT specification. Use 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 4-194.

After you create an `AUTOSAR.Signal` object, you can reference it in a Data Store Memory block. For more information, see

- “Per-Instance Memory” on page 2-31
- “Configure AUTOSAR Per-Instance Memory” on page 4-230
- “Configure AUTOSAR Data for Measurement and Calibration” on page 4-190

See Also

- `Data Store Memory`
- `Simulink.Signal`
- `AUTOSAR4.Signal`

autosar_ui_close

Close Configure AUTOSAR Interface dialog box

Syntax

```
autosar_ui_close(model)
```

Description

`autosar_ui_close(model)` closes the Configure AUTOSAR Interface dialog box for the specified open model.

Examples

Close AUTOSAR Configuration Dialog Box for Example Model

Open the AUTOSAR Interface Configuration dialog box with settings for an AUTOSAR example model, and then close the dialog box.

```
open_system('rtwdemo_autosar_multirunnables')
autosar_ui_launch('rtwdemo_autosar_multirunnables')
autosar_ui_close('rtwdemo_autosar_multirunnables')
```

- “AUTOSAR Interface Configuration” on page 4-3

Input Arguments

`model` — Model for which to close the Configure AUTOSAR Interface dialog box

handle | character vector

Model for which to close the Configure AUTOSAR Interface dialog box, specified as a handle or character vector representing the model name.

Example: `'rtwdemo_autosar_multirunnables'`

See Also

`autosar.api.create` | `autosar_ui_launch`

Introduced in R2014b

autosar_ui_launch

Open Configure AUTOSAR Interface dialog box

Syntax

```
autosar_ui_launch(model)
```

Description

`autosar_ui_launch(model)` opens the Configure AUTOSAR Interface dialog box with settings for the specified open model.

Note: Configuring an AUTOSAR interface requires an Embedded Coder license. If Embedded Coder is not licensed, the Configure AUTOSAR Interface dialog box runs in read-only mode.

Examples

Display AUTOSAR Interface Configuration Settings for Example Model

Open the AUTOSAR Interface Configuration dialog box with settings for an AUTOSAR example model.

```
open_system('rtwdemo_autosar_multirunnables')
autosar_ui_launch('rtwdemo_autosar_multirunnables')
```

- “AUTOSAR Interface Configuration” on page 4-3

Input Arguments

`model` — Model for which to display AUTOSAR interface configuration settings

handle | character vector

Model for which to display AUTOSAR interface configuration settings, specified as a handle or character vector representing the model name.

Example: `'rtwdemo_autosar_multirunnables'`

See Also

`autosar.api.create` | `autosar_ui_close`

Introduced in R2013b

AUTOSAR4.Parameter

Specify value, data type, code generation options, other properties of parameter

Description

With this class, you can create workspace objects for mapping internal global parameters to AUTOSAR memory sections. You can create an `AUTOSAR4.Parameter` object in the base MATLAB workspace.

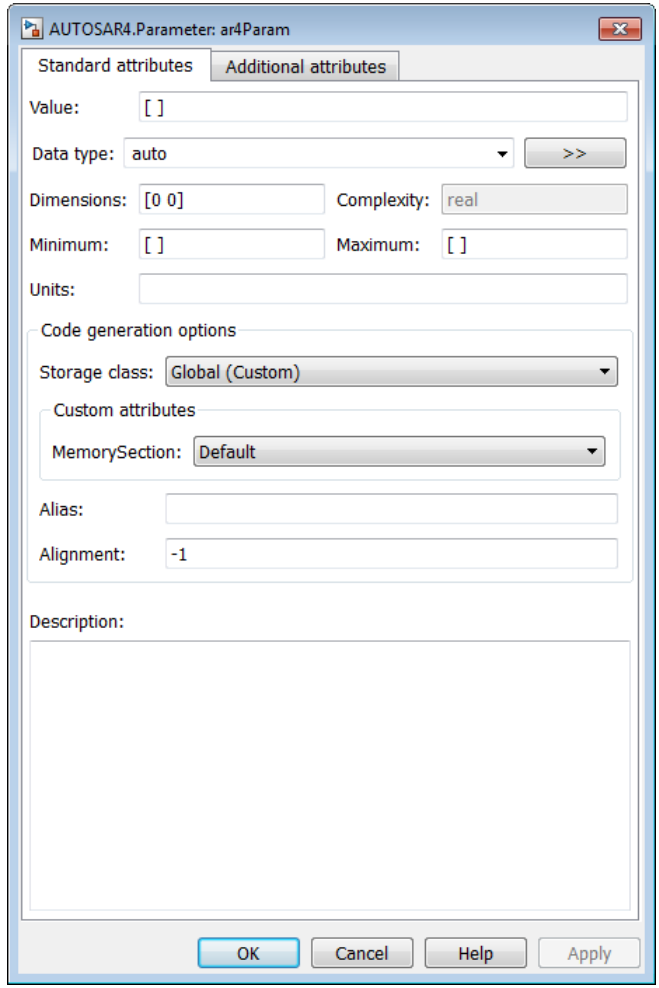
This class extends the `Simulink.Parameter` class. With parameter objects, you can specify the value of a parameter and other information about the parameter, such as its purpose, its dimensions, or its minimum and maximum values. Some Simulink products use this information, for example, to determine whether the parameter is tunable (see "Tune and Experiment with Block Parameter Values").

Simulink performs range checking of parameter values. The software alerts you when the parameter object value lies outside a range that corresponds to its specified minimum and maximum values and data type. For more information, see the Simulink.Parameter reference page.

You can use the AUTOSAR4.Parameter dialog box to define an `AUTOSAR4.Parameter` object. To open the dialog box:

1. In Model Explorer, select the base workspace. Select **Add > Add Custom**.
2. In the Model Explorer - Select Object dialog box, set **Object class** to `AUTOSAR4.Parameter`. Optionally, you can modify the default object name. Click **OK**.

Property Dialog Box

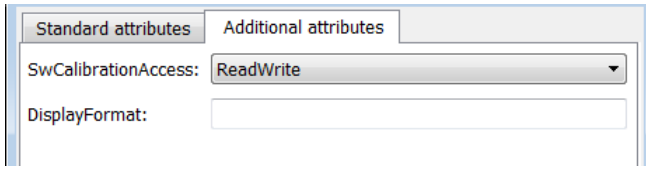


The Simulink.Parameter reference page describes the parameter attributes in detail. The `AUTOSAR4.Parameter` class extends the `Simulink.Parameter` class with the following additional selection for the **Storage class** attribute:

- `Global (Custom)` — Allows you to map internal global parameters to AUTOSAR memory sections. Selecting this storage class enables the custom attribute **MemorySection**.

  MemorySection allows you to explicitly select AUTOSAR memory section `VAR`, `CAL`, `CONST`, `VOLATILE`, or `CONST_VOLATILE`, or accept the `Default`.

The `AUTOSAR4.Parameter` class also provides the following attributes, which are independent of storage class, on the **Additional attributes** tab:



- **SwCalibrationAccess** allows you to specify measurement and calibration software access to the data as `NotAccessible`, `ReadOnly`, or `ReadWrite`.
- **DisplayFormat** allows you to specify the format to be used by measurement and calibration tools to display the data. If you specify a display format, exporting `arxml` code generates a corresponding DISPLAY-FORMAT specification. Use 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 4-194.

For more information, see

- “Static and Constant Memory” on page 2-32
- “Configure AUTOSAR Static or Constant Memory” on page 4-234
- “Configure AUTOSAR Data for Measurement and Calibration” on page 4-190

See Also

- Simulink.Parameter
- `AUTOSAR.Parameter`
- AUTOSAR.DualScaledParameter

AUTOSAR4.Signal

Specify data type, code generation options, other attributes of signal

Description

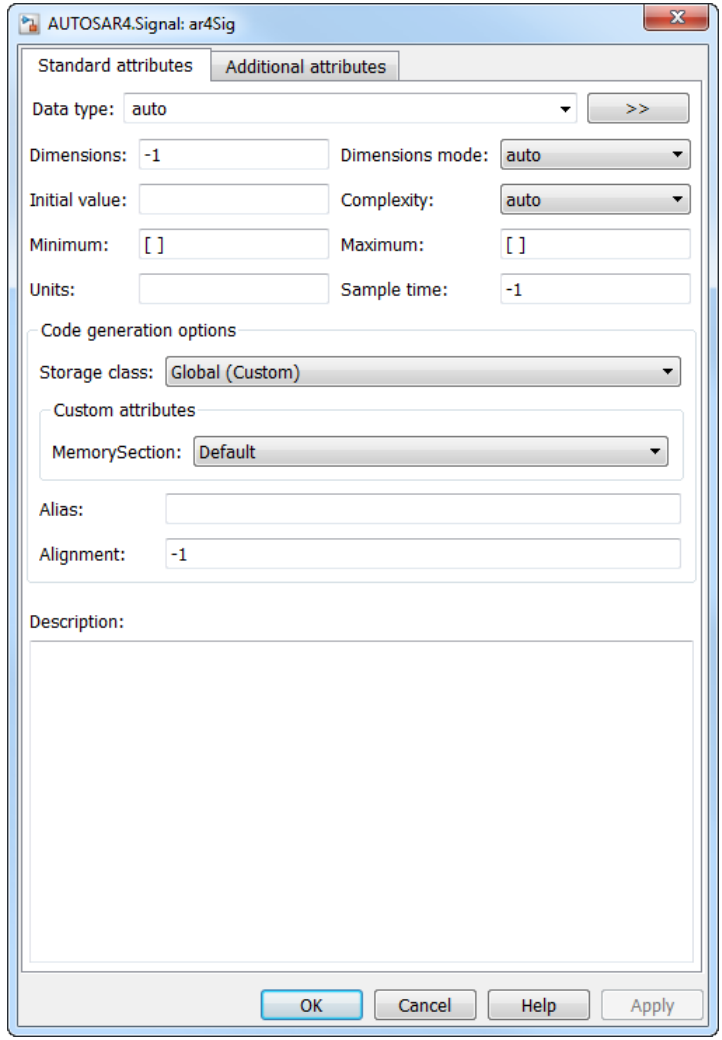
With this class, you can create workspace objects for mapping internal global signals to AUTOSAR memory sections. You can create an `AUTOSAR4.Signal` object in the base MATLAB workspace.

This class extends the `Simulink.Signal` class. With signal objects, you can assign or validate the attributes of a signal or discrete state, such as its data type, numeric type, dimensions, and so on. For more information, see the `Simulink.Signal` reference page.

You can use the AUTOSAR4.Signal dialog box to define an `AUTOSAR4.Signal` object. To open the dialog box:

1. In Model Explorer, select the base workspace. Select **Add > Add Custom**.
2. In the Model Explorer - Select Object dialog box, set **Object class** to `AUTOSAR4.Signal`. Optionally, you can modify the default object name. Click **OK**.

Property Dialog Box

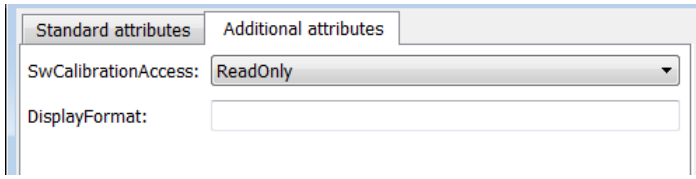


The `Simulink.Signal` reference page describes the signal attributes in detail. The `AUTOSAR4.Signal` class extends the `Simulink.Signal` class with the following additional selection for the **Storage class** attribute:

- `Global (Custom)` — Allows you to map internal global signals to AUTOSAR memory sections. Selecting this storage class enables the custom attribute **MemorySection**.

  MemorySection allows you to explicitly select AUTOSAR memory section `VAR`, `CAL`, `CONST`, `VOLATILE`, or `CONST_VOLATILE`, or accept the `Default`.

The `AUTOSAR4.Signal` class also provides the following attributes, which are independent of storage class, on the **Additional attributes** tab:



- **SwCalibrationAccess** allows you to specify measurement and calibration software access to the data as `NotAccessible`, `ReadOnly`, or `ReadWrite`.
- **DisplayFormat** allows you to specify the format to be used by measurement and calibration tools to display the data. If you specify a display format, exporting `arxml` code generates a corresponding DISPLAY-FORMAT specification. Use 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 4-194.

For more information, see

- “Static and Constant Memory” on page 2-32
- “Configure AUTOSAR Static or Constant Memory” on page 4-234
- “Configure AUTOSAR Data for Measurement and Calibration” on page 4-190

See Also

- `Simulink.Signal`
- `AUTOSAR.Signal`

createCalibrationComponentObjects

Class: arxml.importer
Package: arxml

Create Simulink calibration objects from AUTOSAR calibration component

Syntax

```
createCalibrationComponentObjects(importerObj,ComponentName)
success =
createCalibrationComponentObjects(importerObj,ComponentName,
Property1,Value1,Property2,Value2,...)
```

Description

`createCalibrationComponentObjects(importerObj,ComponentName)` creates Simulink calibration objects from an AUTOSAR calibration component. This imports your parameters into the MATLAB base workspace or a Simulink data dictionary and you can then assign them to block parameters in your Simulink model.

`success = createCalibrationComponentObjects(importerObj,ComponentName, Property1,Value1,Property2,Value2,...)` specifies additional options for Simulink calibration object creation.

Note: This method requires the Embedded Coder Support Package for AUTOSAR Standard. You install support packages using the MATLAB **Add-Ons** menu. On the MATLAB **Home** tab, in the **Environment** section, click **Add-Ons > Get Hardware Support Packages**. For more information, see "Support Package for AUTOSAR Standard" on page 1-3.

Tip

If you enter the `arxml.importer` constructor 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 can specify in calls to `createComponentAsModel` and `createCalibrationComponentObjects`.

Input Arguments

`importerObj` Handle to imported AUTOSAR information created in a previous call to `arxml.importer`.

`componentName` Absolute short-name path of calibration parameter component.

`PropertyN,ValueN` Optional property/value pairs representing creation options. You can specify values for the following properties:

`'CreateSimulinkObject'`

`true` (default) or `false`. If `true`, the function creates `Simulink.AliasType` and `Simulink.NumericType` data objects corresponding to the AUTOSAR data types in the XML file.

`'DataDictionary'`

Character vector specifying a Simulink data dictionary into which to import calibration objects corresponding to AUTOSAR data types in the XML file. If the specified dictionary does not already exist, the importer creates it. The model becomes associated with the specified data dictionary.

Output Arguments

`success` True if function is successful. False otherwise.

Examples

Create calibration objects from an AUTOSAR calibration component and import them into the MATLAB base workspace:

```
obj = arxml.importer('mySWC.arxml')
createCalibrationComponentObjects(obj,'/ComponentType/MyCalibComp1')
```

Create calibration objects from an AUTOSAR calibration component and import them into Simulink data dictionary `ardata.sldd`:

```
obj = arxml.importer('mySWC.arxml')
createCalibrationComponentObjects(obj,'/ComponentType/MyCalibComp1',...
  'DataDictionary','ardata.sldd')
```

How To

- “Import AUTOSAR Software Component” on page 3-4

Introduced in R2008b

createComponentAsModel

Class: arxml.importer
Package: arxml

Create AUTOSAR atomic software component as Simulink model

Syntax

```
createComponentAsModel(importerObj,ComponentName)
[modelH,success] = createComponentAsModel(importerObj,ComponentName,
Property1,Value1,Property2,Value2,...)
```

Description

`createComponentAsModel(importerObj,ComponentName)` creates a Simulink model corresponding to the AUTOSAR atomic software component `'COMPONENT'` described in the XML file imported by the `arxml.importer` object *`importerObj`*.

`[modelH,success] = createComponentAsModel(importerObj,ComponentName, Property1,Value1,Property2,Value2,...)` specifies additional options for Simulink model creation.

Note: This method requires the Embedded Coder Support Package for AUTOSAR Standard. You install support packages using the MATLAB **Add-Ons** menu. On the MATLAB **Home** tab, in the **Environment** section, click **Add-Ons > Get Hardware Support Packages**. For more information, see "Support Package for AUTOSAR Standard" on page 1-3.

Tips

- If you enter the `arxml.importer` constructor 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 can specify in calls to `createComponentAsModel` and `createCalibrationComponentObjects`.

- When you import an AUTOSAR component into a model, in most cases, the importer requires that an AUTOSAR port be accessed by a runnable in order to generate the corresponding Simulink elements. For example, if a sender-receiver or client-server port in `arxml` code is not accessed by a runnable, the importer does not create the corresponding inports, outports, or Simulink functions.

  An exception is when you import a single-runnable AUTOSAR component to create a rate-based, single-tasking model (setting property `'CreateInternalBehavior'` to `false`). In this case, the importer creates inports, outports, and so on even when an AUTOSAR port is not accessed by the runnable.

Input Arguments

`importerObj` Handle to imported AUTOSAR information created in a previous call to `arxml.importer`.

`ComponentName` Absolute short-name path of the atomic software component.

`PropertyN,ValueN` Optional property/value pairs representing creation options. You can specify values for the following properties:

`'AutoSave'`

`true` or `false` (default). If `true`, the function automatically saves the generated Simulink model.

`'CreateInternalBehavior'`

`true` or `false` (default). By default, the function imports a multi-runnable AUTOSAR software component and maps it into a new rate-based model. If the `arxml` code contains periodic runnables, the importer adds rate-based model content, including periodic functions and data transfer lines with rate transitions, and maps them to corresponding periodic runnables and inter-runnable variables (IRVs) imported from the AUTOSAR software component.

If `true`, the function imports a multi-runnable AUTOSAR software component and maps it into a new function-call-based model. The importer adds function-call subsystem or

function blocks and signal lines in the model and maps them to corresponding runnables and IRVs imported from the AUTOSAR software component.

For more information, see "Import Multi-Runnable Software Component" on page 3-6 and "Multi-Runnable Software Components" on page 2-5.

`'CreateSimulinkObject'`

`true` (default) or `false`. If `true`, the function creates `Simulink.AliasType` and `Simulink.NumericType` data objects corresponding to the AUTOSAR data types in the XML file.

`'DataDictionary'`

Character vector specifying a Simulink data dictionary into which to import data objects corresponding to AUTOSAR data types in the XML file. If the specified dictionary does not already exist, the importer creates it. The model becomes associated with the specified data dictionary.

`'InitializationRunnable'`

Character vector specifying the name of an existing runnable as the initialization runnable for the component.

`'NameConflictAction'`

`'overwrite'` (default), `'makenameunique'`, or `'error'`. Use this property to specify the action to take if a Simulink model with the same name as the component already exists.

Output Arguments

`modelH`	Model handle.
`success`	True if the function is successful. Otherwise, it is false.

Examples

Import an AUTOSAR software component and map it into a new Simulink model:

```
obj = arxml.importer('mySWC.arxml')
createComponentAsModel(obj,'/pkg/swc')
```

Import an AUTOSAR software component into a new model, and import data objects for AUTOSAR data into Simulink data dictionary `ardata.sldd`:

```
obj = arxml.importer('mySWC.arxml')
createComponentAsModel(obj,'/pkg/swc','DataDictionary','ardata.sldd')
```

Import a multi-runnable AUTOSAR software component and map it into a new function-call-based model.

```
obj = arxml.importer('mySWC.arxml')
createComponentAsModel(obj,'/pkg/swc','CreateInternalBehavior',true)
```

Import an AUTOSAR software component and designate `Runnable1` as the initialization runnable.

```
obj = arxml.importer('mySWC.arxml')
createComponentAsModel(obj,'/pkg/swc','InitializationRunnable','Runnable1')
```

How To

- "Import AUTOSAR Software Component" on page 3-4

Introduced in R2008a

delete

Delete AUTOSAR element

Syntax

```
delete(arProps,elementPath)
```

Description

`delete(arProps,elementPath)` deletes the AUTOSAR element at `elementPath`.

Note: This function requires the Embedded Coder Support Package for AUTOSAR Standard. You install support packages using the MATLAB **Add-Ons** menu. On the MATLAB **Home** tab, in the **Environment** section, click **Add-Ons > Get Hardware Support Packages**. For more information, see "Support Package for AUTOSAR Standard" on page 1-3.

Examples

Delete Sender-Receiver Interface

Delete the sender-receiver interface `Interface1` from the AUTOSAR configuration for a model.

```
rtwdemo_autosar_multirunnables
arProps=autosar.api.getAUTOSARProperties('rtwdemo_autosar_multirunnables');
ifPaths=find(arProps,[],'SenderReceiverInterface','PathType','FullyQualified')
```

```
ifPaths =
    '/pkg/if/Interface1'    '/pkg/if/Interface2'
```

```
delete(arProps,'Interface1');
ifPaths=find(arProps,[],'SenderReceiverInterface','PathType','FullyQualified')
```

```
ifPaths =
    '/pkg/if/Interface2'
```

- "Configure and Map AUTOSAR Component Programmatically" on page 4-267

- “AUTOSAR Interface Configuration” on page 4-3

Input Arguments

`arProps` — AUTOSAR properties information for a model

handle

AUTOSAR properties information for a model, previously returned by *`arProps`* = `autosar.api.getAUTOSARProperties(`*`model`*`)`. *`model`* is a handle or character vector representing the model name.

Example: `arProps`

`elementPath` — Path to AUTOSAR element

character vector

Path to the AUTOSAR element to delete.

Example: `'Input'`

See Also

`add`

Introduced in R2013b

deleteUnmappedComponents

Delete unmapped AUTOSAR components from model

Syntax

```
deleteUnmappedComponents(arProps)
```

Description

`deleteUnmappedComponents(arProps)` deletes atomic software components that are not mapped to the model. Use this to remove unused imported components that you do not want preserved in the model and exported in `arxml` code. This function does not remove calibration components.

Note: This function requires the Embedded Coder Support Package for AUTOSAR Standard. You install support packages using the MATLAB **Add-Ons** menu. On the MATLAB **Home** tab, in the **Environment** section, click **Add-Ons > Get Hardware Support Packages**. For more information, see "Support Package for AUTOSAR Standard" on page 1-3.

Examples

Remove Unmapped Atomic Software Components From AUTOSAR Model

After importing AUTOSAR information from `arxml` files and configuring a model for AUTOSAR, remove atomic software components that were imported but are not mapped to the model. This prevents unmapped components from being exported back to `arxml`.

```
arProps=autosar.api.getAUTOSARProperties('my_autosar_model');
deleteUnmappedComponents(arProps);
```

- "Configure and Map AUTOSAR Component Programmatically" on page 4-267
- "Import AUTOSAR Software Component" on page 3-4
- "AUTOSAR Interface Configuration" on page 4-3

Input Arguments

`arProps` — AUTOSAR properties information for a model

handle

AUTOSAR properties information for a model, previously returned by *`arProps`* = `autosar.api.getAUTOSARProperties(`*`model`*`)`. *`model`* is a handle or character vector representing the model name.

Example: `arProps`

See Also

arxml.importer

Introduced in R2014b

find

Find AUTOSAR elements

Syntax

```
paths=find(arProps,rootPath,category)
paths=find(arProps,rootPath,category,'PathType',value)
paths=find(arProps,rootPath,category,property,value)
```

Description

`paths=find(arProps,rootPath,category)` returns `paths` to AUTOSAR elements matching `category`, starting at path `rootPath`.

`paths=find(arProps,rootPath,category,'PathType',value)` specifies whether the returned `paths` are fully qualified or partially qualified.

`paths=find(arProps,rootPath,category,property,value)` specifies a constraining value on a property of the specified `category` of elements, narrowing the search.

Note: This function requires the Embedded Coder Support Package for AUTOSAR Standard. You install support packages using the MATLAB **Add-Ons** menu. On the MATLAB **Home** tab, in the **Environment** section, click **Add-Ons > Get Hardware Support Packages**. For more information, see "Support Package for AUTOSAR Standard" on page 1-3.

Examples

Find Sender-Receiver Interfaces That Are Not Services

For a model, find sender-receiver interfaces for which the property `IsService` is `false` and return fully qualified paths.

```
rtwdemo_autosar_multirunnables
arProps=autosar.api.getAUTOSARProperties('rtwdemo_autosar_multirunnables');
ifPaths=find(arProps,[],'SenderReceiverInterface',...
  'IsService',false,'PathType','FullyQualified')

ifPaths =
    '/pkg/if/Interface1'    '/pkg/if/Interface2'
```

Find Mode-Switch Interface Paths

For a model, add a mode-switch interface and then use `find` to list paths for mode-switch interfaces in the model.

```
rtwdemo_autosar_multirunnables
arProps=autosar.api.getAUTOSARProperties('rtwdemo_autosar_multirunnables');
addPackageableElement(arProps,'ModeSwitchInterface','/pkg/if','Interface3',...
  'IsService',true);
ifPaths=find(arProps,[],'ModeSwitchInterface','PathType','FullyQualified')

ifPaths =
    '/pkg/if/Interface3'
```

Input Arguments

`arProps` — AUTOSAR properties information for a model

handle

AUTOSAR properties information for a model, previously returned by *`arProps`* `= autosar.api.getAUTOSARProperties(`*`model`*`)`. *`model`* is a handle or character vector representing the model name.

Example: `arProps`

`rootPath` — Starting point of the search

character vector

Path specifying the starting point at which to look for the specified type of AUTOSAR elements. `[ ]` indicates the root of the component.

Example: `[ ]`

`category` — Type of AUTOSAR element

character vector

Type of AUTOSAR element for which to return paths.

Example: `'SenderReceiverInterface'`

`'PathType',value` — Whether the returned paths are fully qualified or partially qualified
`'PartiallyQualified'` (default) | `'FullyQualified'`

Specify `FullyQualified` to return fully qualified paths.

Example: `'PathType','FullyQualified'`

`property,value` — Property and value
name (character vector), value

Valid property of the specified category of elements, and a value to match for that property in the search. Table "Properties of AUTOSAR Elements" on page 4-269 lists properties that are associated with AUTOSAR elements.

Example: `'IsService',true`

Output Arguments

`paths` — Return structure
cell array of character vectors

Structure to which paths are returned.

Example: `ifPaths`

See Also

`add` | `delete` | `get` | `set`

Introduced in R2013b

get

Get property of AUTOSAR element

Syntax

```
pValue=get(arProps,elementPath,property)
```

Description

`pValue=get(arProps,elementPath,property)` returns the value of the property of the AUTOSAR element at `elementPath`.

Note: This function requires the Embedded Coder Support Package for AUTOSAR Standard. You install support packages using the MATLAB **Add-Ons** menu. On the MATLAB **Home** tab, in the **Environment** section, click **Add-Ons** > **Get Hardware Support Packages**. For more information, see "Support Package for AUTOSAR Standard" on page 1-3.

Examples

Get Value of IsService Property of Sender-Receiver Interface

For a model, get the value of the `IsService` property for the sender-receiver interface `Interface1`. The variable `IsService` returns `false` (0), indicating that the sender-receiver interface is not a service.

```
rtwdemo_autosar_multirunnables
arProps=autosar.api.getAUTOSARProperties('rtwdemo_autosar_multirunnables');
isService=get(arProps,'Interface1','IsService')
```

```
isService =
     0
```

- "Configure and Map AUTOSAR Component Programmatically" on page 4-267
- "AUTOSAR Interface Configuration" on page 4-3

Input Arguments

`arProps` — AUTOSAR properties information for a model

handle

AUTOSAR properties information for a model, previously returned by *`arProps`* `= autosar.api.getAUTOSARProperties(`*`model`*`)`. *`model`* is a handle or character vector representing the model name.

Example: `arProps`

`elementPath` — Path to AUTOSAR element

character vector

Path to the AUTOSAR element for which to return the value of a property.

Example: `'Input'`

`property` — Type of property

character vector

Type of property to add for which to return a value, among valid properties for the AUTOSAR element.

Example: `'IsService'`

Output Arguments

`pValue` — Return value

value of property | path to composite property or property that references other properties

Variable that returns the value of the specified AUTOSAR property. For composite properties or properties that reference other properties, the return value is the path to the property.

Example: `ifPaths`

See Also

`set`

Introduced in R2013b

getApplicationComponentNames

Class: arxml.importer
Package: arxml

Get list of application software component names

Syntax

```
applicationSoftwareComponentNames =
getApplicationComponentNames(importerObj)
```

Description

`applicationSoftwareComponentNames = getApplicationComponentNames(importerObj)` returns the names of application software component names found in the XML files associated with `importerObj`, an `arxml.importer` object.

Note: This method requires the Embedded Coder Support Package for AUTOSAR Standard. You install support packages using the MATLAB **Add-Ons** menu. On the MATLAB **Home** tab, in the **Environment** section, click **Add-Ons > Get Hardware Support Packages**. For more information, see "Support Package for AUTOSAR Standard" on page 1-3.

Input Arguments

`importerObj`	Handle to imported AUTOSAR information created in a previous call to `arxml.importer`.

Output Arguments

`applicationSoftwareComponentNames`	Cell array of character vectors. Each element is absolute short-name path of corresponding application software component: `'/root_package_name[/sub_package_name]/component_short_name'`

See Also

arxml.importer.getSensorActuatorComponentNames |
arxml.importer.getComponentNames

How To

- “Import AUTOSAR Software Component” on page 3-4

Introduced in R2011b

getCalibrationComponentNames

Class: arxml.importer
Package: arxml

Get calibration component names

Syntax

```
calibrationComponentNames =
getCalibrationComponentNames(importerObj)
```

Description

`calibrationComponentNames = getCalibrationComponentNames(importerObj)` returns the list of calibration component names found in the XML files associated with the `arxml.importer` object, *importerObj*.

Note: This method requires the Embedded Coder Support Package for AUTOSAR Standard. You install support packages using the MATLAB **Add-Ons** menu. On the MATLAB **Home** tab, in the **Environment** section, click **Add-Ons > Get Hardware Support Packages**. For more information, see "Support Package for AUTOSAR Standard" on page 1-3.

Input Arguments

importerObj	Handle to imported AUTOSAR information created in a previous call to `arxml.importer`.

Output Arguments

calibrationComponentNames	Cell array of character vectors in which each element is the absolute short-name path of the corresponding calibration parameter component:

```
'/root_package_name[/sub_package_name]/component_short_name'
```

How To

- “Import AUTOSAR Software Component” on page 3-4

Introduced in R2008b

getClientServerInterfaceNames

Class: arxml.importer
Package: arxml

Get list of client-server interfaces

Syntax

```
interfaceNames = getClientServerInterfaceNames(importerObj)
```

Description

`interfaceNames = getClientServerInterfaceNames(importerObj)` returns the names of client-server interfaces found in the XML files associated with `importerObj`, an `arxml.importer` object.

Note: This method requires the Embedded Coder Support Package for AUTOSAR Standard. You install support packages using the MATLAB **Add-Ons** menu. On the MATLAB **Home** tab, in the **Environment** section, click **Add-Ons > Get Hardware Support Packages**. For more information, see "Support Package for AUTOSAR Standard" on page 1-3.

Input Arguments

`importerObj`	Handle to imported AUTOSAR information created in a previous call to `arxml.importer`.

Output Arguments

`interfaceNames`	Cell array of character vectors. Each element is absolute short-name path of corresponding client-server interface: `'/root_package_name[/sub_package_name]/client_server_interface_short_name'`

How To

- "Model AUTOSAR Communication" on page 2-11
- "Import AUTOSAR Software Component" on page 3-4
- "Configure AUTOSAR Client-Server Communication" on page 4-101

Introduced in R2011a

getComponentNames

Class: arxml.importer
Package: arxml

Get application and sensor/actuator software component names

Syntax

```
componentNames = getComponentNames(importerObj)
```

Description

`componentNames = getComponentNames(importerObj)` returns the list of application and sensor/actuator software component names in the XML file associated with the `arxml.importer` object, *`importerObj`*.

`getComponentNames` finds only the application and sensor/actuator software components defined in the XML file specified when constructing the `arxml.importer` object or the XML file specified by the method `setFile`. The application software components and sensor/actuator software components described in the XML file dependencies are ignored.

Note: This method requires the Embedded Coder Support Package for AUTOSAR Standard. You install support packages using the MATLAB **Add-Ons** menu. On the MATLAB **Home** tab, in the **Environment** section, click **Add-Ons > Get Hardware Support Packages**. For more information, see "Support Package for AUTOSAR Standard" on page 1-3.

Input Arguments

`importerObj`	Handle to imported AUTOSAR information created in a previous call to `arxml.importer`.

Output Arguments

componentNames Cell array of character vectors in which each element is the absolute short-name path of the corresponding application software component or sensor/actuator software component:

```
'/root_package_name[/sub_package_name]/component_short_name'
```

See Also

arxml.importer.getSensorActuatorComponentNames |
arxml.importer.getApplicationComponentNames

How To

- “Import AUTOSAR Software Component” on page 3-4

Introduced in R2008a

getDataTransfer

Get AUTOSAR mapping information for Simulink data transfer

Syntax

```
[arIrvName,arDataAccessMode]=getDataTransfer(slMap,slDataTransfer)
```

Description

`[arIrvName,arDataAccessMode]=getDataTransfer(slMap,slDataTransfer)` returns the values of the AUTOSAR inter-runnable variable `arIrvName` and AUTOSAR data access mode `arDataAccessMode` that are mapped to Simulink data transfer line or Rate Transition block `slDataTransfer`.

Note: This function requires the Embedded Coder Support Package for AUTOSAR Standard. You install support packages using the MATLAB **Add-Ons** menu. On the MATLAB **Home** tab, in the **Environment** section, click **Add-Ons > Get Hardware Support Packages**. For more information, see "Support Package for AUTOSAR Standard" on page 1-3.

Examples

Get AUTOSAR Mapping Information for Simulink Data Transfer Line

Get AUTOSAR mapping information for a data transfer line in the example model `rtwdemo_autosar_multirunnables`. The model has data transfer lines named `irv1`, `irv2`, `irv3`, and `irv4`.

```
rtwdemo_autosar_multirunnables
slMap=autosar.api.getSimulinkMapping('rtwdemo_autosar_multirunnables');
[arIrvName,arDataAccessMode]=getDataTransfer(slMap,'irv4')
```

```
arIrvName =
IRV4
```

```
arDataAccessMode =
Implicit
```

Get AUTOSAR Mapping Information for Rate Transition Block

Get AUTOSAR mapping information for a Rate Transition block in the example model `mMultitasking_4rates`. The model has Rate Transition blocks named `RateTransition`, `RateTransition1`, and `RateTransition2`, which are located at the top level of the model.

```
open_system(fullfile(matlabroot,'/help/toolbox/ecoder/examples/autosar/mMultitasking_4rates'))
slMap=autosar.api.getSimulinkMapping('mMultitasking_4rates');
[arIrvName,arDataAccessMode]=getDataTransfer(slMap,'mMultitasking_4rates/RateTransition')
```

```
arIrvName =
IRV1

arDataAccessMode =
Implicit
```

- “Configure and Map AUTOSAR Component Programmatically” on page 4-267
- “AUTOSAR Interface Configuration” on page 4-3

Input Arguments

`slMap` — Simulink to AUTOSAR mapping information for a model

handle

Simulink to AUTOSAR mapping information for a model, previously returned by *`slMap`* `= autosar.api.getSimulinkMapping(`*`model`*`).`*`model`* is a handle or character vector representing the model name.

Example: `slMap`

`slDataTransfer` — Simulink data transfer line name or Rate Transition full block path

character vector

Name of the Simulink data transfer line or full block path to the Rate Transition block for which to return AUTOSAR mapping information.

Example: `'irv4'`

Example: `'myModel/RateTransition2'`

Output Arguments

`arIrvName` — Name of AUTOSAR inter-runnable variable

character vector

Variable that returns the name of AUTOSAR inter-runnable variable mapped to the specified Simulink data transfer.

Example: `arIrvName`

`arDataAccessMode` — Value of AUTOSAR data access mode

character vector

Variable that returns the value of the AUTOSAR data access mode mapped to the specified Simulink data transfer. The value is `Implicit` or `Explicit`.

Example: `arDataAcessMode`

See Also

`mapDataTransfer`

Introduced in R2013b

getDependencies

Class: arxml.importer
Package: arxml

Get list of XML dependency files

Syntax

```
Dependencies = getDependencies(importerObj)
```

Description

`Dependencies = getDependencies(importerObj)` returns the list of XML dependency files associated with the `arxml.importer` object, *`importerObj`*.

Note: This method requires the Embedded Coder Support Package for AUTOSAR Standard. You install support packages using the MATLAB **Add-Ons** menu. On the MATLAB **Home** tab, in the **Environment** section, click **Add-Ons > Get Hardware Support Packages**. For more information, see "Support Package for AUTOSAR Standard" on page 1-3.

Input Arguments

`importerObj`	Handle to imported AUTOSAR information created in a previous call to `arxml.importer`.

Output Arguments

`Dependencies`	Cell array of character vectors.

How To

- "Import AUTOSAR Software Component" on page 3-4

Introduced in R2008a

getFile

Class: arxml.importer
Package: arxml

Return software component XML file name

Syntax

```
filename = getFile(importerObj)
```

Description

`filename = getFile(importerObj)` returns the name of the main software component XML file associated with the `arxml.importer` object, *`importerObj`*.

Note: This method requires the Embedded Coder Support Package for AUTOSAR Standard. You install support packages using the MATLAB **Add-Ons** menu. On the MATLAB **Home** tab, in the **Environment** section, click **Add-Ons > Get Hardware Support Packages**. For more information, see "Support Package for AUTOSAR Standard" on page 1-3.

Input Arguments

`importerObj`	Handle to imported AUTOSAR information created in a previous call to `arxml.importer`.

Output Arguments

`filename`	XML file name

Examples

Get the name of the main software component file associated with an `arxml.importer` object.

```
>> obj = arxml.importer({'control_system_component.arxml',...
                         'control_system_interface.arxml', ...
                         'control_system_datatype.arxml'})

obj =

The file "H:\wrk\control_system_component.arxml" contains:
  1 Application-Software-Component-Type:
    '/ComponentType/controlSystem'

  0 Sensor-Actuator-Software-Component-Type.
  0 CalPrm-Component-Type.
  0 Client-Server-Interface.
>> getFile(obj)

ans =
H:\wrk\control_system_component.arxml

>>
```

See Also

arxml.importer.setFile

How To

- “Import AUTOSAR Software Component” on page 3-4

Introduced in R2008a

getFunction

Get AUTOSAR mapping information for Simulink entry-point function

Syntax

```
arRunnableName=getFunction(slMap,slFcnName)
```

Description

`arRunnableName=getFunction(slMap,slFcnName)` returns the value of the AUTOSAR runnable `arRunnableName` mapped to the Simulink entry-point function `slFcnName`.

Note: This function requires the Embedded Coder Support Package for AUTOSAR Standard. You install support packages using the MATLAB **Add-Ons** menu. On the MATLAB **Home** tab, in the **Environment** section, click **Add-Ons > Get Hardware Support Packages**. For more information, see "Support Package for AUTOSAR Standard" on page 1-3.

Examples

Get AUTOSAR Mapping Information for Simulink Entry-Point Function

Get AUTOSAR mapping information for a Simulink entry point function in the example model `rtwdemo_autosar_multirunnables`. The model has an initialization entry-point function named `InitializeFunction` and three exported entry-point functions named `Runnable1`, `Runnable2`, and `Runnable3`.

```
open_system('rtwdemo_autosar_multirunnables')
slMap=autosar.api.getSimulinkMapping('rtwdemo_autosar_multirunnables');
arRunnableName=getFunction(slMap,'InitializeFunction')
```

```
arRunnableName =
Runnable_Init
```

- "Configure and Map AUTOSAR Component Programmatically" on page 4-267

- "AUTOSAR Interface Configuration" on page 4-3

Input Arguments

`slMap` — Simulink to AUTOSAR mapping information for a model

handle

Simulink to AUTOSAR mapping information for a model, previously returned by *`slMap`* `= autosar.api.getSimulinkMapping(`*`model`*`).`*`model`* is a handle or character vector representing the model name.

Example: `slMap`

`slFcnName` — Name of Simulink entry point function

character vector

Name of the Simulink entry point function for which to return AUTOSAR mapping information, specified as follows:

- For an initialization function, `'InitializeFunction'`.
- For a rate-based function, `'StepFunction'` for a base-rate task or `'StepFunction`*`N`*`'` for a sub-rate task, where *`N`* is the task identifier.
- For an exported function, `'`*`FunctionCallName`*`'`, where *`FunctionCallName`* is the name of the Inport block that drives the control port of the function-call subsystem. For example, `'Runnable1'` in example model `rtwdemo_autosar_multirunnables` or `'FunctionTrigger'` in example model `rtwdemo_autosar_swc_fcncalls`.
- For a Simulink function in a client-server configuration, `'`*`SlFunctionName`*`'`, where*`SlFunctionName`* is the name of the Simulink function. For example, `'readData'` in the example model used in "Configure AUTOSAR Server" on page 4-101.

Example: `'StepFunction2'`

Output Arguments

`arRunnableName` — Name of AUTOSAR runnable

character vector

Variable that returns the name of the AUTOSAR runnable mapped to the specified Simulink entry-point function.

Example: `arRunnableName`

See Also

`mapFunction`

Introduced in R2013b

getFunctionCaller

Get AUTOSAR mapping information for Simulink function-caller block

Syntax

```
[arPortName,arOperationName] = getFunctionCaller(slMap,slFcnName)
```

Description

`[arPortName,arOperationName] = getFunctionCaller(slMap,slFcnName)` returns the value of the AUTOSAR client port `arPortName` and AUTOSAR operation `arOperationName` mapped to the Simulink function caller block for Simulink function `slFcnName`.

Note: This function requires the Embedded Coder Support Package for AUTOSAR Standard. You install support packages using the MATLAB **Add-Ons** menu. On the MATLAB **Home** tab, in the **Environment** section, click **Add-Ons > Get Hardware Support Packages**. For more information, see "Support Package for AUTOSAR Standard" on page 1-3.

Examples

Get AUTOSAR Mapping Information for Function Caller Block

Get AUTOSAR mapping information for a function-caller block in a model in which AUTOSAR client function invocation is being modeled. The model has a function-caller block for Simulink function `readData`.

```
open_system('mControllerWithInterface_client')
slMapC = autosar.api.getSimulinkMapping('mControllerWithInterface_client');
mapFunctionCaller(slMapC,'readData','cPort','readData');
[arPort,arOp] = getFunctionCaller(slMapC,'readData')
```

```
arPort =
cPort

arOp =
```

`readData`

- "Configure and Map AUTOSAR Component Programmatically" on page 4-267
- "AUTOSAR Interface Configuration" on page 4-3

Input Arguments

`slMap` — Simulink to AUTOSAR mapping information for a model

handle

Simulink to AUTOSAR mapping information for a model, previously returned by *`slMap`* `= autosar.api.getSimulinkMapping(`*`model`*`).`*`model`* is a handle or character vector representing the model name.

Example: `slMap`

`slFcnName` — Name of Simulink function

character vector

Name of the Simulink function for the function-caller block for which to return AUTOSAR mapping information.

Example: `'readData'`

Output Arguments

`arPortName` — Name of AUTOSAR client port

character vector

Variable that returns the name of the AUTOSAR client port mapped to the specified function-caller block.

Example: `arPort`

`arOperationName` — Name of AUTOSAR operation

character vector

Variable that returns the name of the AUTOSAR operation mapped to the specified function-caller block.

Example: `arOp`

See Also

`mapFunctionCaller`

Introduced in R2014b

getInport

Get AUTOSAR mapping information for Simulink inport

Syntax

```
[arPortName,arDataElementName,arDataAccessMode]=getInport(slMap,
slPortName)
```

Description

`[arPortName,arDataElementName,arDataAccessMode]=getInport(slMap, slPortName)` returns the values of the AUTOSAR port `arPortName`, AUTOSAR data element `arDataElementName`, and AUTOSAR data access mode `arDataAccessMode` mapped to Simulink inport `slPortName`.

Note: This function requires the Embedded Coder Support Package for AUTOSAR Standard. You install support packages using the MATLAB **Add-Ons** menu. On the MATLAB **Home** tab, in the **Environment** section, click **Add-Ons > Get Hardware Support Packages**. For more information, see "Support Package for AUTOSAR Standard" on page 1-3.

Examples

Get AUTOSAR Mapping Information for Model Inport

Get AUTOSAR mapping information for a model inport in the example model `rtwdemo_autosar_multirunnable`. The model has an inport named `RPort_DE1`.

```
rtwdemo_autosar_multirunnables
slMap=autosar.api.getSimulinkMapping('rtwdemo_autosar_multirunnables');
[arPortName,arDataElementName,arDataAccessMode]=getInport(slMap,'RPort_DE1')
```

```
arPortName =
RPort

arDataElementName =
```

```
DE1

arDataAccessMode =
ImplicitReceive
```

- "Configure and Map AUTOSAR Component Programmatically" on page 4-267
- "AUTOSAR Interface Configuration" on page 4-3

Input Arguments

`slMap` — Simulink to AUTOSAR mapping information for a model

handle

Simulink to AUTOSAR mapping information for a model, previously returned by *`slMap`* `= autosar.api.getSimulinkMapping(`*`model`*`).` *`model`* is a handle or character vector representing the model name.

Example: `slMap`

`slPortName` — Name of model inport

character vector

Name of the model inport for which to return AUTOSAR mapping information.

Example: `'Input'`

Output Arguments

`arPortName` — Name of AUTOSAR port

character vector

Variable that returns the name of the AUTOSAR port mapped to the specified Simulink inport.

Example: `arPortName`

`arDataElementName` — Name of AUTOSAR data element

character vector

Variable that returns the name of the AUTOSAR data element mapped to the specified Simulink inport.

Example: `arDataElementName`

`arDataAccessMode` — Value of AUTOSAR data access mode

character vector

Variable that returns the value of the AUTOSAR data access mode mapped to the specified Simulink inport. The value can be `ImplicitReceive`, `ExplicitReceive`, `QueuedExplicitReceive`, `ErrorStatus`, `ModeReceive`, `IsUpdated`, `EndToEndRead`, or `ExplicitReceiveByVal`

Example: `arDataAccessMode`

See Also

`mapInport`

Introduced in R2013b

getLookupTable

Get AUTOSAR mapping information for Simulink lookup table

Syntax

```
[arParameterAccessMode,arPortName,arParameterData]=getLookupTable(
slMap,slParam)
```

Description

`[arParameterAccessMode,arPortName,arParameterData]=getLookupTable(slMap,slParam)` returns the values of the AUTOSAR parameter access mode `arParameterAccessMode`, AUTOSAR parameter receiver port `arPortName`, and AUTOSAR parameter data item `arParameterData` mapped to Simulink lookup table `slParam`.

Note: This function requires the Embedded Coder Support Package for AUTOSAR Standard. You install support packages using the MATLAB **Add-Ons** menu. On the MATLAB **Home** tab, in the **Environment** section, click **Add-Ons > Get Hardware Support Packages**. For more information, see "Support Package for AUTOSAR Standard" on page 1-3.

Examples

Get AUTOSAR Mapping Information for Simulink Lookup Tables

Get AUTOSAR mapping information for Simulink lookup tables. The model has lookup tables named `L_4x6_single` and `L_4_single`.

- Simulink lookup table `L_4x6_single` is mapped to AUTOSAR parameter data item `L_4x6_single_ar`, which the AUTOSAR software component defines and accesses internally. (The parameter is not associated with a port-based parameter interface.) Parameter `L_4x6_single_ar` uses `Shared` parameter access mode.

- Simulink lookup table `L_4_single` is mapped to AUTOSAR parameter data item `prmDE1`, which is a data element associated with AUTOSAR parameter receiver port `prmRPort1`. Parameter `prmDE1` uses `PortParameter` parameter access mode.

```
open_system('mySWC')
slMap=autosar.api.getSimulinkMapping('mySWC');
[arParameterAccessMode,arPortName,arParameterData]=getLookupTable(slMap,'L_4x6_single')

arParameterAccessmode =
Shared

arPortName =
''

arParameterData =
L_4x6_single_ar

[arParameterAccessMode,arPortName,arParameterData]=getLookupTable(slMap,'L_4_single')

arParameterAccessmode =
PortParameter

arPortName =
prmRPort1

arParameterData =
prmDE1
```

- “Configure and Map AUTOSAR Component Programmatically” on page 4-267
- “AUTOSAR Interface Configuration” on page 4-3

Input Arguments

`slMap` — Simulink to AUTOSAR mapping information for a model

handle

Simulink to AUTOSAR mapping information for a model, previously returned by *`slMap`* `= autosar.api.getSimulinkMapping(`*`model`*`)`. *`model`* is a handle or character vector representing the model name.

Example: `slMap`

`slParam` — Name of Simulink lookup table

character vector

Name of the Simulink lookup table for which to return AUTOSAR mapping information.

Example: `'L_4x6_single'`

Output Arguments

`arParameterAccessMode` — Value of AUTOSAR parameter access mode

character vector

Variable that returns the value of the AUTOSAR parameter access mode mapped to the specified Simulink lookup table. The value can be `PortParameter`, `Shared`, `PerInstance`, or `Const`.

Example: `arParameterAccessMode`

`arPortName` — Name of AUTOSAR port

character vector

Variable that returns the name of the AUTOSAR parameter receiver port mapped to the specified Simulink lookup table. If the parameter returned by `arParameterData` is internal to the AUTOSAR software component, and not associated with a port-based parameter interface, `arPortName` returns an empty character vector.

Example: `arPortName`

`arParameterData` — Name of AUTOSAR parameter data item

character vector

Variable that returns the name of the AUTOSAR parameter data item mapped to the specified Simulink lookup table. The parameter can be internal to the AUTOSAR software component or associated with a port-based parameter interface.

Example: `arParameterData`

See Also

`mapLookupTable`

Introduced in R2016b

getOutport

Get AUTOSAR mapping information for Simulink outport

Syntax

```
[arPortName,arDataElementName,arDataAccessMode]=getOutport(slMap,
slPortName)
```

Description

`[arPortName,arDataElementName,arDataAccessMode]=getOutport(slMap, slPortName)` returns the values of the AUTOSAR provider port `arPortName`, AUTOSAR data element `arDataElementName`, and AUTOSAR data access mode `arDataAccessMode` mapped to Simulink outport `slPortName`.

Note: This function requires the Embedded Coder Support Package for AUTOSAR Standard. You install support packages using the MATLAB **Add-Ons** menu. On the MATLAB **Home** tab, in the **Environment** section, click **Add-Ons > Get Hardware Support Packages**. For more information, see "Support Package for AUTOSAR Standard" on page 1-3.

Examples

Get AUTOSAR Mapping Information for Model Outport

Get AUTOSAR mapping information for a model outport in the example model `rtwdemo_autosar_multirunnables`. The model has an outport named `PPort_DE1`.

```
rtwdemo_autosar_multirunnables
slMap=autosar.api.getSimulinkMapping('rtwdemo_autosar_multirunnables');
[arPortName,arDataElementName,arDataAccessMode]=getOutport(slMap,'PPort_DE1')
```

```
arPortName =
PPort

arDataElementName =
```

```
DE1

arDataAccessMode =
ImplicitSend
```

- "Configure and Map AUTOSAR Component Programmatically" on page 4-267
- "AUTOSAR Interface Configuration" on page 4-3

Input Arguments

`slMap` — Simulink to AUTOSAR mapping information for a model

handle

Simulink to AUTOSAR mapping information for a model, previously returned by *`slMap`* `= autosar.api.getSimulinkMapping(`*`model`*`)`. *`model`* is a handle or character vector representing the model name.

Example: `slMap`

`slPortName` — Name of model outport

character vector

Name of the model outport for which to return AUTOSAR mapping information.

Example: `'Output'`

Output Arguments

`arPortName` — Name of AUTOSAR port

character vector

Variable that returns the name of the AUTOSAR port mapped to the specified Simulink outport.

Example: `arPortName`

`arDataElementName` — Name of AUTOSAR data element

character vector

Variable that returns the name of the AUTOSAR data element mapped to the specified Simulink outport.

Example: `arDataElementName`

`arDataAccessMode` — Value of AUTOSAR data access mode

character vector

Variable that returns the value of the AUTOSAR data access mode mapped to the specified Simulink outport. The value can be `ImplicitSend`, `ImplicitSendByRef`, `ExplicitSend`, `EndToEndWrite`, or `ModeSend`.

Example: `arDataAcessMode`

See Also

`mapOutport`

Introduced in R2013b

getSensorActuatorComponentNames

Class: arxml.importer
Package: arxml

Get list of sensor/actuator software component names

Syntax

```
sensoractuatorSoftwareComponentNames =
getSensorActuatorComponentNames(importerObj)
```

Description

`sensoractuatorSoftwareComponentNames = getSensorActuatorComponentNames(importerObj)` returns the names of sensor/actuator software component names found in the XML files associated with `importerObj`, an `arxml.importer` object.

Note: This method requires the Embedded Coder Support Package for AUTOSAR Standard. You install support packages using the MATLAB **Add-Ons** menu. On the MATLAB **Home** tab, in the **Environment** section, click **Add-Ons > Get Hardware Support Packages**. For more information, see "Support Package for AUTOSAR Standard" on page 1-3.

Input Arguments

`importerObj`	Handle to imported AUTOSAR information created in a previous call to `arxml.importer`.

Output Arguments

`sensoractuator Software ComponentNames`	Cell array of character vectors. Each element is absolute short-name path of corresponding sensor/actuator software component: `'/root_package_name[/sub_package_name]/component_short_name'`

See Also

arxml.importer.getApplicationComponentNames | arxml.importer.getComponentNames

How To

- “Import AUTOSAR Software Component” on page 3-4

Introduced in R2011b

mapDataTransfer

Map Simulink data transfer to AUTOSAR inter-runnable variable

Syntax

```
mapDataTransfer(slMap,slDataTransfer,arIrvName,arDataAccessMode)
```

Description

`mapDataTransfer(slMap,slDataTransfer,arIrvName,arDataAccessMode)` maps the Simulink data transfer line or Rate Transition block `slDataTransfer` to AUTOSAR inter-runnable variable `arIrvName` and AUTOSAR data access mode `arDataAccessMode`.

Note: This function requires the Embedded Coder Support Package for AUTOSAR Standard. You install support packages using the MATLAB **Add-Ons** menu. On the MATLAB **Home** tab, in the **Environment** section, click **Add-Ons** > **Get Hardware Support Packages**. For more information, see "Support Package for AUTOSAR Standard" on page 1-3.

Examples

Set AUTOSAR Mapping Information for Simulink Data Transfer Line

Set AUTOSAR mapping information for a data transfer line in the example model `rtwdemo_autosar_multirunnables`. The model has data transfer lines named `irv1`, `irv2`, `irv3`, and `irv4`. This example changes the AUTOSAR data access mode for `irv4` from `Implicit` to `Explicit`.

```
rtwdemo_autosar_multirunnables
slMap=autosar.api.getSimulinkMapping('rtwdemo_autosar_multirunnables');
mapDataTransfer(slMap,'irv4','IRV4','Explicit');
[arIrvName,arDataAccessMode]=getDataTransfer(slMap,'irv4')
```

```
arIrvName =
IRV4
```

```
arDataAccessMode =
Explicit
```

Set AUTOSAR Mapping Information for Rate Transition Block

Set AUTOSAR mapping information for a Rate Transition block in the example model `mMultitasking_4rates`. The model has Rate Transition blocks named `RateTransition`, `RateTransition1`, and `RateTransition2`, which are located at the top level of the model. This example changes the AUTOSAR data access mode for `RateTransition` from `Implicit` to `Explicit`.

```
open_system(fullfile(matlabroot,'/help/toolbox/ecoder/examples/autosar/mMultitasking_4rates'))
slMap=autosar.api.getSimulinkMapping('mMultitasking_4rates');
mapDataTransfer(slMap,'mMultitasking_4rates/RateTransition','IRV1','Explicit');
[arIrvName,arDataAccessMode]=getDataTransfer(slMap,'mMultitasking_4rates/RateTransition')
```

```
arIrvName =
IRV1

arDataAccessMode =
Explicit
```

- “Configure and Map AUTOSAR Component Programmatically” on page 4-267
- “AUTOSAR Interface Configuration” on page 4-3

Input Arguments

`slMap` — Simulink to AUTOSAR mapping information for a model

handle

Simulink to AUTOSAR mapping information for a model, previously returned by *`slMap`* `= autosar.api.getSimulinkMapping(`*`model`*`)`. *`model`* is a handle or character vector representing the model name.

Example: `slMap`

`slDataTransfer` — Simulink data transfer line name or Rate Transition full block path

character vector

Name of the Simulink data transfer line or full block path to the Rate Transition block for which to set AUTOSAR mapping information.

Example: `'irv4'`

Example: `'myModel/RateTransition2'`

`arIrvName` — Name of AUTOSAR inter-runnable variable

character vector

Name of the AUTOSAR inter-runnable variable to which to map the specified Simulink data transfer.

Example: `'IRV4'`

`arDataAccessMode` — Value of AUTOSAR data access mode

character vector

Value of the AUTOSAR data access mode to which to map the specified Simulink data transfer. The value can be `Implicit` or `Explicit`.

Example: `'Explicit'`

See Also

`getDataTransfer`

Introduced in R2013b

mapFunction

Map Simulink entry-point function to AUTOSAR runnable

Syntax

```
mapFunction(slMap,slFcnName,arRunnableName)
```

Description

`mapFunction(slMap,slFcnName,arRunnableName)` maps the Simulink entry-point function `slFcnName` to the AUTOSAR runnable `arRunnableName`.

Note: This function requires the Embedded Coder Support Package for AUTOSAR Standard. You install support packages using the MATLAB **Add-Ons** menu. On the MATLAB **Home** tab, in the **Environment** section, click **Add-Ons > Get Hardware Support Packages**. For more information, see "Support Package for AUTOSAR Standard" on page 1-3.

Examples

Set AUTOSAR Mapping Information for Simulink Entry-Point Function

Set AUTOSAR mapping information for a Simulink entry point function in the example model `rtwdemo_autosar_multirunnables`. The model has an initialization entry-point function named `InitializeFunction` and three exported entry-point functions named `Runnable1`, `Runnable2`, and `Runnable3`.

```
open_system('rtwdemo_autosar_multirunnables')
slMap=autosar.api.getSimulinkMapping('rtwdemo_autosar_multirunnables');
mapFunction(slMap,'InitializeFunction','Runnable_Init');
arRunnableName=getFunction(slMap,'InitializeFunction')
```

```
arRunnableName =
Runnable_Init
```

- "Configure and Map AUTOSAR Component Programmatically" on page 4-267

- "AUTOSAR Interface Configuration" on page 4-3

Input Arguments

`slMap` — Simulink to AUTOSAR mapping information for a model

handle

Simulink to AUTOSAR mapping information for a model, previously returned by *`slMap`* `= autosar.api.getSimulinkMapping(`*`model`*`).`*`model`* is a handle or character vector representing the model name.

Example: `slMap`

`slFcnName` — Name of Simulink entry point function

character vector

Name of the Simulink entry point function for which to set AUTOSAR mapping information, specified as follows.

- For an initialization function, `'InitializeFunction'`.
- For a rate-based function, `'StepFunction'` for a base-rate task or `'StepFunction`*`N`*`'` for a sub-rate task, where *`N`* is the task identifier.
- For an exported function, `'`*`FunctionCallName`*`'`, where *`FunctionCallName`* is the name of the Inport block that drives the control port of the function-call subsystem. For example, `'Runnable1'` in example model `rtwdemo_autosar_multirunnables` or `'FunctionTrigger'` in example model `rtwdemo_autosar_swc_fcncalls`.
- For a Simulink function in a client-server configuration, `'`*`SlFunctionName`*`'`, where*`SlFunctionName`* is the name of the Simulink function. For example, `'readData'` in the example model used in "Configure AUTOSAR Server" on page 4-101.

Example: `'StepFunction2'`

`arRunnableName` — Name of AUTOSAR runnable

character vector

Name of the AUTOSAR runnable to which to map the specified Simulink entry-point function.

Example: `'Runnable2'`

See Also

`getFunction`

Introduced in R2013b

mapFunctionCaller

Map Simulink function-caller block to AUTOSAR client port and operation

Syntax

```
mapFunctionCaller(slMap,slFcnName,arPortName,arOperationName)
```

Description

`mapFunctionCaller(slMap,slFcnName,arPortName,arOperationName)` maps the Simulink function-caller block for Simulink function `slFcnName` to AUTOSAR client port `arPortName` and AUTOSAR operation `arOperationName`.

If your model has multiple callers of Simulink function `slFcnName`, this function maps all of them to the AUTOSAR client port and operation.

Note: This function requires the Embedded Coder Support Package for AUTOSAR Standard. You install support packages using the MATLAB **Add-Ons** menu. On the MATLAB **Home** tab, in the **Environment** section, click **Add-Ons > Get Hardware Support Packages**. For more information, see "Support Package for AUTOSAR Standard" on page 1-3.

Examples

Set AUTOSAR Mapping Information for Function Caller Block

Set AUTOSAR mapping information for a function-caller block in a model in which AUTOSAR client function invocation is being modeled. The model has a function-caller block for Simulink function `readData`.

```
open_system('mControllerWithInterface_client')
slMapC = autosar.api.getSimulinkMapping('mControllerWithInterface_client');
mapFunctionCaller(slMapC,'readData','cPort','readData');
[arPort,arOp] = getFunctionCaller(slMapC,'readData')
```

```
arPort =
```

```
cPort

arOp =
readData
```

- “Configure and Map AUTOSAR Component Programmatically” on page 4-267
- “AUTOSAR Interface Configuration” on page 4-3

Input Arguments

`slMap` — Simulink to AUTOSAR mapping information for a model

handle

Simulink to AUTOSAR mapping information for a model, previously returned by *`slMap`* `= autosar.api.getSimulinkMapping(`*`model`*`).`*`model`* is a handle or character vector representing the model name.

Example: `slMap`

`slFcnName` — Name of Simulink function

character vector

Name of the Simulink function for the function-caller block for which to set AUTOSAR mapping information.

Example: `'readData'`

`arPortName` — Name of AUTOSAR client port

character vector

Name of the AUTOSAR client port to which to map the specified function-caller block.

Example: `'cPort'`

`arOperationName` — Name of AUTOSAR operation

character vector

Name of the AUTOSAR operation to which to map the specified function-caller block.

Example: `'readData'`

See Also

`getFunctionCaller`

Introduced in R2014b

mapInport

Map Simulink inport to AUTOSAR port

Syntax

```
mapInport(slMap,slPortName,arPortName,arDataElementName,
arDataAccessMode)
```

Description

`mapInport(slMap,slPortName,arPortName,arDataElementName,
arDataAccessMode)` maps the Simulink inport `slPortName` to the AUTOSAR data element `arDataElementName` at AUTOSAR receiver port `arPortName`. The AUTOSAR data access mode for the receiver port is set to `arDataAccessMode`.

Note: This function requires the Embedded Coder Support Package for AUTOSAR Standard. You install support packages using the MATLAB **Add-Ons** menu. On the MATLAB **Home** tab, in the **Environment** section, click **Add-Ons** > **Get Hardware Support Packages**. For more information, see "Support Package for AUTOSAR Standard" on page 1-3.

Examples

Set AUTOSAR Mapping Information for Model Inport

Set AUTOSAR mapping information for a model inport in the example model `rtwdemo_autosar_multirunnables`. The model has an inport named `RPort_DE1`. This example changes the AUTOSAR data access mode for `RPort_DE1` from `ImplicitReceive` to `ExplicitReceive`.

```
rtwdemo_autosar_multirunnables
slMap=autosar.api.getSimulinkMapping('rtwdemo_autosar_multirunnables');
mapInport(slMap,'RPort_DE1','RPort','DE1','ExplicitReceive');
[arPortName,arDataElementName,arDataAccessMode]=getInport(slMap,'RPort_DE1')
```

```
arPortName =
RPort
```

```
arDataElementName =
DE1

arDataAccessMode =
ExplicitReceive
```

- "Configure and Map AUTOSAR Component Programmatically" on page 4-267
- "AUTOSAR Interface Configuration" on page 4-3

Input Arguments

`slMap` — Simulink to AUTOSAR mapping information for a model

handle

Simulink to AUTOSAR mapping information for a model, previously returned by *`slMap`* `= autosar.api.getSimulinkMapping(`*`model`*`).`*`model`* is a handle or character vector representing the model name.

Example: `slMap`

`slPortName` — Name of model inport

character vector

Name of the model inport for which to set AUTOSAR mapping information.

Example: `'Input'`

`arPortName` — Name of AUTOSAR port

character vector

Name of the AUTOSAR port to which to map the specified Simulink inport.

Example: `'Input'`

`arDataElementName` — Name of AUTOSAR data element

character vector

Name of the AUTOSAR data element to which to map the specified Simulink inport.

Example: `'Input'`

`arDataAccessMode` — Value of AUTOSAR data access mode

character vector

Value of the AUTOSAR data access mode to which to map the specified Simulink inport. The value can be `ImplicitReceive`, `ExplicitReceive`, `QueuedExplicitReceive`, `ErrorStatus`, `ModeReceive`, `IsUpdated`, `EndToEndRead`, or `ExplicitReceiveByVal`.

Example: `'ExplicitReceive'`

See Also

`getInport`

Introduced in R2013b

mapLookupTable

Map Simulink lookup table to AUTOSAR parameter

Syntax

```
mapLookupTable(slMap,slParam,arParameterAccessMode,arPortName,
arParameterData)
```

Description

`mapLookupTable(slMap,slParam,arParameterAccessMode,arPortName, arParameterData)` maps the Simulink lookup table `slParam` to the AUTOSAR parameter data item `arParameterData` and, if defined, AUTOSAR parameter receiver port `arPortName`. The AUTOSAR parameter access mode for the parameter is set to `arParameterAccessMode`.

Note: This function requires the Embedded Coder Support Package for AUTOSAR Standard. You install support packages using the MATLAB **Add-Ons** menu. On the MATLAB **Home** tab, in the **Environment** section, click **Add-Ons > Get Hardware Support Packages**. For more information, see "Support Package for AUTOSAR Standard" on page 1-3.

Examples

Set AUTOSAR Mapping Information for Simulink Lookup Tables

Set AUTOSAR mapping information for Simulink lookup tables. The model has lookup tables named `L_4x6_single` and `L_4_single`. This example:

- Maps Simulink lookup table `L_4x6_single` to AUTOSAR parameter data item `L_4x6_single_ar`, which the AUTOSAR software component defines and accesses internally. (The parameter is not associated with a port-based parameter interface.) Parameter `L_4x6_single_ar` uses `Shared` parameter access mode.

- Maps Simulink lookup table `L_4_single` to AUTOSAR parameter data item `prmDE1`, which is a data element associated with AUTOSAR parameter receiver port `prmRPort1`. Parameter `prmDE1` uses `PortParameter` parameter access mode.

```
open_system('mySWC')
slMap=autosar.api.getSimulinkMapping('mySWC');
mapLookupTable(slMap,'L_4x6_single','Shared','','L_4x6_single_ar');
[arParameterAccessMode,arPortName,arParameterData]=getLookupTable(slMap,'L_4x6_single')

arParameterAccessmode =
Shared

arPortName =
''

arParameterData =
L_4x6_single_ar

mapLookupTable(slMap,'L_4_single','PortParameter','prmRPort1','prmDE1');
[arParameterAccessMode,arPortName,arParameterData]=getLookupTable(slMap,'L_4_single')

arParameterAccessmode =
PortParameter

arPortName =
prmRPort1

arParameterData =
prmDE1
```

- "Configure and Map AUTOSAR Component Programmatically" on page 4-267
- "AUTOSAR Interface Configuration" on page 4-3

Input Arguments

`slMap` — Simulink to AUTOSAR mapping information for a model

handle

Simulink to AUTOSAR mapping information for a model, previously returned by *`slMap`* `= autosar.api.getSimulinkMapping(`*`model`*`)`. *`model`* is a handle or character vector representing the model name.

Example: `slMap`

`slParam` — Name of Simulink lookup table

character vector

Name of the Simulink lookup table for which to set AUTOSAR mapping information.

Example: `'L_4x6_single'`

`arParameterAccessMode` — Value of AUTOSAR parameter access mode

character vector

Value of the AUTOSAR parameter access mode to which to map the specified Simulink lookup table. The value can be `PortParameter`, `Shared`, `PerInstance`, or `Const`.

Example: `'Shared'`

`arPortName` — Name of AUTOSAR port

character vector

Name of the AUTOSAR parameter receiver port to which to map the specified Simulink lookup table. If the parameter is internal to the AUTOSAR software component, and not associated with a port-based parameter interface, specify `''`.

Example: `'PrmRPort1'`

`arParameterData` — Name of AUTOSAR parameter data item

character vector

Name of the AUTOSAR parameter data item to which to map the specified Simulink lookup table. The parameter can be internal to the AUTOSAR software component or associated with a port-based parameter interface.

Example: `'prmDE1'`

See Also

`getLookupTable`

Introduced in R2016b

mapOutport

Map Simulink outport to AUTOSAR port

Syntax

```
mapOutport(slMap,slPortName,arPortName,arDataElementName,
arDataAccessMode)
```

Description

`mapOutport(slMap,slPortName,arPortName,arDataElementName, arDataAccessMode)` maps the Simulink outport `slPortName` to the AUTOSAR data element `arDataElementName` at AUTOSAR provider port `arPortName`. The AUTOSAR data access mode for the provider port is set to `arDataAccessMode`.

Note:

This function requires the Embedded Coder Support Package for AUTOSAR Standard. You install support packages using the MATLAB **Add-Ons** menu. On the MATLAB **Home** tab, in the **Environment** section, click **Add-Ons** > **Get Hardware Support Packages**. For more information, see "Support Package for AUTOSAR Standard" on page 1-3.

Examples

Set AUTOSAR Mapping Information for Model Outport

Set AUTOSAR mapping information for a model outport in the example model `rtwdemo_autosar_multirunnables`. The model has an outport named `PPort_DE1`. This example changes the AUTOSAR data access mode for `PPort_DE1` from `ImplicitSend` to `ExplicitSend`.

```
rtwdemo_autosar_multirunnables
slMap=autosar.api.getSimulinkMapping('rtwdemo_autosar_multirunnables');
```

```
mapOutport(slMap,'PPort_DE1','PPort','DE1','ExplicitSend');
[arPortName,arDataElementName,arDataAccessMode]=getOutport(slMap,'PPort_DE1')

arPortName =
PPort

arDataElementName =
DE1

arDataAccessMode =
ExplicitSend
```

- “Configure and Map AUTOSAR Component Programmatically” on page 4-267
- “AUTOSAR Interface Configuration” on page 4-3

Input Arguments

`slMap` — Simulink to AUTOSAR mapping information for a model

handle

Simulink to AUTOSAR mapping information for a model, previously returned by *`slMap`* `= autosar.api.getSimulinkMapping(`*`model`*`).`*`model`* is a handle or character vector representing the model name.

Example: `slMap`

`slPortName` — Name of model outport

character vector

Name of the model outport for which to set AUTOSAR mapping information.

Example: `'Output'`

`arPortName` — Name of AUTOSAR port

character vector

Name of the AUTOSAR port to which to map the specified Simulink outport.

Example: `'Output'`

`arDataElementName` — Name of AUTOSAR data element

character vector

Name of the AUTOSAR data element to which to map the specified Simulink outport.

Example: `'Output'`

`arDataAccessMode` — Value of AUTOSAR data access mode

character vector

Value of the AUTOSAR data access mode to which to map the specified Simulink outport. The value can be `ImplicitSend`, `ImplicitSendByRef`, `ExplicitSend`, `EndToEndWrite`, or `ModeSend`.

Example: `'ExplicitSend'`

See Also

`getOutport`

Introduced in R2013b

set

Set property of AUTOSAR element

Syntax

```
set(arProps,elementPath,property,value)
```

Description

`set(arProps,elementPath,property,value)` sets the specified `property` of the AUTOSAR element at `elementPath` to `value`. For properties that reference other elements, `value` is s path. To set XML packaging options, specify `elementPath` as `XmlOptions`.

Note: This function requires the Embedded Coder Support Package for AUTOSAR Standard. You install support packages using the MATLAB **Add-Ons** menu. On the MATLAB **Home** tab, in the **Environment** section, click **Add-Ons** > **Get Hardware Support Packages**. For more information, see "Support Package for AUTOSAR Standard" on page 1-3.

Examples

Set IsService Property for Sender-Receiver Interface

For a model, set the `IsService` property for sender-receiver interface `Interface1` to `true` (1).

```
rtwdemo_autosar_multirunnables
arProps=autosar.api.getAUTOSARProperties('rtwdemo_autosar_multirunnables');
set(arProps,'Interface1','IsService',true);
isService=get(arProps,'Interface1','IsService')
```

```
isService =
     1
```

Set Runnable Symbol Name

For a model, set the `symbol` property for runnable `Runnable1` to `test_symbol`.

```
rtwdemo_autosar_multirunnables
arProps=autosar.api.getAUTOSARProperties('rtwdemo_autosar_multirunnables');
compQName=get(arProps,'XmlOptions','ComponentQualifiedName');
runnables=find(arProps,compQName,'Runnable','PathType','FullyQualified');
runnables(2)

ans =
    '/pkg/swc/ASWC/Behavior/Runnable1'

get(arProps,runnables{2},'symbol')

ans =
Runnable1

set(arProps,runnables{2},'symbol','test_symbol2')
get(arProps,runnables{2},'symbol')

ans =
test_symbol2
```

- "Configure and Map AUTOSAR Component Programmatically" on page 4-267
- "AUTOSAR Interface Configuration" on page 4-3

Input Arguments

`arProps` — AUTOSAR properties information for a model

handle

AUTOSAR properties information for a model, previously returned by *`arProps`* = `autosar.api.getAUTOSARProperties(`*`model`*`)`. *`model`* is a handle or character vector representing the model name.

Example: `arProps`

`elementPath` — Path to an AUTOSAR element

character vector

Path to an AUTOSAR element for which to set a property. To set XML packaging options, specify `XmlOptions`,

Example: `'Input'`

`property` — Type of property

character vector

Type of property for which to specify a value, among valid properties for the AUTOSAR element.

Example: `'IsService'`

`value` — Value of property

value of property | path to composite property or property that references other properties

Value to set for the specified property. For properties that reference other elements, specify a path.

Example: `true`

See Also

`get`

Introduced in R2013b

setDependencies

Class: arxml.importer
Package: arxml

Set XML file dependencies

Syntax

```
setDependencies(importerObj,dependencies)
```

Description

`setDependencies(importerObj,dependencies)` sets the XML file dependencies associated with the `arxml.importer` object, *`importerObj`*.

Note: This method requires the Embedded Coder Support Package for AUTOSAR Standard. You install support packages using the MATLAB **Add-Ons** menu. On the MATLAB **Home** tab, in the **Environment** section, click **Add-Ons > Get Hardware Support Packages**. For more information, see "Support Package for AUTOSAR Standard" on page 1-3.

Input Arguments

`importerObj` — Handle to imported AUTOSAR information created in a previous call to `arxml.importer`.

`dependencies` — Can be:

- a cell array of character vectors (for a list of dependencies)
- a char array (for a single dependency)
- or the empty array [] (for removing a dependency)

Note: The atomic software components described in the XML file dependencies are ignored.

How To

- “Import AUTOSAR Software Component” on page 3-4

Introduced in R2008a

setFile

Class: arxml.importer
Package: arxml

Set software component XML file name

Syntax

```
setFile(importerObj,filename)
```

Description

`setFile(importerObj,filename)` sets the name of the main software component XML file associated with the `arxml.importer` object, *`importerObj`*.

Note: This method requires the Embedded Coder Support Package for AUTOSAR Standard. You install support packages using the MATLAB **Add-Ons** menu. On the MATLAB **Home** tab, in the **Environment** section, click **Add-Ons > Get Hardware Support Packages**. For more information, see "Support Package for AUTOSAR Standard" on page 1-3.

Input Arguments

`importerObj`	Handle to imported AUTOSAR information created in a previous call to `arxml.importer`.
`filename`	XML file name. Only atomic software components described in this file can be imported.

Examples

Set the name of the main software component file associated with an `arxml.importer` object.

```
>> obj = arxml.importer({'control_system_component.arxml',...
                         'control_system_interface.arxml', ...
                         'control_system_datatype.arxml'});
>> setFile(obj,'control_system_component2.arxml')
>> obj

obj =

The file "H:\wrk\control_system_component2.arxml" contains:
  1 Application-Software-Component-Type:
    '/ComponentType/controlSystem'

  0 Sensor-Actuator-Software-Component-Type.
  0 CalPrm-Component-Type.
  0 Client-Server-Interface.
>>
```

See Also

arxml.importer.getFile

How To

- “Import AUTOSAR Software Component” on page 3-4

Introduced in R2008a

updateModel

Class: arxml.importer
Package: arxml

Update AUTOSAR model with `arxml` changes

Syntax

`updateModel(`*`importerObj`*`,`*`modelname`*`)`

Description

`updateModel(`*`importerObj`*`,`*`modelname`*`)` updates the specified open model with changes detected in the imported AUTOSAR description represented by the `arxml.importer` object, *`importerObj`*. The update generates and opens a report that details the updates applied to the model, and required changes that were not made automatically. The imported description must contain the AUTOSAR software component mapped by the model.

Note: This method requires the Embedded Coder Support Package for AUTOSAR Standard. You install support packages using the MATLAB **Add-Ons** menu. On the MATLAB **Home** tab, in the **Environment** section, click **Add-Ons > Get Hardware Support Packages**. For more information, see "Support Package for AUTOSAR Standard" on page 1-3.

Input Arguments

`importerObj`	Handle to imported AUTOSAR information created in a previous call to `arxml.importer`.
`modelname`	Name of the model to be updated with changes in the imported AUTOSAR description represented by *`importerObj`*. The model must be open.

Examples

Update model `mySWC` with AUTOSAR `arxml` changes described in `updatedSWC.arxml` and open an update report.

```
>> open_system('mySWC')
>> obj = arxml.importer('updatedSWC.arxml');
>> updateModel(obj,'mySWC');
### Updating model mySWC
### Saving original model as mySWC_backup.slx
### Creating HTML report mySWC_update_report.html
>>
```

How To

- “Import AUTOSAR Software Component Updates” on page 3-8

Introduced in R2014a

updateReferences

Class: arxml.importer
Package: arxml

Update model with `arxml` definitions of AUTOSAR reference elements

Syntax

```
updateReferences(importerObj,modelname)
updateReferences(importerObj,modelname,Property,Value)
```

Description

`updateReferences(importerObj,modelname)` updates the specified open model with AUTOSAR reference elements. Reference elements are definitions of packageable AUTOSAR elements that multiple components and services can share. The update operation generates a report that details the elements added to the model. The model can reference the imported elements, on a read-only basis, and export the references to AUTOSAR software component `arxml` files.

`updateReferences(importerObj,modelname,Property,Value)` updates the specified open model with AUTOSAR reference elements, using a property-value pair to specify a specific element category, package, or path.

Note: This method requires the Embedded Coder Support Package for AUTOSAR Standard. You install support packages using the MATLAB **Add-Ons** menu. On the MATLAB **Home** tab, in the **Environment** section, click **Add-Ons > Get Hardware Support Packages**. For more information, see "Support Package for AUTOSAR Standard" on page 1-3.

Input Arguments

importerObj	Handle to imported AUTOSAR information created in a previous call to `arxml.importer`.

modelname — Name of the model to be updated with definitions of AUTOSAR reference elements, which are contained in AUTOSAR information represented by *importerObj*. The model must be open.

Property,Value — Optional property/value pair specifying AUTOSAR element categories, packages, or paths from which to import reference element definitions. You can specify values for the following properties:

`'Category'`

Character vector specifying one or more AUTOSAR element categories from which to import reference elements. For example:

```
'Category',{'ImplementationDataType'}
```

`'Package'`

Character vector specifying one or more AUTOSAR element packages from which to import reference elements. For example:

```
'Package',{'/AUTOSAR_PlatformTypes/CompuMethods'}
```

`'RootPath'`

Character vector specifying paths to one or more specific AUTOSAR reference elements to import. For example:

```
'RootPath',{'/ExternalElements/CompuMethods/RpmCm',...
    '/AUTOSAR_PlatformTypes/ImplementationDataTypes/uint16'}
```

To refine a category or package import, you can specify both a category and a package from which to import reference elements. For example:

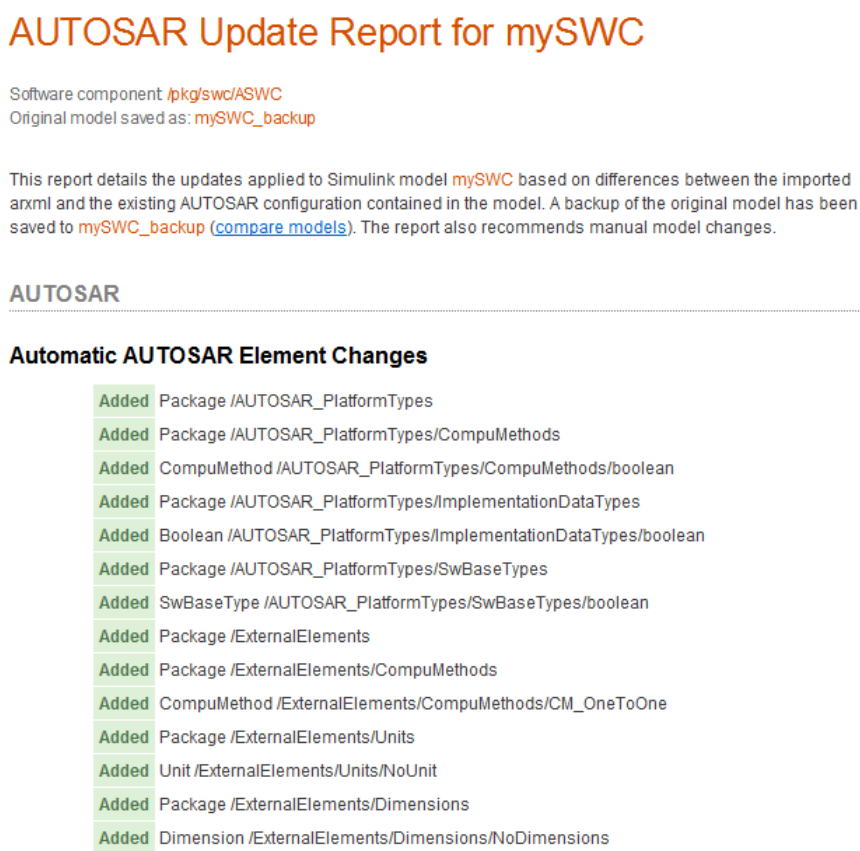
```
'Category',{'ImplementationDataType'}, ...
'Package',{'/AUTOSAR_PlatformTypes/ImplementationDataTypes'}
```

Examples

Update model `mySWC` with AUTOSAR reference elements described in `ExternalElements.arxml` and generate an update report.

```
>> open_system('mySWC')
```

```
>> obj = arxml.importer('ExternalElements.arxml')
obj =
The file ".../ExternalElements.arxml" contains:
  0 Application-Software-Component-Type.
  0 Sensor-Actuator-Software-Component-Type.
  0 Parameter-Software-Component-Type.
  0 Client-Server-Interface.
>> updateReferences(obj,'mySWC');
### Updating references in model mySWC
### Saving original model as mySWC_backup.slx
### Creating HTML report mySWC_update_report.html
>>
```



AUTOSAR Update Report for mySWC

Software component /pkg/swc/ASWC
Original model saved as: mySWC_backup

This report details the updates applied to Simulink model mySWC 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 mySWC_backup (compare models). The report also recommends manual model changes.

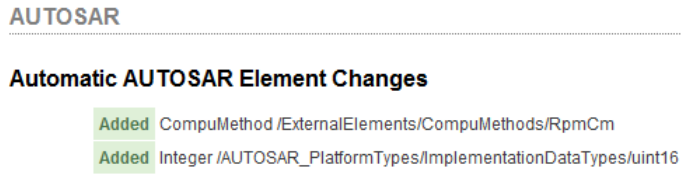
AUTOSAR

Automatic AUTOSAR Element Changes

Added	Package /AUTOSAR_PlatformTypes
Added	Package /AUTOSAR_PlatformTypes/CompuMethods
Added	CompuMethod /AUTOSAR_PlatformTypes/CompuMethods/boolean
Added	Package /AUTOSAR_PlatformTypes/ImplementationDataTypes
Added	Boolean /AUTOSAR_PlatformTypes/ImplementationDataTypes/boolean
Added	Package /AUTOSAR_PlatformTypes/SwBaseTypes
Added	SwBaseType /AUTOSAR_PlatformTypes/SwBaseTypes/boolean
Added	Package /ExternalElements
Added	Package /ExternalElements/CompuMethods
Added	CompuMethod /ExternalElements/CompuMethods/CM_OneToOne
Added	Package /ExternalElements/Units
Added	Unit /ExternalElements/Units/NoUnit
Added	Package /ExternalElements/Dimensions
Added	Dimension /ExternalElements/Dimensions/NoDimensions

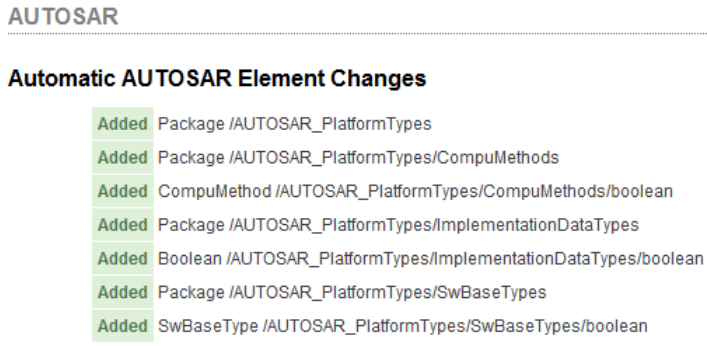
Update model `mySWC` with two AUTOSAR reference elements, specified by root paths `/ExternalElements/CompuMethods/RpmCm` and `/AUTOSAR_PlatformTypes/ImplementationDataTypes/uint16`.

```
>> updateReferences(obj,'mySWC','RootPath',{'/ExternalElements/CompuMethods/RpmCm',...
                            '/AUTOSAR_PlatformTypes/ImplementationDataTypes/uint16'});
```



Update model `mySWC` with AUTOSAR reference elements from package `/AUTOSAR_PlatformTypes/CompuMethods`.

```
>> updateReferences(obj,'mySWC','Package',{'/AUTOSAR_PlatformTypes/CompuMethods'});
```



Update model `mySWC` with AUTOSAR reference elements of category `ImplementationDataType`. Importing `ImplementationDataType` elements also imports dependent elements, such as `SwBaseType` elements.

```
>> updateReferences(obj,'mySWC','Category',{'ImplementationDataType'});
```

See Also

"AUTOSAR reference element definitions for sharing among components and services"

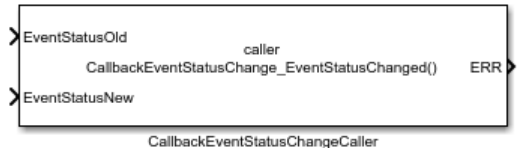
Introduced in R2016b

Blocks — Alphabetical List

CallbackEventStatusChangeCaller
DiagnosticInfoCaller
DiagnosticMonitorCaller
NvMAdminCaller
NvMServiceCaller
Signal Invalidation

CallbackEventStatusChangeCaller

AUTOSAR Diagnostic Event Manager (Dem) event status change callback



Library

Embedded Coder Support Package for AUTOSAR / Basic Software / Diagnostic Event Manager (Dem)

Description

Call AUTOSAR Dem service interface `CallbackEventStatusChange` using a specified operation.

Note: This block requires the Embedded Coder Support Package for AUTOSAR Standard. You install support packages using the MATLAB **Add-Ons** menu. On the MATLAB **Home** tab, in the **Environment** section, click **Add-Ons > Get Hardware Support Packages**. For more information, see "Support Package for AUTOSAR Standard" on page 1-3.

Parameters

Client port name

Name of the client port used by the AUTOSAR software component for the function call, set to `CallbackEventStatusChange` by default.

Operation

Name of an operation defined by the AUTOSAR standard for the Dem service interface `CallbackEventStatusChange`. One operation is supported: `EventStatusChanged`.

Data type for EventStatusOld and EventStatusNew

Name of a data type that defines Dem event status values for the function inputs `EventStatusOld` and `EventStatusNew`. By default, the data type is set to `Enum: Dem_EventStatusExtendedType`. For more information about event status values, see the AUTOSAR standard *Specification of Diagnostic Event Manager*.

Sample time

Block sample time, set to -1 (inherited) by default.

See Also

`DiagnosticInfoCaller`

`DiagnosticMonitorCaller`

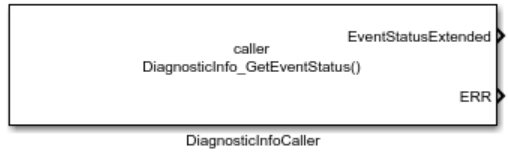
Related Examples

"Configure Calls to AUTOSAR Diagnostic Event Manager (Dem) Service" on page 4-153

Introduced in R2016b

DiagnosticInfoCaller

Get AUTOSAR Diagnostic Event Manager (Dem) event information



Library

Embedded Coder Support Package for AUTOSAR / Basic Software / Diagnostic Event Manager (Dem)

Description

Call AUTOSAR Dem service interface `DiagnosticInfo` using a specified operation.

Note: This block requires the Embedded Coder Support Package for AUTOSAR Standard. You install support packages using the MATLAB **Add-Ons** menu. On the MATLAB **Home** tab, in the **Environment** section, click **Add-Ons > Get Hardware Support Packages**. For more information, see "Support Package for AUTOSAR Standard" on page 1-3.

Parameters

Client port name

Name of the client port used by the AUTOSAR software component for the function call, set to `DiagnosticInfo` by default.

Operation

Name of an operation defined by the AUTOSAR standard for the Dem service interface `DiagnosticInfo`:

- `GetEventStatus`

- `GetEventFailed`
- `GetEventTested`
- `GetDTCOfEvent`
- `GetFaultDetectionCounter`
- `GetEventExtendedDataRecord`
- `GetEventFreezeFrameData`

Data type for EventStatusExtended (`GetEventStatus` only)

Name of a data type that defines Dem event status values for the function output `EventStatusExtended`. By default, the data type is set to `Enum: Dem_EventStatusExtendedType`. For more information about event status values, see the AUTOSAR standard *Specification of Diagnostic Event Manager*.

Data type for DTCFormat (`GetDTCOfEvent` only)

Name of a data type that defines Dem format type values for the function input `DTCFormat`. By default, the data type is set to `Enum: Dem_DTCFormatType`. For more information about format type values, see the AUTOSAR standard *Specification of Diagnostic Event Manager*.

Sample time

Block sample time, set to -1 (inherited) by default.

See Also

`CallbackEventStatusChangeCaller`

`DiagnosticMonitorCaller`

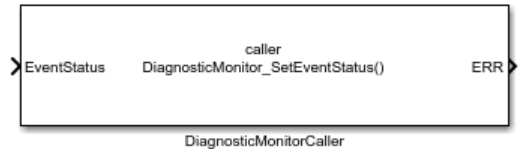
Related Examples

“Configure Calls to AUTOSAR Diagnostic Event Manager (Dem) Service” on page 4-153

Introduced in R2016b

DiagnosticMonitorCaller

Set AUTOSAR Diagnostic Event Manager (Dem) event information



Library

Embedded Coder Support Package for AUTOSAR / Basic Software / Diagnostic Event Manager (Dem)

Description

Call AUTOSAR Dem service interface `DiagnosticMonitor` using a specified operation.

Note: This block requires the Embedded Coder Support Package for AUTOSAR Standard. You install support packages using the MATLAB **Add-Ons** menu. On the MATLAB **Home** tab, in the **Environment** section, click **Add-Ons > Get Hardware Support Packages**. For more information, see "Support Package for AUTOSAR Standard" on page 1-3.

Parameters

Client port name

Name of the client port used by the AUTOSAR software component for the function call, set to `DiagnosticMonitor` by default.

Operation

Name of an operation defined by the AUTOSAR standard for the Dem service interface `DiagnosticMonitor`:

- `SetEventStatus`

- `ResetEventStatus`
- `PrestoreFreezeFrame`
- `ClearPrestoredFreezeFrame`
- `SetEventDisabled`

Data type for EventStatus (`SetEventStatus` only)

Name of a data type that defines Dem event status values for the function input `EventStatus`. By default, the data type is set to `Enum: Dem_EventStatusType`. For more information about event status values, see the AUTOSAR standard *Specification of Diagnostic Event Manager*.

Sample time

Block sample time, set to -1 (inherited) by default.

See Also

`CallbackEventStatusChangeCaller`

`DiagnosticInfoCaller`

Related Examples

"Configure Calls to AUTOSAR Diagnostic Event Manager (Dem) Service" on page 4-153

Introduced in R2016b

NvMAdminCaller

Configure AUTOSAR NVRAM Manager (NvM) administration

Library

Embedded Coder Support Package for AUTOSAR / Basic Software / NVRAM Manager (NvM)

Description

Call AUTOSAR NvM service interface `NvMAdmin` using a specified operation.

Note: This block requires the Embedded Coder Support Package for AUTOSAR Standard. You install support packages using the MATLAB **Add-Ons** menu. On the MATLAB **Home** tab, in the **Environment** section, click **Add-Ons** > **Get Hardware Support Packages**. For more information, see "Support Package for AUTOSAR Standard" on page 1-3.

Parameters

Client port name

Name of the client port used by the AUTOSAR software component for the function call, set to `NvMAdmin` by default.

Operation

Name of an operation defined by the AUTOSAR standard for the NvM service interface `NvMAdmin`. One operation is supported: `SetBlockProtection`.

Sample time

Block sample time, set to -1 (inherited) by default.

See Also

`NvMServiceCaller`

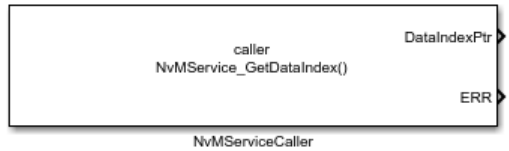
Related Examples

“Configure Calls to AUTOSAR NVRAM Manager (NvM) Service” on page 4-161

Introduced in R2016b

NvMServiceCaller

Configure AUTOSAR NVRAM Manager (NvM) service



Library

Embedded Coder Support Package for AUTOSAR / Basic Software / NVRAM Manager (NvM)

Description

Call AUTOSAR NvM service interface `NvMService` using a specified operation.

Note: This block requires the Embedded Coder Support Package for AUTOSAR Standard. You install support packages using the MATLAB **Add-Ons** menu. On the MATLAB **Home** tab, in the **Environment** section, click **Add-Ons > Get Hardware Support Packages**. For more information, see "Support Package for AUTOSAR Standard" on page 1-3.

Parameters

Client port name

Name of the client port used by the AUTOSAR software component for the function call, set to `NvMService` by default.

Operation

Name of an operation defined by the AUTOSAR standard for the NvM service interface `NvMService`:

- `GetDataIndex`

- `GetErrorStatus`
- `EraseNvBlock`
- `InvalidateNvBlock`
- `ReadBlock`
- `RestoreBlockDefaults`
- `SetDataIndex`
- `SetRamBlockStatus`
- `WriteBlock`

Sample time

Block sample time, set to -1 (inherited) by default.

See Also

`NvMAdminCaller`

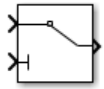
Related Examples

“Configure Calls to AUTOSAR NVRAM Manager (NvM) Service” on page 4-161

Introduced in R2016b

Signal Invalidation

Control AUTOSAR root outport data element invalidation



Library

Embedded Coder / AUTOSAR

Description

Pass through the first input, the data input, when the second input, invalidation control, is false (that is, when the signal is valid). When the second input is true (signal is invalid), keep the previous good value.

In AUTOSAR generated code, the `Rte_Write_`*`Port_Element`* function is called when the signal is valid and `Rte_Invalidate_`*`Port_Element`* when it is not.

This block must be connected directly to a root outport block.

The following modeling rules apply to using a Signal Invalidation block.

- The block outport:
  - Destination should be a root outport mapped to an AUTOSAR interface data element for which the invalidation policy is set to `Keep` or `Replace`.
  - Should not be branched.
  - Should not be connected to a merge block.
- The block:
  - Can be located at the root-level of a model or inside a virtual subsystem.
  - Cannot be located within a reusable subsystem.
  - Can be located inside a Simulink function (connected to an outport, not to an ArgOut block).

- Can be located inside a nonvirtual subsystem only if function packaging is set to `Inline` or `Nonreusable function` (`void-void` only).

See Also

"Configure AUTOSAR Sender Port for Data Element Invalidation" on page 4-90

Introduced in R2015b