

Embedded Coder<sup>®</sup>

AUTOSAR



MATLAB<sup>®</sup>&SIMULINK<sup>®</sup>

R2015b



## How to Contact MathWorks



Latest news: [www.mathworks.com](http://www.mathworks.com)  
Sales and services: [www.mathworks.com/sales\\_and\\_services](http://www.mathworks.com/sales_and_services)  
User community: [www.mathworks.com/matlabcentral](http://www.mathworks.com/matlabcentral)  
Technical support: [www.mathworks.com/support/contact\\_us](http://www.mathworks.com/support/contact_us)



Phone: 508-647-7000



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

*Embedded Coder*<sup>®</sup> AUTOSAR

© COPYRIGHT 2014–2015 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](http://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](http://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)

## 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/](http://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.



## Overview of AUTOSAR Support

**1**

<b>AUTOSAR Standard</b> .....	<b>1-2</b>
<b>Support Package for AUTOSAR Standard</b> .....	<b>1-3</b>
<b>Workflows for AUTOSAR</b> .....	<b>1-4</b>
<b>Sample Workflows</b> .....	<b>1-6</b>
<b>AUTOSAR Terminology</b> .....	<b>1-7</b>

## Modeling Patterns for AUTOSAR

**2**

<b>Simulink Modeling Patterns for AUTOSAR</b> .....	<b>2-2</b>
<b>Model AUTOSAR Software Components</b> .....	<b>2-3</b>
About AUTOSAR Software Components .....	<b>2-3</b>
Multi-Instance Software Components .....	<b>2-4</b>
Multi-Runnable Software Components .....	<b>2-5</b>
<b>Model AUTOSAR Communication</b> .....	<b>2-10</b>
About AUTOSAR Communication .....	<b>2-10</b>
Sender-Receiver Interface .....	<b>2-10</b>
Client-Server Interface .....	<b>2-12</b>
Mode-Switch Interface .....	<b>2-14</b>
<b>Model AUTOSAR Calibration Parameters</b> .....	<b>2-19</b>
About AUTOSAR Calibration Parameters .....	<b>2-19</b>
Import and Export Calibration Parameters .....	<b>2-19</b>

Calibration Components .....	2-20
Calibration Parameters for COM_AXIS Lookup Tables .....	2-21
<b>Model AUTOSAR Data Types .....</b>	<b>2-24</b>
About AUTOSAR Data Types .....	2-24
Enumerated Data Types .....	2-27
Structure Parameters .....	2-28
Release 2.x and 3.x Data Types .....	2-28
Release 4.x Data Types .....	2-29
CompuMethod Categories for Data Types .....	2-32
<b>Model AUTOSAR Component Behavior .....</b>	<b>2-34</b>
AUTOSAR Elements for Modeling Component Behavior .....	2-34
Runnables .....	2-34
Inter-Runnable Variables .....	2-35
Variation Point Proxies .....	2-36
System Constants .....	2-37
Per-Instance Memory .....	2-37
Static and Constant Memory .....	2-38

## AUTOSAR Component Creation

# 3

<b>Import AUTOSAR Software Component .....</b>	<b>3-2</b>
AUTOSAR arxml Importer Tool .....	3-2
General arxml Import Workflow .....	3-3
Import Multi-Runnable Software Component .....	3-5
Merge AUTOSAR Authoring Tool Changes Into Model .....	3-6
<b>Round-Trip Preservation of AUTOSAR Elements and   UUIDs .....</b>	<b>3-10</b>
<b>Create AUTOSAR Software Component in Simulink .....</b>	<b>3-11</b>
<b>Limitations and Tips .....</b>	<b>3-17</b>
Cannot Save Importer Objects in MAT-Files .....	3-17

<b>Configure the AUTOSAR Interface</b> .....	4-3
AUTOSAR Interface Configuration .....	4-3
Map Model Elements Using Simulink-AUTOSAR Mapping Explorer .....	4-5
Configure AUTOSAR Component Using AUTOSAR Properties Explorer .....	4-10
Configure AUTOSAR Package for Interface or CompuMethod .....	4-51
<b>Configure AUTOSAR Packages</b> .....	4-54
AR-PACKAGE Structure .....	4-54
Configure AUTOSAR Packages and Paths .....	4-56
Export AUTOSAR Packages .....	4-59
AR-PACKAGE Location in Exported ARXML Files .....	4-62
<b>Configure AUTOSAR Provide-Require Port</b> .....	4-66
<b>Configure AUTOSAR Port aliveTimeout Value</b> .....	4-70
<b>Configure AUTOSAR Receiver Port for IsUpdated Service</b>	4-71
<b>Configure AUTOSAR Sender Port for Data Element   Invalidation</b> .....	4-74
<b>Configure AUTOSAR S-R Interface Port for End-To-End   Protection</b> .....	4-78
<b>Configure AUTOSAR Receiver Port for   DataReceiveErrorEvent</b> .....	4-81
<b>Configure AUTOSAR Mode Receiver Ports and Mode-Switch   Events</b> .....	4-85
<b>Configure AUTOSAR Client-Server Communication</b> .....	4-92
Configure AUTOSAR Server .....	4-92
Configure AUTOSAR Client .....	4-103
Configure AUTOSAR Client-Server Error Handling .....	4-113
Concurrency Constraints for AUTOSAR Server Runnables	4-118

Configure and Map AUTOSAR Server and Client Programmatically .....	4-121
<b>Configure AUTOSAR Calibration Parameters .....</b>	<b>4-123</b>
<b>Configure AUTOSAR Calibration Component .....</b>	<b>4-125</b>
<b>Configure COM_AXIS Lookup Table for Measurement and Calibration .....</b>	<b>4-129</b>
<b>Configure AUTOSAR Data for Measurement and Calibration .....</b>	<b>4-135</b>
About Software Data Definition Properties (SwDataDefProps) .....	4-135
Configure SwCalibrationAccess .....	4-135
Configure swAddrMethod .....	4-139
Configure swAlignment .....	4-141
swImplPolicy for Exported Data .....	4-142
<b>Configure AUTOSAR Display Format for Measurement and Calibration .....</b>	<b>4-143</b>
<b>Configure AUTOSAR Release 4.x Data Types .....</b>	<b>4-148</b>
Control Application Data Type Generation .....	4-148
Configure DataTypeMappingSet Package and Name .....	4-149
Initialize Data with ApplicationValueSpecification .....	4-151
<b>Configure AUTOSAR CompuMethods .....</b>	<b>4-152</b>
Configure AUTOSAR CompuMethod Properties .....	4-152
Create AUTOSAR CompuMethods .....	4-154
Configure CompuMethod Direction for Linear Functions ..	4-155
Export CompuMethod Unit References .....	4-157
Configure Rational Function CompuMethod for Dual-Scaled Parameter .....	4-157
<b>Configure AUTOSAR Internal Data Constraints Export ..</b>	<b>4-162</b>
<b>Configure AUTOSAR Multiple Runnables .....</b>	<b>4-165</b>
Configure Multiple Runnables Using Rate-Based Multitasking .....	4-165
Configure Multiple Runnables Using Function Calls .....	4-172
<b>Configure AUTOSAR Initialization Runnable .....</b>	<b>4-175</b>



<b>Configure Disabled Mode for AUTOSAR Runnable Event</b>	4-178
<b>Configure AUTOSAR Per-Instance Memory</b> . . . . .	4-179
<b>Configure AUTOSAR Static or Constant Memory</b> . . . . .	4-182
<b>Configure AUTOSAR Variation Point Proxies</b> . . . . .	4-185
<b>Configure and Map AUTOSAR Component</b>	
<b>Programmatically</b> . . . . .	4-188
AUTOSAR Property and Mapping Functions . . . . .	4-188
Tree View of AUTOSAR Configuration . . . . .	4-188
Properties of AUTOSAR Elements . . . . .	4-190
Specify AUTOSAR Element Location . . . . .	4-193
Sample Scripts That Use AUTOSAR Functions . . . . .	4-194
<b>Limitations and Tips</b> . . . . .	4-197
Source of Initial Output Value for Function-Call Subsystem Output . . . . .	4-197
AUTOSAR Client Block in Referenced Model . . . . .	4-197
Use the Merge Block for Inter-Runnable Variables . . . . .	4-197

## AUTOSAR Code Generation

# 5

<b>Export AUTOSAR Component XML and C Code</b> . . . . .	5-2
Inspect XML Options . . . . .	5-2
Select an AUTOSAR Schema . . . . .	5-2
Specify Maximum SHORT-NAME Length . . . . .	5-3
Configure AUTOSAR Compiler Abstraction Macros . . . . .	5-4
Root-Level Matrix I/O . . . . .	5-5
Export AUTOSAR Software Component . . . . .	5-5
<b>Code Replacement for AUTOSAR</b> . . . . .	5-8
AUTOSAR Code Replacement . . . . .	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
<b>Verify AUTOSAR C Code with SIL and PIL .....</b>	<b>5-29</b>
<b>Limitations and Tips .....</b>	<b>5-30</b>
Generate Code Only Check Box .....	5-30
AUTOSAR Compiler Abstraction Macros .....	5-30
Relative File Paths in AUTOSAR Code Descriptors .....	5-30

## Functions — Alphabetical List

6

## Blocks — Alphabetical List

7

# 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<sup>®</sup> 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](http://www.autosar.org) for details on the AUTOSAR standard.
- “Simulink Modeling Patterns for AUTOSAR” on page 2-2 to model AUTOSAR software components and related concepts in Simulink<sup>®</sup>.
- “Workflows for AUTOSAR” on page 1-4 to use Embedded Coder software to generate code and description files that are compliant with AUTOSAR.
- <http://www.mathworks.com/automotive/standards/autosar.html> to learn about using MathWorks<sup>®</sup> 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 download and install the support package,

- 1** On the MATLAB® **Home** tab, click **Add-Ons > Get Hardware Support Packages**.
- 2** Select **Install from Internet** and click **Next**.
- 3** From the list of available support packages, select **AUTOSAR Standard**.
- 4** To complete the installation, follow the instructions provided by Support Package Installer.

For more information, see “Support Package Installation”.

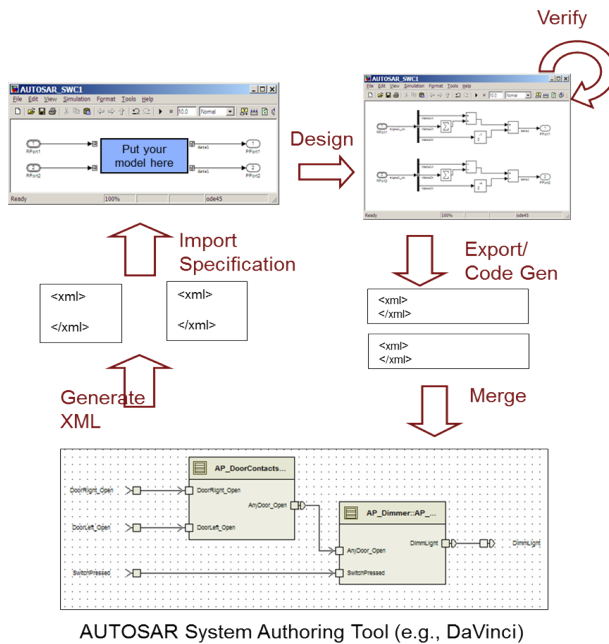
# Workflows for AUTOSAR

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 3-2
  - “Configure AUTOSAR Calibration Parameters” on page 4-123
- 2** Develop the model using Simulink model-based design. This process includes mapping Simulink model elements to AUTOSAR component elements, configuring the AUTOSAR interface, and validating the interface. See:
  - “Configure the AUTOSAR Interface” on page 4-3
  - “Configure AUTOSAR Client-Server Communication” on page 4-92
  - “Configure and Map AUTOSAR Component Programmatically” on page 4-188
- 3** Export the AUTOSAR component from Simulink, generating XML description files and C code files. See:
  - “Export AUTOSAR Software Component” on page 5-5
  - “Configure AUTOSAR Multiple Runnables” on page 4-165

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. See example “Generate AUTOSAR-Compliant Code for Multiple Runnable Entities”.

You can use the authoring tool to export specifications, which can be imported back into Simulink. During the import, you can request that `arxml` code changes be detected and merged into the model. See “Merge AUTOSAR Authoring Tool Changes Into Model” on page 3-6.

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 3-11. Subsequent tasks in the workflow are as listed above.

## Sample Workflows

Embedded Coder provides the following examples to demonstrate AUTOSAR workflows.

<b>Example</b>	<b>How to ...</b>
“Generate Code That Is Compliant with the AUTOSAR Standard”	Generate AUTOSAR-compliant code and export AUTOSAR software component description XML files from a Simulink model.
“Generate AUTOSAR-Compliant Code for Multiple Runnable Entities”	Configure and generate AUTOSAR-compliant code and export AUTOSAR software component description XML files for a Simulink model with multiple runnables.
“Model for AUTOSAR Platform”	Model AUTOSAR atomic software components in Simulink, using rate-based and function-call-based approaches.



## AUTOSAR Terminology

Term	Notes
AUTOSAR Runtime Environment (RTE)	<ul style="list-style-type: none"> <li>• Layer between Application and Basic Software layers</li> <li>• Realizes communication between:               <ul style="list-style-type: none"> <li>• AUTOSAR software components</li> <li>• AUTOSAR software components and Basic Software</li> </ul> </li> </ul>
AUTOSAR Software Component	<ul style="list-style-type: none"> <li>• A software component containing one or more algorithms, which communicates with its environment through ports</li> <li>• Connected to the AUTOSAR Runtime Environment (RTE)</li> <li>• Relocatable (not tied to a particular ECU)</li> </ul>
Characteristics	Values of characteristics can be changed on an ECU through a calibration data management tool or an offline calibration tool.
Client-Server Interface	<ul style="list-style-type: none"> <li>• PortInterface for client-server communication</li> <li>• Defines operations provided by server and used by client</li> </ul>
Composite data types	Category of data types, such as one of the following: <ul style="list-style-type: none"> <li>• Array — Contains more than one element of the same type, and has zero-based indexing</li> <li>• Record — Non-empty set of objects, where each object has a unique identifier</li> </ul>
ComSpec	Defines specific communication attributes.
DataElementPrototype (data element)	Data value (signal) exchanged between a sender and a receiver.
Data types	<ul style="list-style-type: none"> <li>• Either primitive or composite</li> <li>• Types data elements, arguments of operations in a Client-Server Interface, and constants</li> </ul>

Term	Notes
ErrorStatus	<p>Indicates errors detected by communication system. Runtime Environment defines the following macros for sender-receiver communication:</p> <ul style="list-style-type: none"> <li>• RTE_E_OK: no errors</li> <li>• RTE_E_INVALID: data element invalid</li> <li>• RTE_E_MAX_AGE_EXCEEDED: data element outdated</li> </ul>
OperationPrototype (operation)	<ul style="list-style-type: none"> <li>• Invoked by a client</li> <li>• Provides value for each argument with direction <code>in</code> or <code>inout</code>, which must be of the corresponding data type</li> <li>• Client expects to receive a response to the invoked operation, part of which is a value with direction <code>out</code> or <code>inout</code></li> </ul>
PortInterface	<ul style="list-style-type: none"> <li>• Characterizes information provided or required by a port</li> <li>• Can be either Sender-Receiver Interface or Client-Server Interface</li> </ul>
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.
RTEEvent	<p>Event or situation that triggers execution of a runnable by the Runtime Environment (RTE). The software supports the following RTEEvents:</p> <ul style="list-style-type: none"> <li>• OperationInvokedEvent (applicable to server operations)</li> <li>• TimingEvent</li> <li>• DataReceivedEvent</li> </ul>
Runnable entity (runnable)	Part of AUTOSAR Software-Component that can be executed and scheduled independently of other runnable entities (runnables).

Term	Notes
Sender-Receiver Interface	<ul style="list-style-type: none"><li>• PortInterface for sender-receiver communication</li><li>• Defines data elements sent by sending component (with <b>Provide</b> port providing Sender-Receiver Interface) or received by receiving component (with <b>Require</b> requiring Sender-Receiver Interface)</li></ul>
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 2-10
- “Model AUTOSAR Calibration Parameters” on page 2-19
- “Model AUTOSAR Data Types” on page 2-24
- “Model AUTOSAR Component Behavior” on page 2-34

# 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 2-10
- “Model AUTOSAR Calibration Parameters” on page 2-19
- “Model AUTOSAR Data Types” on page 2-24
- “Model AUTOSAR Component Behavior” on page 2-34

# Model AUTOSAR Software Components

**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

## About AUTOSAR Software Components

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

---

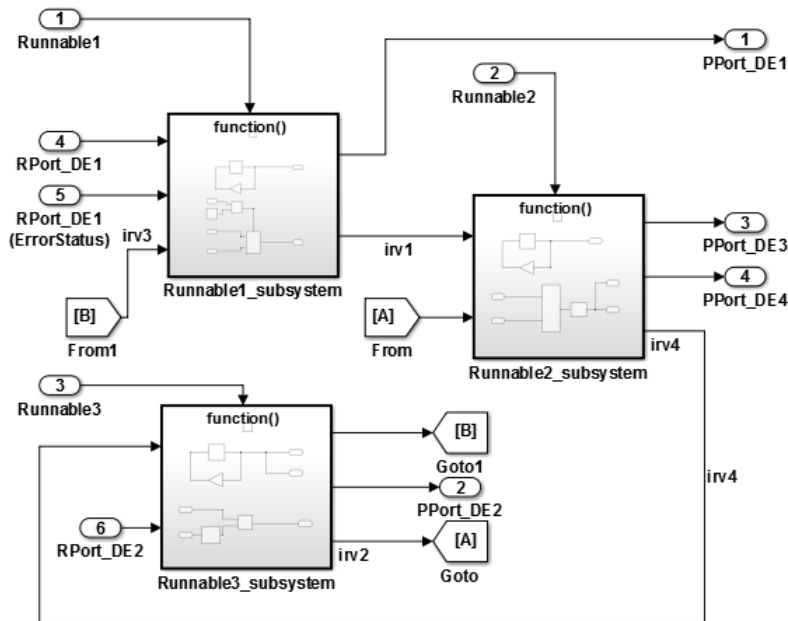
**Note:** An AUTOSAR software component is sometimes referred to as *atomic*, because it is never split across more than one Electronic Control Unit (ECU). Do not confuse *atomic* in this context with the Simulink concept of atomic subsystems.

For information about AUTOSAR calibration components, see “Model AUTOSAR Calibration Parameters” on page 2-19

---

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 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”.

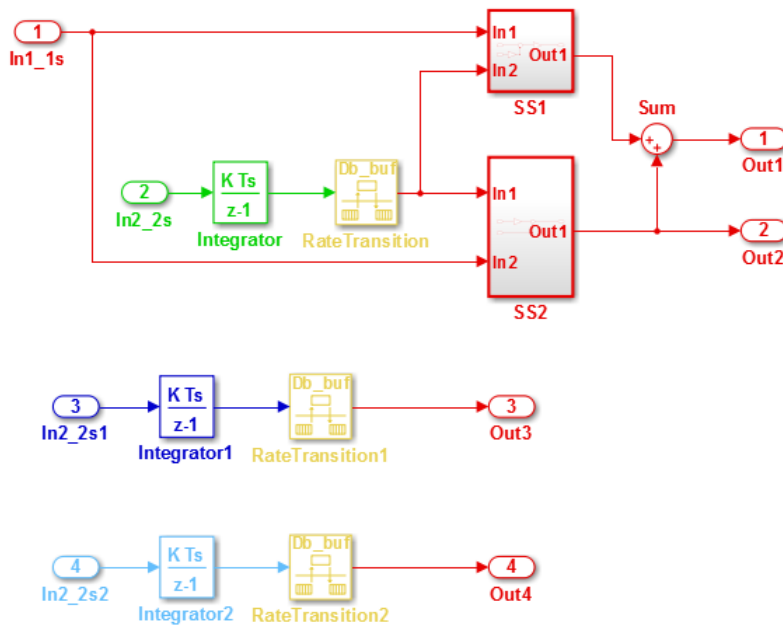
### Model Multiple Runnables Using Rate-Based Multitasking

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.

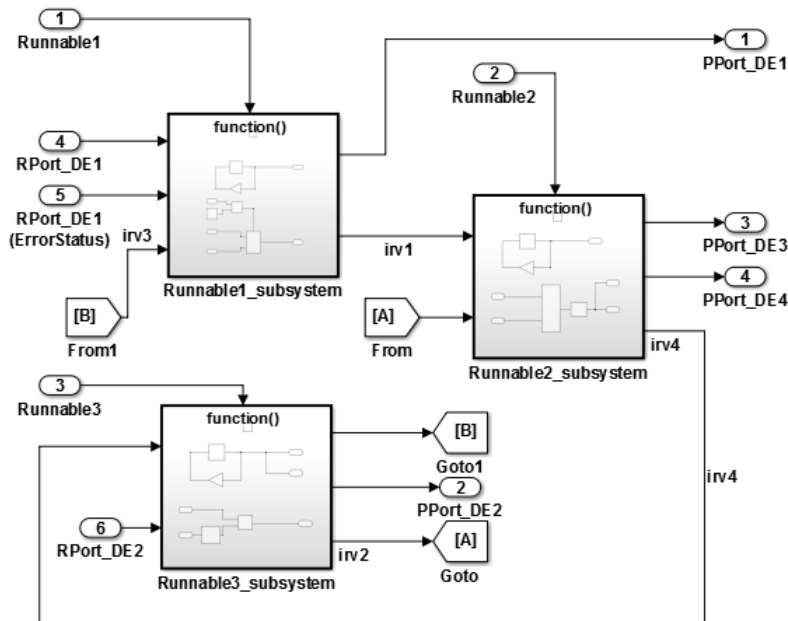


When you generate code, the model C code contains rate-grouped model step functions corresponding to AUTOSAR runnables, one for each discrete rate in the model. (The periodic step functions must be called in the manner of a rate-monotonic scheduler.) For more information, see “Configure Multiple Runnables Using Rate-Based Multitasking” on page 4-165.

### 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 4-172.

### See Also

Rate Transition | Simulink Function

### Related Examples

- “Import AUTOSAR Software Component” on page 3-2
- “Model for AUTOSAR Platform”
- “Configure Multiple Runnables Using Rate-Based Multitasking” on page 4-165
- “Configure Multiple Runnables Using Function Calls” on page 4-172
- “Export AUTOSAR Component XML and C Code” on page 5-2

## **More About**

- “AUTOSAR Interface Configuration” on page 4-3

## Model AUTOSAR Communication

### In this section...

“About AUTOSAR Communication” on page 2-10

“Sender-Receiver Interface” on page 2-10

“Client-Server Interface” on page 2-12

“Mode-Switch Interface” on page 2-14

### About AUTOSAR Communication

AUTOSAR software components provide well-defined connection points, ports. There are two types of AUTOSAR ports:

- Require
- Provide

AUTOSAR ports can reference the following kinds of interfaces:

- Sender-Receiver
- Client-Server
- Mode-Switch

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

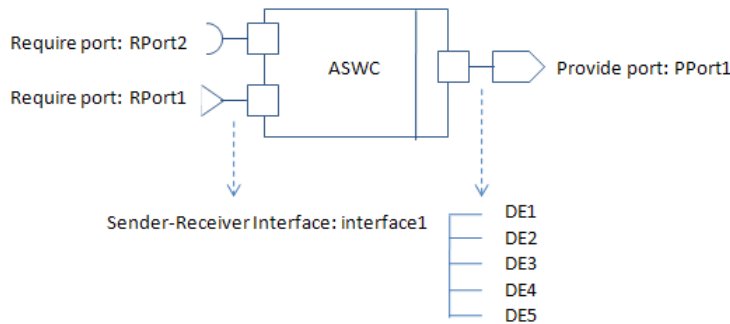


A Require port that references a Mode-Switch interface is called a *mode-receiver port*.

### Sender-Receiver Interface

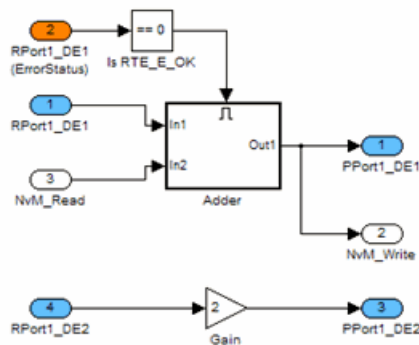
A Sender-Receiver Interface consists of one or more data elements. Although a **Require**, **Provide**, or **Provide-Require** port may reference a Sender-Receiver Interface, the

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 4-66.

*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 “Configure the AUTOSAR Interface” on page 4-3.

### Client-Server Interface

AUTOSAR allows client-server communication between:

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

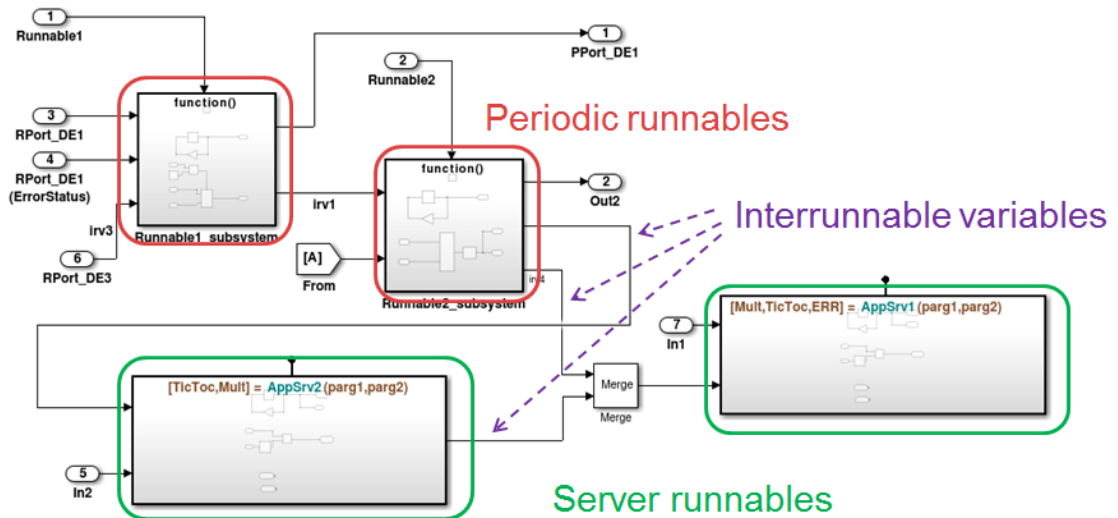
An AUTOSAR Client-Server Interface defines the interaction between a software component that *provides* the interface and a software component that *requires* the interface. The component that provides the interface is the server. The component that requires the interface is the client.

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 a detailed example, see “Configure AUTOSAR Client-Server Communication” on page 4-92.

## Mode-Switch Interface

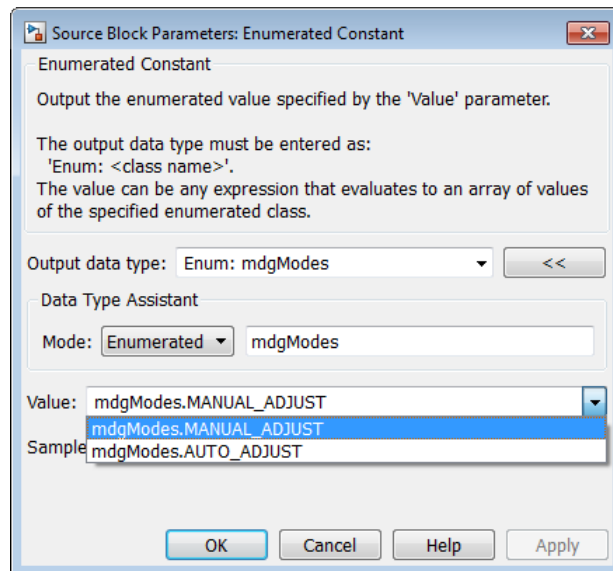
To model an AUTOSAR mode-switch interface, with mode receiver ports and mode-switch events, in Simulink:

- Model the mode receiver port for an AUTOSAR software component using a Simulink inport.
- Specify a mode-switch event to trigger an initialize runnable or other runnable.

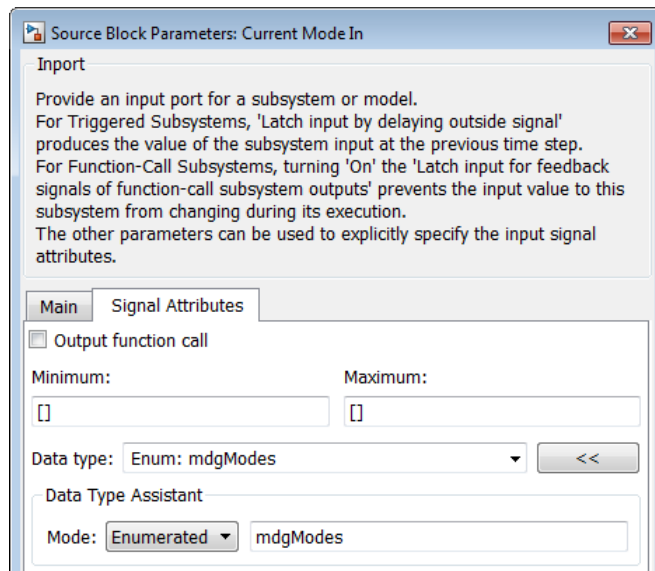
To model an AUTOSAR software component mode-receiver port, general steps might include:

- 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`.

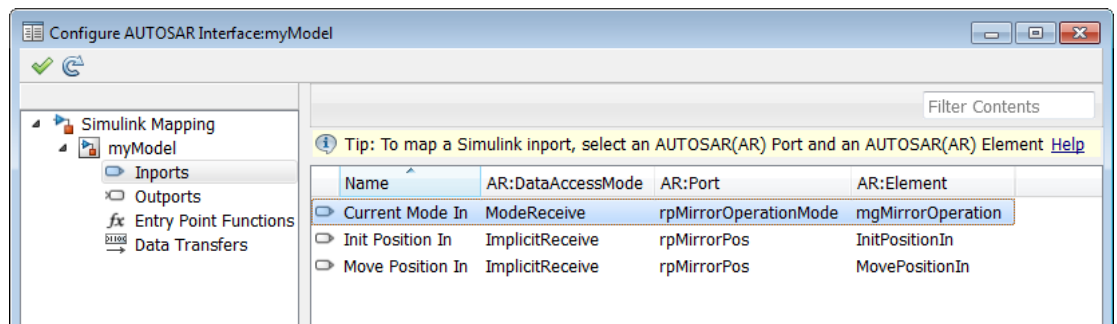
```
Simulink.defineIntEnumType('mdgModes', ...
    {'MANUAL_ADJUST', 'AUTO_ADJUST'}, ...
    [18 28], ...
    'Description', 'Type definition of mdgModes.', ...
    'HeaderFile', 'Rte_Type.h', ...
    'DefaultValue', 'MANUAL_ADJUST', ...
    'AddClassNameToEnumNames', false' ...
);
```



- 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.

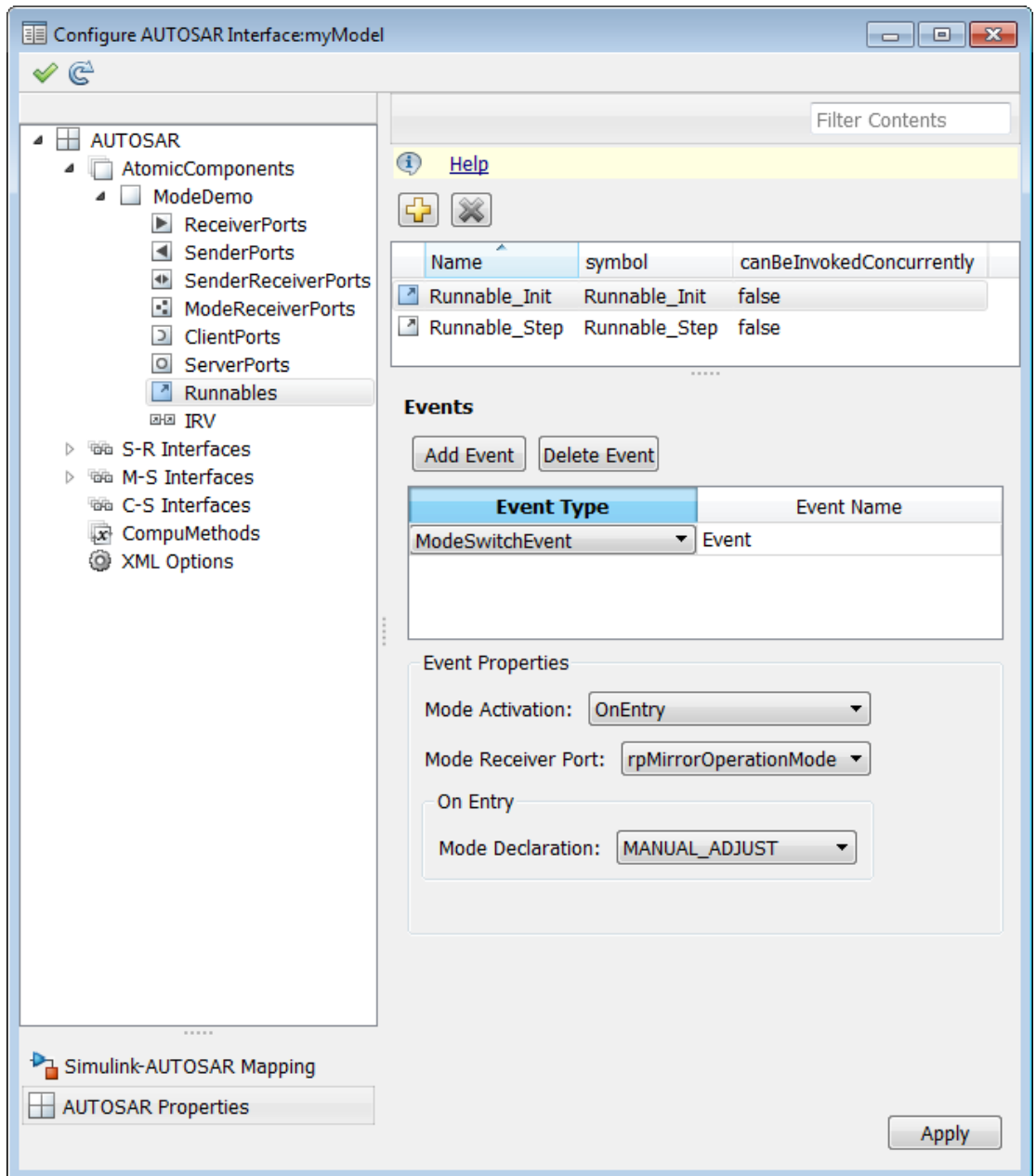


- Use the Configure AUTOSAR Interface dialog box in the **Simulink-AUTOSAR Mapping** view (or equivalent API commands) to specify the mapping of the Simulink inport to the AUTOSAR mode-receiver port. In the following example, Simulink inport `Current Mode In` is mapped to AUTOSAR mode-receiver port `rpMirrorOperationMode` and AUTOSAR element `mgMirrorOperation`.



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\_Init**, 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 a detailed example, see “Configure AUTOSAR Mode Receiver Ports and Mode-Switch Events” on page 4-85.

# Model AUTOSAR Calibration Parameters

## In this section...

“About AUTOSAR Calibration Parameters” on page 2-19

“Import and Export Calibration Parameters” on page 2-19

“Calibration Components” on page 2-20

“Calibration Parameters for COM\_AXIS Lookup Tables” on page 2-21

## 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. See “Calibration Components” on page 2-20.
- Internal calibration parameters, which are defined and accessed by only one AUTOSAR software component. See “Configure AUTOSAR Calibration Parameters” on page 4-123

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 3-2.

You can specify the type of calibration parameter exported by configuring properties of the corresponding block parameter in the base workspace. See “Configure AUTOSAR Calibration Parameters” on page 4-123 and “Generate Code That Is Compliant with the AUTOSAR Standard”.

### Calibration Components

An AUTOSAR calibration component (`ParameterSwComponent`) contains calibration parameters that can be accessed by AUTOSAR SWCs using an associated provider port. If the `arxml` importer finds a calibration component in `arxml` code, it is imported into Simulink and preserved for export. Alternatively, you can create a calibration component in Simulink. To do this, open AUTOSAR calibration parameters that are described by `AUTOSAR.Parameter` data objects in your model. For each parameter, 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.



AUTOSAR.Parameter: K

Standard attributes    Additional attributes

Value: 2

Data type: UInt8

Dimensions: [ 1 1 ]    Complexity: real

Minimum: [ ]    Maximum: [ ]

Units: [ ]

Code generation options

Storage class: CalPrm (Custom)

Custom attributes

HeaderFile: [ ]

ElementName: K

PortName: rCounter

InterfacePath: /CalibrationComponents/counter\_if

CalibrationComponent: /CalibrationComponents/counter\_swk/counter

ProviderPortName: pCounter

Alias: [ ]

Alignment: -1

For a detailed example, see “Configure AUTOSAR Calibration Component” on page 4-125.

## Calibration Parameters for COM\_AXIS Lookup Tables

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 common axis (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 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.

Embedded Coder provides the ability to model COM\_AXIS lookup tables for AUTOSAR applications. You can:

- Import AUTOSAR calibration parameters of category CURVE, MAP, CUBOID, and COM\_AXIS from `arxml` files into Simulink. The importer creates corresponding model content, including n-D Lookup Table blocks and parameter objects.
- Use Simulink to create a COM\_AXIS table and configure it for AUTOSAR run-time calibration.
- Export COM\_AXIS lookup table information in `arxml` code, including calibration parameters of category CURVE, MAP, CUBOID, and COM\_AXIS.

If you import `arxml` code that contains calibration parameters of category CURVE, MAP, CUBOID, and COM\_AXIS, the importer adds an `n-D Lookup Table` block to the model, and creates calibration parameters in the MATLAB workspace. In the lookup table block, table and axis data reference the calibration parameters. 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 inputs to lookup table block inputs.

To create a COM\_AXIS lookup table in Simulink and configure it for run-time calibration, use the following general modeling pattern:

- 1** Add an `n-D Lookup Table` block to your model.
- 2** For table data and axis data that you want to tune or manipulate at run-time, create AUTOSAR calibration parameters in the MATLAB workspace. They can include internal calibration parameters, calibration parameters in a calibration component, and constant memory parameters.
- 3** Configure the lookup table block for COM\_AXIS data. Reference the AUTOSAR calibration parameters from the lookup table block. You can also model AUTOSAR input variables, which measure the input of lookup tables.

When you generate code, the `arxml` exporter generates 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.
- Software record layouts (`SwRecordLayouts`) referenced by the application data types of category CURVE, MAP, CUBOID, and COM\_AXIS.

For examples of lookup table configuration in Simulink, see “Configure COM\_AXIS Lookup Table for Measurement and Calibration” on page 4-129.

## Model AUTOSAR Data Types

In this section...
“About AUTOSAR Data Types” on page 2-24
“Enumerated Data Types” on page 2-27
“Structure Parameters” on page 2-28
“Release 2.x and 3.x Data Types” on page 2-28
“Release 4.x Data Types” on page 2-29
“CompuMethod Categories for Data Types” on page 2-32

### 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.

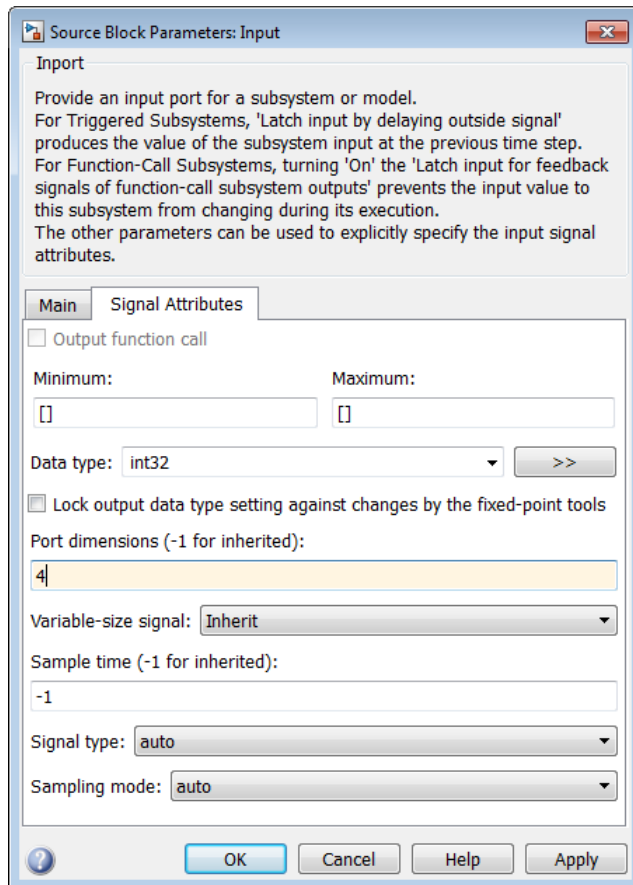
You can use Simulink data types to define AUTOSAR primitive types.

AUTOSAR Data Type	Simulink Data Type
boolean	boolean
float32	single
float64	double
sint8	int8
sint16	int16
sint32	int32
uint8	uint8

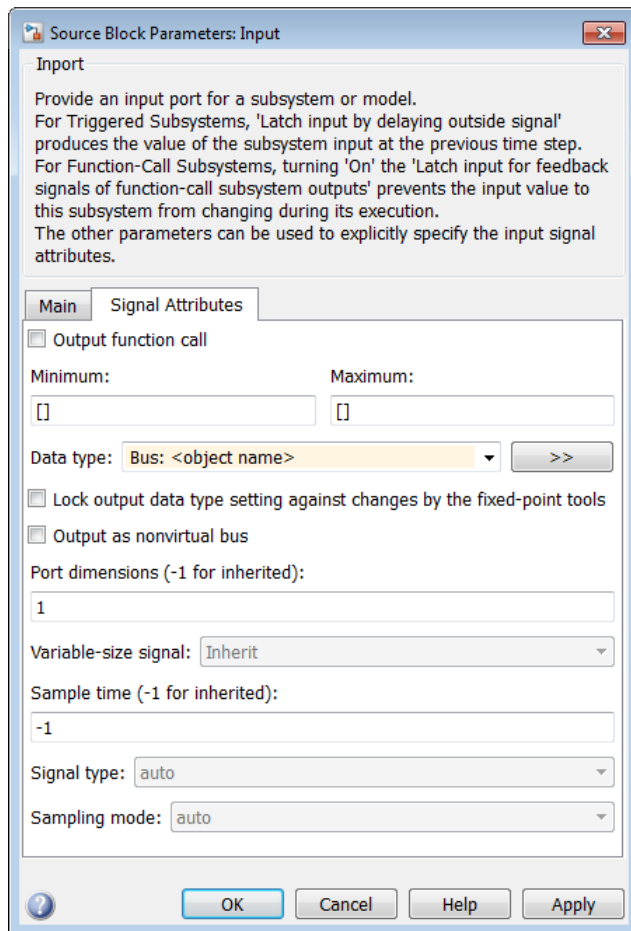
AUTOSAR Data Type	Simulink Data Type
uint16	uint16
uint32	uint32

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.

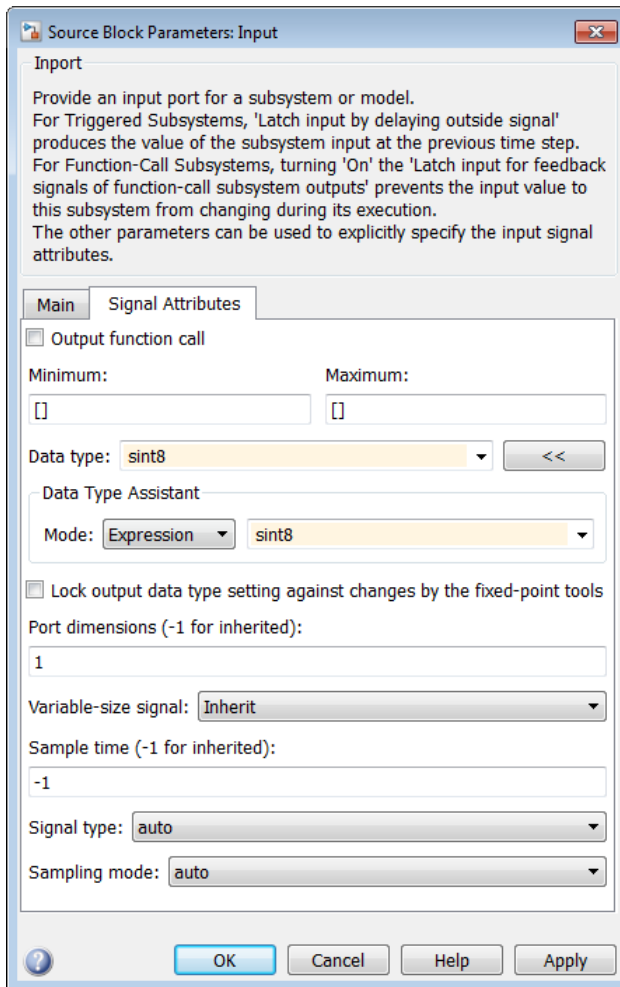
The following figure shows how to specify a wide signal, which corresponds to an AUTOSAR composite array.



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 “Structure Parameters and 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

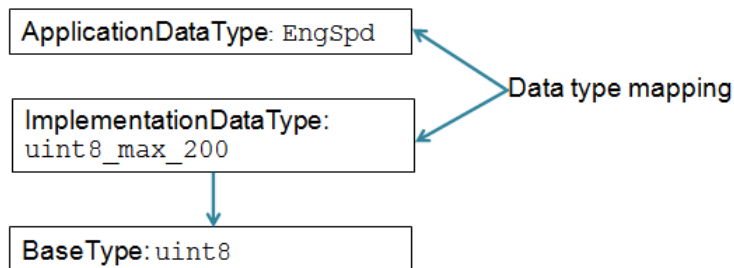
The following table shows how the software translates AUTOSAR R2.x and R3.x data types to Simulink data types. For information about Release 4.x data types, see “Release 4.x Data Types” on page 2-29.



AUTOSAR	Simulink
Primitive types (excluding fixed point), for example, <code>myInt16</code>  Covers Boolean, integer, real	<pre>&gt;&gt; myInt16 = Simulink.AliasType; &gt;&gt; myInt16.BaseType = 'int16'; &gt;&gt; myInt16.IsAlias = true; &gt;&gt; myInt16.HeaderFile = 'Rte_Type.h';</pre>
Primitive type (fixed point), for example, <code>myFixPt</code>	<pre>&gt;&gt; myFixPt = Simulink.NumericType; &gt;&gt; myFixPt.DataTypeMode = ...; &gt;&gt; myFixPt.IsAlias = true; &gt;&gt; myFixPt.HeaderFile = 'Rte_Type.h';</pre>
Enumerations, for example, <code>myEnum</code>	<pre>Simulink.defineIntEnumType('myEnum',... {'Red','Green','Blue'},... [1;2;3],...);</pre>
Record types, for example, <code>myRecord</code>	<pre>&gt;&gt; myRecord = Simulink.Bus;</pre>

## 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 more details about workflow support, see “R4.x Data Types in Simulink Originated Workflow” on page 2-30 and “R4.x Data Types in Round-Trip Workflow” on page 2-31.

For information about mapping value constraints between AUTOSAR application data types and Simulink data types, see “Application Data Type Physical Constraint Mapping” on page 2-32.

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 4-148.

### R4.x Data Types in Simulink Originated Workflow

In this 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'; >> myInt16.IsAlias = true;	<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

With this workflow, you first import an AUTOSAR software component using the XML description generated by an AUTOSAR authoring tool. Later, you generate AUTOSAR code.

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:

```
App1DT1
```

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

```
Impl1DT1
```

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

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

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

### Application Data Type Physical Constraint Mapping

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

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

### CompuMethod Categories for Data Types

AUTOSAR 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 generated by Embedded Coder 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.
- `TEXTTABLE` — Transform internal value into textual elements.

The arxml exporter generates `CompuMethods` for every primitive application type, allowing measurement and calibration tools to monitor and interact with the application

data. The following table shows the **CompuMethod** categories that Embedded Coder software generates for data types in a model that is configured for AUTOSAR.

<b>Data Type</b>	<b>CompuMethod Category</b>	<b>CompuMethod on Application Type</b>	<b>CompuMethod on Implementation Type</b>
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)	Yes	No
Floating-point	IDENTICAL	Yes	No
Integer	IDENTICAL	Yes	No

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

For information about configuring **CompuMethods** for code generation, see “Configure AUTOSAR **CompuMethods**” on page 4-152.

# Model AUTOSAR Component Behavior

### In this section...

“AUTOSAR Elements for Modeling Component Behavior” on page 2-34

“Runnables” on page 2-34

“Inter-Runnable Variables” on page 2-35

“Variation Point Proxies” on page 2-36

“System Constants” on page 2-37

“Per-Instance Memory” on page 2-37

“Static and Constant Memory” on page 2-38

## 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:

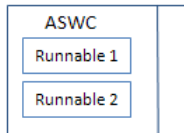
- Runnables and the events to which they respond
- Inter-runnable variables, used to communicate data between runnables in the same component
- Variation point proxies, used to specify variant condition logic inside a runnable
- System constants, used to specify system-level constant values that are 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.

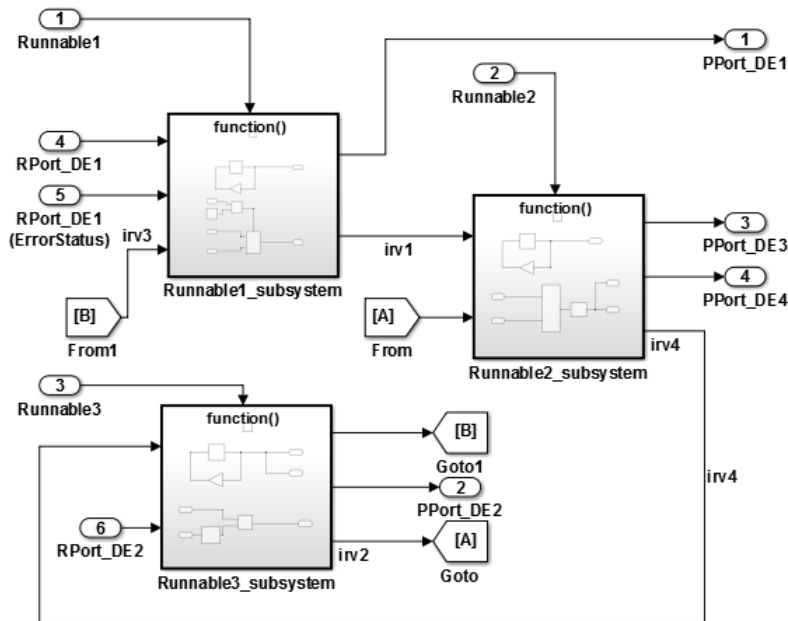
---

**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.

### Variation Point Proxies

AUTOSAR supports variant condition logic inside a runnable, using a `VariationPointProxy`, which was introduced in AUTOSAR schema version 4.0. You can model an AUTOSAR `VariationPointProxy` using the following Simulink elements:

- Variant Subsystem or Model Variant block to model a `VariationPointProxy` inside an AUTOSAR runnable.
- `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 a detailed example, see “Configure AUTOSAR Variation Point Proxies” on page 4-185.



## 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 Variation Point Proxies” on page 4-185.

## 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 non-volatile 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 4-179.

### 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 4-182 and “Configure AUTOSAR Data for Measurement and Calibration” on page 4-135.

# AUTOSAR Component Creation

---

- “Import AUTOSAR Software Component” on page 3-2
- “Round-Trip Preservation of AUTOSAR Elements and UUIDs” on page 3-10
- “Create AUTOSAR Software Component in Simulink” on page 3-11
- “Limitations and Tips” on page 3-17

## Import AUTOSAR Software Component

### In this section...

“AUTOSAR arxml Importer Tool” on page 3-2

“General arxml Import Workflow” on page 3-3

“Import Multi-Runnable Software Component” on page 3-5

“Merge AUTOSAR Authoring Tool Changes Into Model” on page 3-6

### AUTOSAR arxml Importer Tool

The AUTOSAR arxml importer tool parse AUTOSAR software component description files produced, for example, by an AUTOSAR authoring tool (AAT) and imports software component information into a Simulink model for further configuration and model-based design.

The tool imports a subset of the elements and objects from arxml files representing an AUTOSAR software component. The subset consists of AUTOSAR elements relevant for Simulink model-based design of an automotive application, for example, AUTOSAR Components, Ports, Interfaces, DataTypes, and Packages.

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 3-10.
- Provides the ability to merge changes found in imported arxml files into the associated Simulink model. For more information, see “Merge AUTOSAR Authoring Tool Changes Into Model” on page 3-6.

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.

## 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'})
```

In the Command Window, you see reports describing the software component content of the specified XML file or files.

```
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')
```

To import Simulink data objects for AUTOSAR data into an existing Simulink data dictionary, you can set the `DataDictionary` property on the model creation. For example:

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

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 3-5 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 an existing Simulink data dictionary, you can set the `DataDictionary` property on the calibration objects creation. For example:

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

- `arxml.importer.updateModel` — Merges `arxml` code changes into the associated mapped Simulink model.

For example:

```
open_system('mySWC')  
updateModel(obj, 'mySWC')
```

For more information on updating a model with `arxml` code changes, see “Merge AUTOSAR Authoring Tool Changes Into Model” on page 3-6.

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 configure AUTOSAR code generation options and XML export options, see:

- “Configure the AUTOSAR Interface” on page 4-3
- “Export AUTOSAR Component XML and C Code” on page 5-2
- “Configure and Map AUTOSAR Component Programmatically” on page 4-188

## 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', '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.

---

## Merge AUTOSAR Authoring Tool Changes Into Model

To support the round trip of AUTOSAR components between an AUTOSAR authoring tool (AAT) and the Simulink design environment, the arxml importer can merge arxml code changes into a model.

Given a model into which you have imported arxml files or from which you have exported arxml code, suppose that changes are made to the arxml information in an AAT. Using the method `arxml.importer.updateModel`, you can import the changed arxml information and request that the changes be merged into the model. The update/merge generates a report that details the updates applied to the model, and required changes that were not made automatically.

To merge AUTOSAR authoring tool changes into a Simulink model:

- 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\\_tic\\_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

- Add** Outport [mySWC/Runnable2/IRV6\\_write](#) corresponding to InterRunnableVariable IRV6 to [mySWC/Runnable2](#)
- Add** Outport [mySWC/Runnable2/IRV5\\_write](#) corresponding to InterRunnableVariable IRV5 to [mySWC/Runnable2](#)
- Add** a fcn-call subsystem corresponding to runnable [Runnable4](#) to the model
- Add** a fcn-call inport to runnable [Runnable4](#)

### 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. For example, you can verify the reported changes and perform the required manual changes.

Required manual changes can include additions and deletions of Simulink elements, such as inports, outports, or subsystems. After you add Simulink elements, map them to the corresponding AUTOSAR elements:

- a Open the Configure AUTOSAR Interface dialog box, for example, by selecting **Code > C/C++ Code > Configure Model as AUTOSAR Component**.

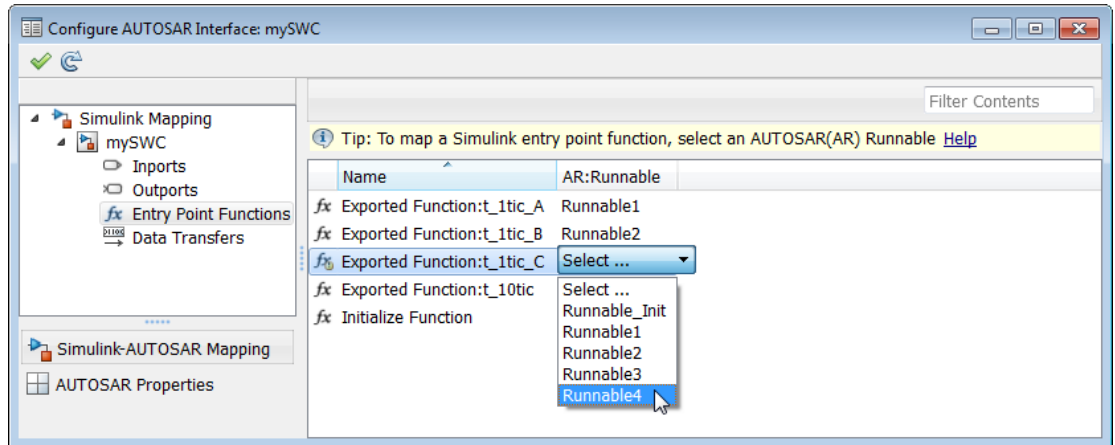
---

**Note:** If you added data transfer lines, function callers, or numeric types, click the **Synchronize** icon . This action loads or updates Simulink data transfers, function callers, and numeric types in your model.

---

- b Select **Simulink-AUTOSAR Mapping**.
- c Use the Simulink-AUTOSAR Mapping Explorer to map the new Simulink elements to the corresponding AUTOSAR elements. For example, in the Inports view, map each new inport to an AUTOSAR port and data element. In the Entry Point Functions view, map each new top-level subsystem to an AUTOSAR runnable.

For example, suppose that you follow the directions in the mySWC update report displayed in the previous step. You manually add a function-call subsystem corresponding to runnable `Runnable4` to the top level of the model. You add a function-call inport `t_ttic_C`, with block option **Output function call** selected, to the new subsystem. In the Entry Point Functions view, you map the new function-call subsystem to AUTOSAR runnable `Runnable4`.



- d After you make manual changes, further configuration could be required to pass validation. Repeat the merge procedure, rerunning the `updateModel` method with the same `arxml` files, to see if more manual model changes are required.

## 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.

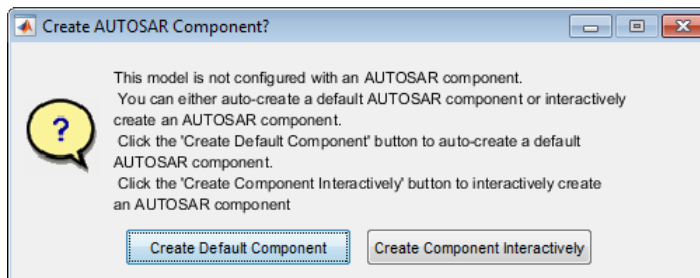
- When `arxml` files created by an AAT are imported into a Simulink model, AUTOSAR element information is preserved, including UUIDs (for Identifiables), properties, references, and packages.
- After import, you can view and edit AUTOSAR objects in a Simulink model window without losing the information 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

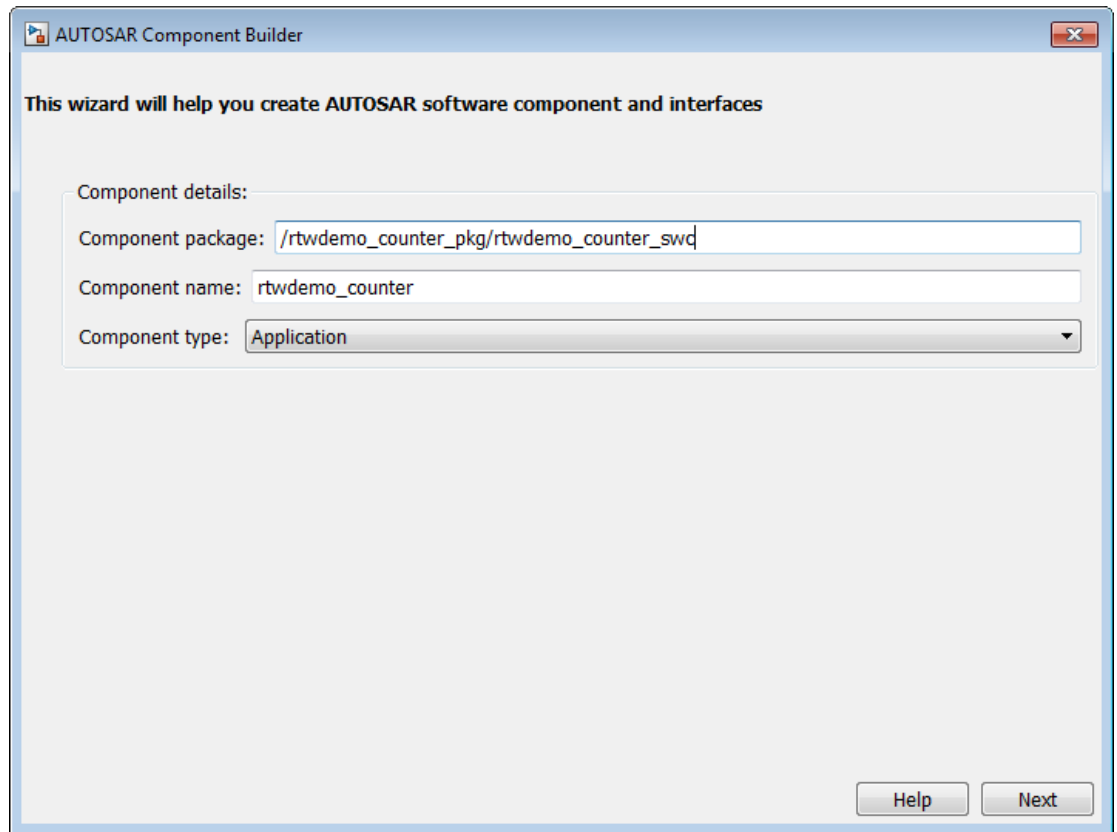
As an alternative to importing an AUTOSAR software component from an AUTOSAR authoring tool (AAT), you can create an AUTOSAR software component in Simulink.

- 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.



- 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 “Configure the AUTOSAR Interface” 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.

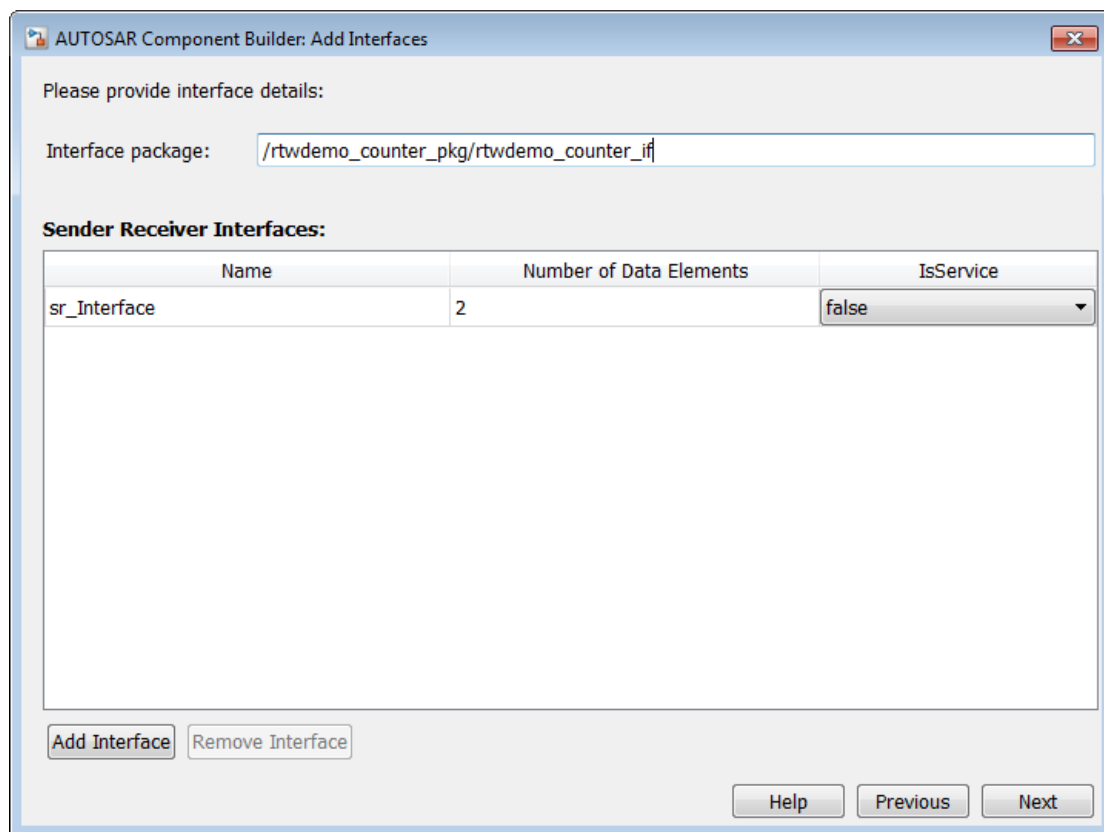
- AUTOSAR component type: **Application** for an AUTOSAR application software component, or **Sensor Accuator** for an AUTOSAR sensor/actuator software component.



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

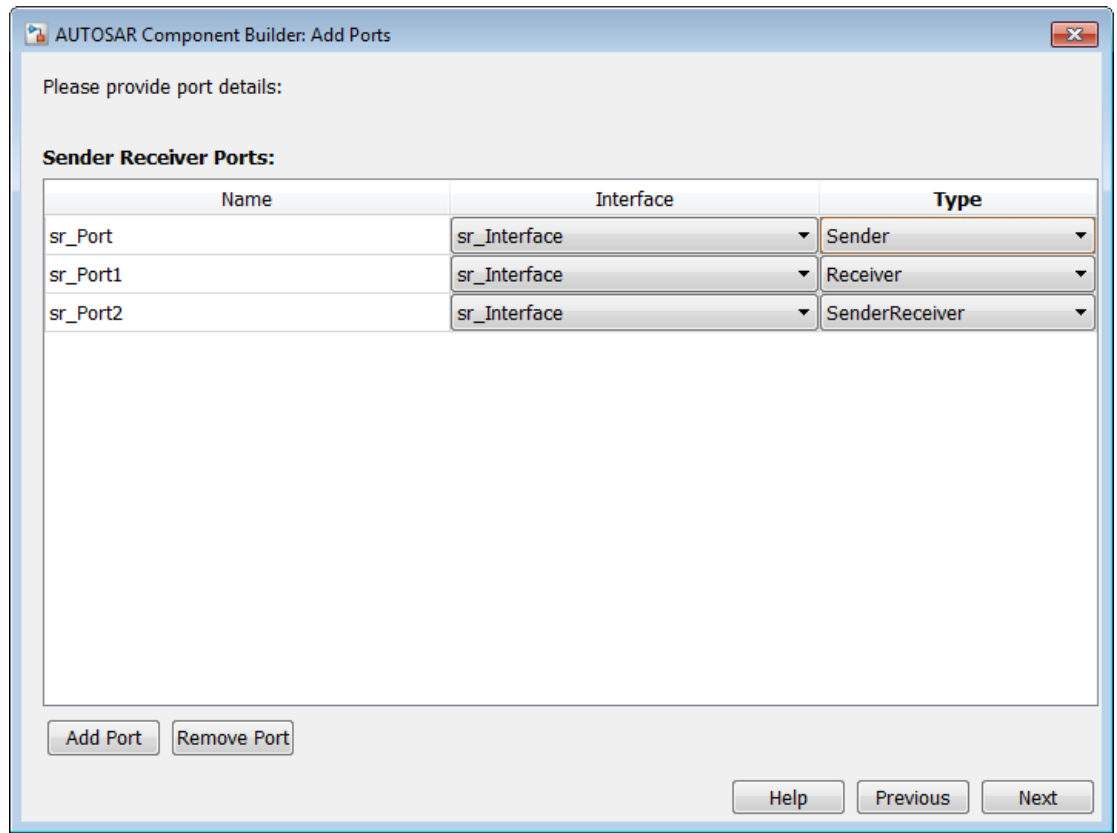
- 7 In the Add Sender Receiver Interfaces view of the AUTOSAR Component Builder dialog box, you can:
  - Modify the name of the Interface package.
  - Click **Add Interface** to add more interfaces to the displayed list.
  - Click **Remove Interface** to remove a selected interface.

- 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.

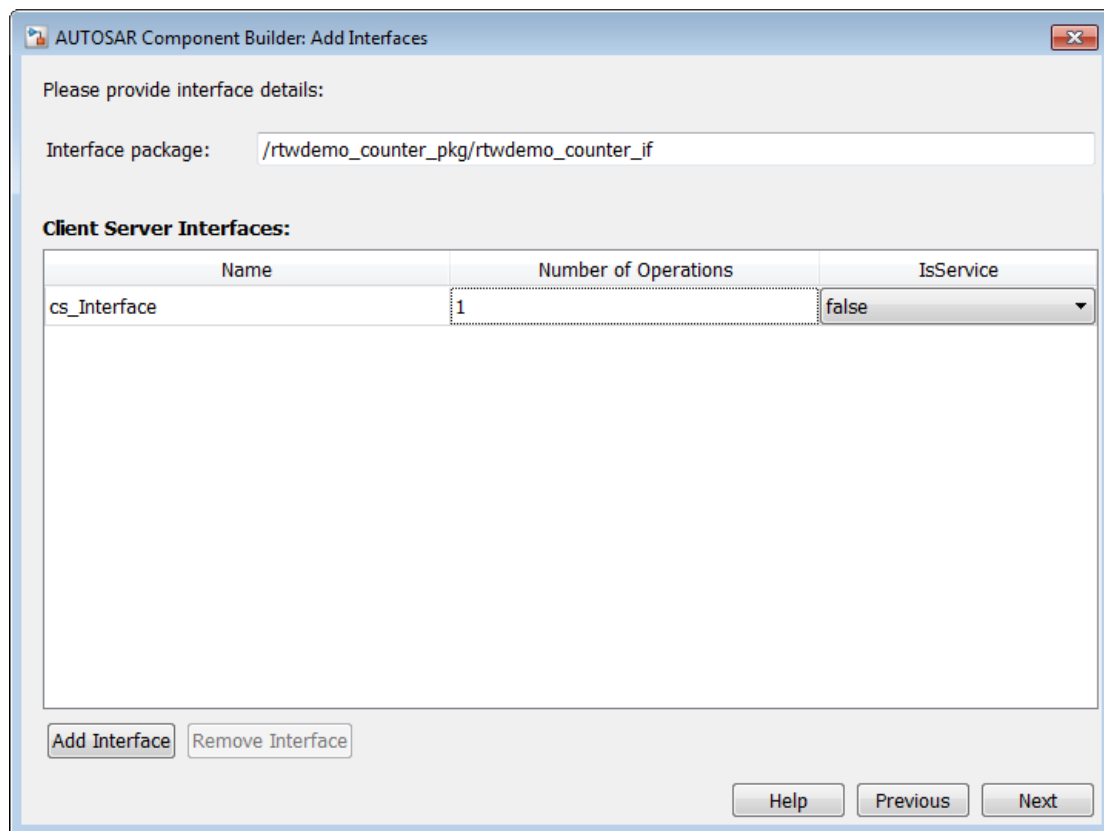
- 8** In the Add Sender Receiver Ports view of the AUTOSAR Component Builder dialog box, you can:
  - Click **Add Port** to add more sender, receiver, or sender-receiver ports to the displayed list.
  - Click **Remove Port** to remove a selected port.
  - For each listed port, edit the name, select the associated S-R interface, and select whether the port type is **Sender**, **Receiver**, or **SenderReceiver**.



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.

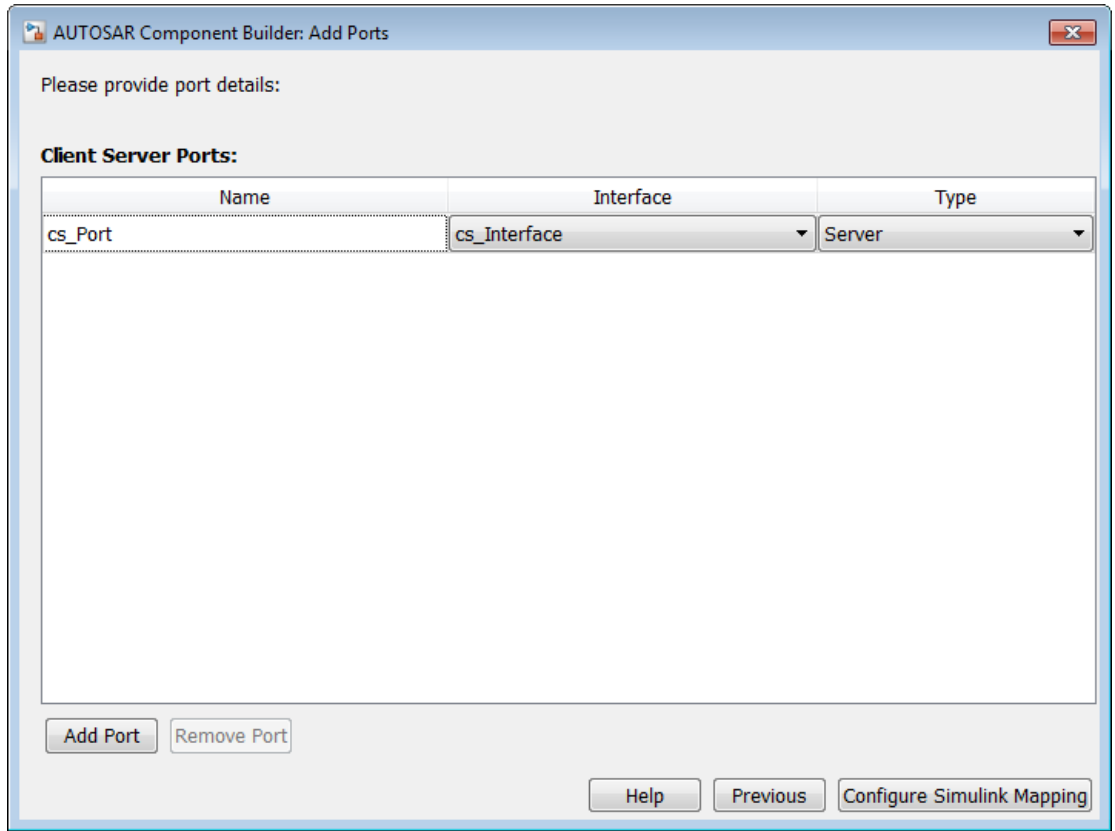




Click **Next** to go to the Add Client Server Ports view.

**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 “Configure the AUTOSAR Interface” on page 4-3.

## Limitations and Tips

### 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<sup>®</sup> objects that compose the `arxml.importer` object.



# AUTOSAR Component Development

---

- “Configure the AUTOSAR Interface” on page 4-3
- “Configure AUTOSAR Packages” on page 4-54
- “Configure AUTOSAR Provide-Require Port” on page 4-66
- “Configure AUTOSAR Port aliveTimeout Value” on page 4-70
- “Configure AUTOSAR Receiver Port for IsUpdated Service” on page 4-71
- “Configure AUTOSAR Sender Port for Data Element Invalidation” on page 4-74
- “Configure AUTOSAR S-R Interface Port for End-To-End Protection” on page 4-78
- “Configure AUTOSAR Receiver Port for DataReceiveErrorEvent” on page 4-81
- “Configure AUTOSAR Mode Receiver Ports and Mode-Switch Events” on page 4-85
- “Configure AUTOSAR Client-Server Communication” on page 4-92
- “Configure AUTOSAR Calibration Parameters” on page 4-123
- “Configure AUTOSAR Calibration Component” on page 4-125
- “Configure COM\_AXIS Lookup Table for Measurement and Calibration” on page 4-129
- “Configure AUTOSAR Data for Measurement and Calibration” on page 4-135
- “Configure AUTOSAR Display Format for Measurement and Calibration” on page 4-143
- “Configure AUTOSAR Release 4.x Data Types” on page 4-148
- “Configure AUTOSAR CompuMethods” on page 4-152
- “Configure AUTOSAR Internal Data Constraints Export” on page 4-162
- “Configure AUTOSAR Multiple Runnables” on page 4-165
- “Configure AUTOSAR Initialization Runnable” on page 4-175
- “Configure Disabled Mode for AUTOSAR Runnable Event” on page 4-178
- “Configure AUTOSAR Per-Instance Memory” on page 4-179
- “Configure AUTOSAR Static or Constant Memory” on page 4-182

- “Configure AUTOSAR Variation Point Proxies” on page 4-185
- “Configure and Map AUTOSAR Component Programmatically” on page 4-188
- “Limitations and Tips” on page 4-197

# Configure the AUTOSAR Interface

## In this section...

“AUTOSAR Interface Configuration” on page 4-3

“Map Model Elements Using Simulink-AUTOSAR Mapping Explorer” on page 4-5

“Configure AUTOSAR Component Using AUTOSAR Properties Explorer” on page 4-10

“Configure AUTOSAR Package for Interface or CompuMethod” on page 4-51

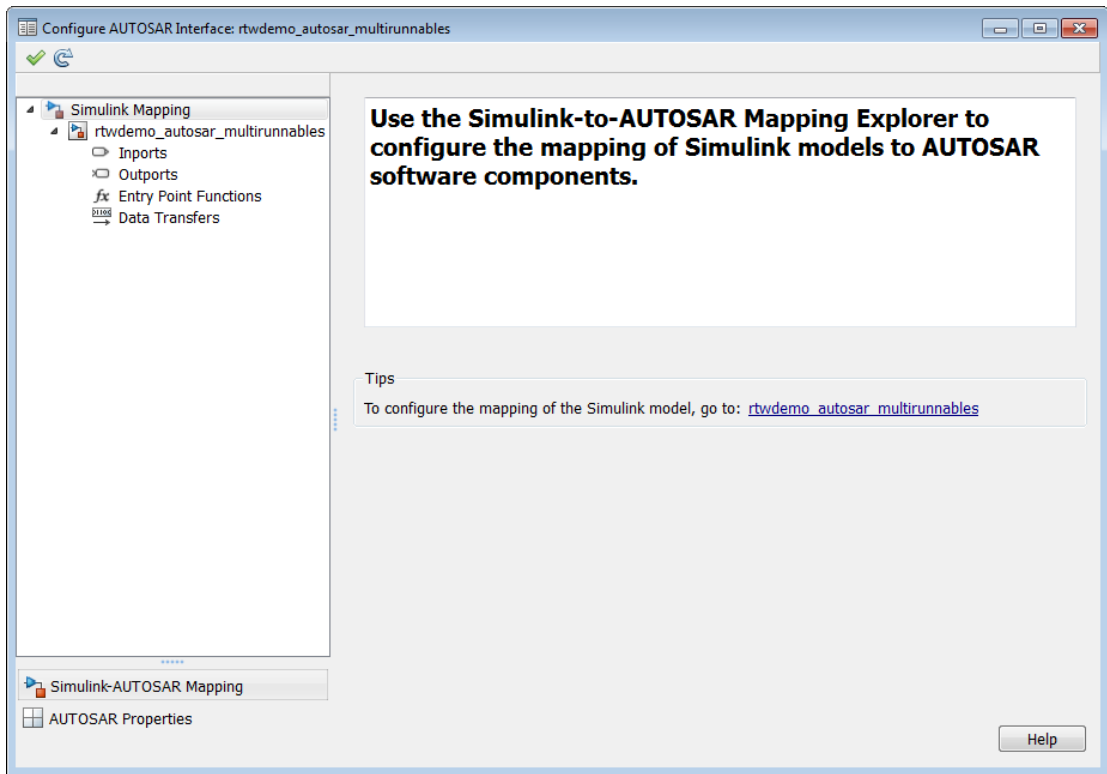
## AUTOSAR Interface Configuration

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, and data transfers 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, computational 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 4-188.

In a model for which the AUTOSAR code generation target (`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 load or update Simulink data transfers, function callers, and numeric types in your model.
- 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.

---

For more information on the Simulink-AUTOSAR Mapping Explorer, see “Map Model Elements Using Simulink-AUTOSAR Mapping Explorer” on page 4-5. For more information on the AUTOSAR Properties Explorer, see “Configure AUTOSAR Component Using AUTOSAR Properties Explorer” on page 4-10.

## Map Model Elements Using Simulink-AUTOSAR Mapping Explorer

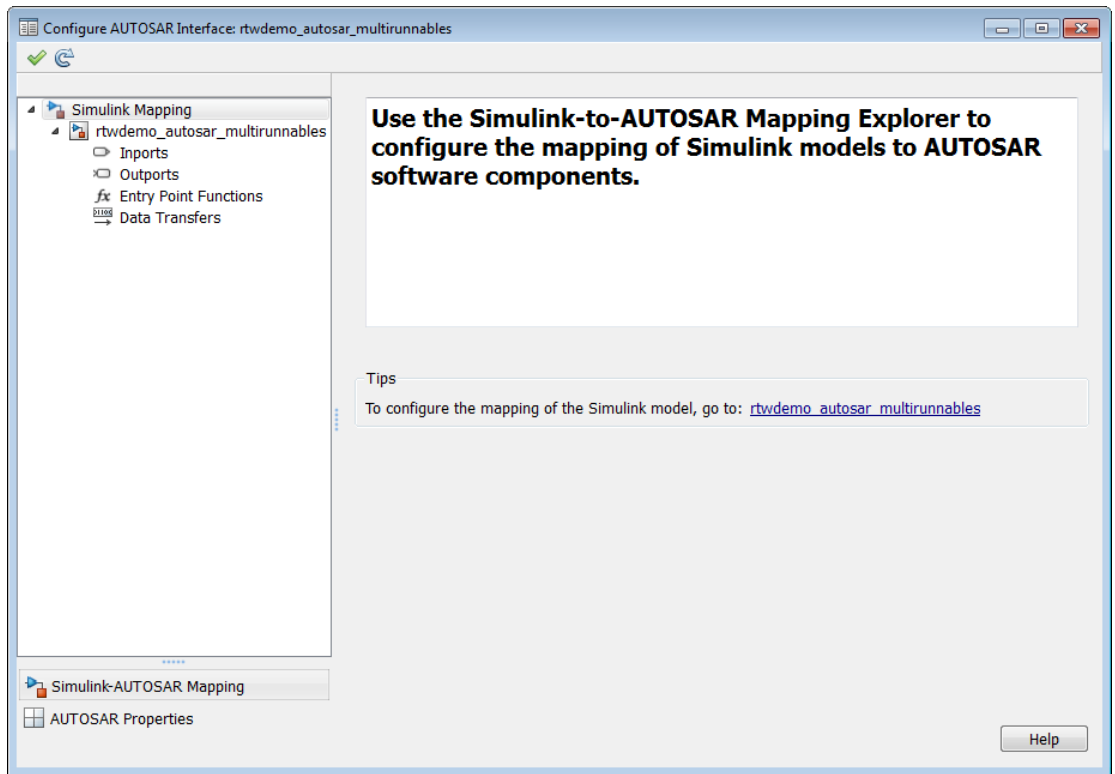
To map Simulink model elements to AUTOSAR software component elements:

- 1 Open a model for which the AUTOSAR code generation target (`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).

---

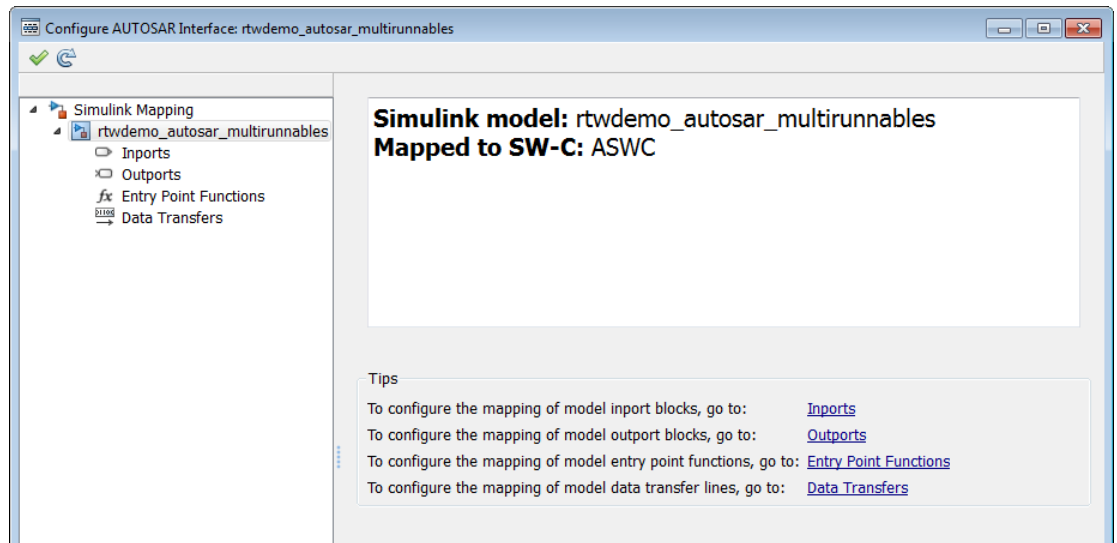
**Note:** Additionally, if the model contains Function Caller blocks in a client/server interface configuration, **Function Callers** appears in the model tree. A Simulink function caller can be mapped to an AUTOSAR client port and an AUTOSAR operation.

---



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

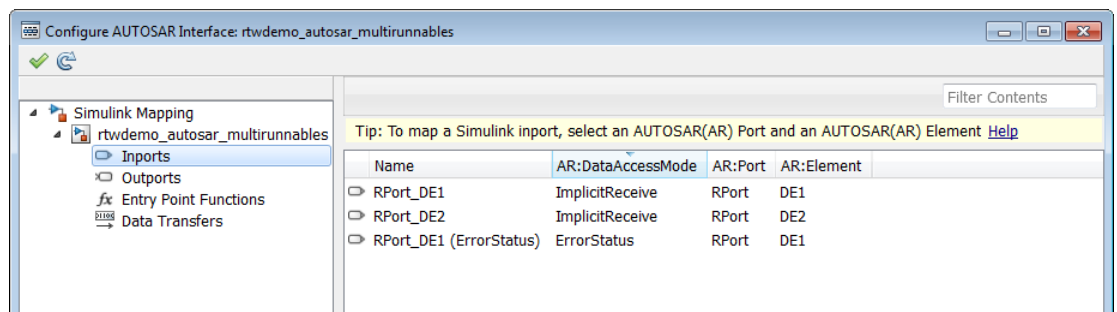
The model view of the Simulink-AUTOSAR Mapping Explorer displays the model name and the name of the AUTOSAR component to which the model is being mapped.



- 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:

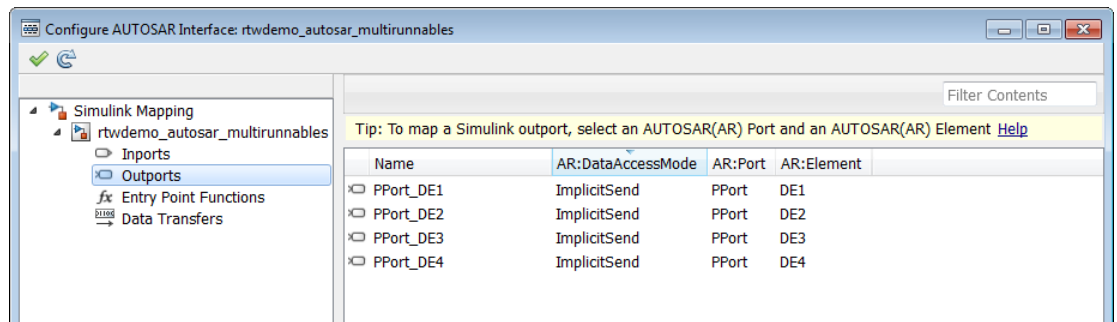
- Map a Simulink inport by selecting the inport and then selecting menu values for an AUTOSAR Port and an AUTOSAR Element.
- Select a port and then select a menu value for its AUTOSAR port data access mode: `ImplicitReceive`, `ExplicitReceive`, `QueuedExplicitReceive`, `ErrorStatus`, `ModeReceive`, `IsUpdated`,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



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

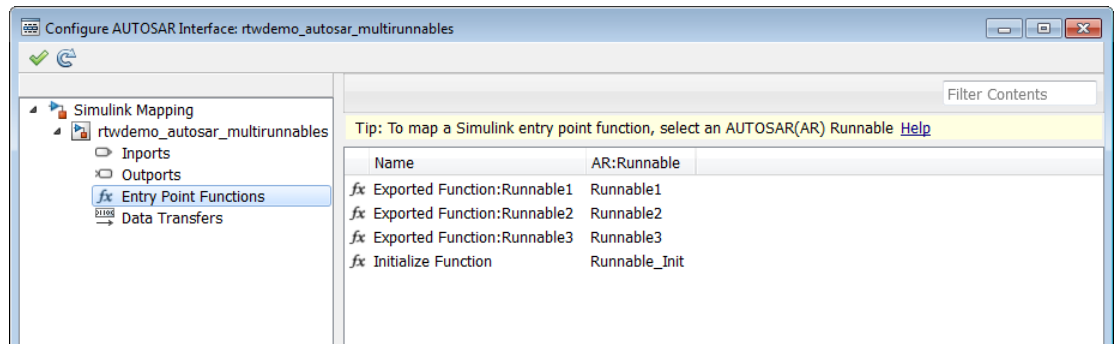
The Outports view of the Simulink-AUTOSAR Mapping Explorer maps each Simulink outport to a data element of an AUTOSAR port. You can:

- 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`, or `EndToEndWrite`.



- 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. 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**.

---

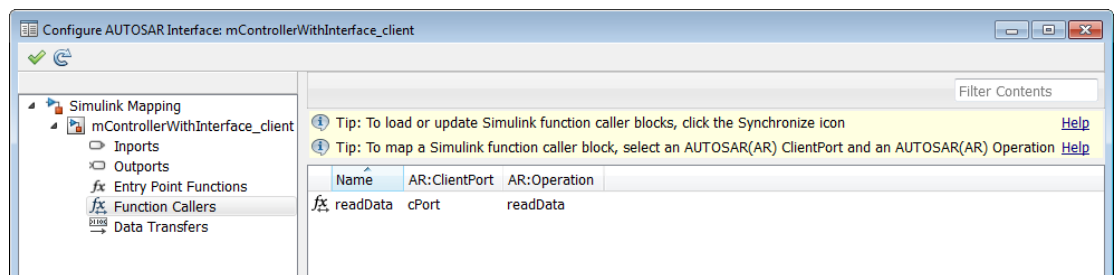
**Note: Function Callers** is available for selection only if the model contains Function Caller blocks in a client/server interface configuration. If **Function Callers** is absent, skip this step.

---


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

First, click the **Synchronize** icon  to load or update Simulink function callers in your model.

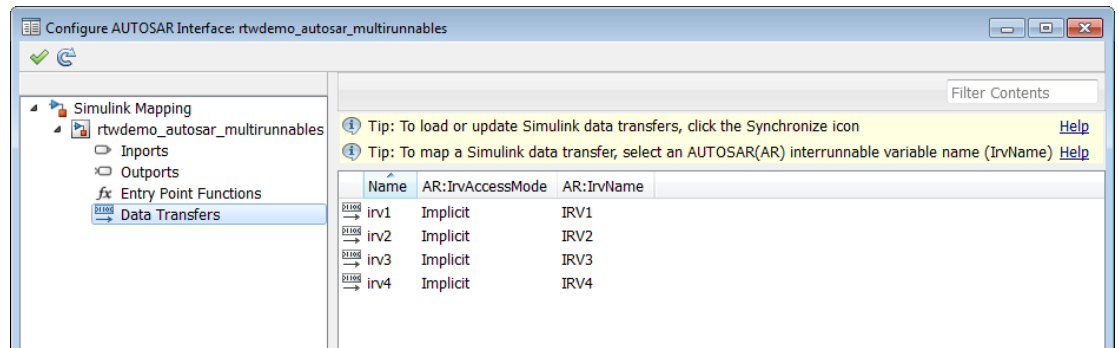
You can map a Simulink function caller by selecting the function name and then selecting menu values for a client port and an operation, among those listed for the AUTOSAR component.




- 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). First, 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.

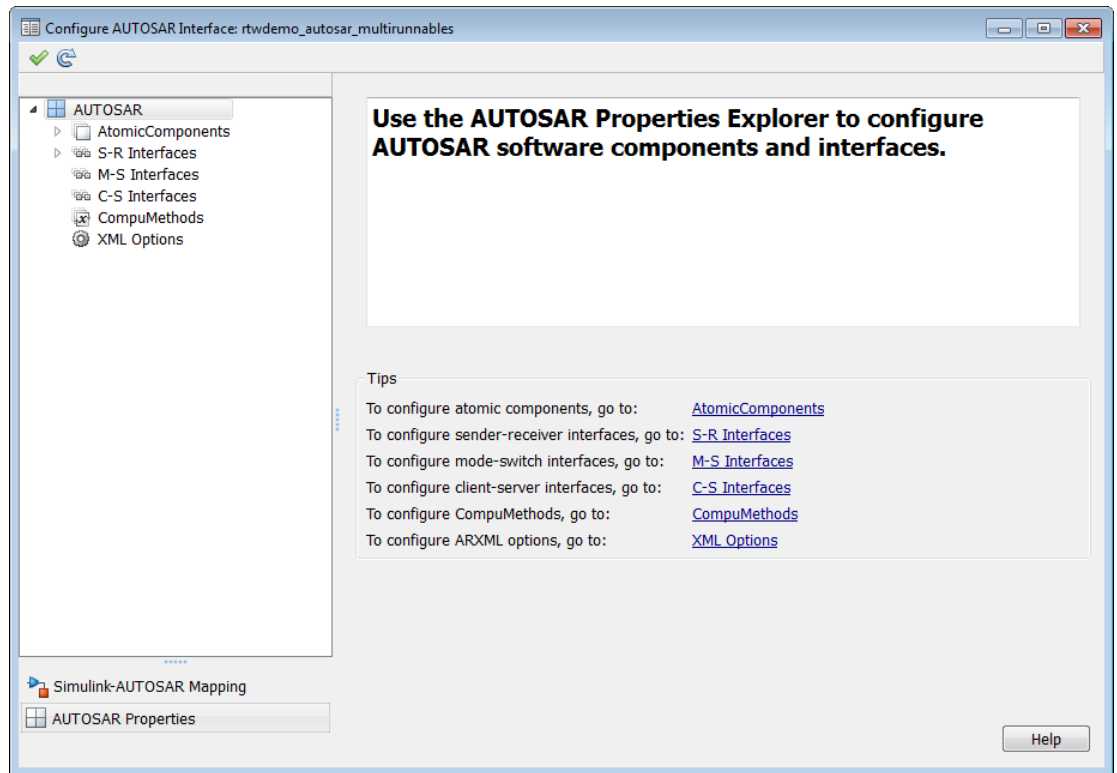


- 9 Click the **Validate** icon  to validate the AUTOSAR interface configuration. If errors are reported, address them and then retry validation.

## Configure AUTOSAR Component Using AUTOSAR Properties Explorer

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

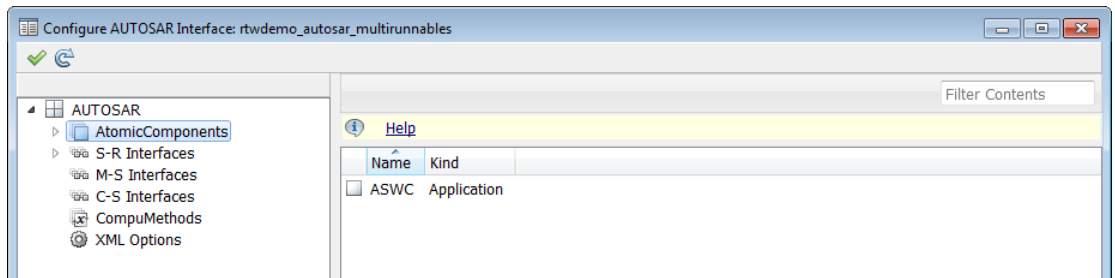
- 1 Open a model for which the AUTOSAR code generation target (`autosar.tlc`) has been selected.
- 2 Open the Configure AUTOSAR Interface dialog box by selecting **Code > C/C++ Code > Configure Model as AUTOSAR Component**. If the AUTOSAR Properties Explorer is not already selected, click **AUTOSAR Properties**.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



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

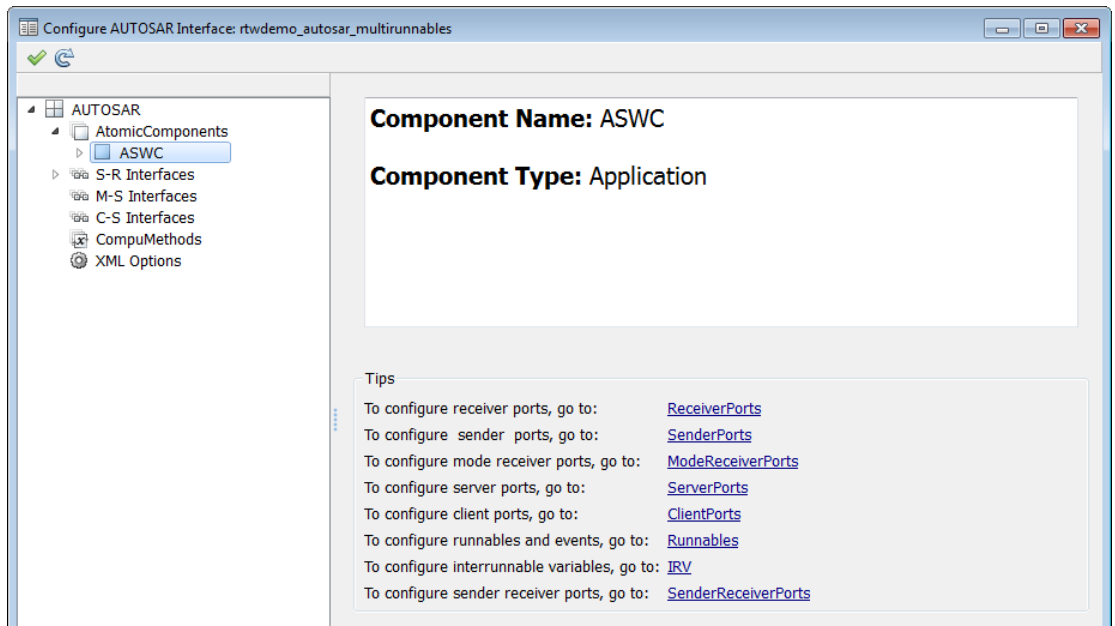
The Atomic Components view of the AUTOSAR Properties Explorer displays atomic components and their types. You can:

- 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 string.



- 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 — receiver ports, sender ports, sender-receiver ports, mode receiver ports, server ports, client ports, runnables and events, and inter-runnable variables.

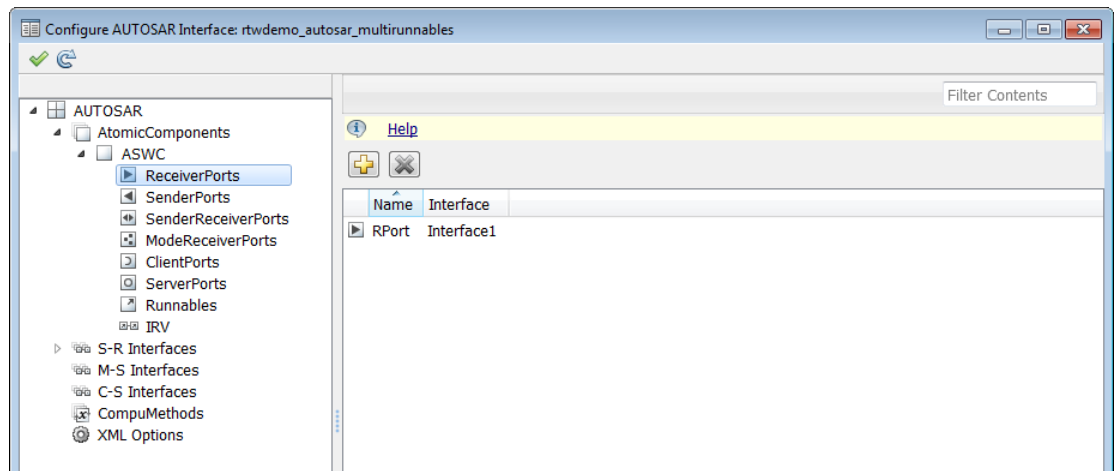




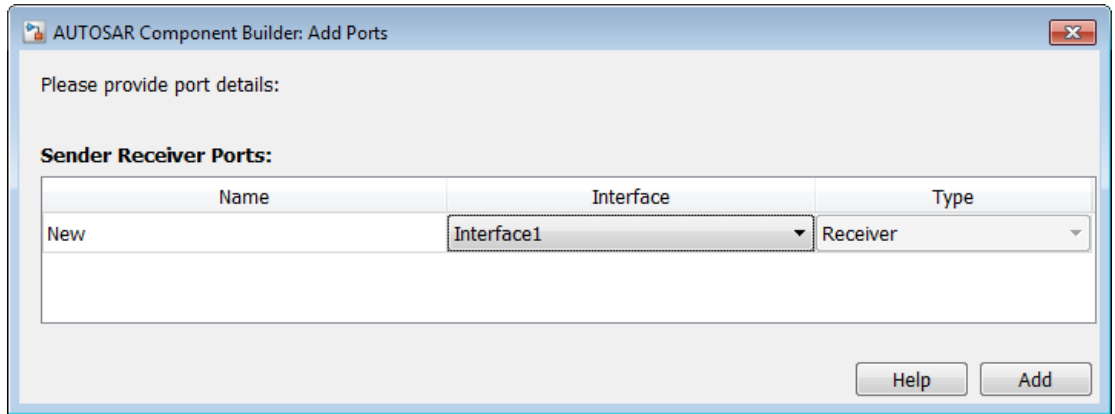
- 5 In the left-hand pane of the Configure AUTOSAR Interface dialog box, expand the component and select **ReceiverPorts**.

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 string.
- 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.





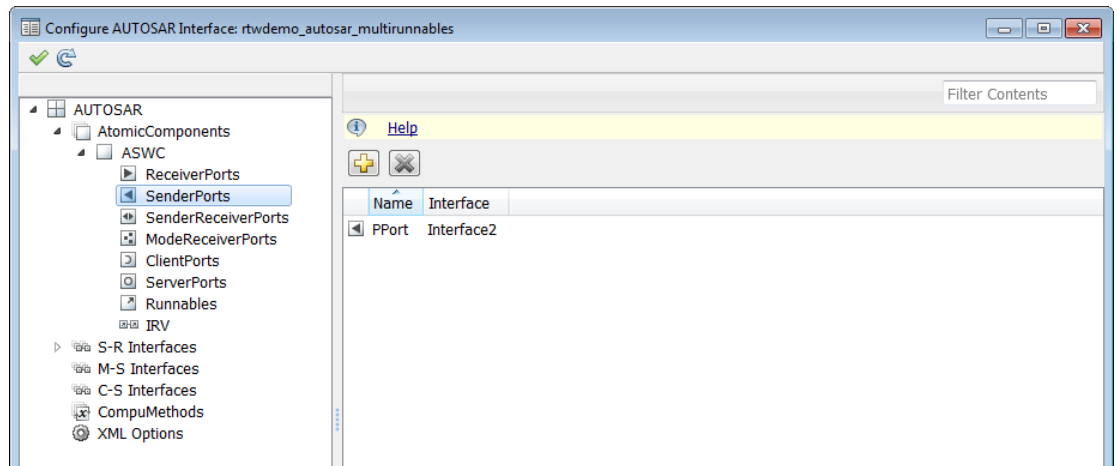
The Add Ports dialog box lets you add a receiver port and associate it with an existing S-R interface. Click **Add** to add the port and return to the Receiver Ports view.



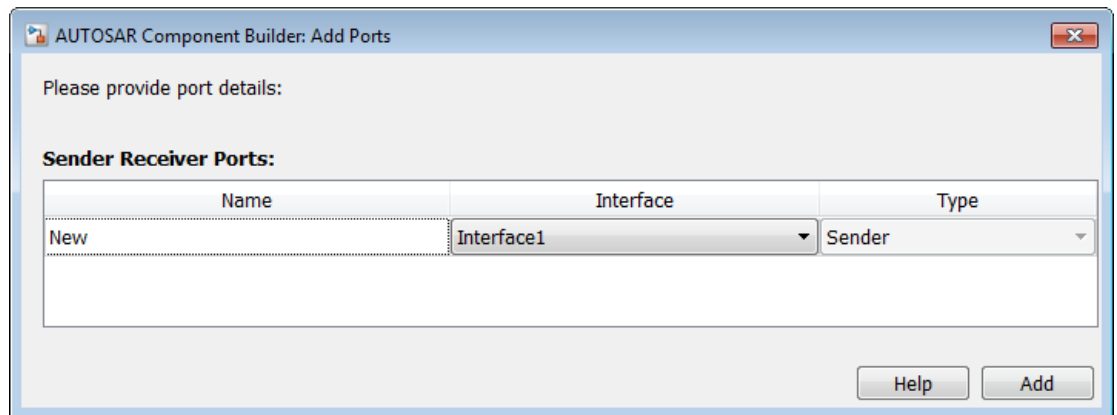
- 6 In the left-hand pane of the Configure AUTOSAR Interface dialog box, select **SenderPorts**.

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 string.
- 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**.

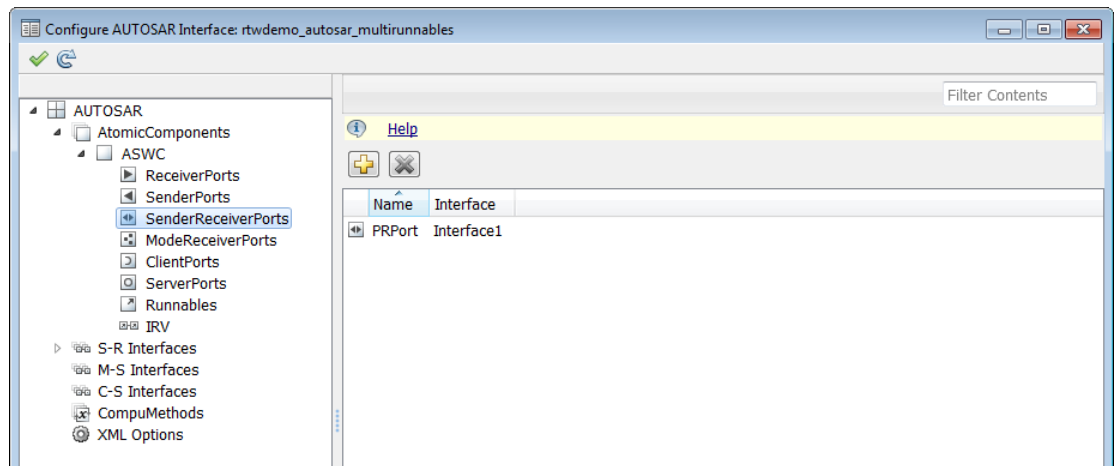
The Sender-Receiver Ports view of the AUTOSAR Properties Explorer lists sender-receiver ports and their properties. You can:

- 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 string.
- 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.

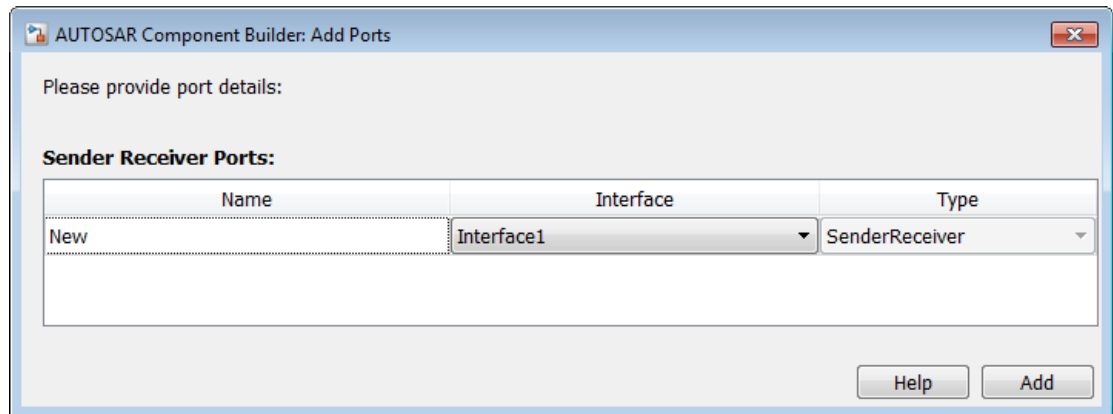
---

**Note:** AUTOSAR sender-receiver ports require AUTOSAR schema version 4.1. To select a schema version for the model, go to **AUTOSAR Code Generation Options** in the Configuration Parameters dialog box.

---





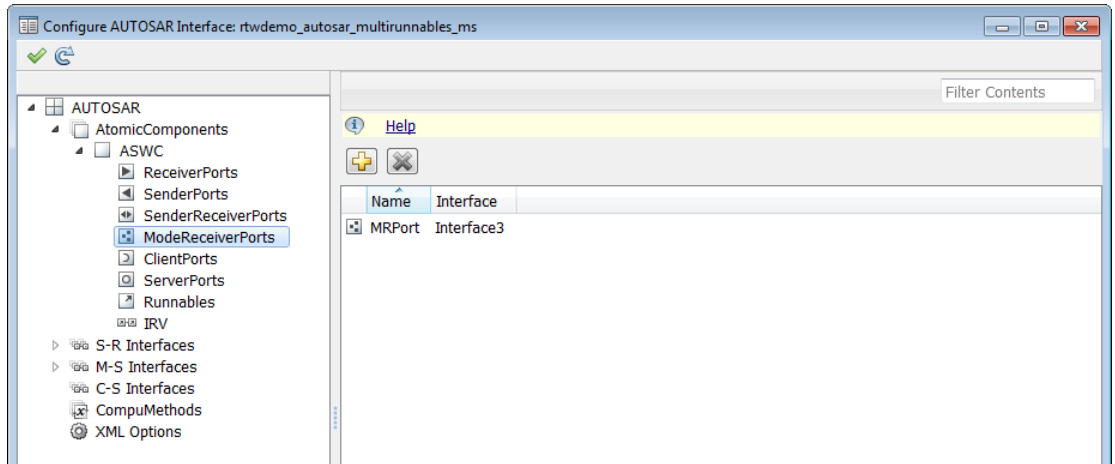
The Add Ports dialog box lets you add a sender-receiver port and associate it with an existing S-R interface. Click **Add** to add the port and return to the Sender-Receiver Ports view.



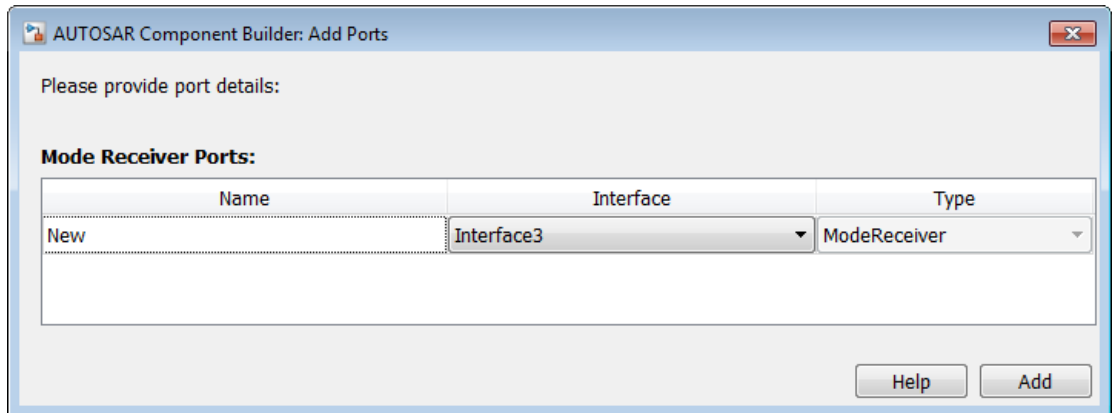
- 8 In the left-hand pane of the Configure AUTOSAR Interface dialog box, select **ModeReceiverPorts**.

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 string.
- 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.

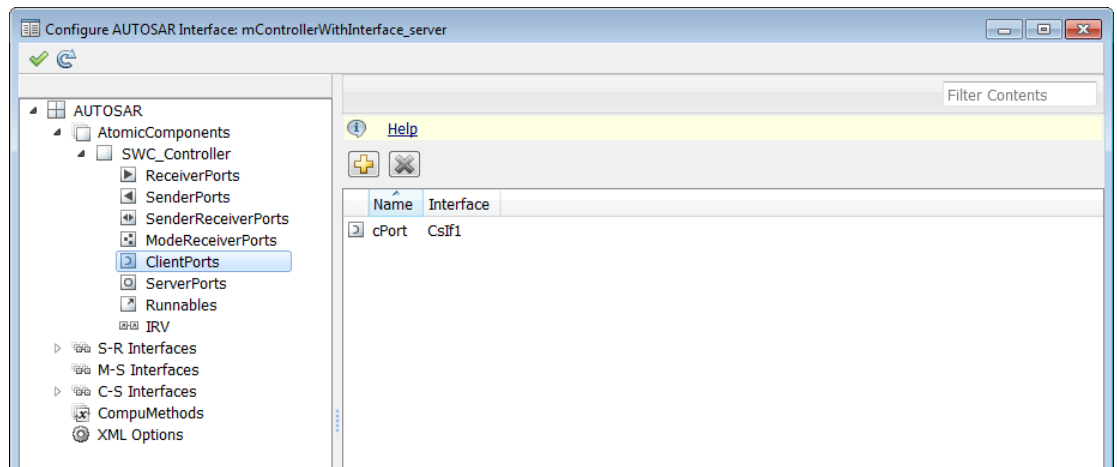


For more information about adding and configuring mode-receiver ports, see “Configure AUTOSAR Mode Receiver Ports and Mode-Switch Events” on page 4-85.

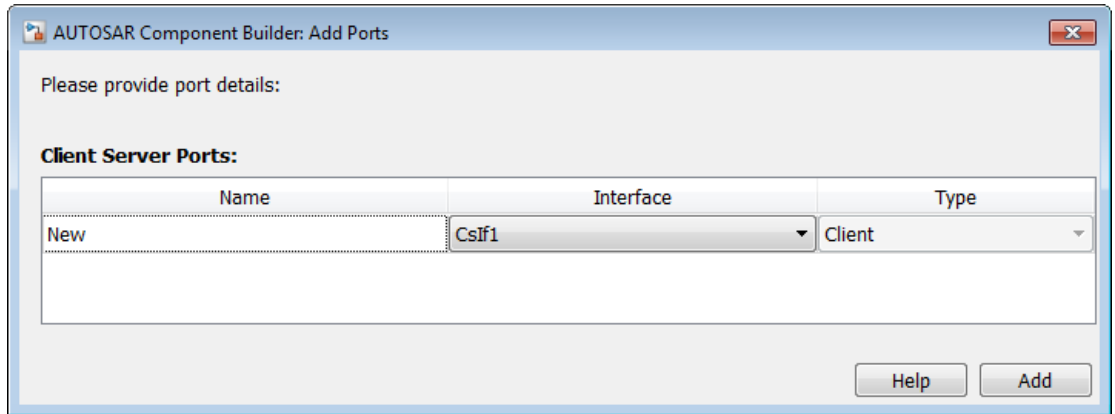
- 9 In the left-hand pane of the Configure AUTOSAR Interface dialog box, select **ClientPorts**.

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 string.
- 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.

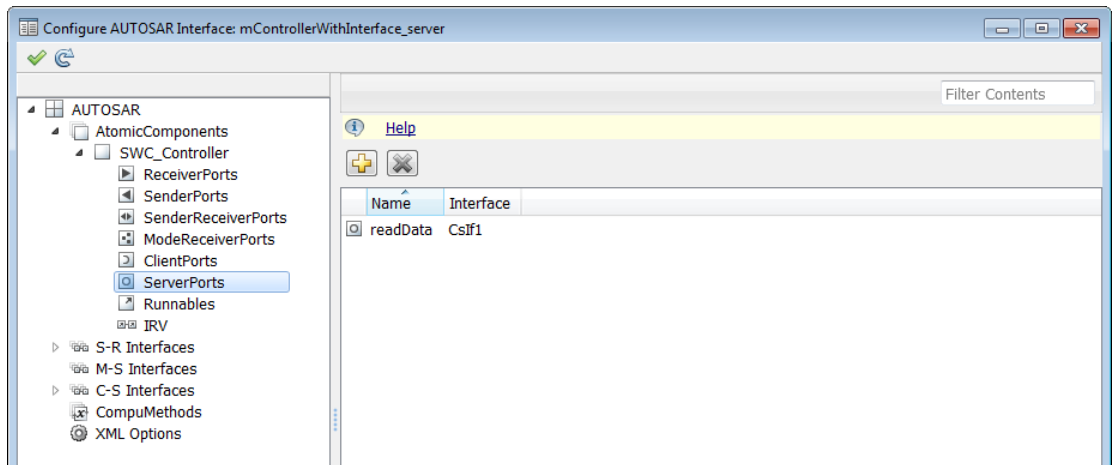


- 10** In the left-hand pane of the Configure AUTOSAR Interface dialog box, select **ServerPorts**.

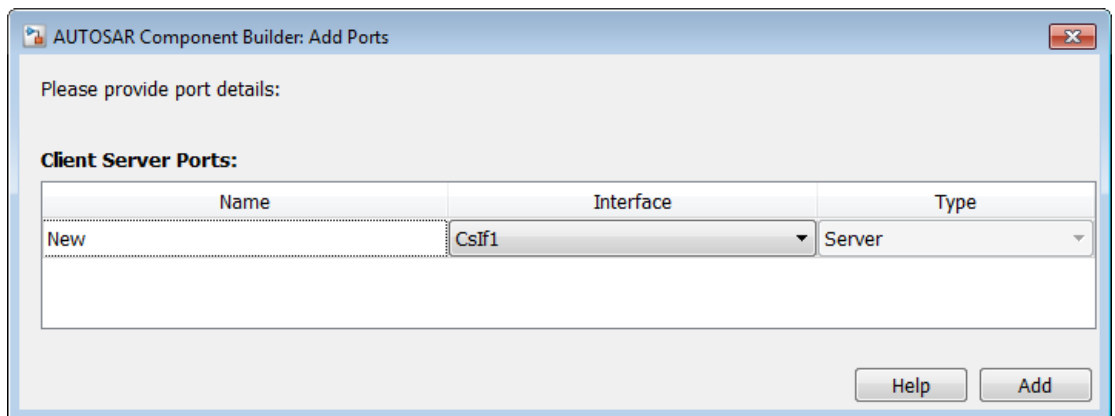
The Server Ports view of the AUTOSAR Properties Explorer lists client ports and their properties. You can

- 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 string. 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.







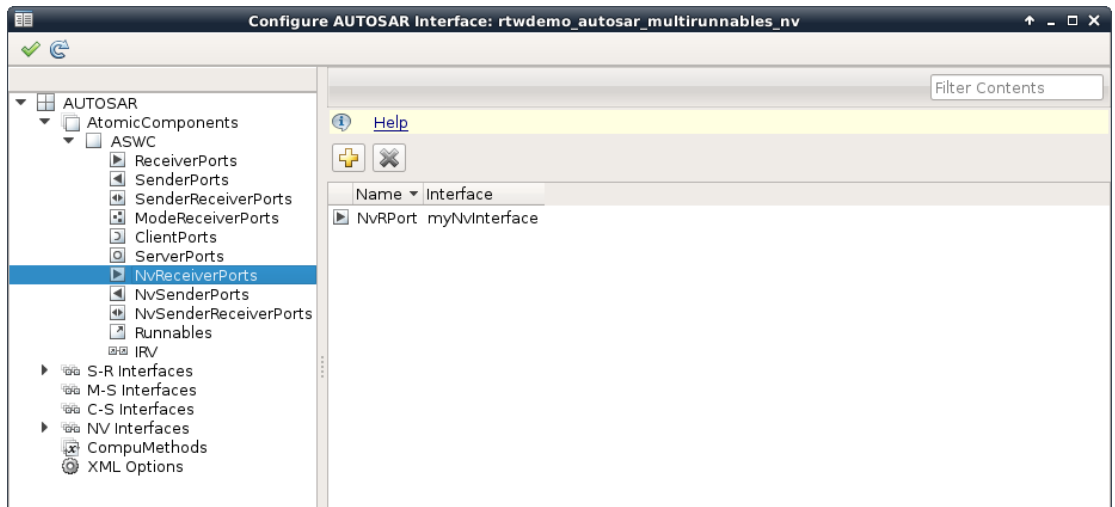
The Add Ports dialog box lets you add a server port and associate it with an existing C-S interface. Set **Name** to the name of the associated server function. If a C-S interface does not exist in the component, you must create one before adding the port. Click **Add** to add the port and return to the Server Ports view.



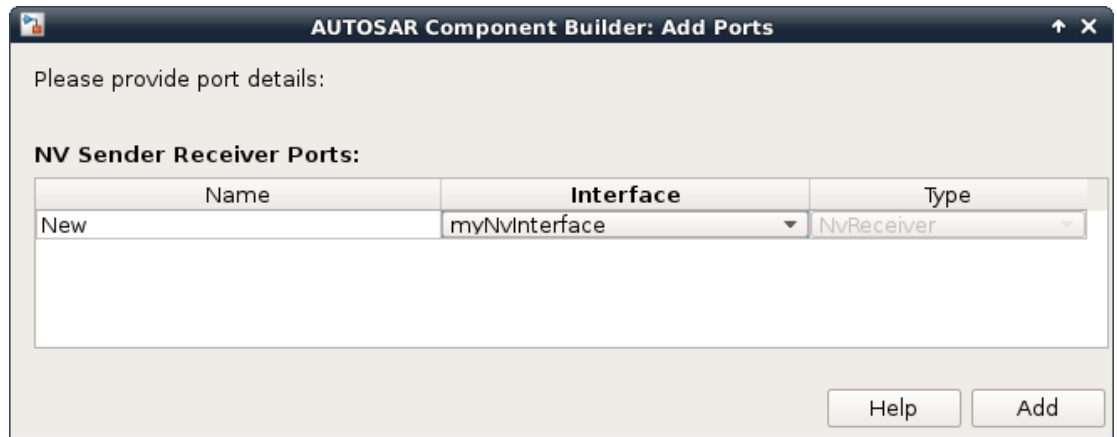
- In the left-hand pane of the Configure AUTOSAR Interface dialog box, expand the component and select **NvReceiverPorts**.

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 string.
- 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.





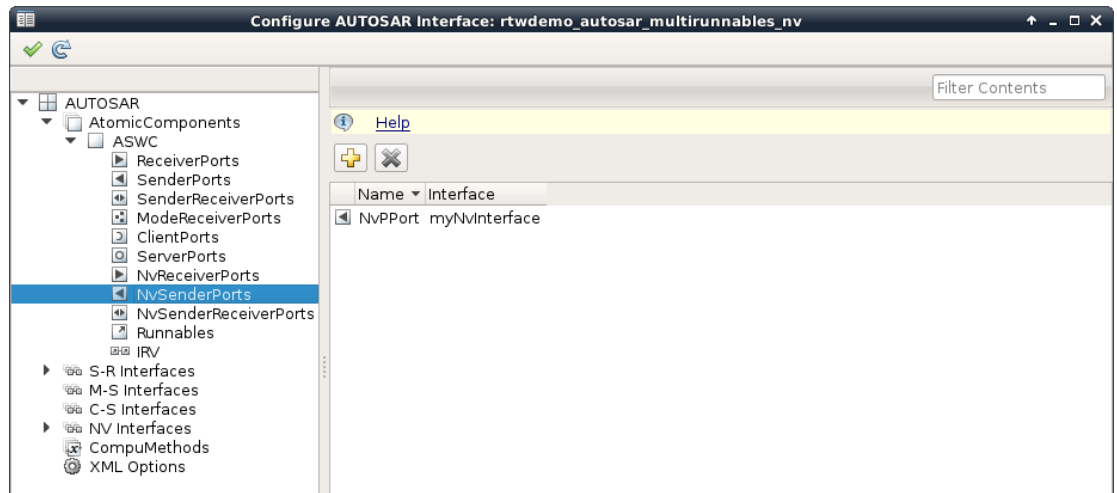
The Add Ports dialog box lets you add an NV receiver port and associate it with an existing NV interface. Click **Add** to add the port and return to the NV Receiver Ports view.



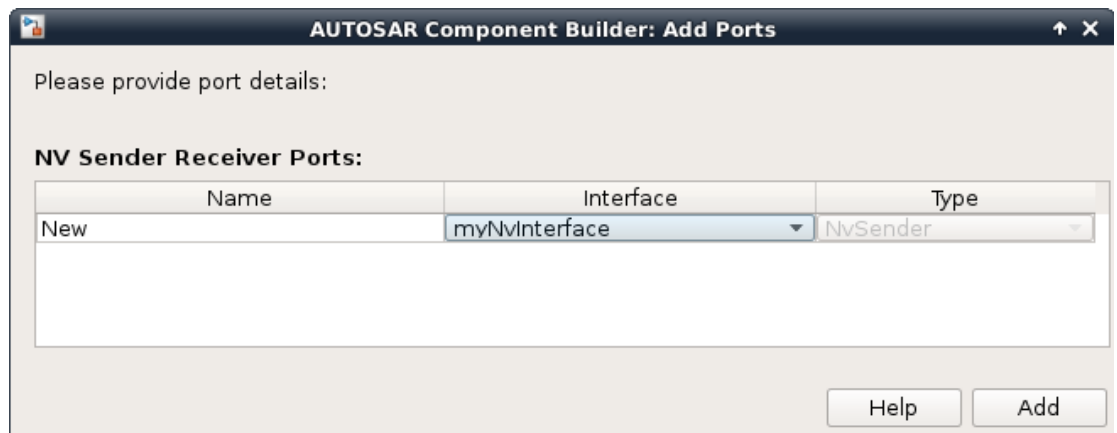
- 12** In the left-hand pane of the Configure AUTOSAR Interface dialog box, select **NvSenderPorts**.

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 string.
- 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.



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

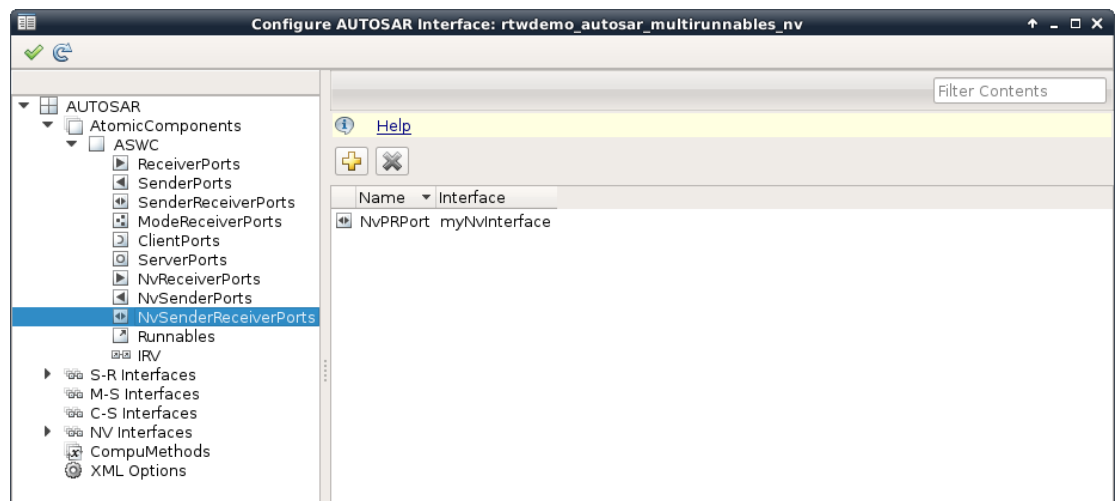
The NV Sender Receiver Ports view of the AUTOSAR Properties Explorer lists nonvolatile (NV) sender-receiver ports and their properties. You can:

- 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 string.
- 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.

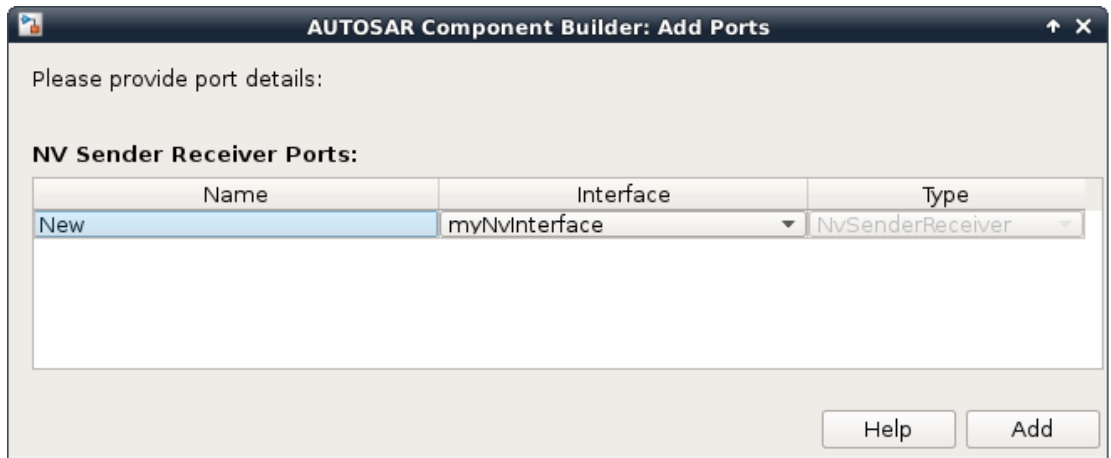
---

**Note:** AUTOSAR NV sender-receiver ports require AUTOSAR schema version 4.1 or later. 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.



- 14** 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 string.
- 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>
```

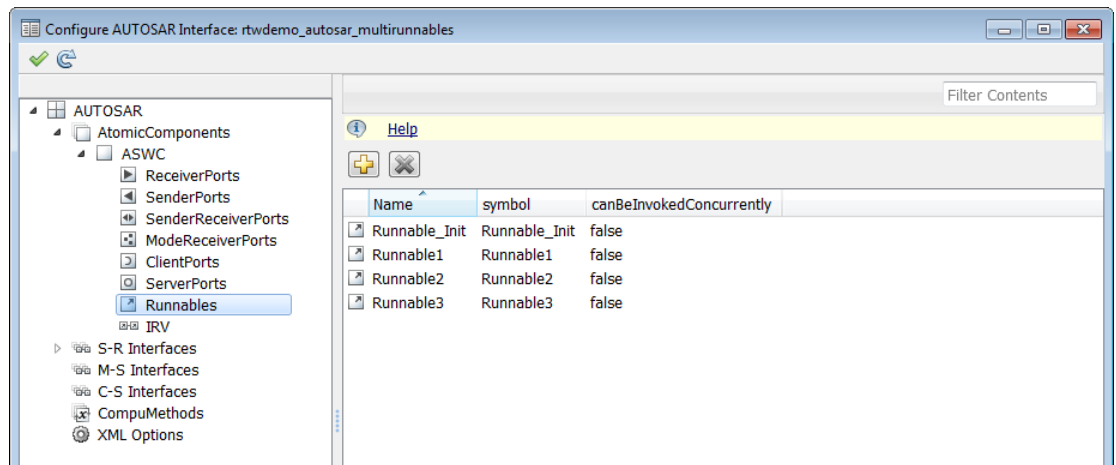
#### **rtwdemo\_autosar\_multirunnables.c**

```
/* 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 4-118.
- 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`, or `DataReceiveErrorEvent` — and name. You can rename an AUTOSAR event by clicking its name and then editing the name string. You can use the buttons **Add Event** and **Delete Event** to add or delete events from a runnable.

**Events**

Add Event Delete Event

Event Type	Event Name
TimingEvent	Event_t_1tic_A

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

**Events**

Add Event Delete Event

Event Type	Event Name
DataReceivedEvent	Event

Event Properties

Trigger RPort.DE1

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 Receiver Ports and Mode-Switch Events” on page 4-85.)



**Events**

Add Event Delete Event

Event Type	Event Name
ModeSwitchEvent	Event_Run

Event Properties

Mode Activation: OnTransition

Mode Receiver Port: MRPort

Transition From

Mode Declaration: Sleep

Transition To

Mode Declaration: Run

If you select an event of type `OperationInvokedEvent`, the runnable becomes an AUTOSAR server runnable. Select the event to display its **Trigger** property. Select a trigger for the event from the list of available trigger ports. The **Operation Signature** is displayed below the **Trigger** property. (For more information on using an `OperationInvokedEvent`, see “Configure AUTOSAR Client-Server Communication” on page 4-92.)

**Events**

Add Event Delete Event

Event Type	Event Name
OperationInvokedEvent	Event_readData

Event Properties

Trigger: readData.readData

If you select an event of type `InitEvent`, you can rename the event by clicking its name and then editing the name string. (For more information on using an `InitEvent`, see “Configure AUTOSAR Initialization Runnable” on page 4-175.)

The screenshot shows a dialog box titled "Events". At the top, there are two buttons: "Add Event" and "Delete Event". Below these is a table with two columns: "Event Type" and "Event Name". The "Event Type" column has a dropdown menu currently showing "InitEvent". The "Event Name" column contains the text "Event".



If you select an event of type `DataReceiveErrorEvent`, the runnable is activated by a `DataReceiveErrorEvent`. Select the event 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 4-81.)

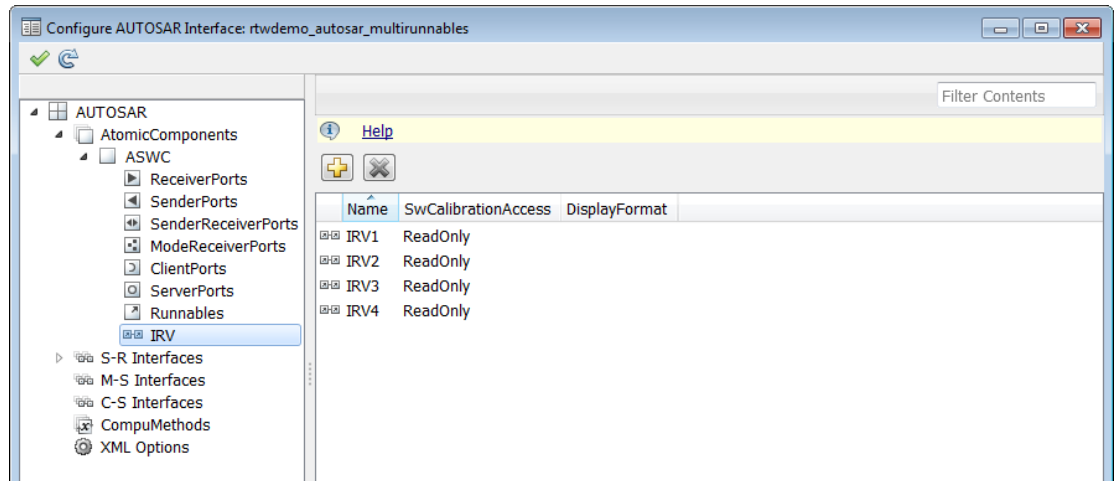
The screenshot shows the "Events" dialog box with the "Event Type" dropdown set to "DataReceiveErrorEvent" and the "Event Name" field containing "DRE\_Evt". Below the table, there is a section titled "Event Properties" with a "Trigger" dropdown menu set to "rPort.DE".

- 15 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 string.
- Specify the level of measurement and calibration tool access to an IRV, by setting 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<sup>®</sup> 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 AUTOSAR Display Format for Measurement and Calibration” on page 4-143.

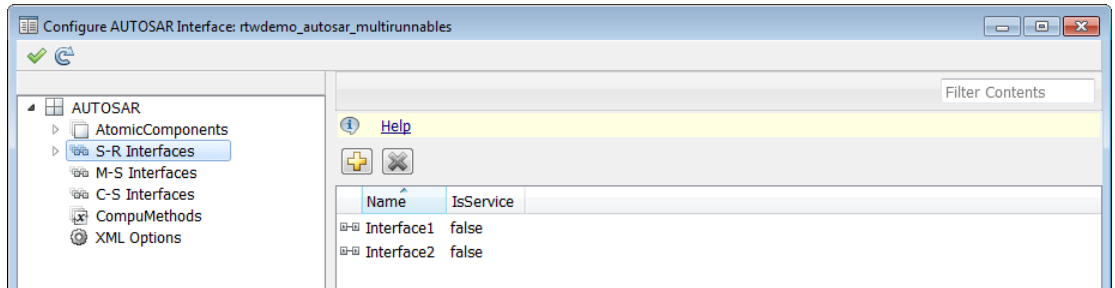
- Click the **Add** icon  to add an AUTOSAR IRV.
- Select an AUTOSAR IRV and then click the **Delete** icon  to remove it.



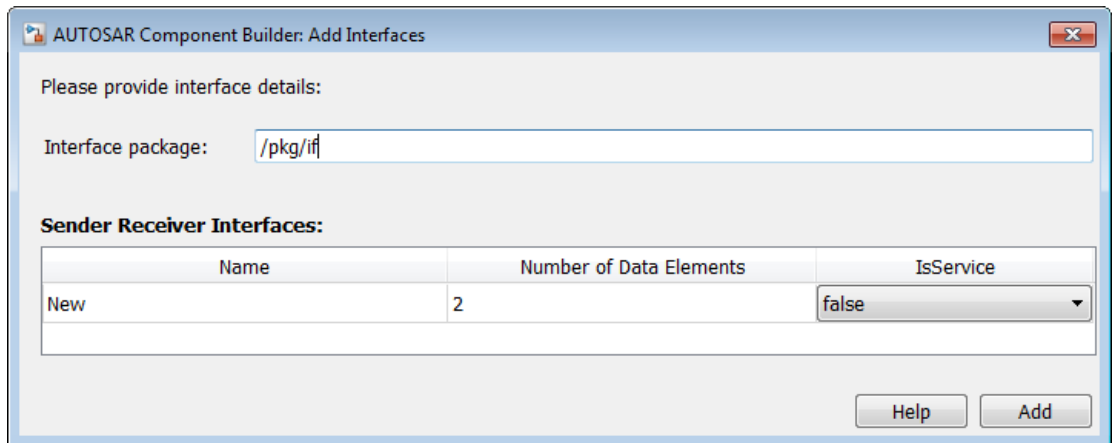
**16** In the left-hand pane of the Configure AUTOSAR Interface dialog box, select **S-R Interfaces**.

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 string.
- 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.

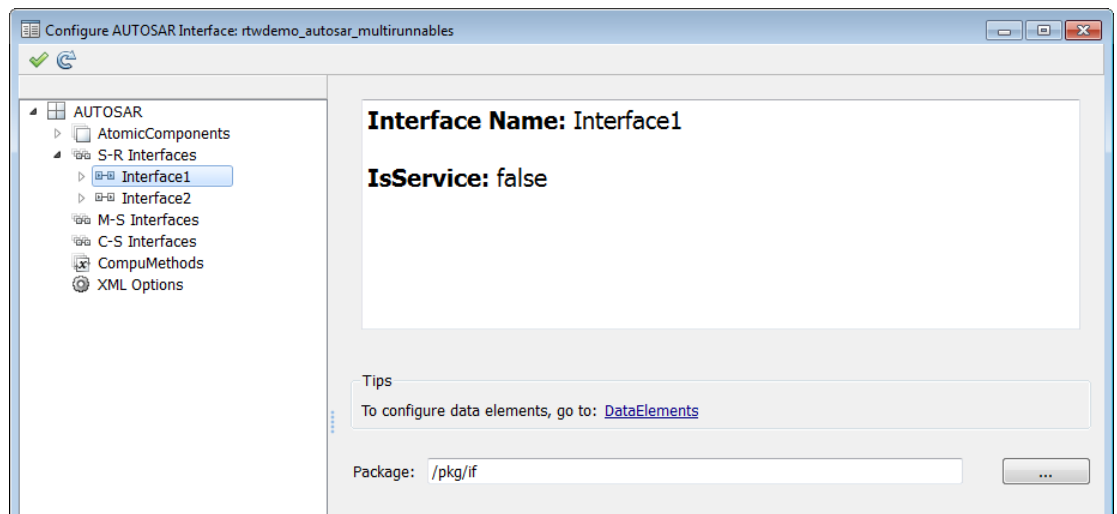


- 17 In the left-hand pane of the Configure AUTOSAR Interface dialog box, expand **S-R Interfaces** and select an S-R interface from the list.

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 4-51.

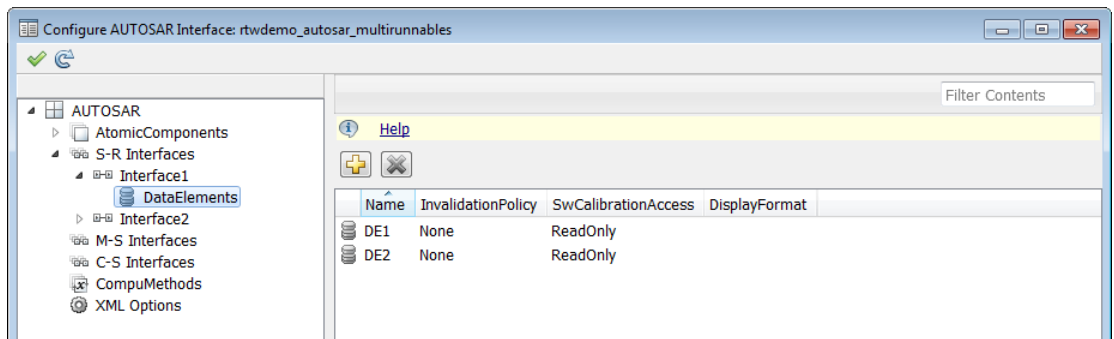


- 18** In the left-hand pane of the Configure AUTOSAR Interface dialog box, expand the selected interface and select **DataElements**.

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 4-74.
- Specify the level of measurement and calibration tool access to the S-R interface data element. Set its **SwCalibrationAccess** value to **ReadOnly**, **ReadWrite**, or **NotAccessible**.



- Optionally specify the format to be used by measurement and calibration tools to display the S-R interface 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 AUTOSAR Display Format for Measurement and Calibration” on page 4-143.
- Click the **Add** icon  to add a data element.
- Select a data element and then click the **Delete** icon  to remove it.

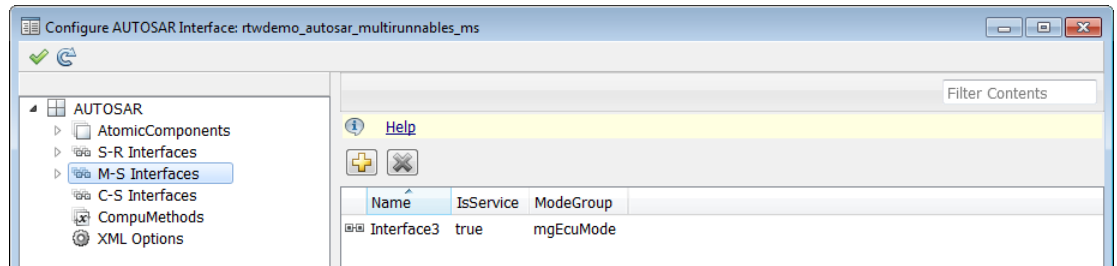


- 19** In the left-hand pane of the Configure AUTOSAR Interface dialog box, select **M-S Interfaces**.

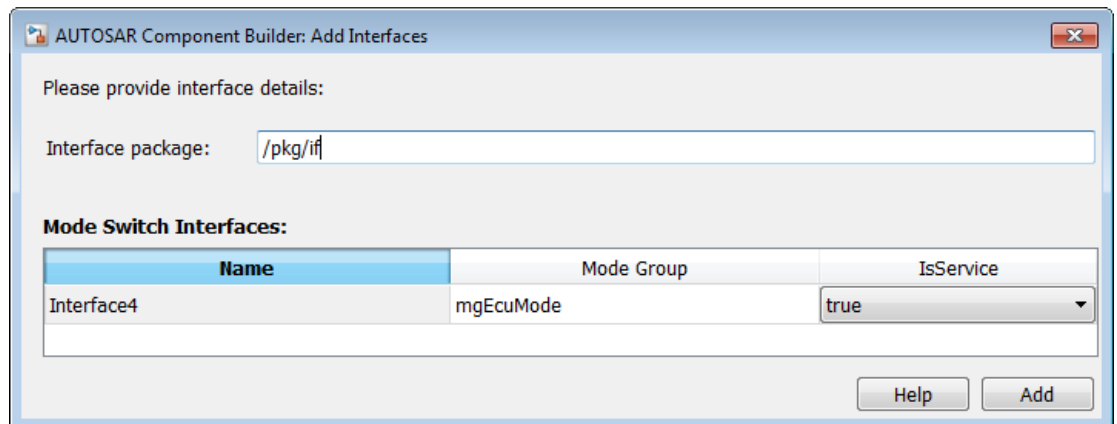
The M-S Interfaces view of the AUTOSAR Properties Explorer lists AUTOSAR mode switch interfaces and their properties. You can

- 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 using Simulink enumeration. For more information, see “Configure AUTOSAR Mode Receiver Ports and Mode-Switch Events” on page 4-85.
- Rename an M-S interface by clicking its name and then editing the name string.

- 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.

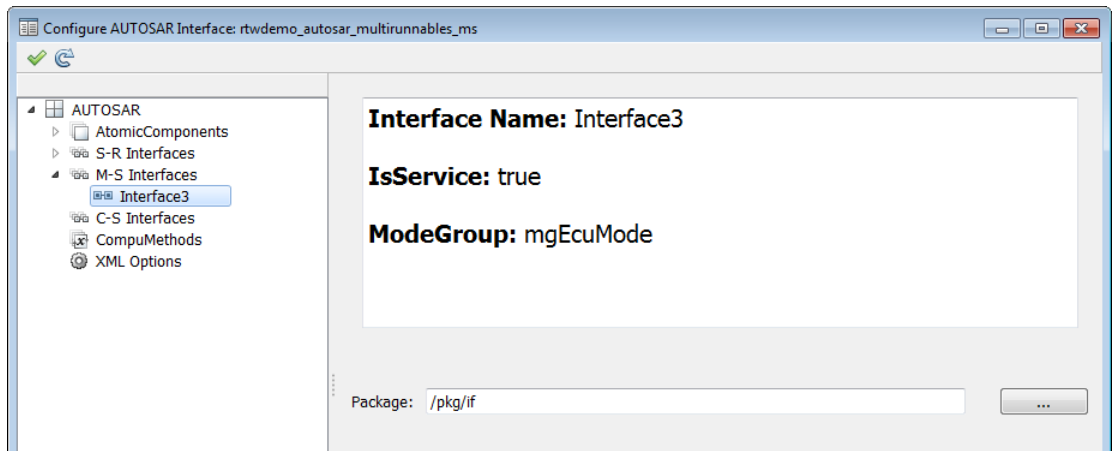


- 20** In the left-hand pane of the Configure AUTOSAR Interface dialog box, expand **M-S Interfaces** and select an M-S interface from the list.

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 4-51.

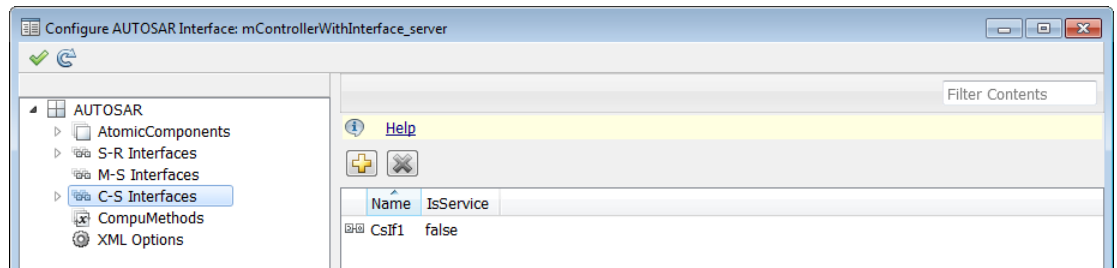


- 21** In the left-hand pane of the Configure AUTOSAR Interface dialog box, select **C-S Interfaces**.

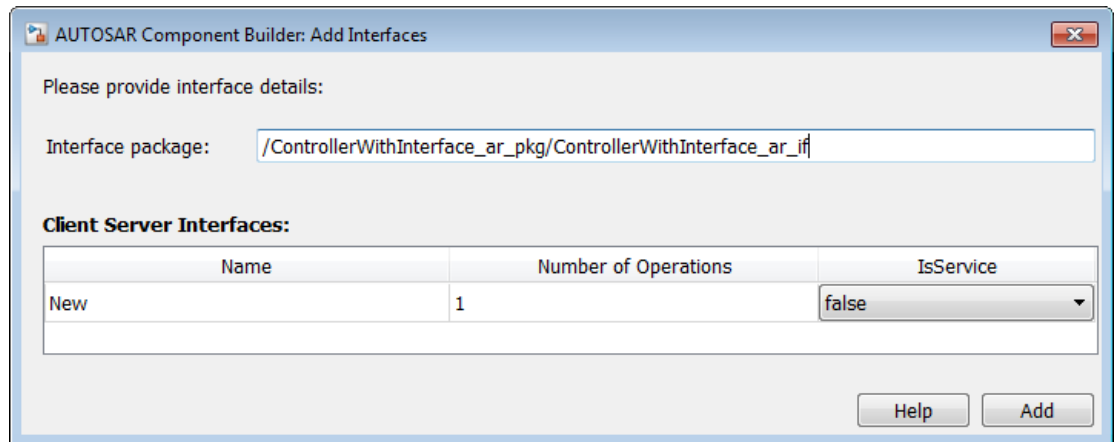
The C-S interfaces view of the AUTOSAR Properties Explorer lists AUTOSAR client-server interfaces and their properties. You can

- 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 string.
- 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.



- 22** 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.

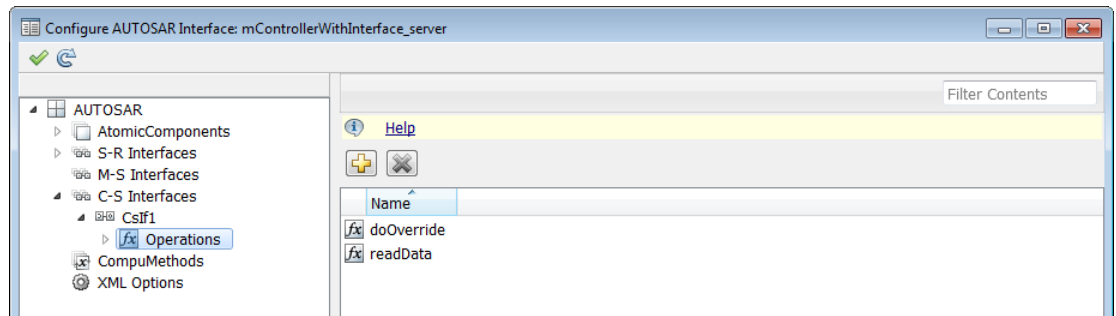
- Click the button to the right of the **Package** field to open the AUTOSAR Package Browser. Use the browser to navigate to an existing package or create a new package. When you select a package in the browser and click **Apply**, the interface **Package** parameter value is updated with your selection. For more information about the AUTOSAR Package Browser, see “Configure AUTOSAR Package for Interface or CompuMethod” on page 4-51.



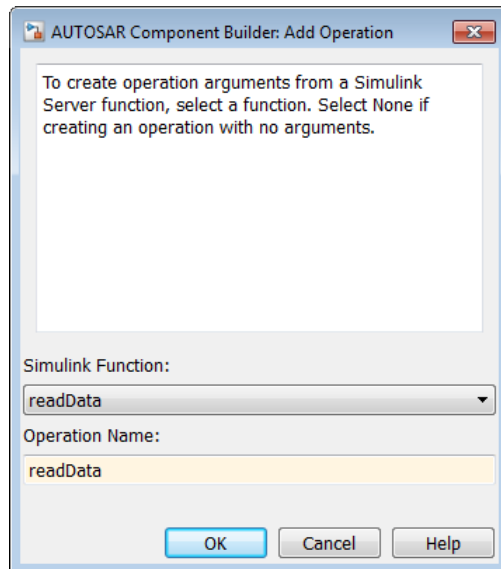
- 23** In the left-hand pane of the Configure AUTOSAR Interface dialog box, expand the selected interface and select **Operations**.

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.




- 24** 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.

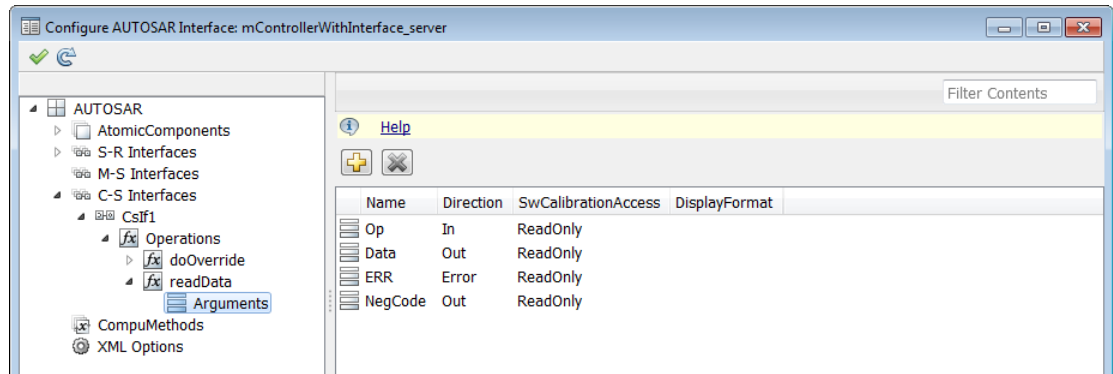


- 25** In the left-hand pane of the Configure AUTOSAR Interface dialog box, expand the selected operation and select **Arguments**.

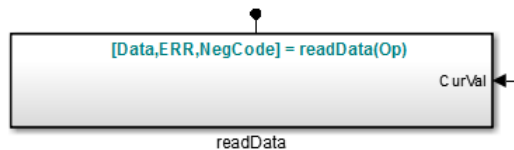
The Arguments view of the AUTOSAR Properties Explorer lists AUTOSAR client-server operation arguments and their properties. You can

- 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 4-113.
- Specify the level of measurement and calibration tool access to the C-S operation argument. Set its **SwCalibrationAccess** value to **ReadOnly**, **ReadWrite**, or **NotAccessible**.
- Optionally specify the format to be used by measurement and calibration tools to display the C-S operation 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 AUTOSAR Display Format for Measurement and Calibration” on page 4-143.
- Click the **Add** icon  to add an argument.

- Select an argument and then click the **Delete** icon  to remove it.





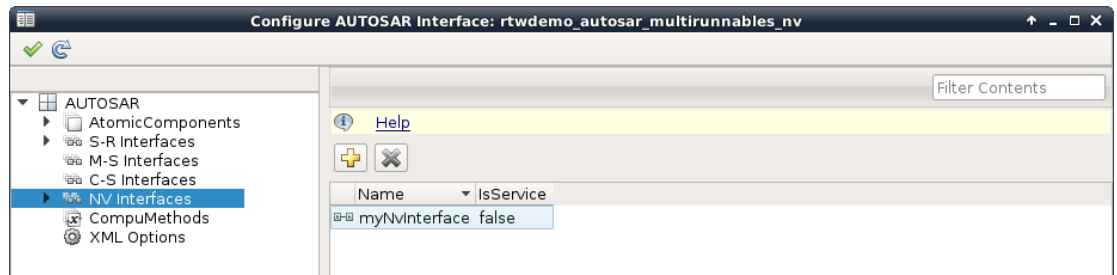
The displayed server operation arguments were created from the following Simulink Function block.



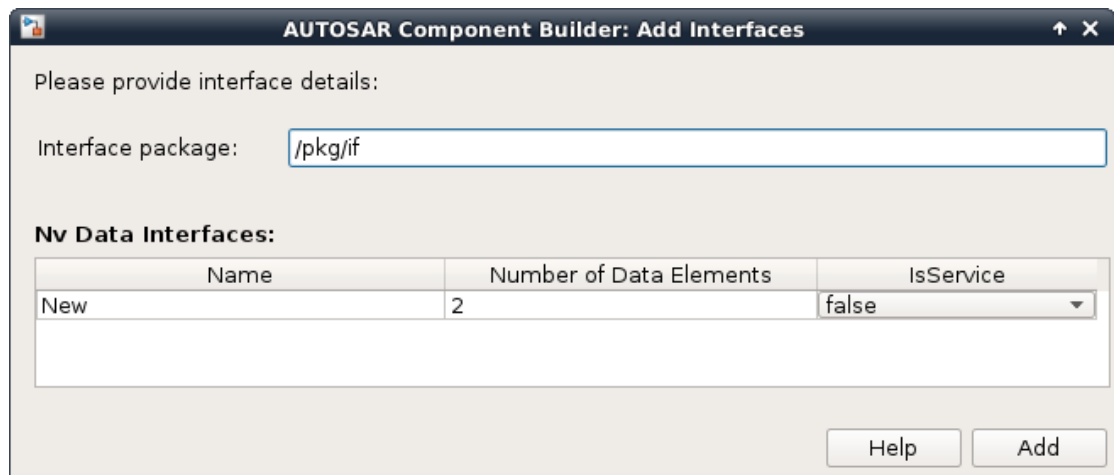
- 26** In the left-hand pane of the Configure AUTOSAR Interface dialog box, select **NV Interfaces**.

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 string.
- 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.

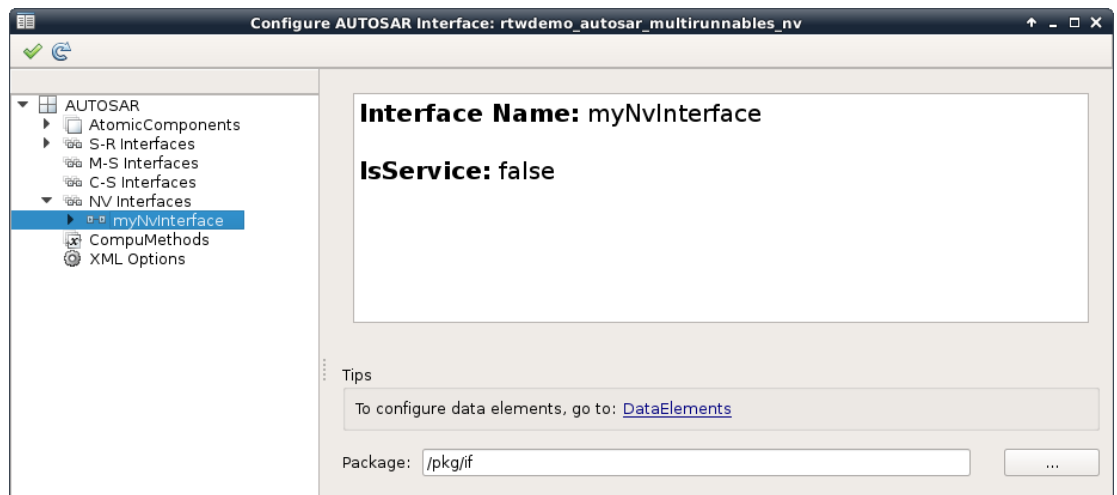


- 27 In the left-hand pane of the Configure AUTOSAR Interface dialog box, expand **NV Interfaces** and select an NV interface from the list.

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 4-51.

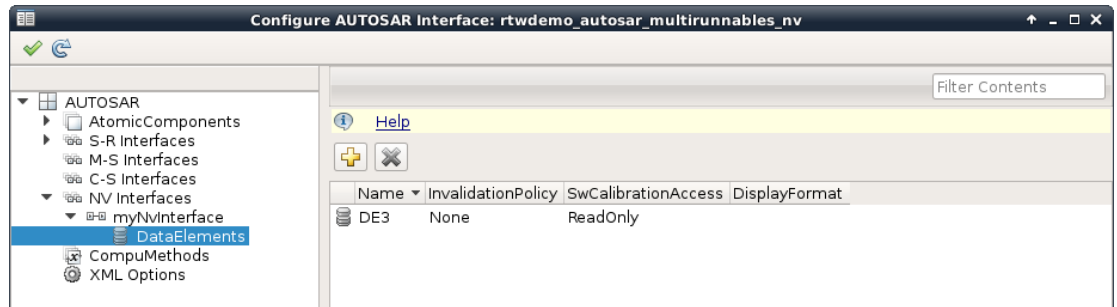


- 28** In the left-hand pane of the Configure AUTOSAR Interface dialog box, expand the selected interface and select **DataElements**.

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.
- If signal invalidation is configured for the data element on an NV 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 4-74.
- Specify the level of measurement and calibration tool access to the NV interface data element. Set its **SwCalibrationAccess** value to **ReadOnly**, **ReadWrite**, or **NotAccessible**.

- Optionally specify the format to be used by measurement and calibration tools to display the NV interface 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 AUTOSAR Display Format for Measurement and Calibration” on page 4-143.
- Click the **Add** icon  to add a data element.
- Select a data element and then click the **Delete** icon  to remove it.



- 29** 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:

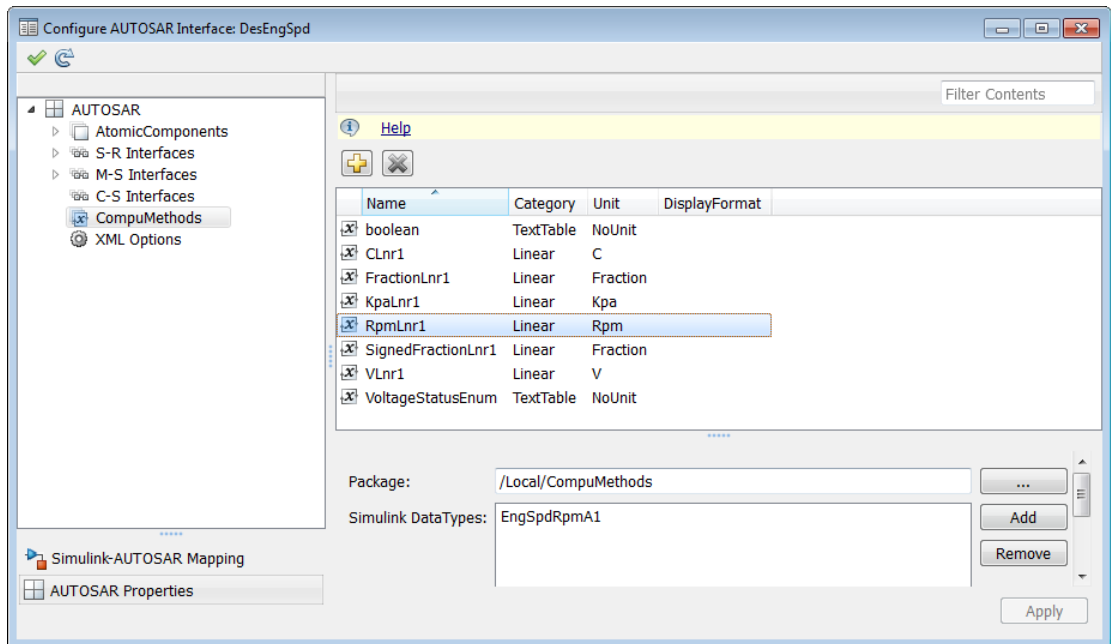
- Select a CompuMethod and modify properties, such as name, category, unit, display format for measurement and calibration, AUTOSAR package to be generated for the CompuMethod, and a list of Simulink data types that reference the CompuMethod. For property descriptions, see “Configure AUTOSAR CompuMethods” on page 4-152.
- 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.



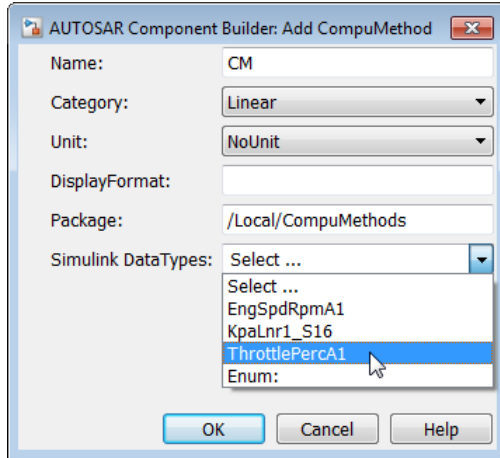
To modify the AUTOSAR package for a CompuMethod, you can do either of the following:

- 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 4-51.

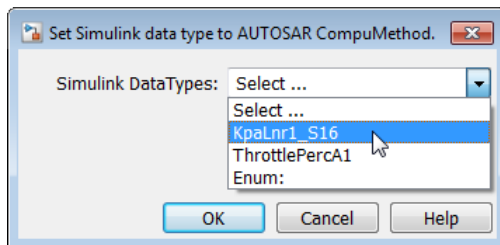
To associate a CompuMethod with a Simulink data type used in the model, select a CompuMethod and click the **Add** button to the right of **Simulink DataTypes**. This action opens a dialog box with a list of available data types. Select a data type and click OK to add it to the **Simulink DataTypes** 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 4-152.

- 30 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...
<i>modelName_component.arxml</i>	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 4-62.
<i>modelName_datatype.arxml</i>	Data types and related elements.
<i>modelName_implementation.arxml</i>	Software component implementation.
<i>modelName_interface.arxml</i>	Interfaces, including S-R, C-S, M-S, and NV interfaces.
<i>modelName_behavior.arxml</i>	Software component internal behavior (generated only for schema 3.x or earlier).

For more information, see “Export AUTOSAR Software Component” on page 5-5.

- 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 4-54.

- 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 **ImplementationDataTypeReference** to `NotAllowed`. For more information, see “Control Application Data Type Generation” on page 4-148.
- 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.

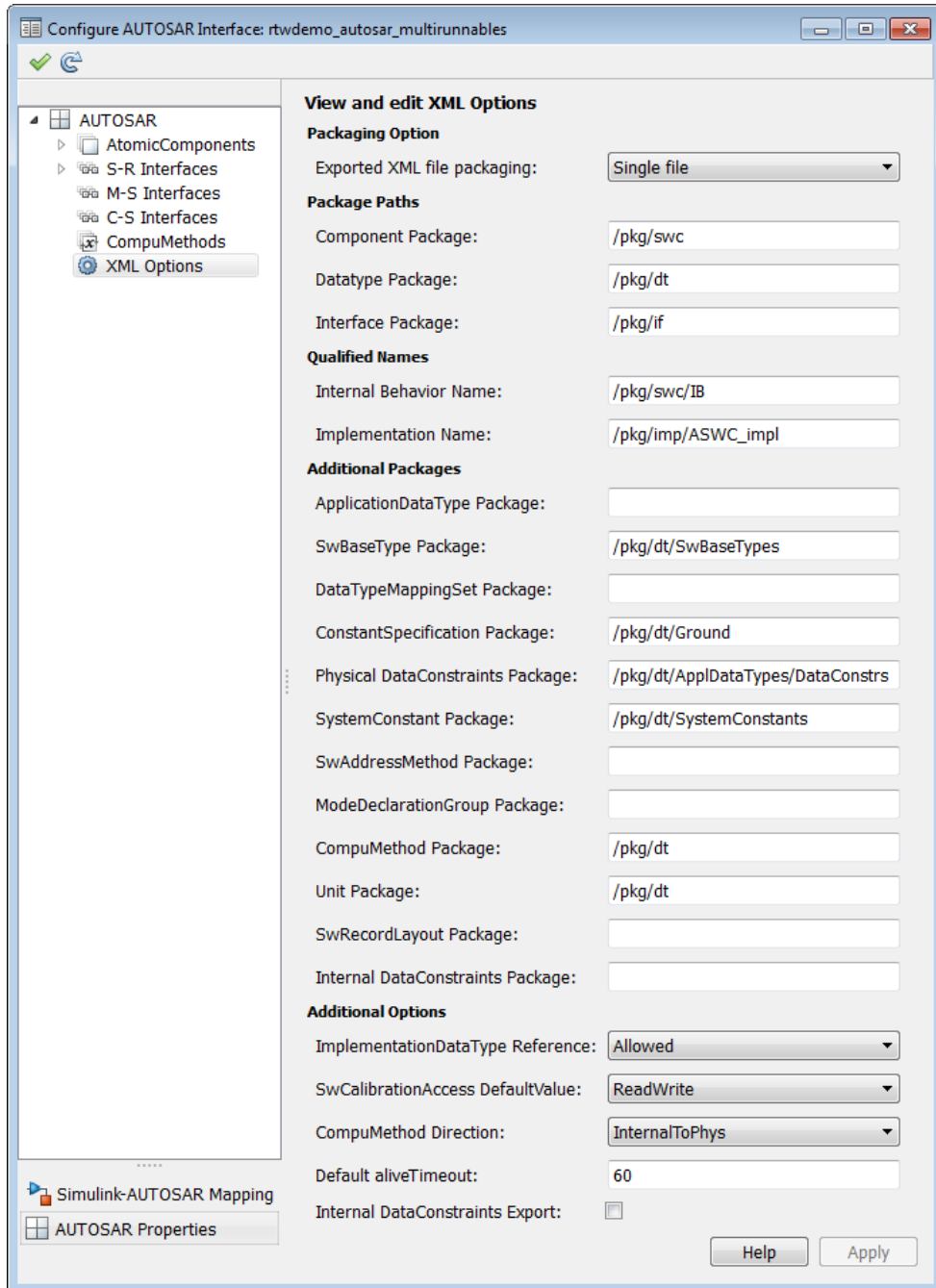
For more information, see “Configure SwCalibrationAccess” on page 4-135.


- Control the direction of **CompuMethod** conversion for linear-function **CompuMethods**. Select one of the following values for **CompuMethod Direction**:
  - `InternalToPhys` (default) — Generate **CompuMethod** sections for conversion of internal values into their physical representations.
  - `PhysToInternal` — Generate **CompuMethod** sections for conversion of physical values into their internal representations.
  - `Bidirectional` — Generate **CompuMethod** sections for both internal-to-physical and physical-to-internal conversion directions.

For more information, see “Configure CompuMethod Direction for Linear Functions” on page 4-155.

- Use **Default aliveTimeout** to control the `aliveTimeout` value to be generated for each AUTOSAR port in subsequent `arxml` exports. In AUTOSAR applications, the `aliveTimeout` value for an AUTOSAR port specifies the amount of time in seconds after which the AUTOSAR software component must be notified if the port has not received data according to a specified timing description. The `arxml` exporter default value is 60 seconds. You can specify a different positive quantity of seconds, or 0 if no timeout monitoring is to be performed. For more information, see “Configure AUTOSAR Port aliveTimeout Value” on page 4-70.

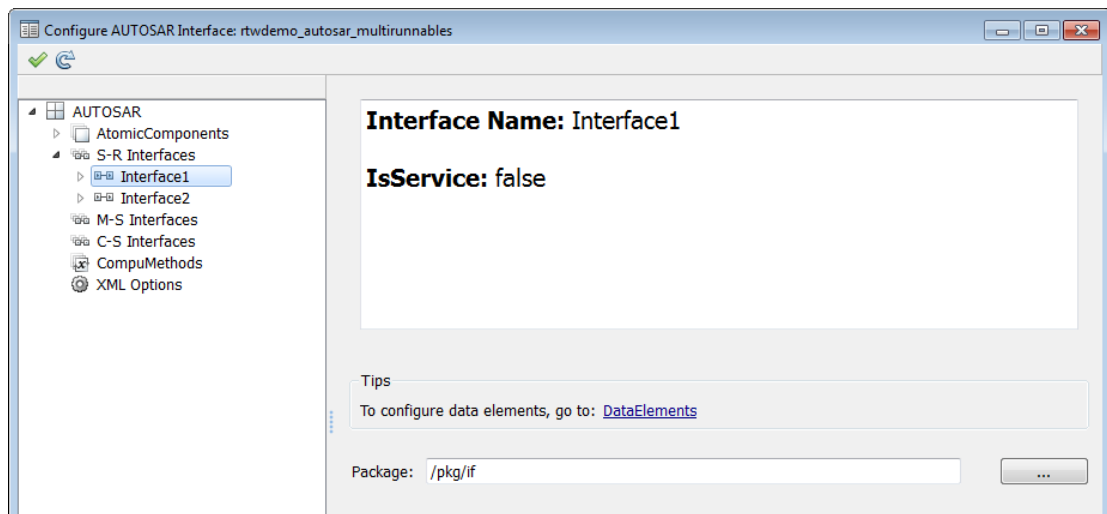
- Optionally override the default behavior for generating internal data constraint information for AUTOSAR implementation data types in arxml code. To force export of internal data constraints for implementation data types, select the option **Internal DataConstraints Export**. For more information, see “Configure AUTOSAR Internal Data Constraints Export” on page 4-162.



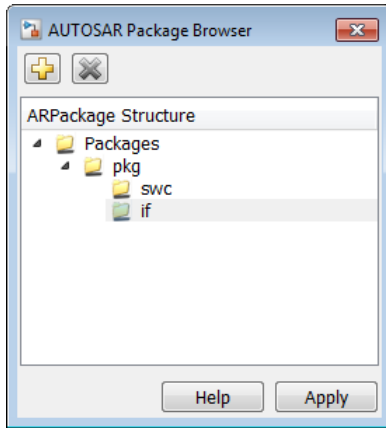
- 31 Click the **Validate** icon  to validate the AUTOSAR interface configuration. If errors are reported, address them and then retry validation.


## Configure AUTOSAR Package for Interface or CompuMethod

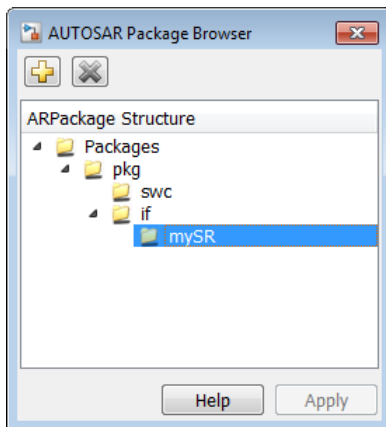
As part of configuring an AUTOSAR interface or an AUTOSAR CompuMethod, you specify the AUTOSAR package (AR-PACKAGE) to be generated for individual S-R, M-S, C-S, and NV interfaces or individual CompuMethods in your configuration. For example, here is the AUTOSAR Properties Explorer view for an individual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

Package:



For more information about AR-PACKAGEs, see “Configure AUTOSAR Packages” on page 4-54.

# Configure AUTOSAR Packages

### In this section...

“AR-PACKAGE Structure” on page 4-54

“Configure AUTOSAR Packages and Paths” on page 4-56

“Export AUTOSAR Packages” on page 4-59

“AR-PACKAGE Location in Exported ARXML Files” on page 4-62

## 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:

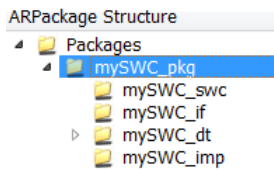
- Conform to an organizational or standardized AR-PACKAGE structure.
- Establish a namespace for elements in a package.
- Provide a basis for relative references to elements.

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, if you create an AUTOSAR component named `mySWC`, 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
- Computational 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.

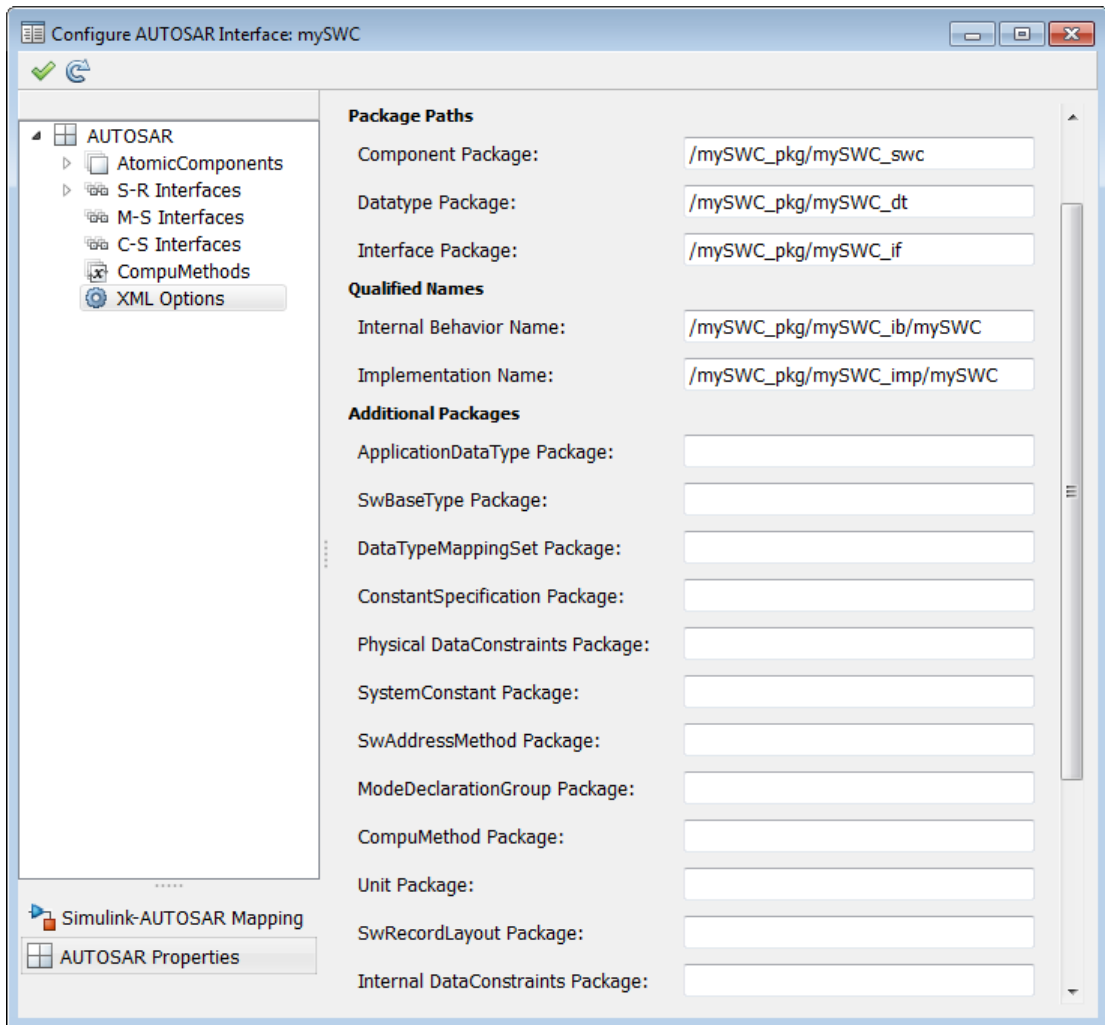
---

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 4-62.

### 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.

If you create an AUTOSAR software component in Simulink, 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 path specification 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	<i>modelName_pkg/modelname_swc/modelname</i> (dialog box displays the component path without the shortname)
DataTypePackage	<i>modelName_pkg/modelname_dt</i>
InterfacePackage	<i>modelName_pkg/modelname_if</i>
InternalBehaviorQualifiedName	<i>modelName_pkg/modelname_ib/modelname</i>
ImplementationQualifiedName	<i>modelName_pkg/modelname_imp/modelname</i>
ApplicationDataTypePackage	<i>DataTypePackage/ApplDataTypes</i>
SwBaseTypePackage	<i>DataTypePackage/SwBaseTypes</i>
DataTypeMappingPackage	<i>DataTypePackage/DataTypeMappings</i>
ConstantSpecificationPackage	<i>DataTypePackage/Ground</i>
DataConstraintPackage	<i>ApplicationDataTypePackage/DataConstrs</i>
SystemConstantPackage	<i>DataTypePackage/SystemConstants</i>
SwAddressMethodPackage	<i>DataTypePackage/SwAddrMethods</i>
ModeDeclarationGroupPackage	<i>DataTypePackage</i>
CompuMethodPackage	<i>DataTypePackage</i>
UnitPackage	<i>DataTypePackage</i>
SwRecordLayoutPackage	<i>ApplicationDataTypePackage/RecordLayouts</i>
InternalDataConstraintPackage	<i>DataTypePackage/DataConstrs</i>

To set a packaging property in the MATLAB Command Window or in a script, use an AUTOSAR property `set` function call similar to the following:

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

For an example of configuring and exporting AUTOSAR packages, see “Export AUTOSAR Packages” on page 4-59.

## 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 target 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.

**View and edit XML Options**

**Packaging Option**

Exported XML file packaging:

**Package Paths**

Component Package:

Datatype Package:

Interface Package:

**Qualified Names**

Internal Behavior Name:


Implementation Name:

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`.)

Additional Packages	
ApplicationDataType Package:	<input type="text"/>
SwBaseType Package:	<input type="text"/>
DataTypeMappingSet Package:	<input type="text"/>
ConstantSpecification Package:	<input type="text" value="/pkg/misc/MyGround"/>
Physical DataConstraints Package:	<input type="text"/>
SystemConstant Package:	<input type="text"/>
SwAddressMethod Package:	<input type="text"/>
ModeDeclarationGroup Package:	<input type="text"/>
CompuMethod Package:	<input type="text"/>
Unit Package:	<input type="text"/>
SwRecordLayout Package:	<input type="text"/>
Internal DataConstraints Package:	<input type="text"/>

- 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 string AR-PACKAGE. For the example model, searching `rtwdemo_autosar_multirunnables.arxml` for the string 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>
          </ELEMENTS>
        </ARRAY-VALUE-SPECIFICATION>
      </VALUE-SPEC>
    </CONSTANT-SPECIFICATION>
  </ELEMENTS>
</AR-PACKAGE>

```

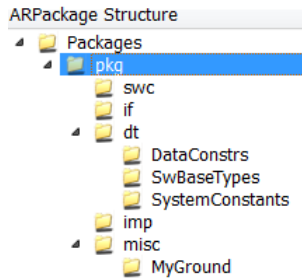
- 8** During the code generation that you initiated in a previous step, the arxml exporter filled in other package paths based on internal rules. Return to the XML Options view in the Configure AUTOSAR Interface dialog box. The display now includes five additional package paths.

Additional Packages	
ApplicationDataType Package:	<input type="text"/>
SwBaseType Package:	<input type="text" value="/pkg/dt/SwBaseTypes"/>
DataTypeMappingSet Package:	<input type="text"/>
ConstantSpecification Package:	<input type="text" value="/pkg/misc/MyGround"/>
Physical DataConstraints Package:	<input type="text" value="/pkg/dt/ApplDataTypes/DataConstrs"/>
SystemConstant Package:	<input type="text" value="/pkg/dt/SystemConstants"/>
SwAddressMethod Package:	<input type="text"/>
ModeDeclarationGroup Package:	<input type="text"/>
CompuMethod Package:	<input type="text" value="/pkg/dt"/>
Unit Package:	<input type="text" value="/pkg/dt"/>
SwRecordLayout Package:	<input type="text"/>
Internal DataConstraints Package:	<input type="text"/>

### 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 4-59). 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 3-2
- “Export AUTOSAR Component XML and C Code” on page 5-2

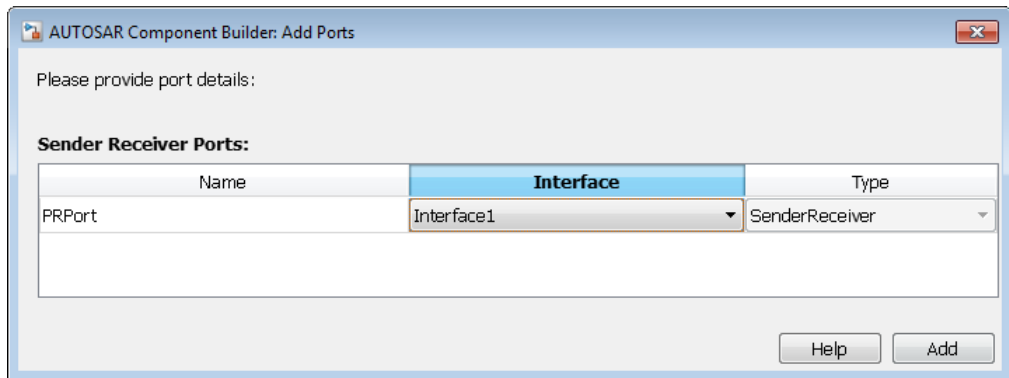
## **More About**

- “AUTOSAR Interface Configuration” on page 4-3

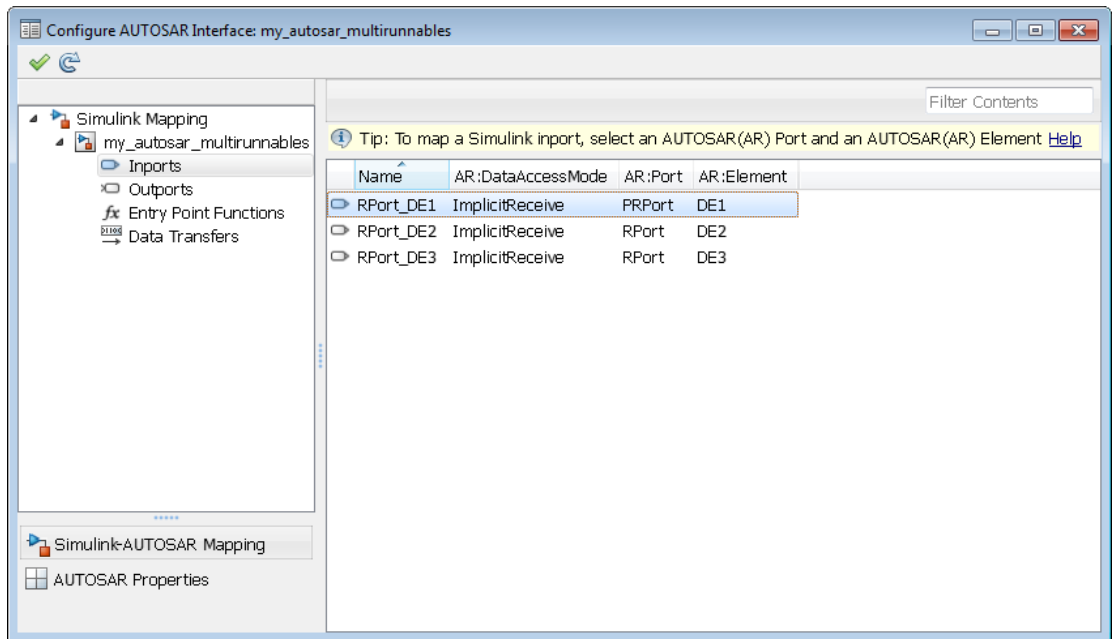
## Configure AUTOSAR Provide-Require Port

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. To configure an AUTOSAR PRPort in Simulink:

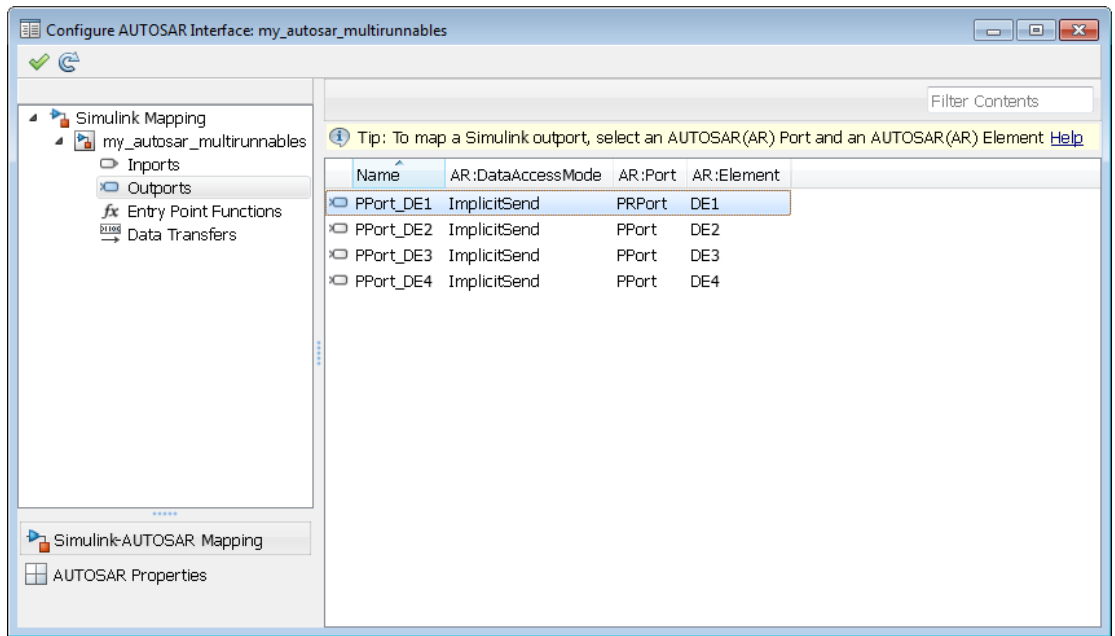
- 1 Open a model that is configured for AUTOSAR, and in which a runnable has an inport and an outport suitable for pairing into an AUTOSAR PRPort. In this example, the RPort\_DE1 inport and PPort\_DE1 outport both use data type `int8`, port dimension 1, and signal type `real`.
- 2 Open the Configure AUTOSAR Interface dialog box, and navigate to the SenderReceiverPorts view of the AUTOSAR Properties Explorer.
- 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 outpost to the AUTOSAR sender-receiver port you created, select the outpost, set **AR:Port** to the value **PRPort**, and set **AR:Element** to the same data element selected in the previous step.



- 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 output that are mapped to the AUTOSAR PRPort.

Alternatively, you can programmatically add and map a PRPort port using AUTOSAR property and mapping functions. The following example adds an AUTOSAR PRPort (sender-receiver port) and then maps it to a Simulink inport and output 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(s1Map, 'RPort_DE1', 'PRPort', 'DE1', 'ImplicitReceive')
mapOutport(s1Map, 'PPort_DE1', 'PRPort', 'DE1', 'ImplicitSend')
[arPortName, arDataElementName, arDataAccessMode] = getOutport(s1Map, 'PPort_DE1')

arPortName =
PRPort

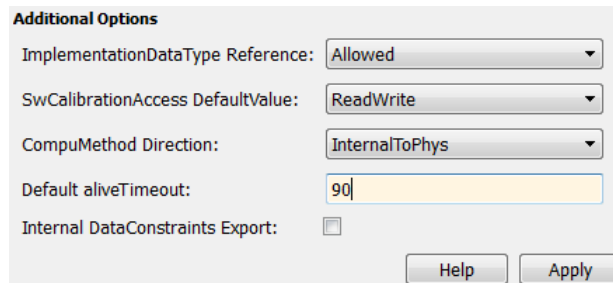
arDataElementName =
DE1

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. By default, `arxml` export generates an `aliveTimeout` value of 60 for each AUTOSAR port. To set a different value to be generated in subsequent `arxml` exports, modify the XML option **Default aliveTimeout**.

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 seconds. Specify a positive integer quantity of seconds, or 0 not to monitor port timeouts. Click **Apply**. The value you specify is generated for each port in subsequent `arxml` exports.



**Additional Options**

ImplementationDataType Reference:

SwCalibrationAccess DefaultValue:

CompuMethod Direction:

Default aliveTimeout:

Internal DataConstraints Export:

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.

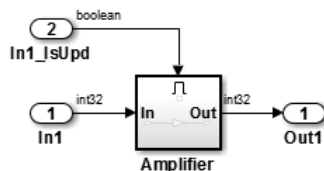
For the sender, the AUTOSAR run-time environment (RTE) sets the status of an update flag, indicating whether the data element has been written. The receiver calls the `Rte_IsUpdated_Port_Element` API, which reads and then clears the update flag.

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:

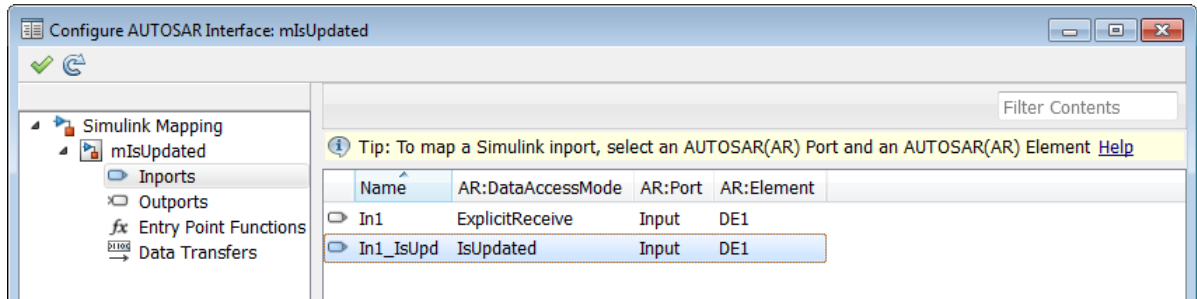
- 1 Open a model for which an AUTOSAR sender-receiver interface is configured.
- 2 Identify the inport that corresponds to the AUTOSAR receiver port for which `IsUpdated` service is required. Create a second inport, set its data type to `boolean`, and connect it to the same block. For example:




- 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 2-10
- “Import AUTOSAR Software Component” on page 3-2
- “Export AUTOSAR Component XML and C Code” on page 5-2

## **More About**

- “AUTOSAR Interface Configuration” on page 4-3

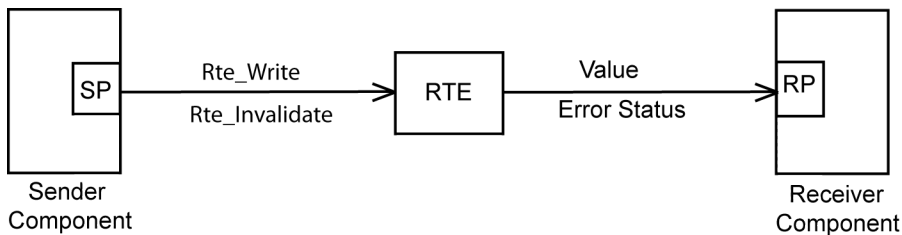
## Configure AUTOSAR Sender Port for Data Element Invalidation

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.

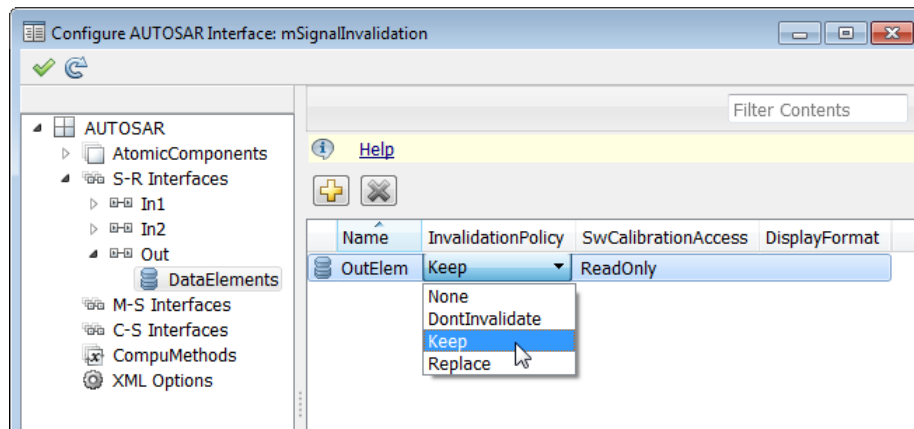
Invalidation Policy	Value	Error Status
Keep	Previous good value	RTE_E_INVALID
Replace	Initial value	RTE_E_OK



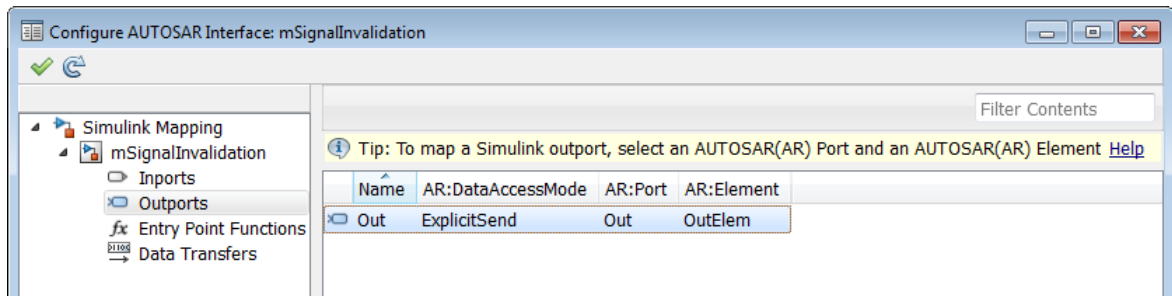
**Note:** The **InvalidationPolicy** values **DontInvalidate** and **None** indicate no invalidation policy. **DontInvalidate** causes the ARXML exporter to generate **INVALIDATION-POLICY** tagging with the property **HANDLE-INVALID** set to **DONT-INVALIDATE**. **None** does not generate **INVALIDATION-POLICY**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

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

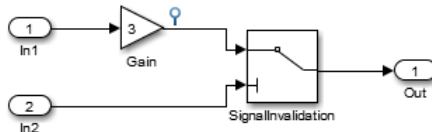
- 1 Open a model for which an AUTOSAR sender-receiver interface is configured.
- 2 Open the Configure AUTOSAR Interface dialog box. Select **Code > C/C++ Code > Configure Model as AUTOSAR Component**.
- 3 Select **AUTOSAR Properties** and expand **S-R Interfaces**.
- 4 Navigate to the AUTOSAR data element 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 **Outputs**.
- 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 2-10
- “Import AUTOSAR Software Component” on page 3-2
- “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.

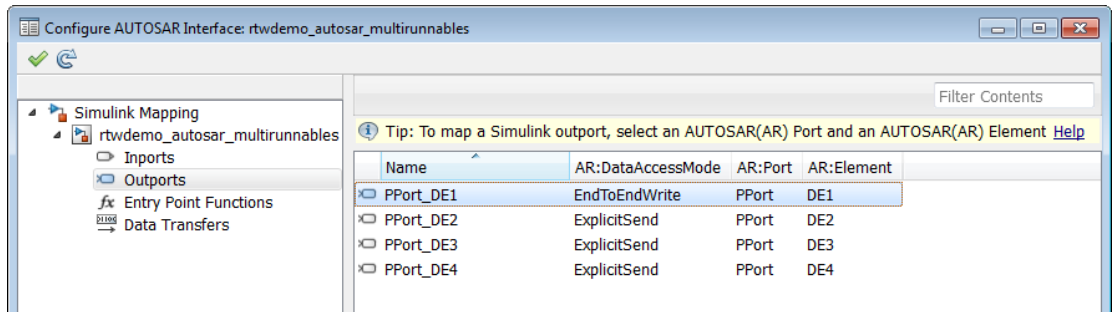
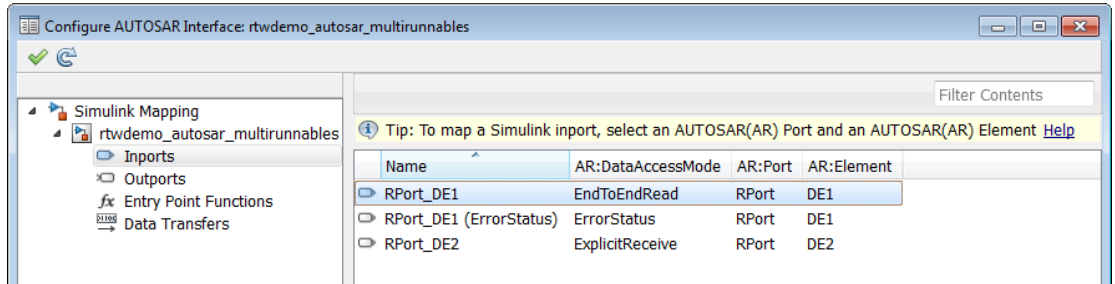
You should configure E2E protection for AUTOSAR sender and receiver ports that use explicit write and read data access modes. When you change the data access mode of an AUTOSAR port from explicit write to end-to-end write, or from explicit read to end-to-end read:


- 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(void)
{
    ...
    uint32_T err = E2EPW_Read_RPort_DE1(...);
    ...
    E2EPW_Write_PPort_DE1(...);
    ...
}
...
void Runnable_Init(void)
{
```

```
...
    E2EPW_ReadInit_RPort_DE1();
...
    E2EPW_WriteInit_PPort_DE1();
...
}
```

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 2-10
- “Import AUTOSAR Software Component” on page 3-2
- “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.

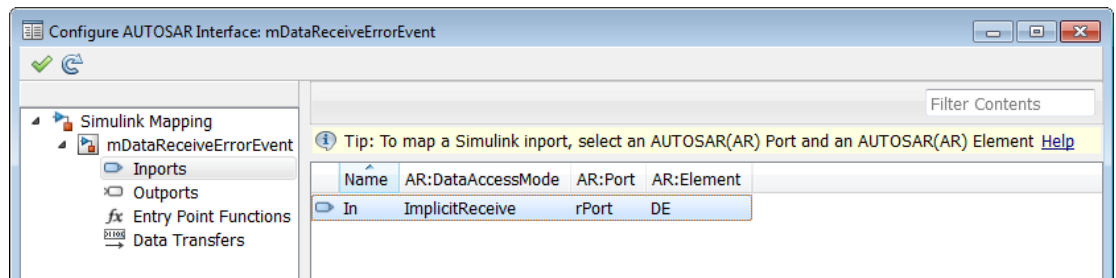
Embedded Coder supports creating `DataReceiveErrorEvents` in AUTOSAR receiver components. In Simulink, you can:

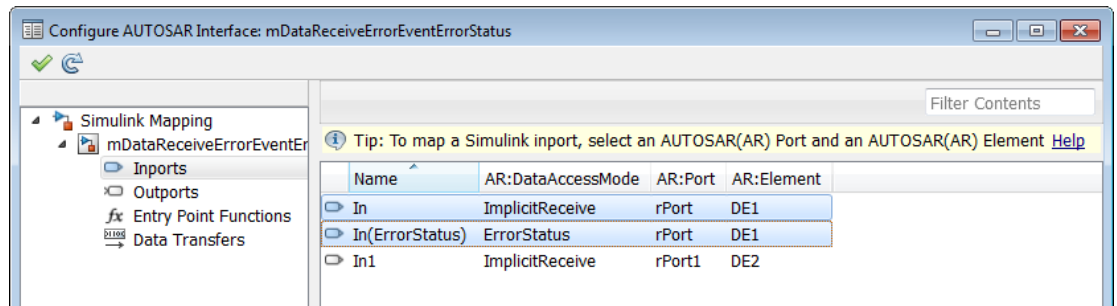
- Import an AUTOSAR `DataReceiveErrorEvent` definition.
- Define a `DataReceiveErrorEvent`.
- Generate arxml code for AUTOSAR receiver ports for which a `DataReceiveErrorEvent` is configured.


You should configure a `DataReceiveErrorEvent` for an AUTOSAR receiver port that uses `ImplicitReceive`, `ExplicitReceive`, or `EndToEndRead` data access mode.

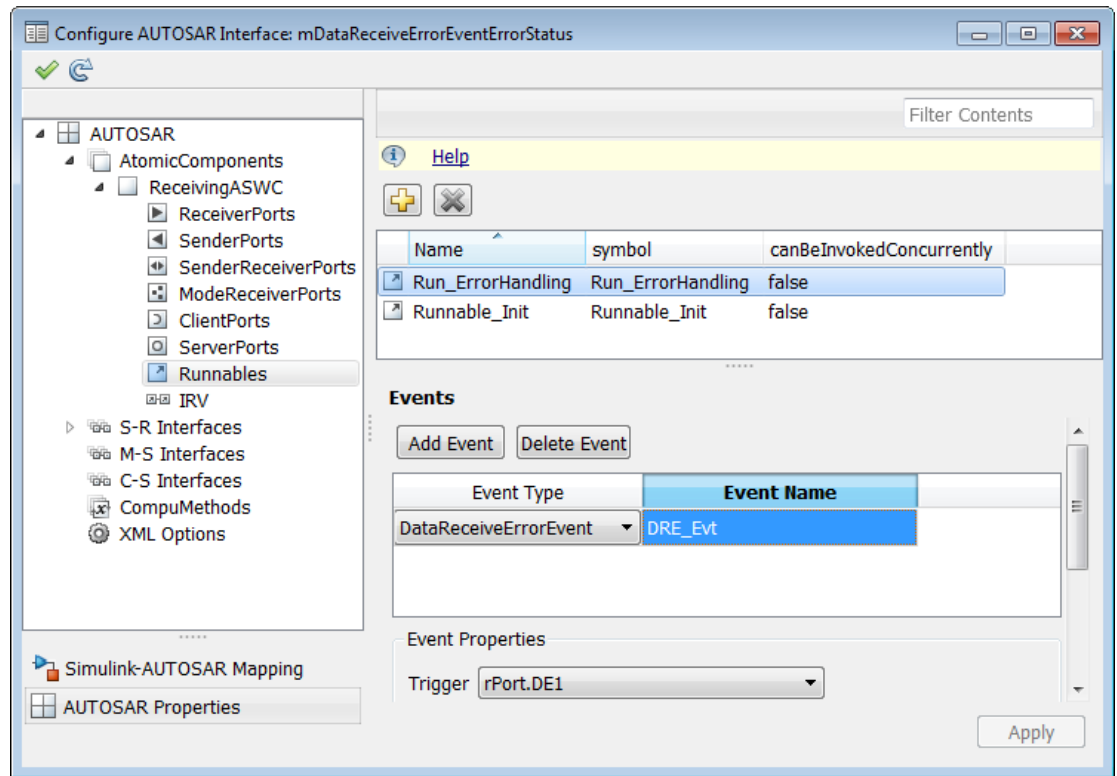
To configure an AUTOSAR receiver port for a `DataReceiveErrorEvent`:

- 1 Open a model for which the receiver side of an AUTOSAR sender-receiver interface is configured.
- 2 Open the 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(mdIname);
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_sw/ReceivingASWC/IB/Run_ErrorHandling</START-ON-EVENT-REF>
    <DATA-IREF>
      <CONTEXT-R-PORT-REF DEST="R-PORT-PROTOTYPE">
        /Root/mDemoModel_sw/ReceivingASWC/rPort</CONTEXT-R-PORT-REF>
      <TARGET-DATA-ELEMENT-REF DEST="VARIABLE-DATA-PROTOTYPE">
        /Root/Interfaces/In/DE</TARGET-DATA-ELEMENT-REF>
    </DATA-IREF>
  </DATA-RECEIVE-ERROR-EVENT>
</EVENTS>
...
<RUNNABLES>
  <RUNNABLE-ENTITY>
    <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 2-10
- “Import AUTOSAR Software Component” on page 3-2
- “Export AUTOSAR Component XML and C Code” on page 5-2

## More About

- “AUTOSAR Interface Configuration” on page 4-3



## Configure AUTOSAR Mode Receiver Ports and Mode-Switch Events

This example shows how to configure an AUTOSAR mode-receiver port and an AUTOSAR mode-switch event, using the example model `rtwdemo_autosar_multirunnables` as a starting point.

---

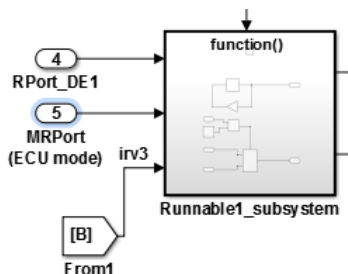
**Note:** This example does not implement a meaningful algorithm for controlling component execution based on the current ECU mode.

---

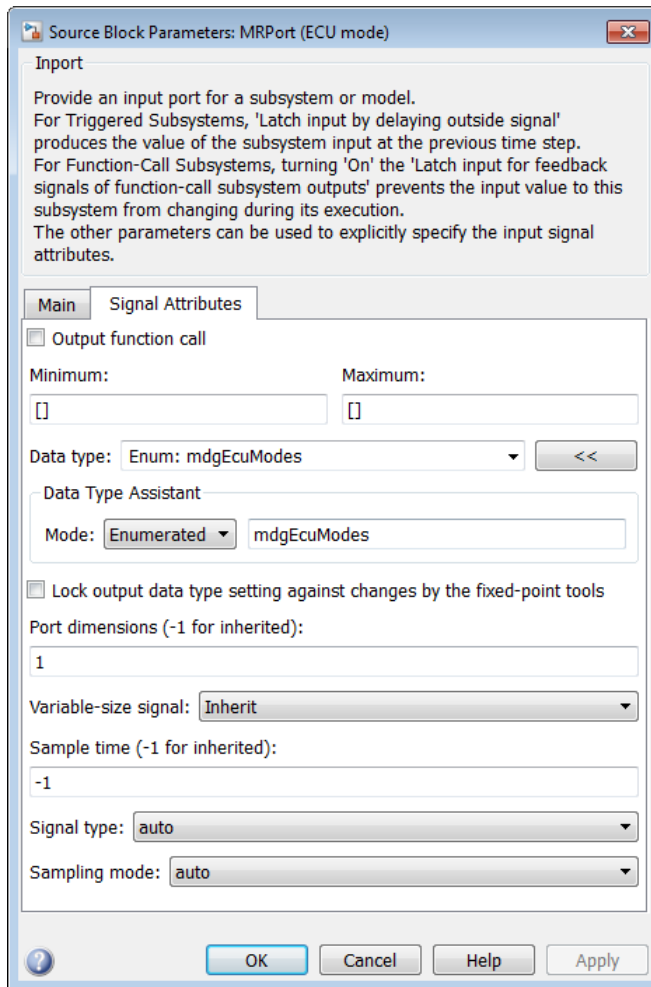
- 1 Open the example model `rtwdemo_autosar_multirunnables`. Save a copy to a writable work folder.
- 2 Declare a mode declaration group — a group of mode values — using Simulink enumeration. Enter the following command in the MATLAB Command Window:

```
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.

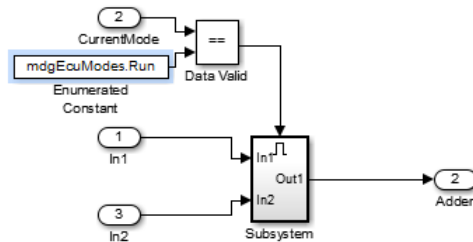



- 4 Next, apply the mode declaration group `mdgEcuModes` to inport `MRPort` (`ECU mode`). Double-click the inport to open the Inport block parameters dialog box. Select the **Signal Attributes** tab. For **Data type**, enter `Enum: mdgEcuModes`. Additionally, set both **Signal type** and **Sampling mode**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

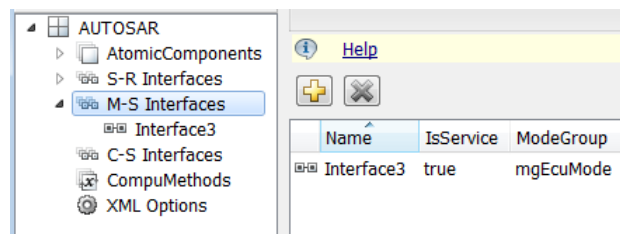
click the block to open its block parameters dialog box. Set **Output data type** to Enum: `mdgEcuModes` and set **Value** to `mdgEcuModes.Run`. Click **OK**.




- 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.

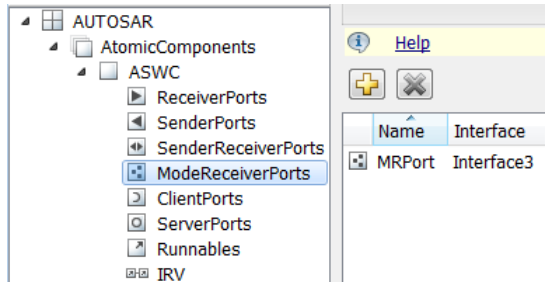
Click **Add**.



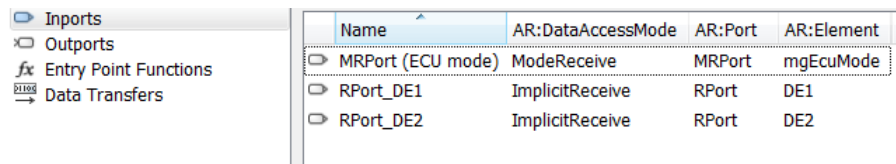
The value you specify for the AUTOSAR mode group is used in a later step, when you map a Simulink inport to an AUTOSAR mode-receiver port and element.


- 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**.



- 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.

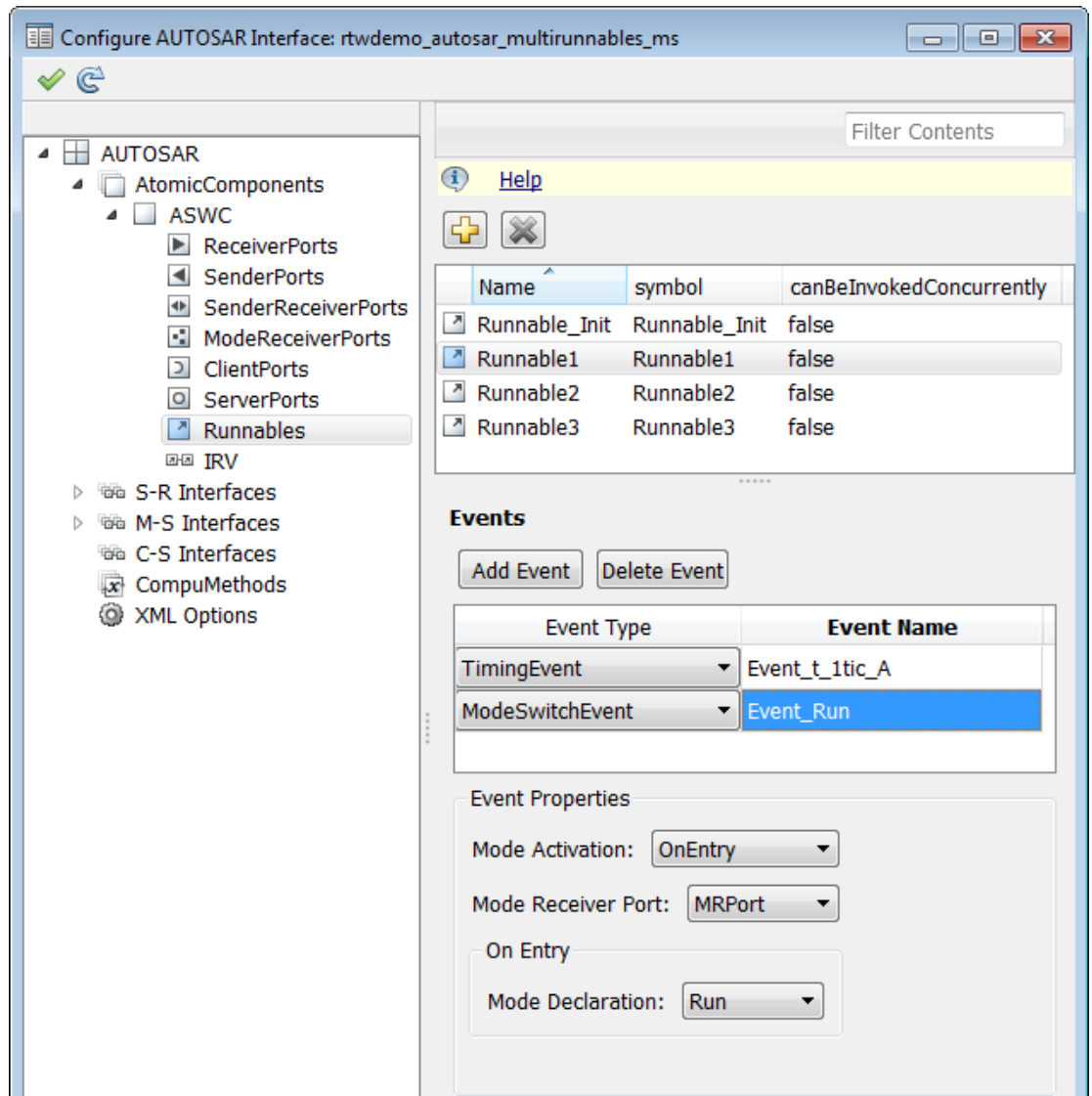
---

**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.

---

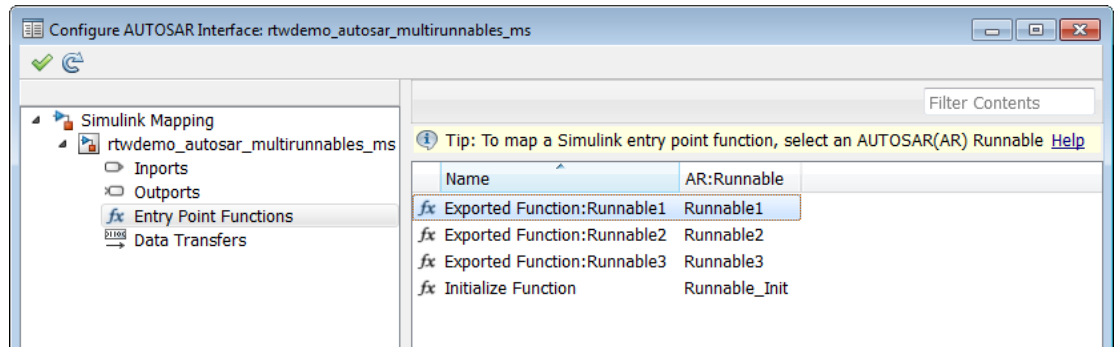
- To add an AUTOSAR mode-switch event for a runnable:


- 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 AUTOSAR Client-Server Communication

### In this section...

“Configure AUTOSAR Server” on page 4-92

“Configure AUTOSAR Client” on page 4-103

“Configure AUTOSAR Client-Server Error Handling” on page 4-113

“Concurrency Constraints for AUTOSAR Server Runnables” on page 4-118

“Configure and Map AUTOSAR Server and Client Programmatically” on page 4-121

You can model AUTOSAR servers and clients in Simulink for simulation and code generation. For information about the Simulink blocks you use and the high-level workflow, see “Client-Server Interface” on page 2-12.

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 section provides detailed examples of AUTOSAR server and client configuration that start from Simulink blocks.

## Configure AUTOSAR Server

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

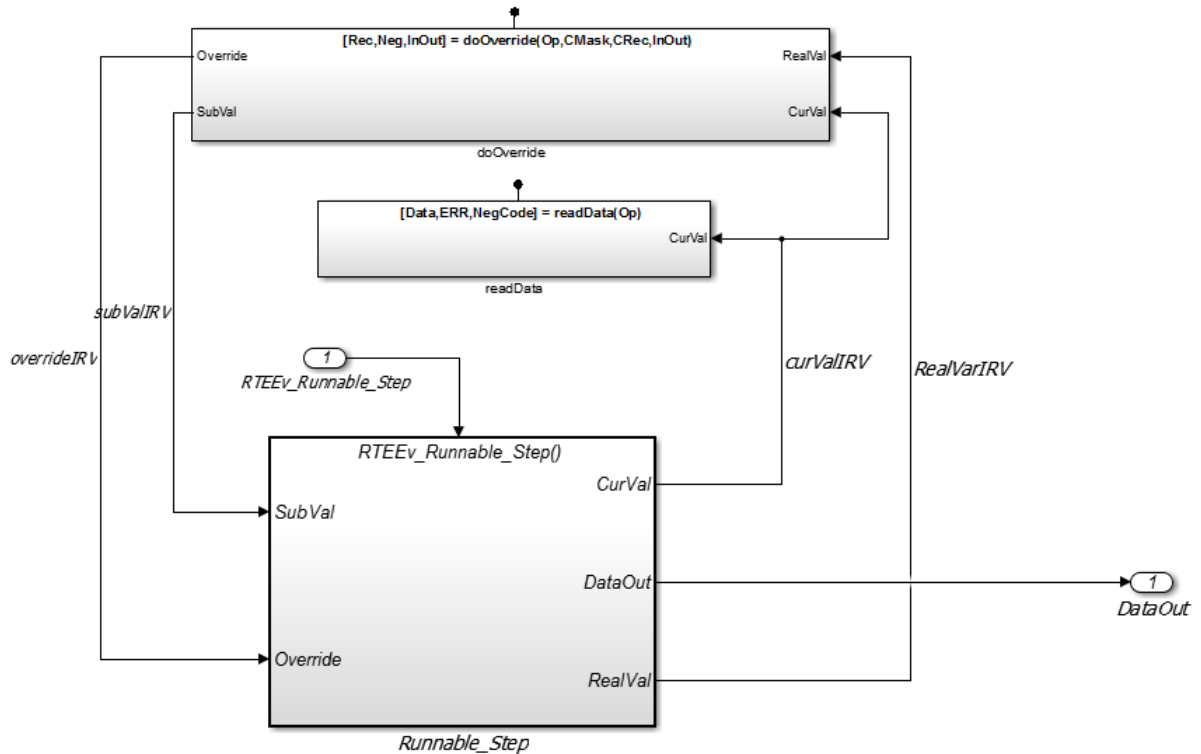
- `mControllerWithInterface_server.slx`
- `ExampleApplicationErrorType.m`

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**.

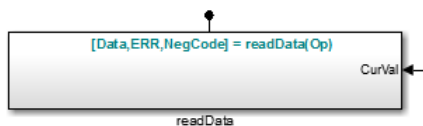


The example model provides two Simulink Function blocks, doOverride and readData.



- 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:

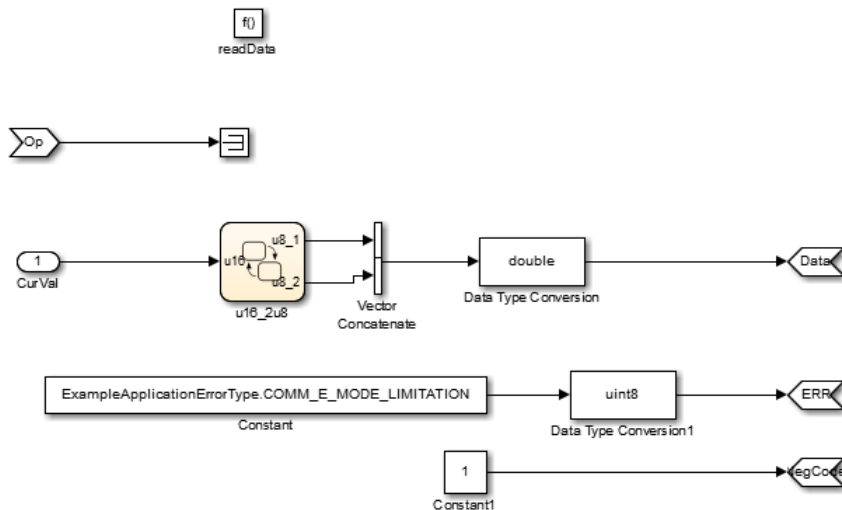
- Trigger block `readData`, representing a trigger port for the server function. In the Trigger block properties, **Trigger type** is set to `Function call`. Also, the option **Treat as Simulink function** is selected.
- Argument Inport block `Op` and Argument Outport blocks `Data`, `ERR`, and `NegCode`, 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.

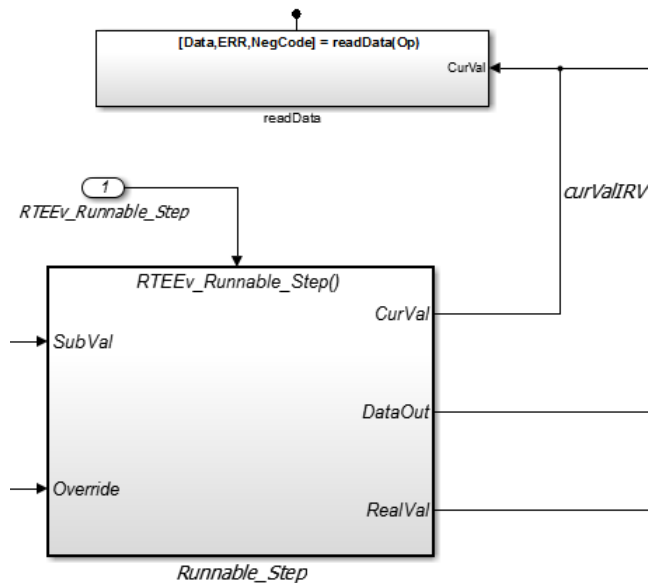
---

- 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.

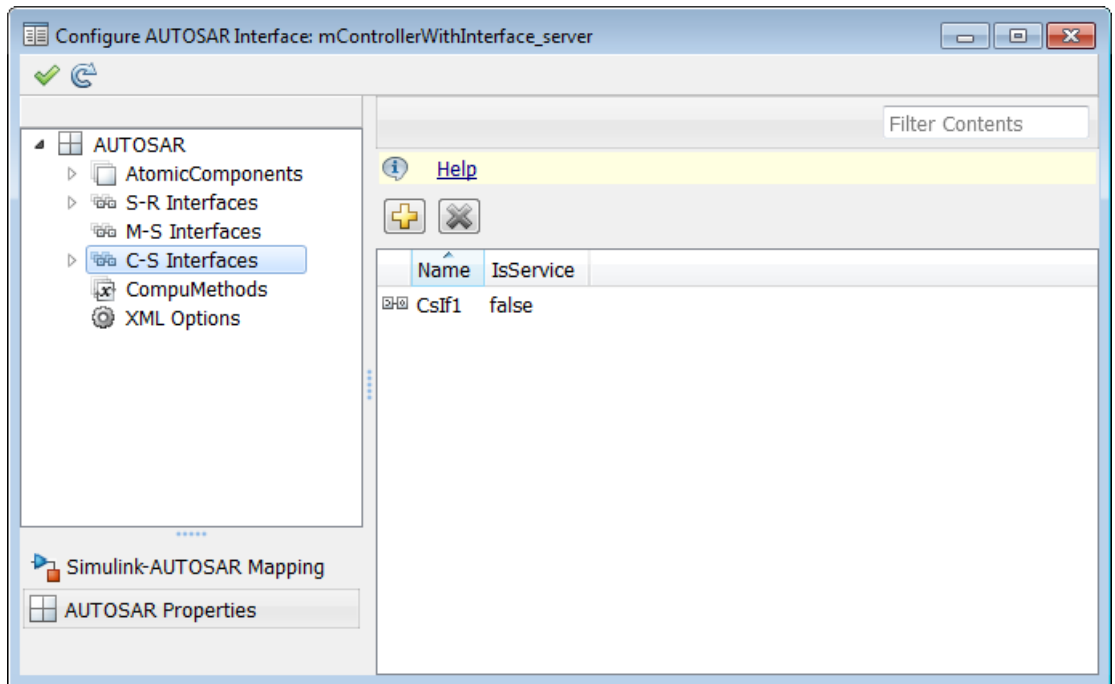
The example model is an AUTOSAR model, into which the Simulink Function block `readData` has been copied. In place of a meaningful `Op` input value for the `readData` function, Simulink data transfer line `CurVal` provides an input value that is used in the function algorithm.




- 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

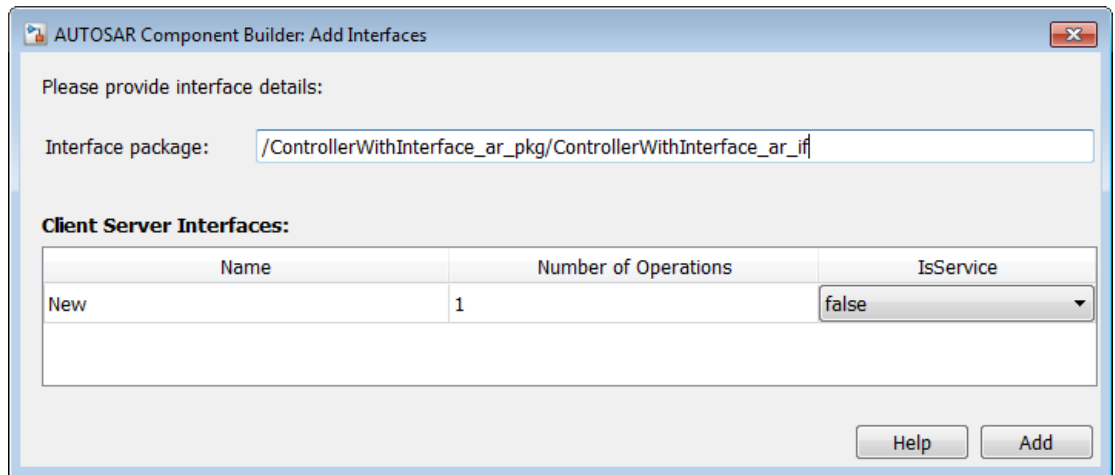
To view the AUTOSAR configuration for the model, in the model window, select **Code > C/C++ Code > Configure Model as AUTOSAR Component**. This action opens the Configure AUTOSAR Interface dialog box.

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 a client-server interface named **CsIf1**.



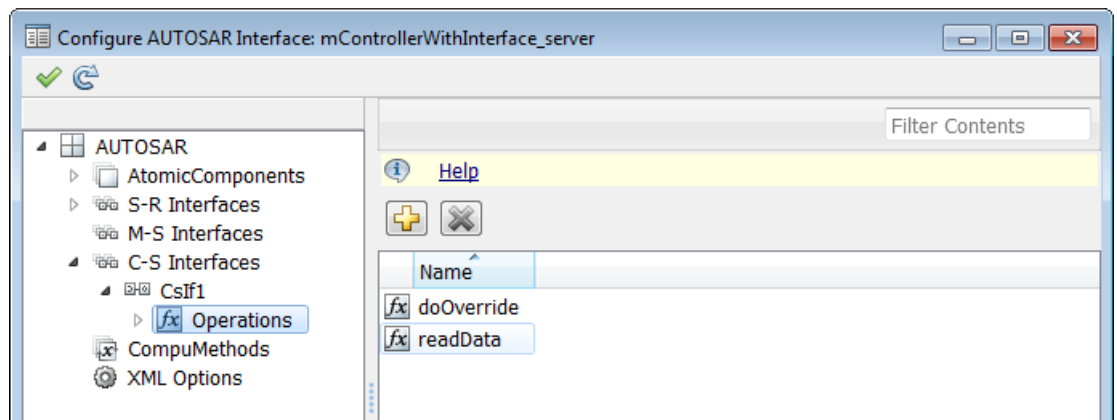
If a C-S interface does not yet exist in your model, create one.

- 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.




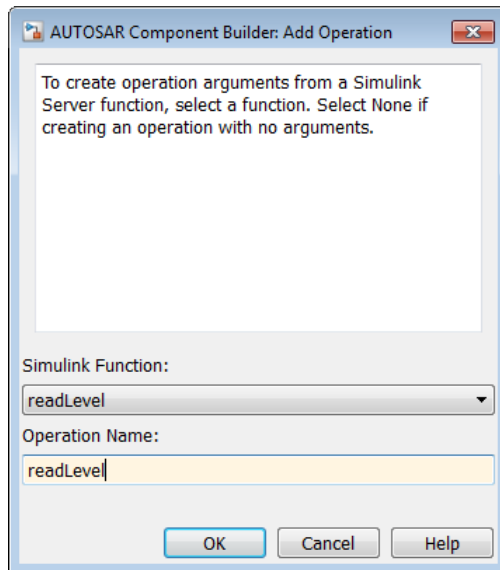
- Under **C-S Interfaces**, create one or more AUTOSAR server operations for which the C-S interface handles client requests. Each operation corresponds to a Simulink server function in the model.

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**.

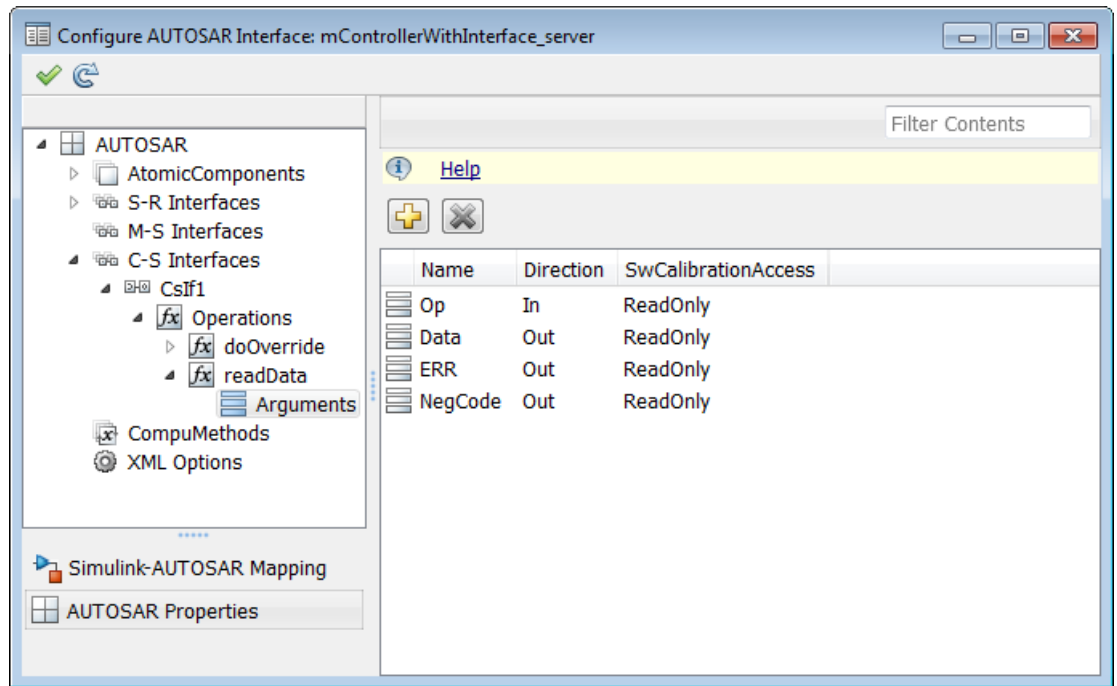


If a server operation does not yet exist in your model, create one. (If there is a placeholder operation named `Operation1` in your C-S interface, you can safely delete it.)

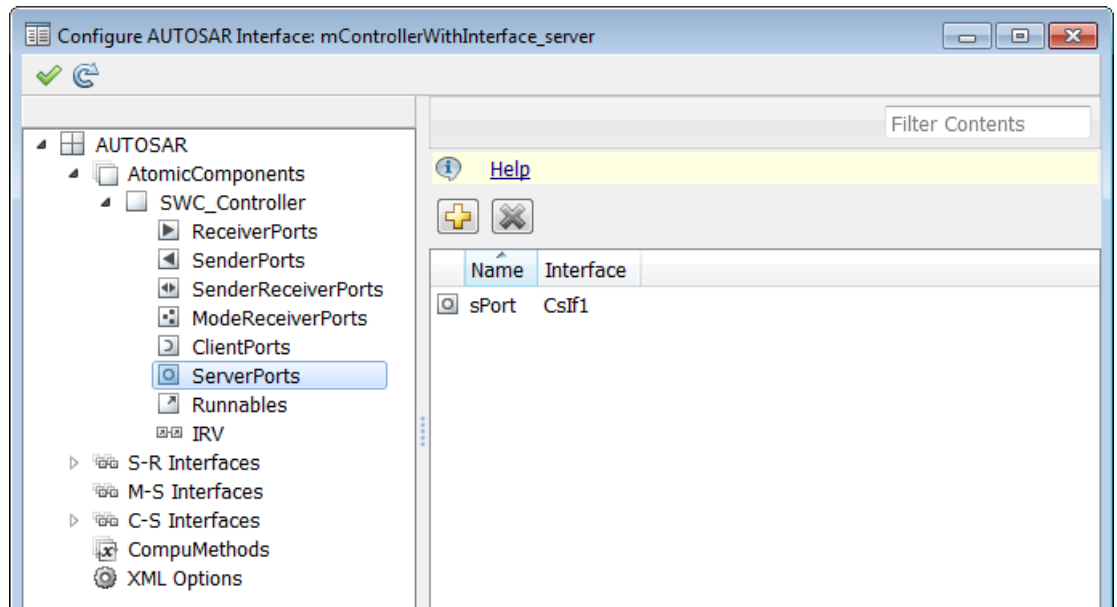
- a In the Operations view, click the **Add** icon . This action opens the Add Operation dialog box.
- b In the dialog box, enter the **Operation Name**. Specify the name of the corresponding Simulink server function.
- c If the corresponding Simulink server function has arguments, select the function 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.



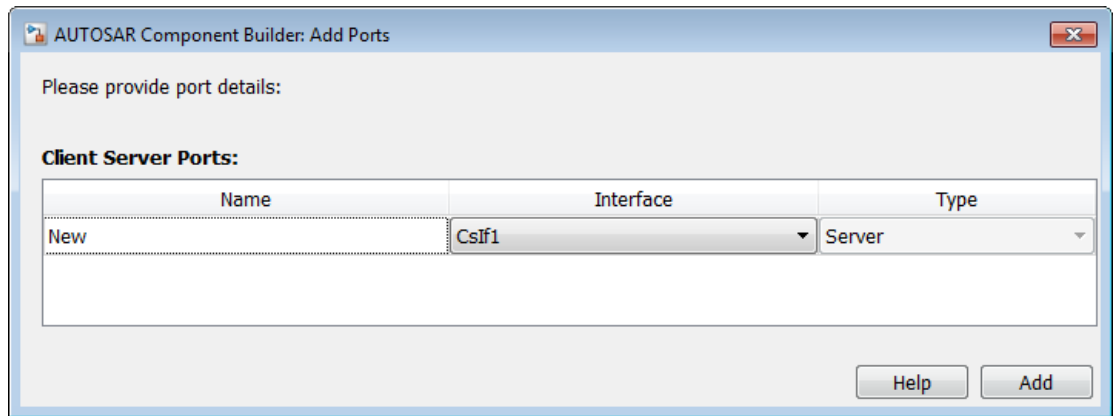
- 8 To view AUTOSAR server ports in the model, go to the ServerPorts view. Expand **AtomicComponents**, expand the individual component that you are configuring, and select **ServerPorts**. The example model already contains an AUTOSAR server port named **sPort**.



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.



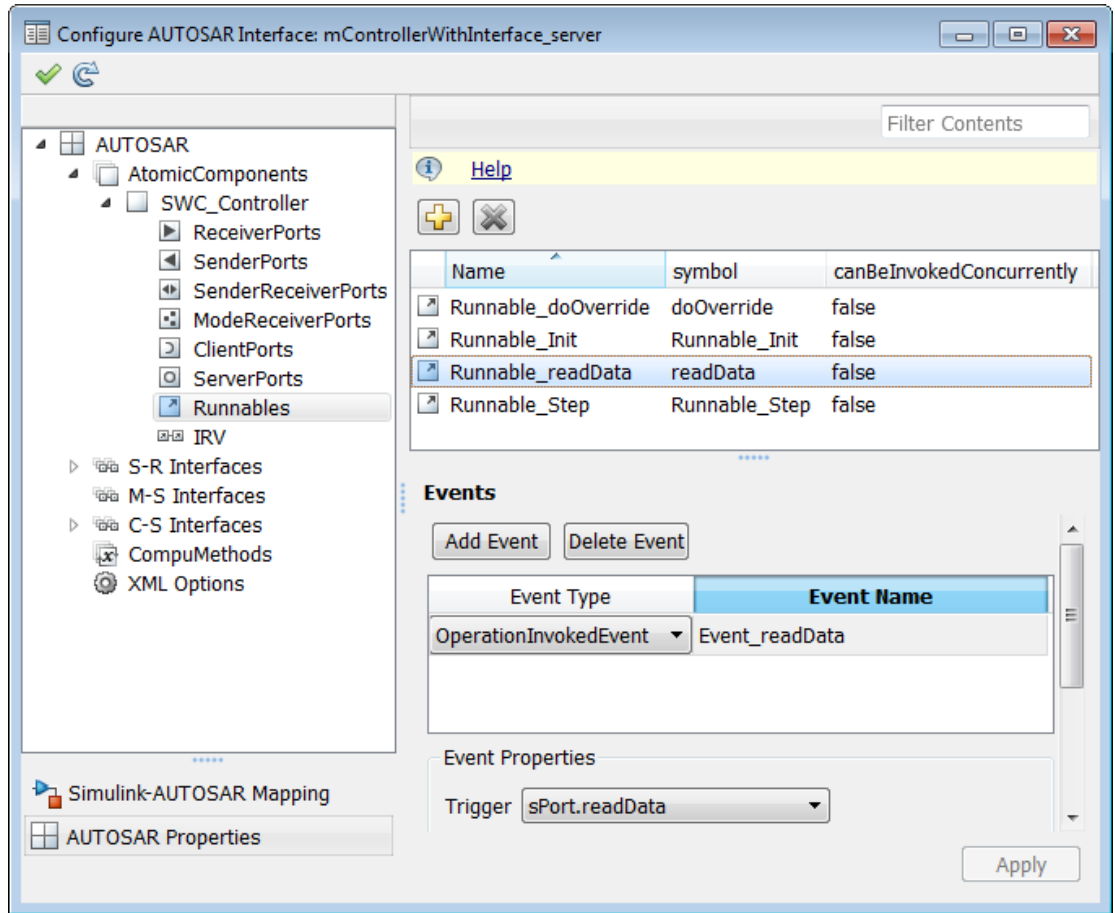


- 9 For each AUTOSAR server operation, configure an AUTOSAR server runnable to execute client requests. To view AUTOSAR runnables in the model, select **Runnables**. The example model already contains a server runnable for `readData`, named `Runnable_readData`.

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

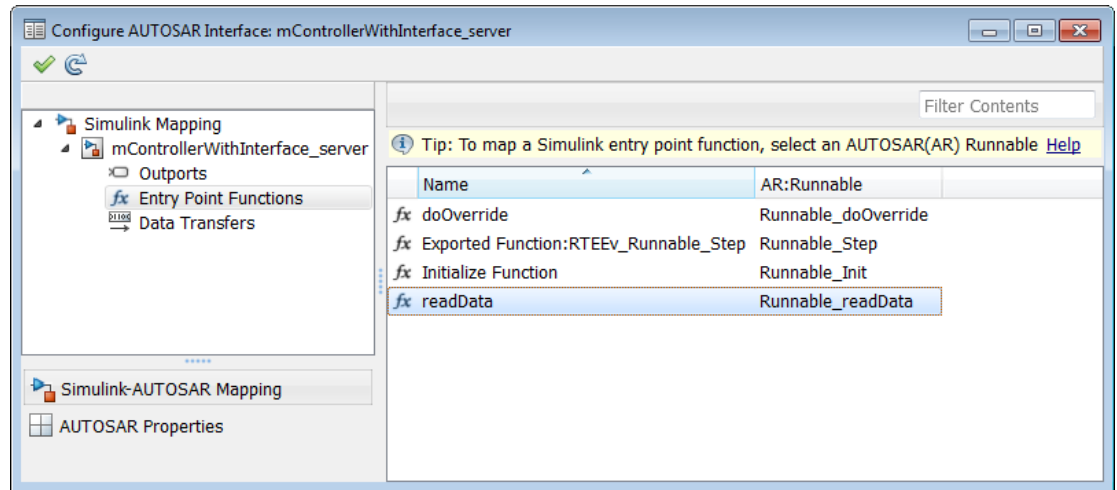
sPort.readData, combining server port sPort with operation readData.)  
Click **Apply**.




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.

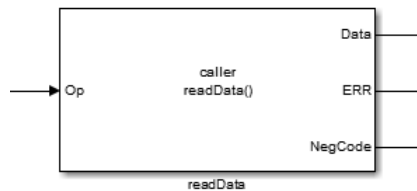
After you configure an AUTOSAR server, configure a corresponding AUTOSAR client invocation, as described in “Configure AUTOSAR Client” on page 4-103.

## Configure AUTOSAR Client

After you configure an AUTOSAR server, as described in “Configure AUTOSAR Server” on page 4-92, 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/client_server/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`.

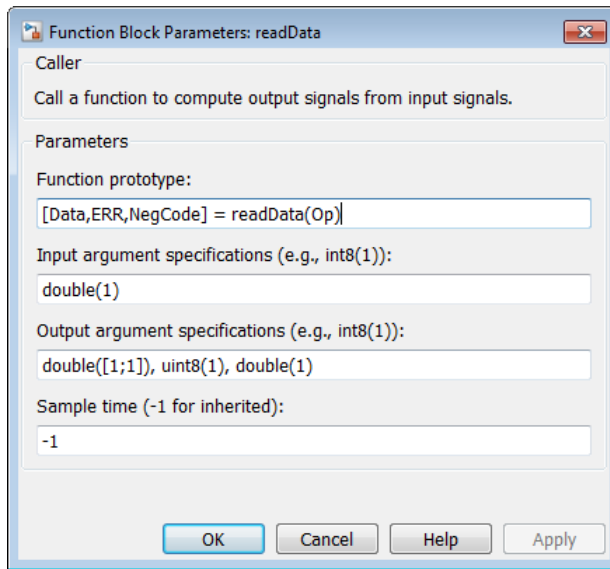


- 3 Configure the Function Caller block to call a corresponding Simulink Function block. Double-click the block to open it, and edit the block parameters to specify the server function prototype.

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 4-92. 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 4-92. 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.

---

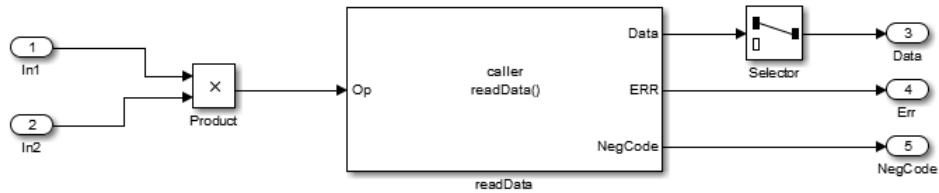
- 4 When the function invocation is completely formed in Simulink, set up the Function Caller block in a model configured for AUTOSAR. For example, configure the current model for AUTOSAR or copy the block into an AUTOSAR model.


---

**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 3-11.

---

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.

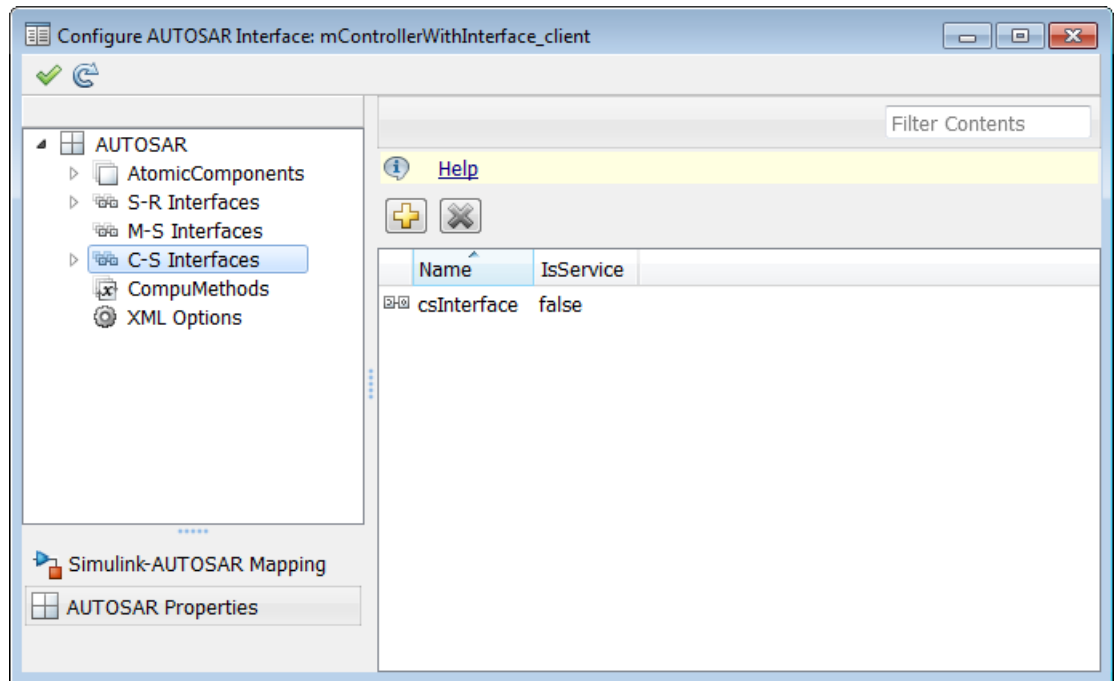


**Note:** Whenever you add or change a Function Caller block in an AUTOSAR model, synchronize function callers in the AUTOSAR configuration. If the Configure AUTOSAR Interface dialog box is not already open, in the model window, select **Code > C/C++ Code > Configure Model as AUTOSAR Component**. In 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

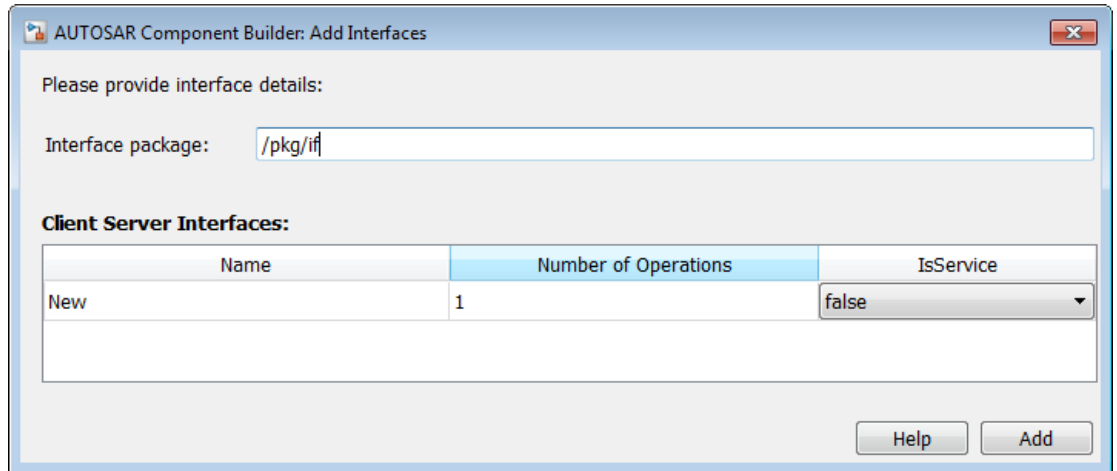
To view the AUTOSAR configuration for the model, in the model window, select **Code > C/C++ Code > Configure Model as AUTOSAR Component**. This action opens the Configure AUTOSAR Interface dialog box.

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`.



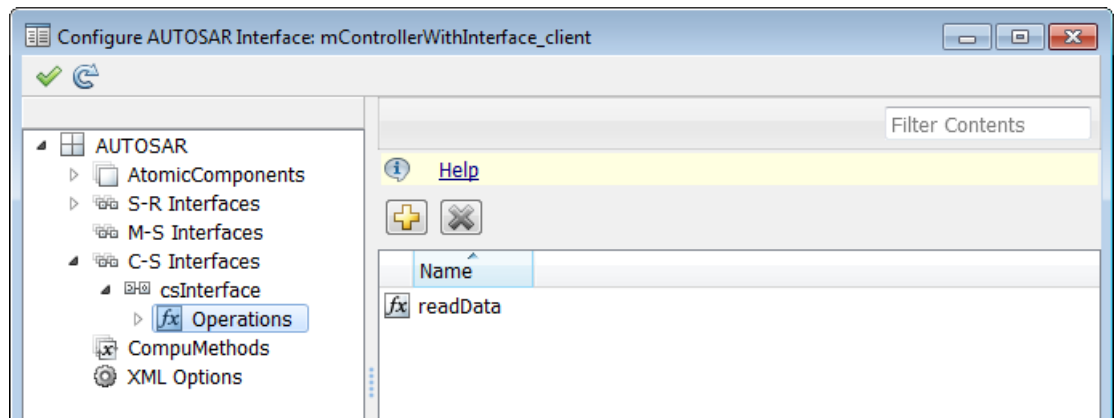
If a C-S interface does not yet exist in the AUTOSAR configuration, create one.

- 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.





- Under **C-S Interfaces**, create one or more AUTOSAR operations matching the Simulink server functions that you defined in the AUTOSAR server model.

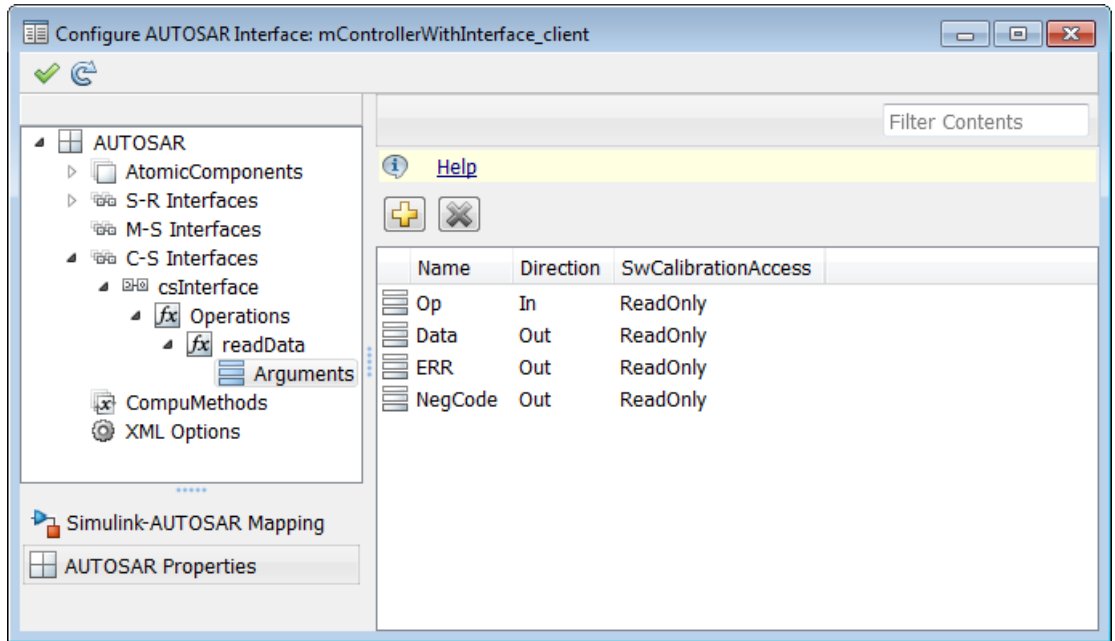
Expand **C-S Interfaces**, and expand the individual C-S interface to which you want to add an AUTOSAR operation. (In the example model, expand `csInterface`.) To view operations for the interface, select **Operations**. The example model already contains an AUTOSAR operation named `readData`.



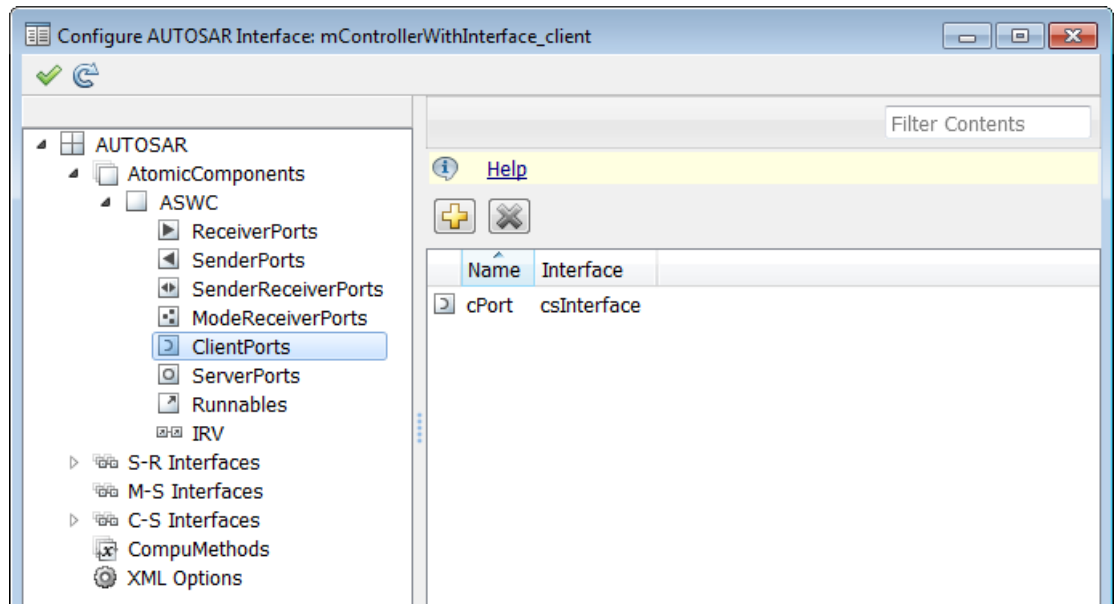


If an AUTOSAR operation does not yet exist in your model, create one. (If there is a placeholder operation named `Operation1` in your C-S interface, you can safely delete it.)

- 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.

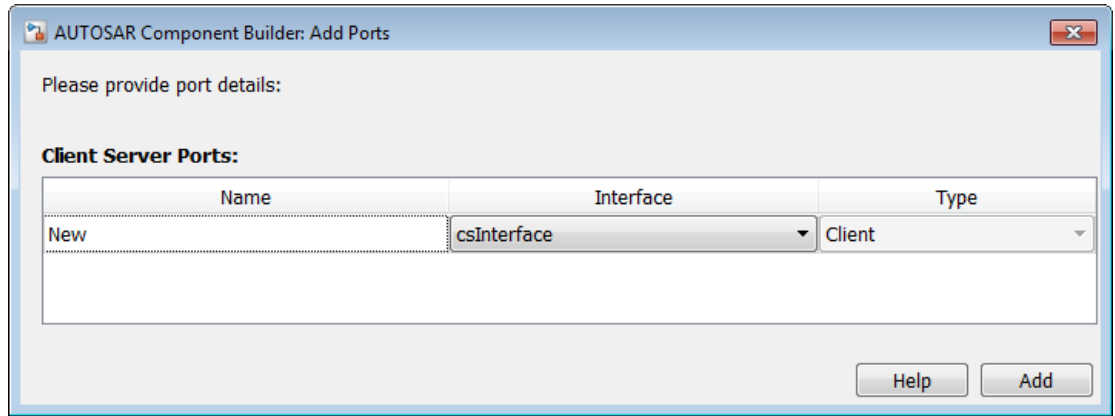


- 8 To view AUTOSAR client ports in the model, go to the ClientPorts view. Expand **AtomicComponents**, expand the individual component that you are configuring, and select **ClientPorts**. The example model already contains an AUTOSAR client port named cPort.



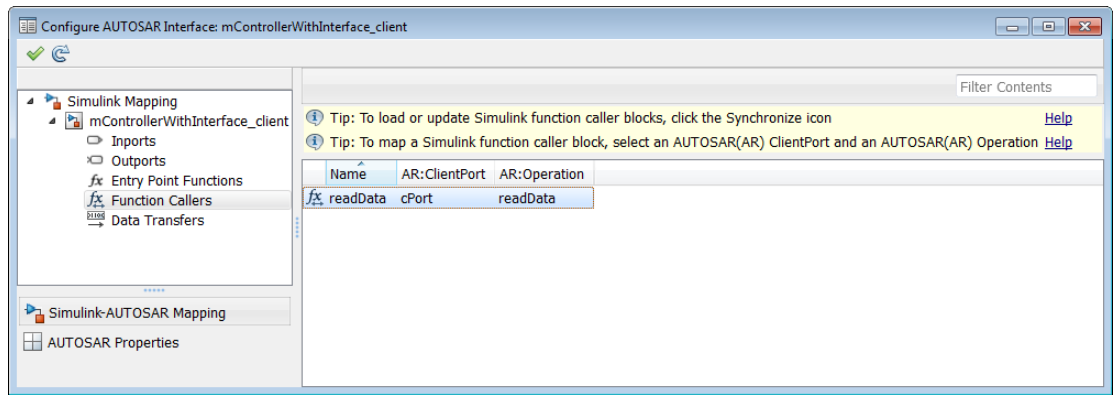
If a client port does not yet exist in your model, create one.


- 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

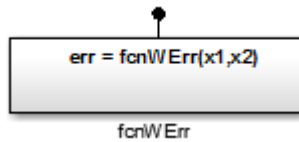
        function dataScope = getDataScope()
            dataScope = 'Exported';
        end
    end
end
```

---

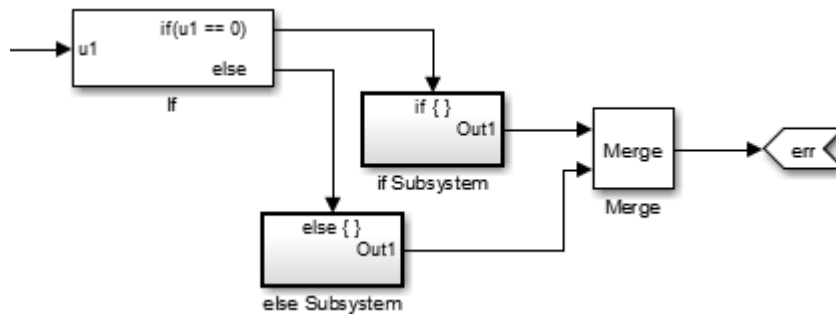
**Note:** The Simulink data type that you define to represent possible errors in the model does not directly affect the AUTOSAR possible errors that are imported and exported in `arxml` code. To modify the exported possible errors for a C-S interface or C-S operation, use AUTOSAR properties functions. This topic provides examples.

---

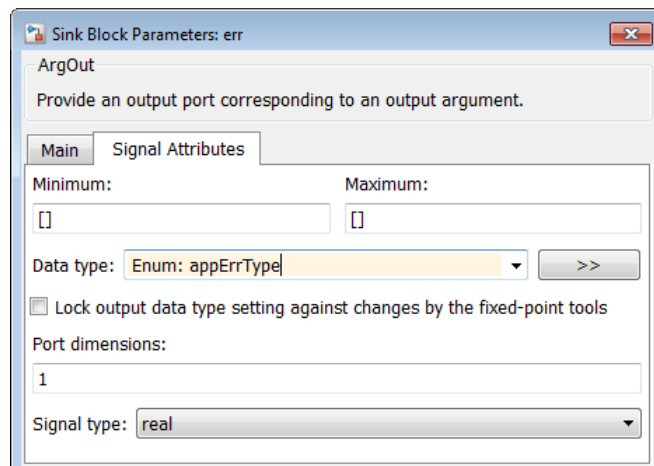
- 2 Define an error status output argument for the Simulink Function block that models the AUTOSAR server. Configure the error status argument as the only function output or add it to other outputs. For example, here is a Simulink Function block that returns an error status value in output `err`.




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

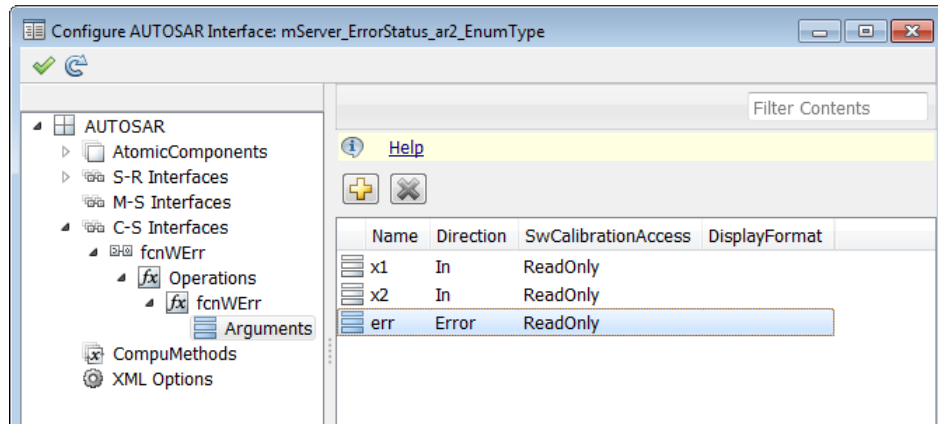


- Reference the possible error values type in the model. In the Argument Output block parameters for the error output, specify the error status data type, in this case, `appErrType`. Set **Port dimensions** to 1 and **Signal type** to real.

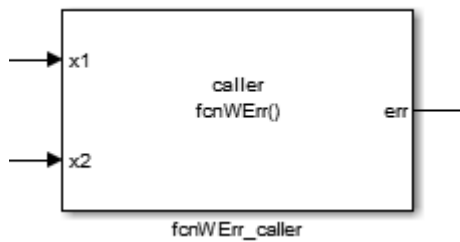


- Configure the AUTOSAR properties of the error argument in the client-server interface. Open the Configure AUTOSAR Interface dialog box, select **AUTOSAR**

**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.

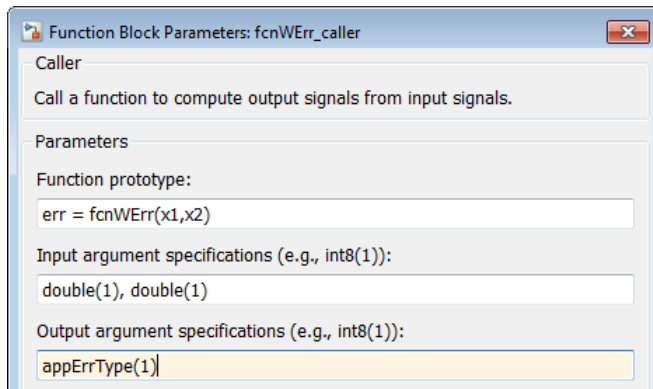


- 5 Create an error port in each Function-Caller block that models an AUTOSAR client invocation. For example, here is a Function-Caller block that models an invocation of fcnWErr.



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;
}
```

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 CUSTOM to the possible errors for a C-S interface named AppSrvr1.

```
>> apiObj = autosar.api.getAUTOSARProperties(bdroot)
>> get(apiObj, 'AppSrvr1', 'PossibleError')
ans =
    'AppSrvr1/SUCCESS'    'AppSrvr1/ERROR'    'AppSrvr1/COMM_MODE_L'
    'AppSrvr1/OVERFLOW'  'AppSrvr1/UNDERFLOW'  'AppSrvr1/VALUE_MOD3'
>> get(apiObj, 'AppSrvr1/ERROR', 'errorCode')
ans =
    1
>> add(apiObj, 'AppSrvr1', 'PossibleError', 'CUSTOM')
>> set(apiObj, 'AppSrvr1/CUSTOM', 'errorCode', 10)
>> get(apiObj, 'AppSrvr1/CUSTOM', 'errorCode')
ans =
    10
```

You can also access possible errors on a C-S operation. The following example lists possible errors for operation AppSrvr1 on C-S interface AppSrvr1.

```
>> apiObj = autosar.api.getAUTOSARProperties(bdroot)
>> get(apiObj, 'AppSrvr1/AppSrvr1', 'PossibleError')
ans =
    'AppSrvr1/SUCCESS'    'AppSrvr1/ERROR'    'AppSrvr1/COMM_MODE_L'
    'AppSrvr1/OVERFLOW'  'AppSrvr1/UNDERFLOW'  'AppSrvr1/VALUE_MOD3'
```

The following example sets the direction of C-S operation argument ERR\_STATUS to Error.

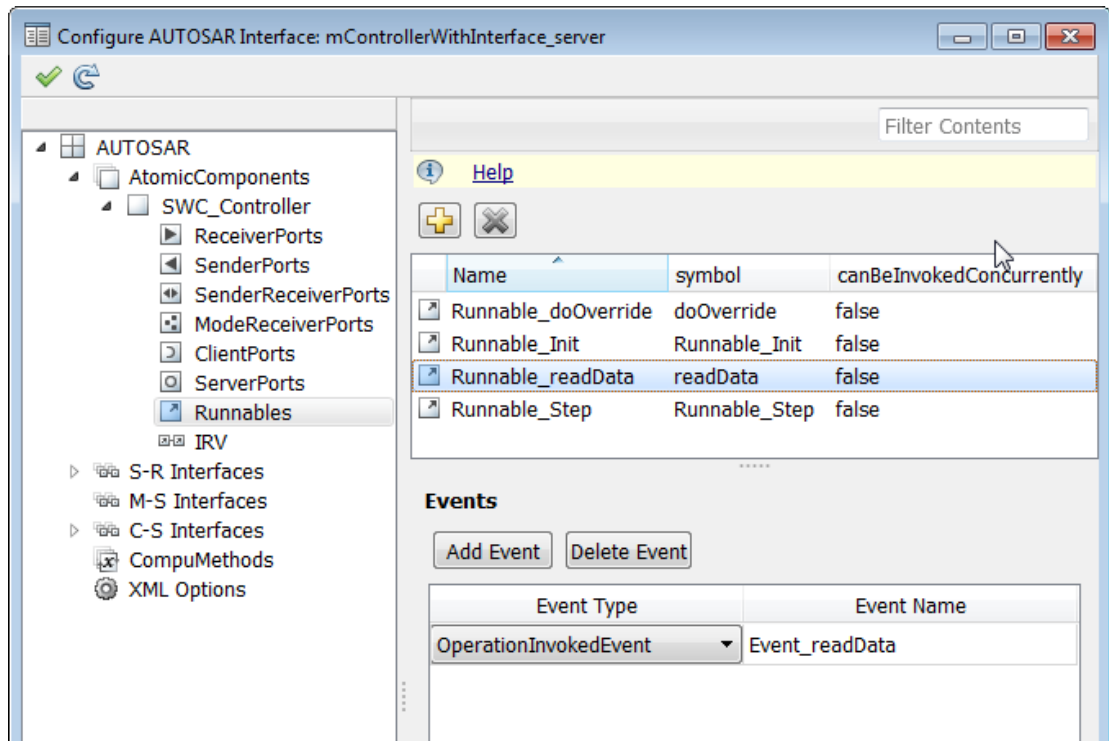
```
>> apiObj = autosar.api.getAUTOSARProperties(bdroot)
>> set(apiObj, 'AppSrvr1/AppSrvr1/ERR_STATUS', 'Direction', 'Error')
>> get(apiObj, 'AppSrvr1/AppSrvr1/ERR_STATUS', 'Direction')
ans =
    Error
```

## 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<sup>®</sup> 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 Simulink to AUTOSAR mapping information for AUTOSAR clients and servers, use these functions:

- `getFunction`
- `getFunctionCaller`
- `mapFunction`
- `mapFunctionCaller`

For an example of configuring AUTOSAR mapping information for a Simulink Function Caller block, see the `mapFunctionCaller` reference page.

To configure AUTOSAR properties of AUTOSAR client-server interfaces, use AUTOSAR property functions such as `set` and `get`.

## See Also

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

## Related Examples

- “Client-Server Interface” on page 2-12

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

### **More About**

- “Model AUTOSAR Communication” on page 2-10
- “AUTOSAR Interface Configuration” on page 4-3

## Configure AUTOSAR 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 4-125.

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

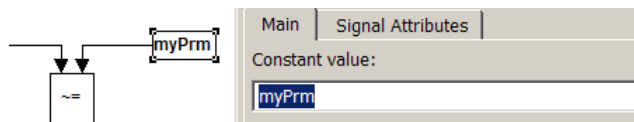
- 1 In the base workspace, create an `AUTOSAR.Parameter` object. For example:

```
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** — For information about creating a data type, for example, a bus object data type, 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:



- 4 Open the Configuration Parameters dialog box and clear the option **Code Generation > Ignore custom storage classes**.
- 5 Generate code.

---

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

---



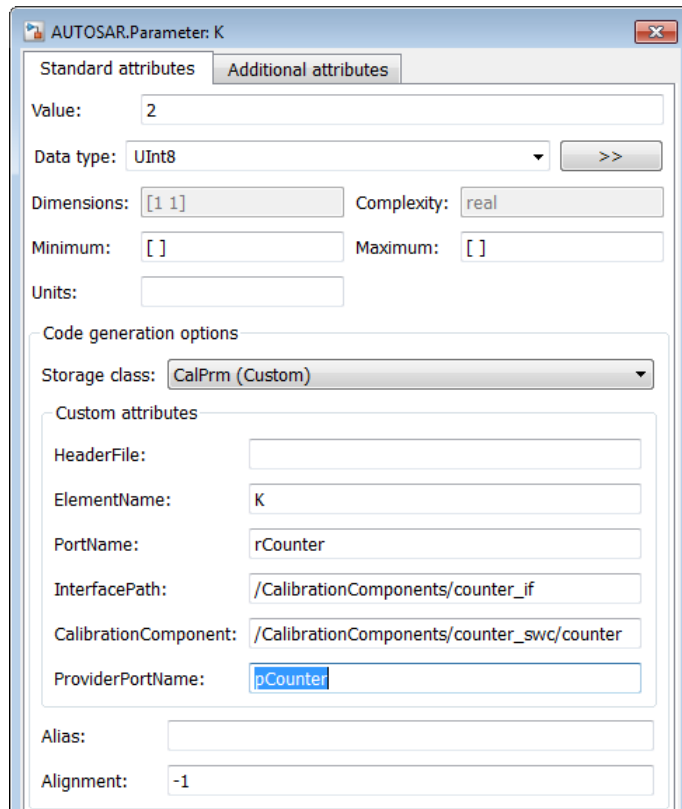
## Configure AUTOSAR Calibration Component

An AUTOSAR calibration 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 new 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`.



AUTOSAR.Parameter: K

Standard attributes    Additional attributes

Value: 2

Data type: UInt8 >>

Dimensions: [1 1]    Complexity: real

Minimum: [ ]    Maximum: [ ]

Units:

Code generation options

Storage class: CalPrm (Custom)

Custom attributes

HeaderFile:

ElementName: K

PortName: rCounter

InterfacePath: /CalibrationComponents/counter\_if

CalibrationComponent: /CalibrationComponents/counter\_sw/counter

ProviderPortName: pCounter

Alias:

Alignment: -1

### 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_sw</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 COM\_AXIS Lookup Table for Measurement and Calibration

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 COM\_AXIS Lookup Tables” on page 2-21.

To create and configure a COM\_AXIS lookup table:

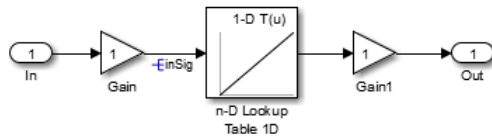
- 1** Add an **n-D Lookup Table** block to your model.
- 2** If AUTOSAR calibration parameters for the lookup table are not already present in your model, create them. They can include:
  - Internal calibration parameters, modeled as described in “Configure AUTOSAR Calibration Parameters” on page 4-123.
  - Calibration parameters in a calibration component, modeled as described in “Configure AUTOSAR Calibration Component” on page 4-125.
  - Constant memory parameters, modeled as described in “Configure AUTOSAR Static or Constant Memory” on page 4-182.
- 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 4-182.
  - 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 Replacement for AUTOSAR” 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
DocUnits: ''
Complexity: 'real'
Dimensions: [1 5]
```

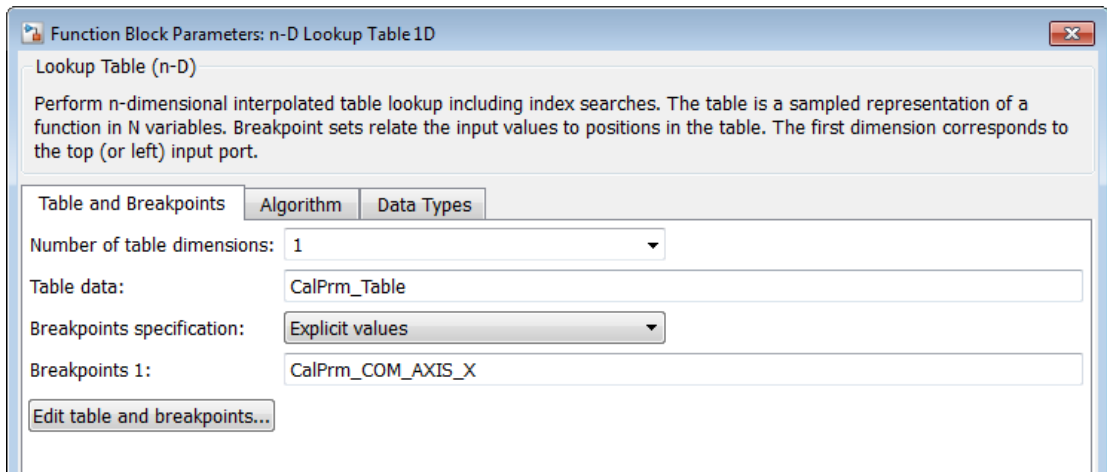
`CalPrm_COM_AXIS_X` represents axis data.

`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
DocUnits: ''
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/AppIDataTypes/App1_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\_App1\_CalPrm\_Table.

```
<APPLICATION-PRIMITIVE-DATA-TYPE UUID="...">
  <SHORT-NAME>App1_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/AppIDataTypes/App1_CalPrm_COM_AXIS_X</SHARED-AXIS-TYPE-REF>
            </SW-AXIS-GROUPED>
          </SW-CALPRM-AXIS>
        </SW-CALPRM-AXIS-SET>
      </SW-DATA-DEF-PROPS-CONDITIONAL>
    </SW-DATA-DEF-PROPS-VARIANTS>
  </SW-DATA-DEF-PROPS>
</APPLICATION-PRIMITIVE-DATA-TYPE>
```

```

        </SW-CALPRM-AXIS>
    </SW-CALPRM-AXIS-SET>
    ...
    <SW-RECORD-LAYOUT-REF DEST="SW-RECORD-LAYOUT">/MW/RecordLayouts/
        RL_App1_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 4-182. For example:

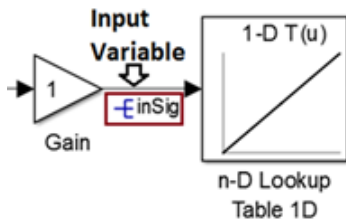
inSig =

Signal with properties:

```

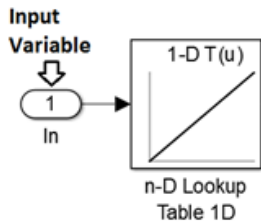
SwCalibrationAccess: 'ReadWrite'
DisplayFormat: ''
CoderInfo: [1x1 Simulink.CoderInfo]
Description: ''
DataType: 'fixdt(1,16,0.1,0)'
Min: []
Max: []
DocUnits: ''
Dimensions: -1
DimensionsMode: 'auto'
Complexity: 'auto'
SampleTime: -1
InitialValue: ''

```



- 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/App1DataTypes/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 3-2
- “Model AUTOSAR Calibration Parameters” on page 2-19
- “Configure AUTOSAR Calibration Parameters” on page 4-123
- “Configure AUTOSAR Calibration Component” on page 4-125
- “Configure AUTOSAR Static or Constant Memory” on page 4-182
- “Code Replacement for AUTOSAR” on page 5-8
- “Export AUTOSAR Component XML and C Code” on page 5-2

### **More About**

- “Calibration Parameters for COM\_AXIS Lookup Tables” on page 2-21
- “Static and Constant Memory” on page 2-38
- “AUTOSAR Interface Configuration” on page 4-3

## Configure AUTOSAR Data for Measurement and Calibration

### In this section...

“About Software Data Definition Properties (SwDataDefProps)” on page 4-135

“Configure SwCalibrationAccess” on page 4-135

“Configure swAddrMethod” on page 4-139

“Configure swAlignment” on page 4-141

“swImplPolicy for Exported Data” on page 4-142

### 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.
- 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, with respect to consistency mechanisms of variables.

In the Simulink environment, you can directly modify the SwCalibrationAccess, swAddrMethod, and swAlignment properties for some forms of AUTOSAR data.

You cannot modify the swImplPolicy property, but the property is set to `standard` or `queued` for AUTOSAR data in exported arxml code.

For more information, see “Configure SwCalibrationAccess” on page 4-135, “Configure swAddrMethod” on page 4-139, “Configure swAlignment” on page 4-141, and “swImplPolicy for Exported Data” on page 4-142

### 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 4-136
- “Specify **SwCalibrationAccess** for Signal and Parameter Data Objects” on page 4-138
- “Specify Default **SwCalibrationAccess** for Application Data Types” on page 4-139

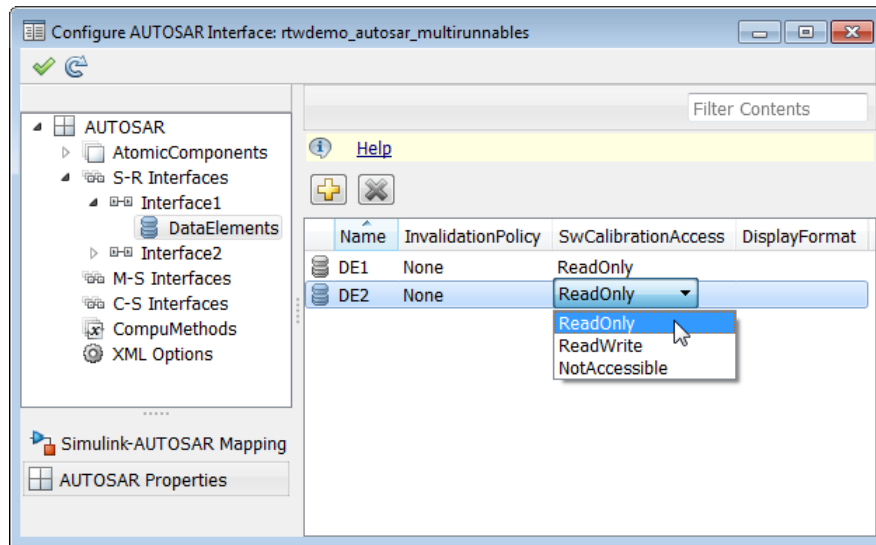
### Specify **SwCalibrationAccess** for AUTOSAR Data Elements

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 view.
- 4 Navigate to the **Data Elements** view under an S-R interface, the **Data Elements** view under an NV interface, the **Arguments** view under a C-S interface, or to the **IRV** view under an atomic component.
- 5 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 `IRV1` to `ReadWrite`.

```
>> open_system('rtwdemo_autosar_multirunnables')
>> dataobj = autosar.api.getAUTOSARProperties('rtwdemo_autosar_multirunnables');
>> get(dataobj, '/pkg/swc/ASWC/Behavior/IRV1', 'SwCalibrationAccess')

ans =
ReadOnly

>> set(dataobj, '/pkg/swc/ASWC/Behavior/IRV1', 'SwCalibrationAccess', 'ReadWrite');
>> get(dataobj, '/pkg/swc/ASWC/Behavior/IRV1', '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

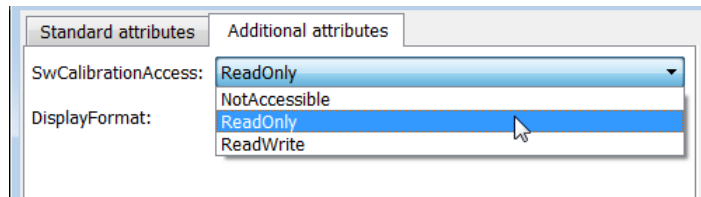
arSig =

  Signal with properties:

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

>>
```

- 3 Open the data object, for example, by double-clicking on the object in the workspace.
- 4 Select the **Additional attributes** tab (or for `AUTOSAR.DualScaledParameter`, the **Calibration Attributes** tab) and 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 `swAddrMethod`

In Simulink, you can specify or modify AUTOSAR software address methods for measurement and calibration tools in the following ways:

- “Edit `swAddrMethod` for AUTOSAR Static or Constant Memory” on page 4-140
- “Add, Find, or Set `swAddrMethod` for Data” on page 4-141

To support the round-trip workflow, the arxml importer imports and preserves the `swAddrMethod` property for the following AUTOSAR data:

- Per-instance memory
- Software component parameters
- Parameter interface data elements
- Client-server interface operation arguments

### **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, the following additional tokens can be used in AUTOSAR memory sections:

- `%<AUTOSAR_COMPONENT>` and `%<MemorySectionName>` — When specified in the Pre/Post Pragma fields within a Memory-Section definition, are expanded during C code generation.
- `MySwc_START_SEC_FLASHMEMORY` and `MySwc_STOP_SEC_FLASHMEMORY` — When specified with an include file within an AUTOSAR memory section, the include file is expanded to the corresponding memory section during the C preprocessor stage. These tokens allow you to separate platform-independent variable declarations from platform-dependent `#pragma` statements in an include file.

The code examples below 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 new `swAddrMethod` with the qualified name `qName` to the AUTOSAR configuration for a model, and if property-value pairs are specified, sets properties for the `swAddrMethod`.

The following example adds a new `swAddrMethod` to the AUTOSAR configuration for example model `rtwdemo_autosar_counter`, with qualified name `'/A/B/C/SwAddressMethods/sw1'`, with `MemoryAllocationKeywordPolicy` set to `'ADDR-METHOD-SHORT-NAME'`, and with `SectionType` set 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 = 'E11';
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. If the property is not defined, the alignment is determined by the `swBaseType` size and the `memoryAllocationKeywordPolicy` of the referenced `swAlignment`. `swAlignment` defines a number of bits, and possible values include 8,

12, 32, UNKNOWN (deprecated), UNSPECIFIED, and BOOLEAN. For numeric data, typical `swAlignment` values are 8, 16, and 32.

You can use the AUTOSAR properties `set` function to set `swAlignment` for S-R interface data elements and inter-runnable variables. For example:

```
interfacePath = '/A/B/C/Interfaces/If1/';
dataElementName = 'E11';
swAlignmentValue = '32';
set(dataObj,[interfacePath dataElementName], 'SwAlignment', swAlignmentValue);
```

To support the round-trip workflow, the `arxml` importer imports and preserves the `swAlignment` property for the following AUTOSAR data:

- Per-instance memory
- Software component parameters
- Parameter interface data elements
- Client-server interface operation arguments
- Static and constant memory

### **swImplPolicy for Exported Data**

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

## Configure AUTOSAR Display Format for Measurement and Calibration

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)
- Interrunnable variables
- Sender-receiver interface data elements
- Client-server interface operation arguments
- CompuMethods

The display format string is a subset of ANSI C `printf` specifiers, with the following form:

```
%[flags][width][.precision]type
```

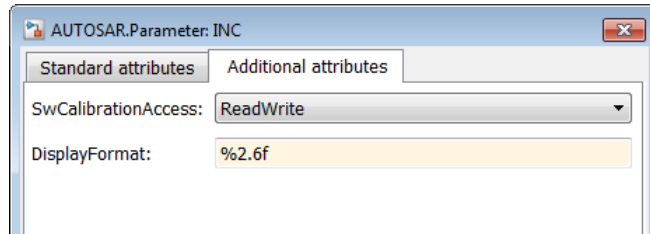
Field	Description
<b>flags</b> (optional)	String specifying flags supported by AUTOSAR schemas: <ul style="list-style-type: none"> <li>• ( ' ' ): Insert a space before the value.</li> <li>• -: Left-justify.</li> <li>• +: Display plus or minus sign, even for positive numbers.</li> <li>• #:                             <ul style="list-style-type: none"> <li>• For types o, x, and X, display 0, 0x, or 0X prefix.</li> <li>• For types e, E, and f, display decimal point even if the precision is 0.</li> <li>• For types g and G, do not remove trailing zeros or decimal point.</li> </ul> </li> </ul>
<b>width</b> (optional)	Positive integer specifying the minimum number of characters to display.
<b>precision</b> (optional)	Positive integer specifying the precision to display: <ul style="list-style-type: none"> <li>• For integer type values (d, i, o, u, x, and X), specifies the minimum number of digits.</li> <li>• For types e, E, and f, specifies the number of digits to the right of the decimal point.</li> <li>• For types g and G, specifies the number of significant digits.</li> </ul>

Field	Description
type	<p>String specifying a numeric conversion type supported by AUTOSAR schemas:</p> <ul style="list-style-type: none"> <li>• <b>d</b>: Signed decimal integer.</li> <li>• <b>i</b>: Signed decimal integer.</li> <li>• <b>o</b>: Unsigned octal integer.</li> <li>• <b>u</b>: Unsigned decimal integer.</li> <li>• <b>x</b>: Unsigned hexadecimal integer, using characters "abcdef".</li> <li>• <b>X</b>: Unsigned hexadecimal integer, using characters "ABCDEF".</li> <li>• <b>e</b>: Signed floating-point value in exponential notation. The value has the form <code>[ - ]d.dddd e [ sign]ddd</code>. <ul style="list-style-type: none"> <li>• <b>d</b> is a single decimal digit.</li> <li>• <b>dddd</b> is one or more decimal digits.</li> <li>• <b>ddd</b> is exactly three decimal digits.</li> <li>• <b>sign</b> is + or -.</li> </ul> </li> <li>• <b>E</b>: Identical to the <b>e</b> format except that <b>E</b>, rather than <b>e</b>, introduces the exponent.</li> <li>• <b>f</b>: Signed floating-point value in fixed-point notation. The value has the form <code>[ - ]dddd.dddd</code>. <ul style="list-style-type: none"> <li>• <b>dddd</b> is one or more decimal digits.</li> <li>• The number of digits before the decimal point depends on the magnitude of the number.</li> <li>• The number of digits after the decimal point depends on the requested precision.</li> </ul> </li> <li>• <b>g</b>: Signed value printed in <b>f</b> or <b>e</b> 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.</li> <li>• <b>G</b>: Identical to the <b>g</b> format, except that <b>E</b>, rather than <b>e</b>, introduces the exponent (where appropriate).</li> </ul>

For example, the format string `%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>
```

### See Also

[AUTOSAR.DualScaledParameter](#) | [AUTOSAR.Parameter](#) | [AUTOSAR.Signal](#) | [AUTOSAR4.Parameter](#) | [AUTOSAR4.Signal](#)

### Related Examples

- “Import AUTOSAR Software Component” on page 3-2
- “Model AUTOSAR Calibration Parameters” on page 2-19
- “Configure AUTOSAR Data for Measurement and Calibration” on page 4-135
- “Configure AUTOSAR CompuMethods” on page 4-152

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

### **More About**

- “AUTOSAR Interface Configuration” on page 4-3

## Configure AUTOSAR Release 4.x Data Types

### In this section...

“Control Application Data Type Generation” on page 4-148

“Configure DataTypeMappingSet Package and Name” on page 4-149

“Initialize Data with ApplicationValueSpecification” on page 4-151

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 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. For more information, see the following topics.

For more information about modeling R4.x data types, see “Release 4.x Data Types” on page 2-29.

### Control Application Data Type Generation

For AUTOSAR data types created in Simulink, by default, the software generates application base types only for fixed-point data types and enumerated data types with storage types. If you want to override the default behavior for generating application types, you can configure the `arxml` exporter to generate an application type, along with the implementation type and base type, for each exported AUTOSAR data type. Use the XML options parameter **ImplementationDataTypeReference** (`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', 'ImplementationDataTypeReference', 'NotAllowed');
get(arProps, 'XmlOptions', 'ImplementationDataTypeReference')
```

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', 'ComponentQualifiedNames');
ib = get(propObj, swc, 'Behavior', 'PathType', 'FullyQualified');
set(propObj, ib, 'DataTypeMapping', newMappingSet);

% Force generation of application data types
set(propObj, 'XmlOptions', 'ImplementationTypeReference', 'NotAllowed');

% Build
rtwbuild(modelName);
```

The exported arxml code uses the specified package and name, as shown below.

```
<INTERNAL-BEHAVIORS>
  <SWC-INTERNAL-BEHAVIOR UUID="...">
    <SHORT-NAME>IB</SHORT-NAME>
    <DATA-TYPE-MAPPING-REFS>
      <DATA-TYPE-MAPPING-REF DEST="DATA-TYPE-MAPPING-SET">
        /myPkg/mySubpkg/MyMappingSets/MappingSetName</DATA-TYPE-MAPPING-REF>
    </DATA-TYPE-MAPPING-REFS>
```

```
...  
</SWC-INTERNAL-BEHAVIOR>  
</INTERNAL-BEHAVIORS>
```

## Initialize Data with ApplicationValueSpecification

To initialize AUTOSAR data objects typed by application data type, R4.1 requires AUTOSAR application value specifications (**ApplicationValueSpecifications**). Embedded Coder provides the following support:

- The **arxml** importer uses **ApplicationValueSpecifications** found in imported **arxml** files to initialize the corresponding data objects in the Simulink model.
- If you select AUTOSAR schema 4.0 or later for a model that contains AUTOSAR parameters typed by application data type, code generation exports **arxml** code that uses **ApplicationValueSpecifications** to specify initial values for AUTOSAR data.

For AUTOSAR parameters typed by implementation data type, code generation exports **arxml** code that uses **NumericalValueSpecifications** and (for enumerated types) **TextValueSpecifications** to specify initial values. If initial values for parameters specify multiple values, generated code uses **ArrayValueSpecifications**.

# Configure AUTOSAR CompuMethods

### In this section...

“Configure AUTOSAR CompuMethod Properties” on page 4-152

“Create AUTOSAR CompuMethods” on page 4-154

“Configure CompuMethod Direction for Linear Functions” on page 4-155

“Export CompuMethod Unit References” on page 4-157

“Configure Rational Function CompuMethod for Dual-Scaled Parameter” on page 4-157

Embedded Coder software imports AUTOSAR computational methods (CompuMethods) 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.

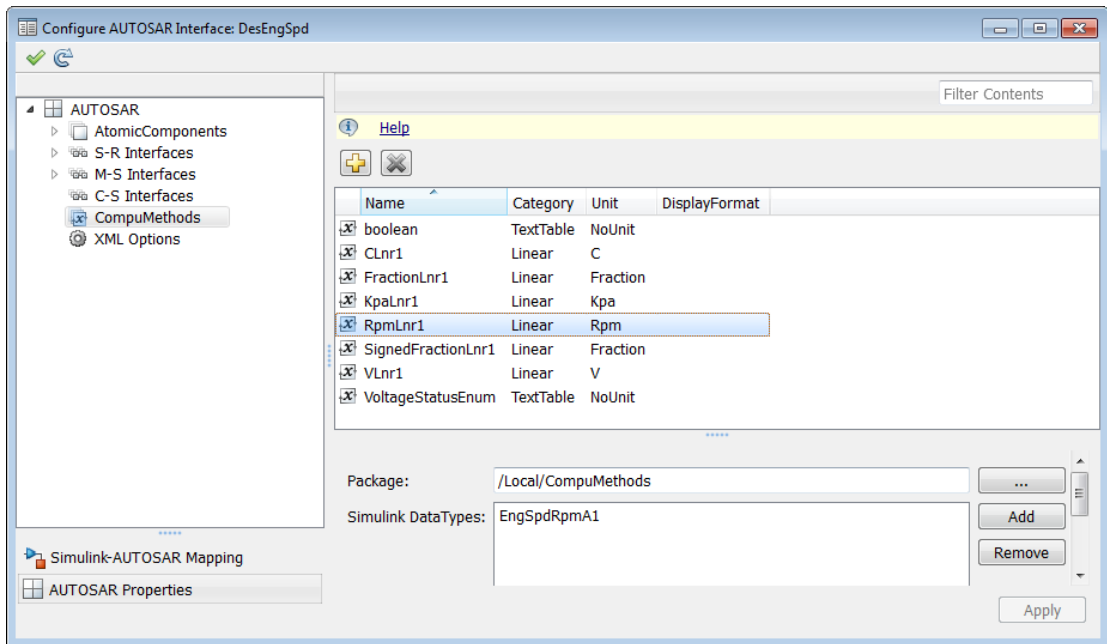
Additionally, you can create CompuMethods in Simulink, initialize their properties, and control the CompuMethod direction exported for linear functions.

This topic provides examples of configuring AUTOSAR CompuMethods in Simulink.

## 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.



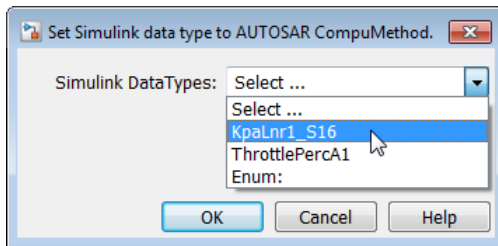
Select a CompuMethod and edit the available fields.

- **Name** — Specify name string
- **Category** — Select **Identical**, **Linear**, **RatFunc**, or **TextTable** (see “CompuMethod Categories for Data Types” on page 2-32)
- **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 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 AUTOSAR Display Format for Measurement and Calibration” on page 4-143.
- **Package** — Specify path of AUTOSAR package to be generated for the CompuMethod
- **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.
- 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 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 4-51.

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. Click **OK** to add it to the **Simulink DataTypes** list.




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('DesEngSpd');
set(arProps, '/Local/CompuMethods/RpmLnr1', 'S1DataTypes', {'EngSpdRpmA1'})
sltypes=get(arProps, '/Local/CompuMethods/RpmLnr1', 'S1DataTypes')

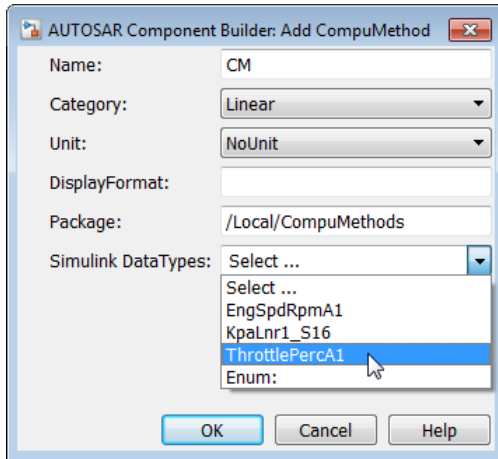
sltypes =
    'EngSpdRpmA1'
```

## 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. Click the **Add** icon  to open the Add CompuMethod dialog box. 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:

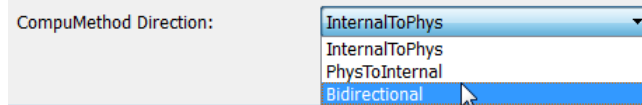
- **InternalToPhys** (default) — Generate CompuMethod sections for conversion of internal values into their physical representations.
- **PhysToInternal** — Generate CompuMethod sections for conversion of physical values into their internal representations.
- **Bidirectional** — Generate CompuMethod 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 **CompuMethods** in the exported arxml code contain the requested directional sections. For example, here is a **CompuMethod** generated with the **CompuMethod** direction set to **Bidirectional**.

```
<COMPU-METHOD UUID="...">
  <SHORT-NAME>COMPU_EngSpdValue</SHORT-NAME>
  <CATEGORY>LINEAR</CATEGORY>
  <COMPU-INTERNAL-TO-PHYS>
    <COMPU-SCALES>
      <COMPU-SCALE>
        <SHORT-LABEL>intToPhys</SHORT-LABEL>
        <LOWER-LIMIT INTERVAL-TYPE="CLOSED">0</LOWER-LIMIT>
        <UPPER-LIMIT INTERVAL-TYPE="CLOSED">24000</UPPER-LIMIT>
        <COMPU-RATIONAL-COEFFS>
          <COMPU-NUMERATOR>
            <V>0</V>
            <V>1</V>
          </COMPU-NUMERATOR>
          <COMPU-DENOMINATOR>
            <V>8</V>
          </COMPU-DENOMINATOR>
        </COMPU-RATIONAL-COEFFS>
      </COMPU-SCALE>
    </COMPU-SCALES>
  </COMPU-INTERNAL-TO-PHYS>
  <COMPU-PHYS-TO-INTERNAL>
    <COMPU-SCALES>
      <COMPU-SCALE>
        <SHORT-LABEL>physToInt</SHORT-LABEL>
        <LOWER-LIMIT INTERVAL-TYPE="CLOSED">0</LOWER-LIMIT>
        <UPPER-LIMIT INTERVAL-TYPE="CLOSED">3000</UPPER-LIMIT>
        <COMPU-RATIONAL-COEFFS>
          <COMPU-NUMERATOR>
            <V>0</V>
            <V>8</V>
          </COMPU-NUMERATOR>
          <COMPU-DENOMINATOR>
            <V>1</V>
          </COMPU-DENOMINATOR>
        </COMPU-RATIONAL-COEFFS>
      </COMPU-SCALE>
    </COMPU-SCALES>
  </COMPU-PHYS-TO-INTERNAL>
</COMPU-METHOD>
```



```

</COMPU-SCALES>
</COMPU-PHYS-TO-INTERNAL>
</COMPU-METHOD>

```

---

**Note:** CompuMethods of category TEXTTABLE, which are generated for Boolean or enumerated data types, use only InternalToPhys, regardless of the direction parameter setting.

---

## Export CompuMethod Unit References

The arxml importer preserves unit and physical dimension information found in imported CompuMethods. The software preserves CompuMethod unit and physical dimension information across round-trips between an AUTOSAR authoring tool (AAT) and Simulink.

For designs originated in Simulink, the exporter generates a unit reference for each CompuMethod. By convention, each CompuMethod references a unit named NoUnit. For example, here is a Boolean data type CompuMethod and the unit it references.

```

<COMPU-METHOD UUID="...">
  <SHORT-NAME>COMPU_Boolean</SHORT-NAME>
  <CATEGORY>TEXTTABLE</CATEGORY>
  <UNIT-REF DEST="UNIT">/mymodel1_pkg/mymodel1_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.

## Configure Rational Function CompuMethod for Dual-Scaled Parameter

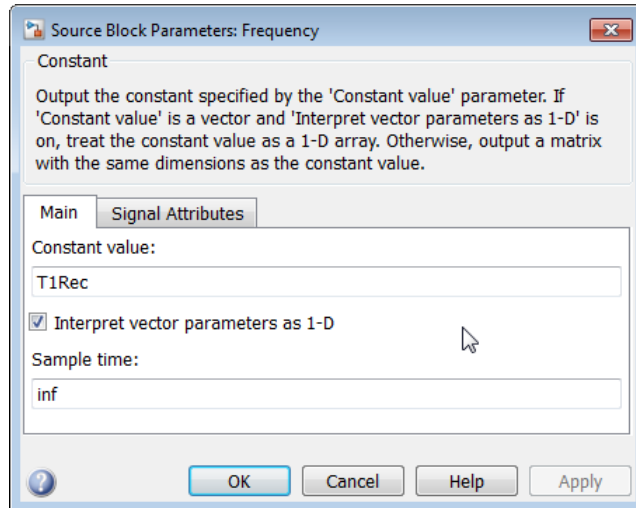
For an AUTOSAR dual-scaled parameter, which stores two scaled values of the same physical value, the software generates the CompuMethod category RAT\_FUNC. The computation method can be a first-order rational function.

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.



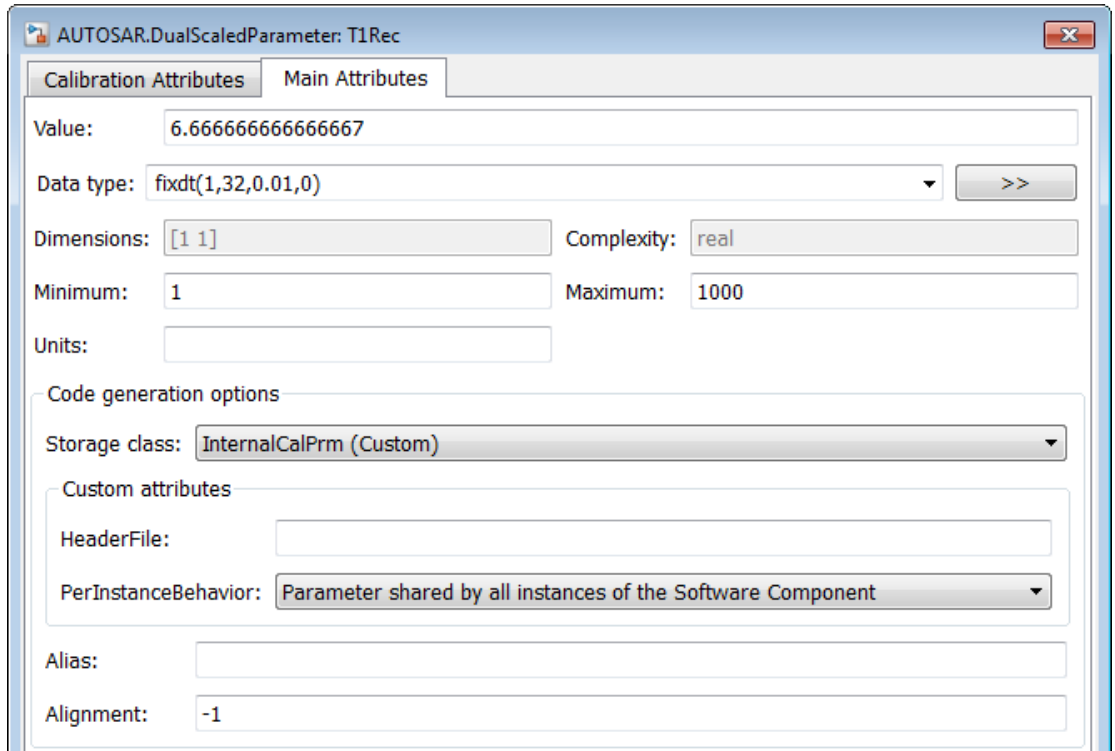
Connect the Constant block to a Simulink output.

- 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.DocUnits = '';
T1Rec.CalibrationDocUnits = 'm/s^2';
```

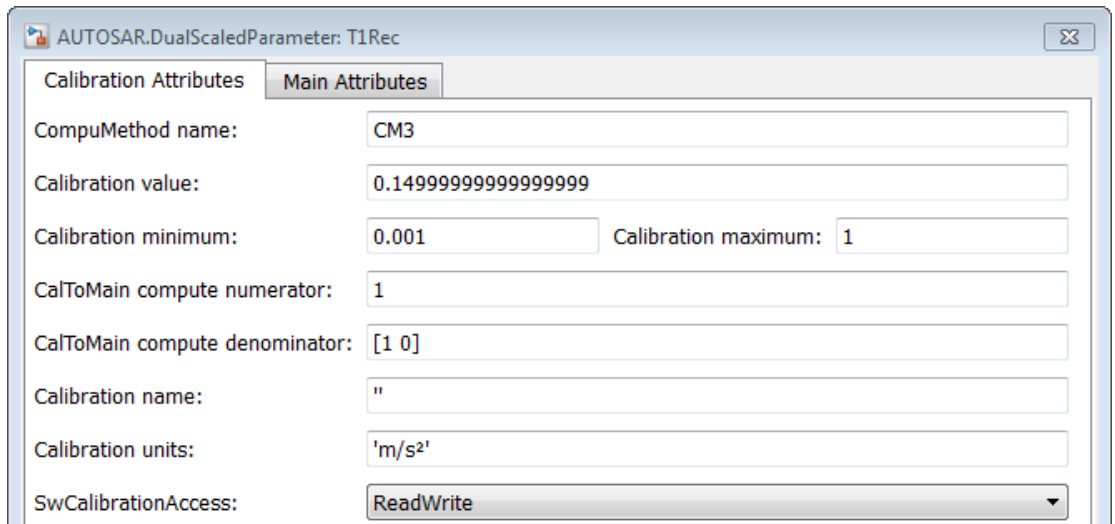
- 4 To open the dual-scaled parameter dialog box, double-click the parameter in the workspace. Here are the main attributes set by the MATLAB code.



The screenshot shows the 'AUTOSAR.DualScaledParameter: T1Rec' dialog box. It has two tabs: 'Calibration Attributes' and 'Main Attributes'. The 'Main Attributes' tab is selected. The dialog contains the following fields and values:

- Value: 6.666666666666667
- Data type: fixdt(1,32,0.01,0)
- Dimensions: [1 1]
- Complexity: real
- Minimum: 1
- Maximum: 1000
- Units: (empty)
- Code generation options:
  - Storage class: InternalCalPrm (Custom)
  - Custom attributes:
    - HeaderFile: (empty)
    - PerInstanceBehavior: Parameter shared by all instances of the Software Component
  - Alias: (empty)
  - Alignment: -1

- 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 for the model, 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>

```

# Configure AUTOSAR Internal Data Constraints Export

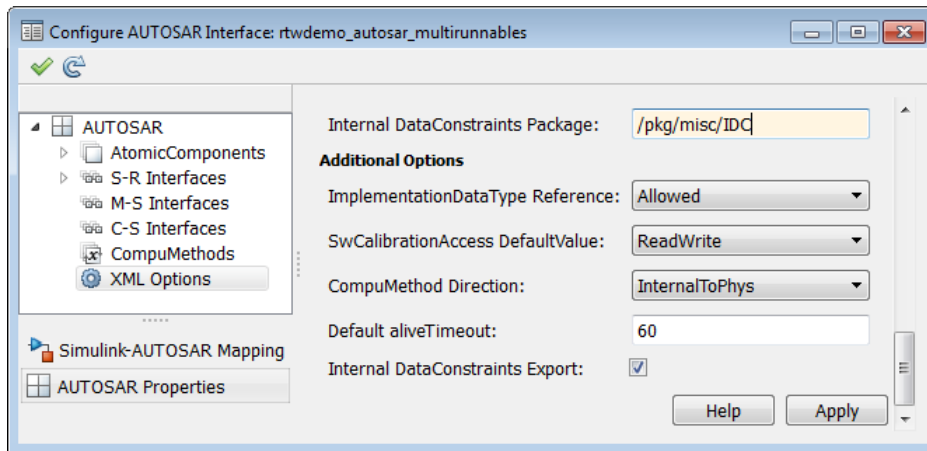
AUTOSAR applications use data constraints to implement limits on data types and provide a controlled range of possible values. Internal data constraints represent minimum and maximum values for implementation data types, reflecting the internal or machine view of the data.

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 4-148
- “Model AUTOSAR Data Types” on page 2-24
- “Export AUTOSAR Component XML and C Code” on page 5-2

### **More About**

- “Release 4.x Data Types” on page 2-29
- “AUTOSAR Interface Configuration” on page 4-3



## Configure AUTOSAR Multiple Runnables

### In this section...

“Configure Multiple Runnables Using Rate-Based Multitasking” on page 4-165

“Configure Multiple Runnables Using Function Calls” on page 4-172

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.

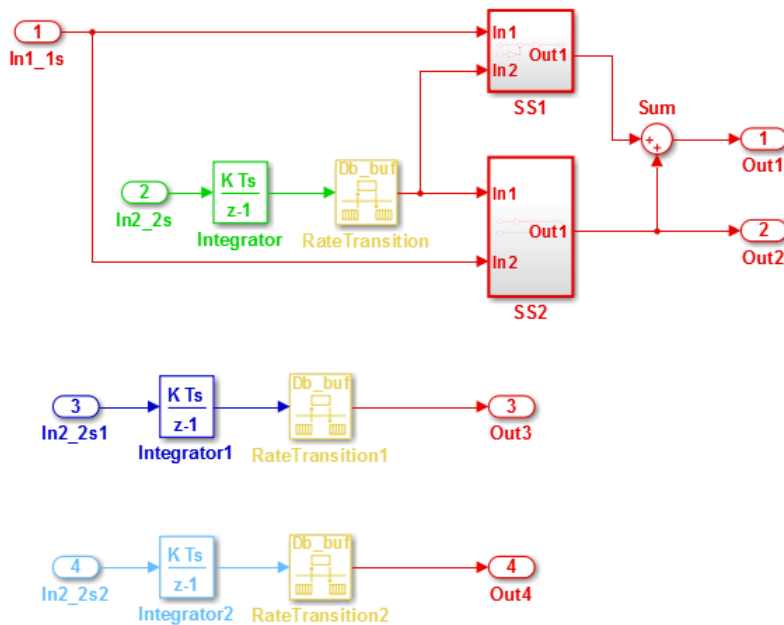
For more information about the modeling styles, see “Multi-Runnable Software Components” on page 2-5 and “Model for AUTOSAR Platform”.

This topic provides examples of configuring AUTOSAR multi-runnables in each available modeling style.

### 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 target.

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.

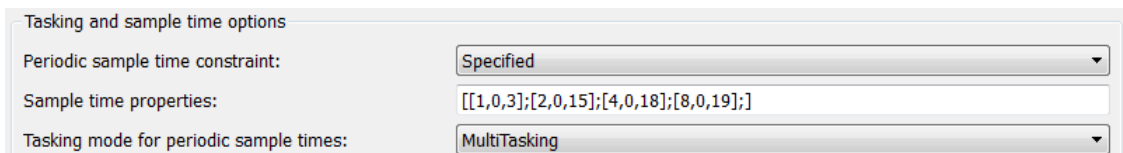
This example shows the general workflow for configuring a rate-based, multitasking Simulink model as an AUTOSAR software component.

---

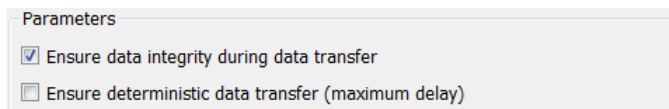
**Note:** The illustrations and code excerpts in this example use the preconfigured model `matlabroot/help/toolbox/ecoder/examples/autosar/mMultitasking_4rates.slx`. If you prefer to follow the example with a model not already configured for the AUTOSAR target, you can use the Simulink Coder example model `rtwdemo_mrmtbb`.

---

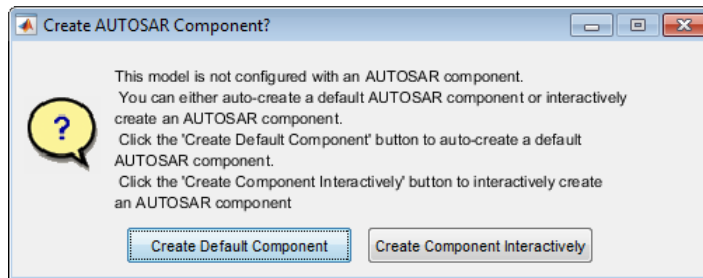
- 1 Configure a Simulink model to generate multitasking code. The minimum requirements are:
  - The model must use multiple rates.
  - In the Configuration Parameters dialog box, **Solver** pane, the model parameter **Tasking mode for periodic sample times** must be set to `MultiTasking` or `Auto`.



- 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 target, 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.

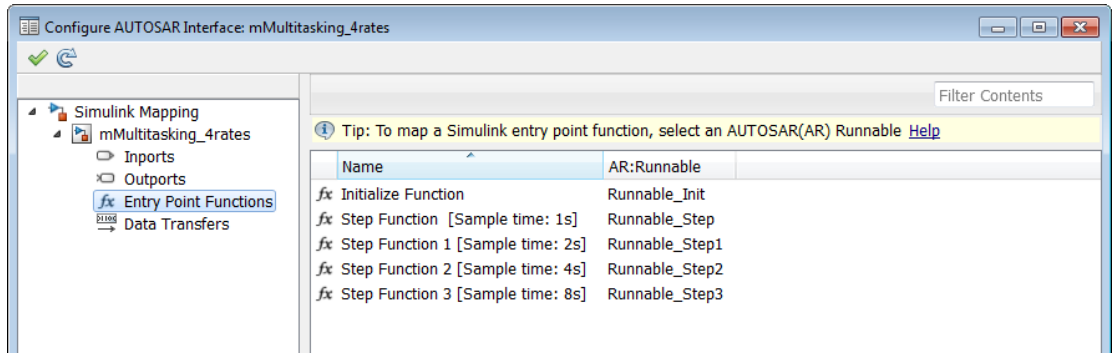
After you click **Create Default Component**, a new AUTOSAR component opens in the Configure AUTOSAR Interface dialog box.

---

**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.

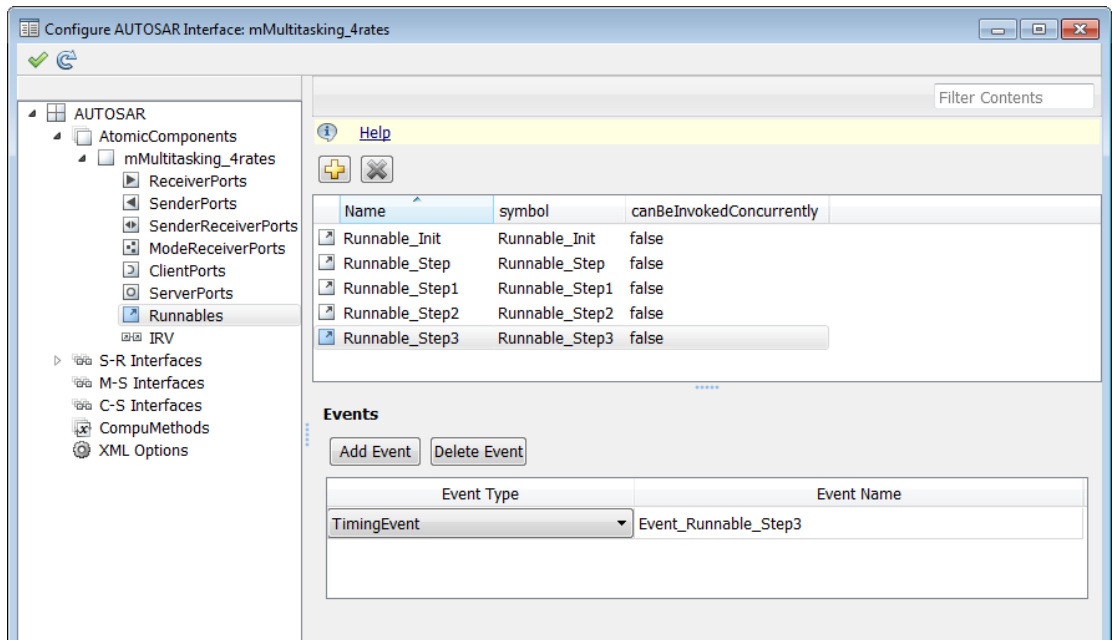
---

- 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.



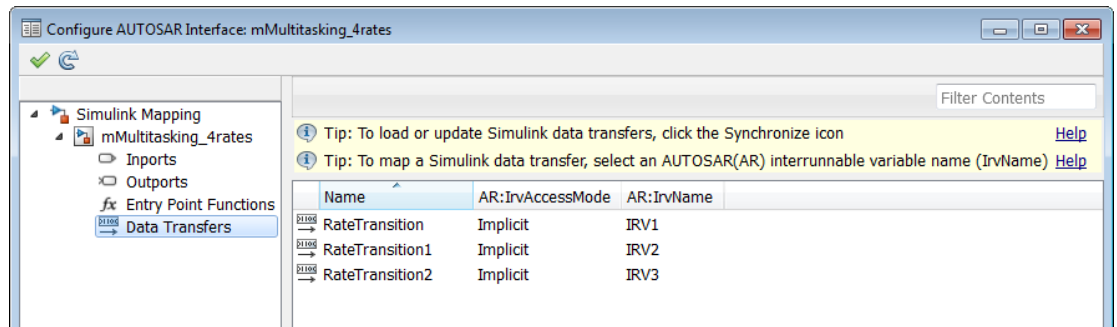
If a step function is not yet mapped to an AUTOSAR runnable, the **AR:Runnable** property displays **Select**.

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.



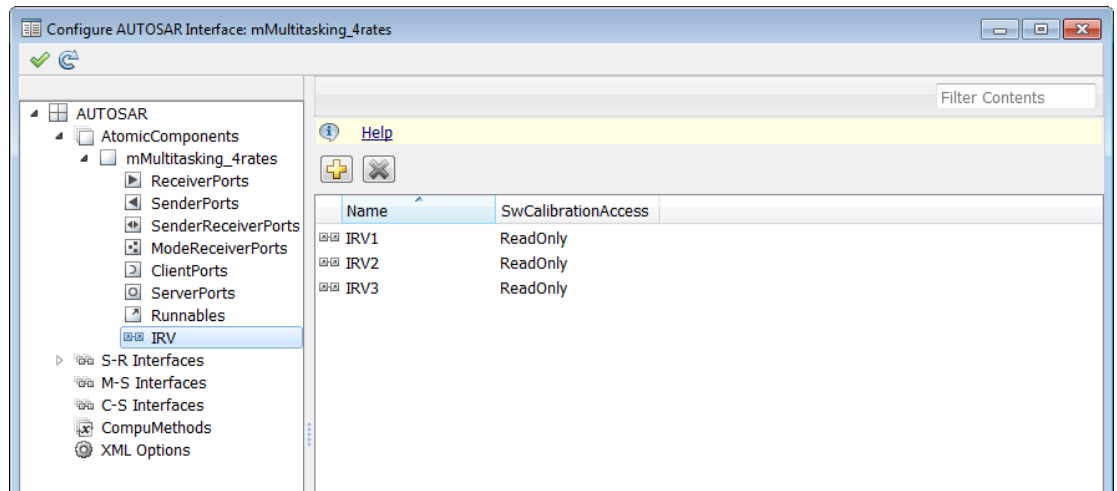
If you previously found, in the **Entry Point Functions** view, that a Simulink periodic step function was not mapped to an AUTOSAR runnable, use the **Runnables** view to add a runnable. Also, create a timing event for the runnable. Then return to the Simulink-AUTOSAR Mapping Explorer, **Entry Point 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>/OutputBufferForTicToc' incorporates:
   * UnitDelay: '<S3>/Delay'
   */
  OutputBufferForTicToc = 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' */

  /* Output: '<Root>/PPort_DE2' */
  Rte_IWrite_Runnable3_PPort_DE2(OutputBufferForTicToc);
}

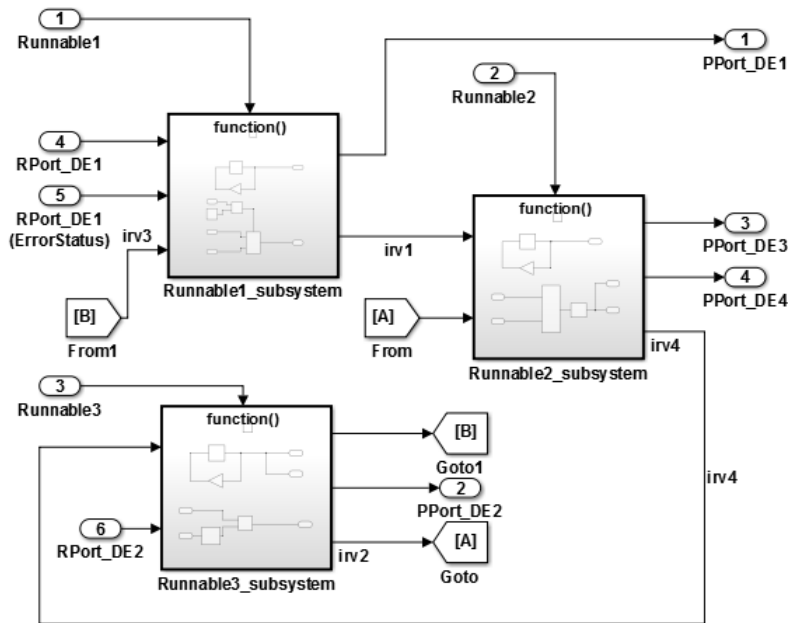
```

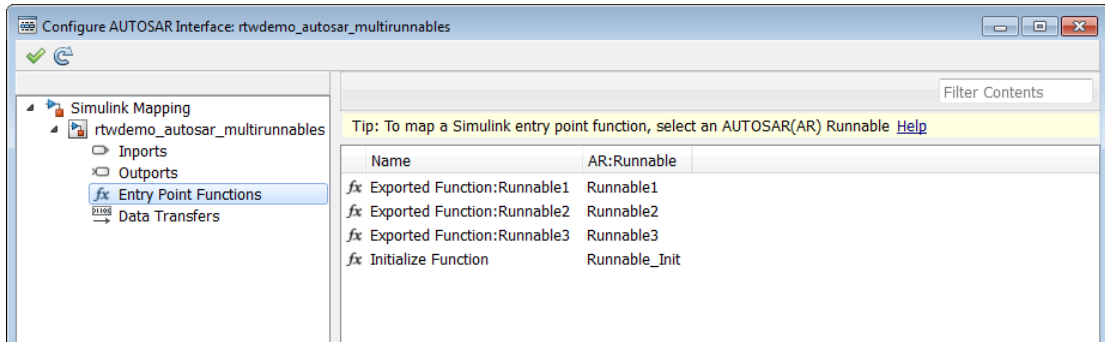


}

**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 3-2
- “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

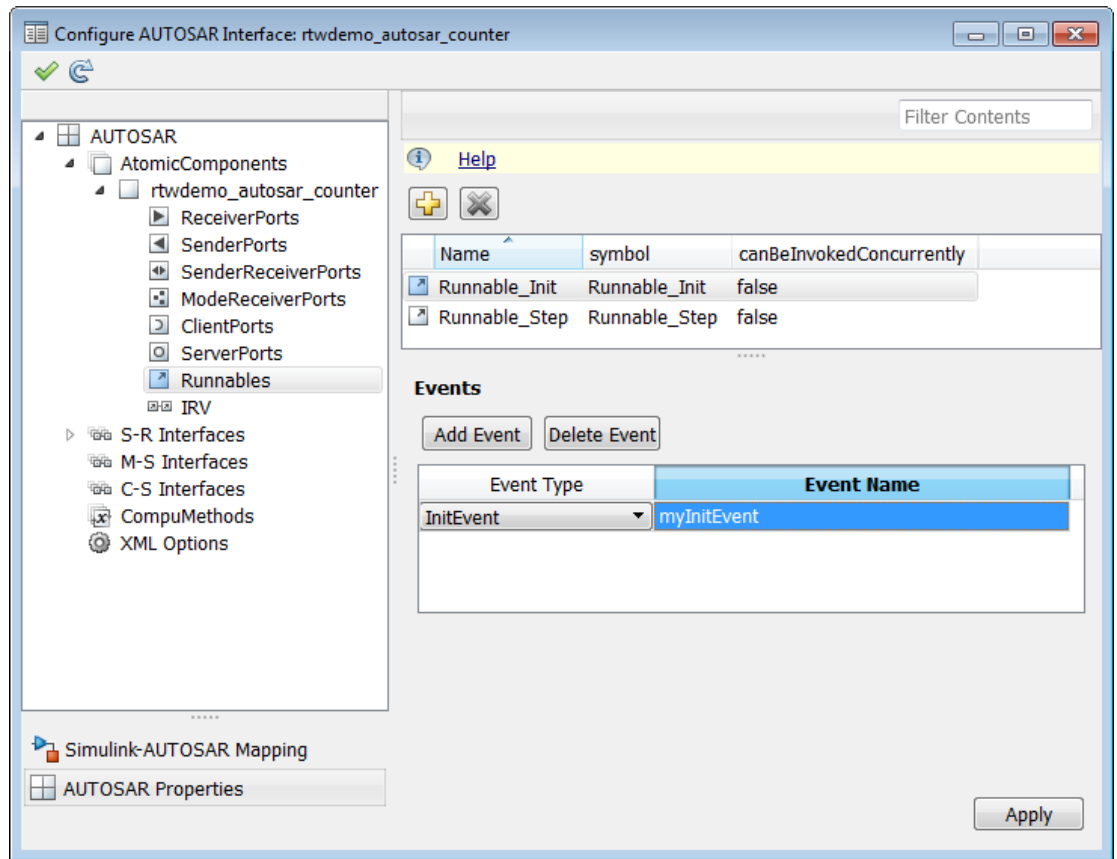
## Configure AUTOSAR Initialization Runnable

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 Receiver Ports and Mode-Switch Events” on page 4-85.)

If you import `arxml` code that describes a runnable with an `InitEvent`, the `arxml` importer configures the runnable in Simulink as an initialization runnable.

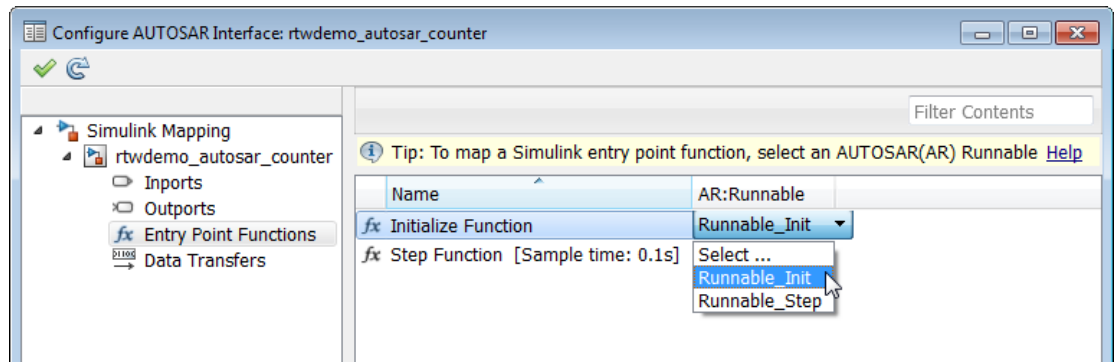
Alternatively, you can configure a runnable to be the initialization runnable in Simulink. For example,

- 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**.
- 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 an **Event Name** string. 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.

With Embedded Coder, you can programmatically get and set the `DisabledMode` property of a `TimingEvent`, `DataReceivedEvent`, `ModeSwitchEvent`, `OperationInvokedEvent`, or `DataReceiveErrorEvent`. The property is not supported for an `InitEvent`.

The value of the `DisabledMode` property of an event is `'portName.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` to a mode defined for a mode-receiver port, `MRPort.Sleep`:

```
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 2-37.

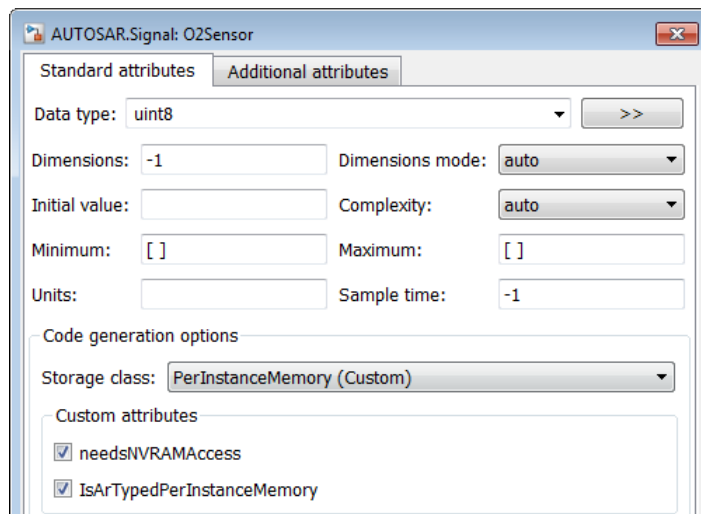
When you import arxml code that contains an AUTOSAR PIM definition into Simulink, the importer:

- 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.

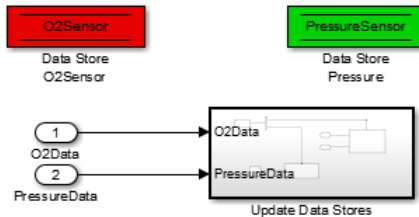
You can use the same modeling pattern to create per-instance memory in your model. For example:

- 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 `rtwdemo_autosar_PIM`, 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 3-2
- “Model AUTOSAR Component Behavior” on page 2-34
- `rtwdemo_autosar_PIM`
- “Export AUTOSAR Component XML and C Code” on page 5-2

## More About

- “Per-Instance Memory” on page 2-37

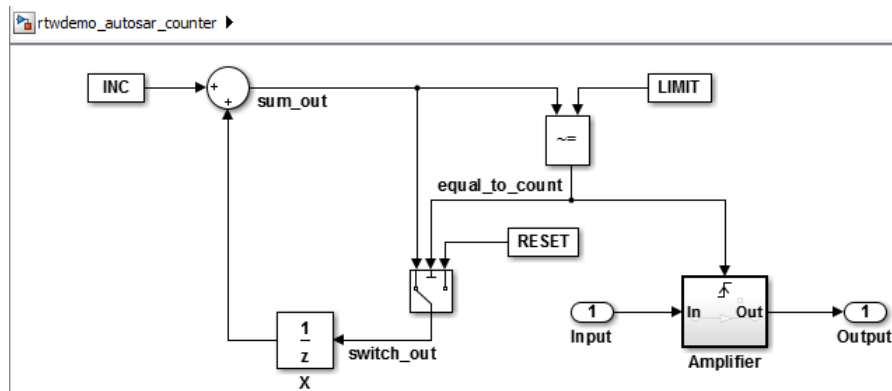
## Configure AUTOSAR Static or Constant Memory

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:

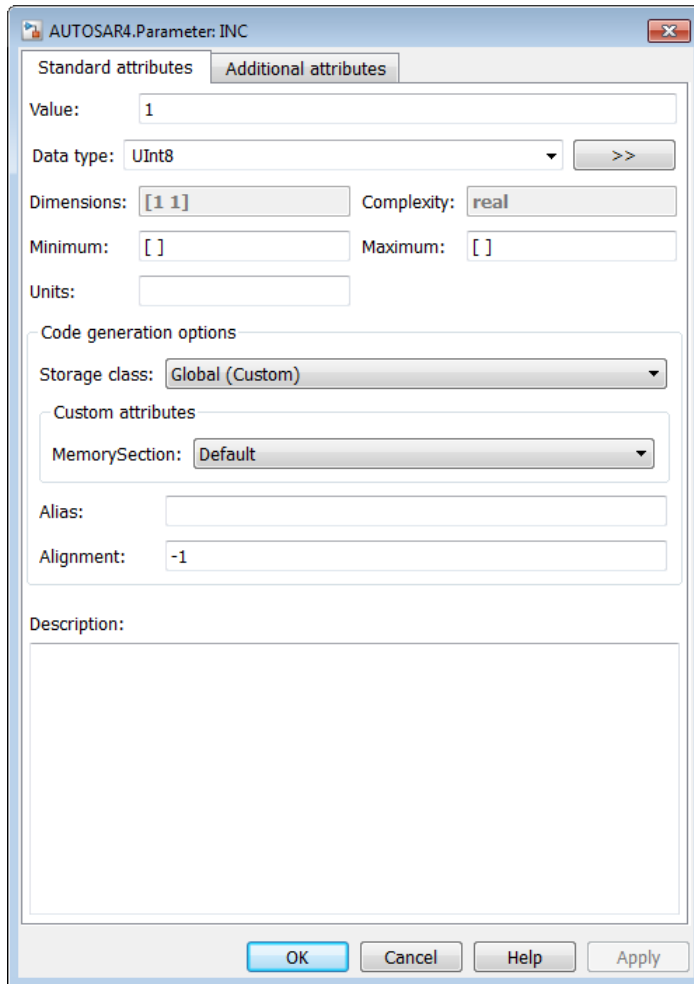
- 1 Open the example model `rtwdemo_autosar_counter`.



- 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.

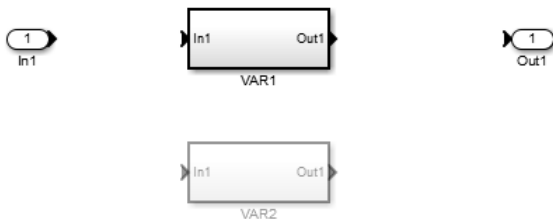
- 5 Generate code, and verify that `INC` and `sum_out` appear in the generated code as Constant Memory and Static Memory respectively. In the generated file `rtwdemo_autosar_counter_component.arxml`:
  - `INC` now appears in a constant memory specification, rather than a calibration parameter.
  - `sum_out` now appears in a static memory specification (previously was absent).

## Configure AUTOSAR Variation Point Proxies

AUTOSAR supports variant condition logic inside a runnable, using a `VariationPointProxy`, which was introduced in AUTOSAR schema version 4.0. You can model an AUTOSAR `VariationPointProxy` using a Simulink variant with `AUTOSAR.Parameter` and `Simulink.Variant` data objects. The general workflow is:

- 1 Inside a function-call subsystem that models an AUTOSAR runnable, configure a Simulink Variant Subsystem or Model Variant block to model an AUTOSAR `VariationPointProxy`.
- 2 Configure `AUTOSAR.Parameter` data objects to model AUTOSAR system constants, representing the conditional values associated with the variant condition logic.
- 3 Create `Simulink.Variant` data objects in the base workspace to define the variant condition logic.

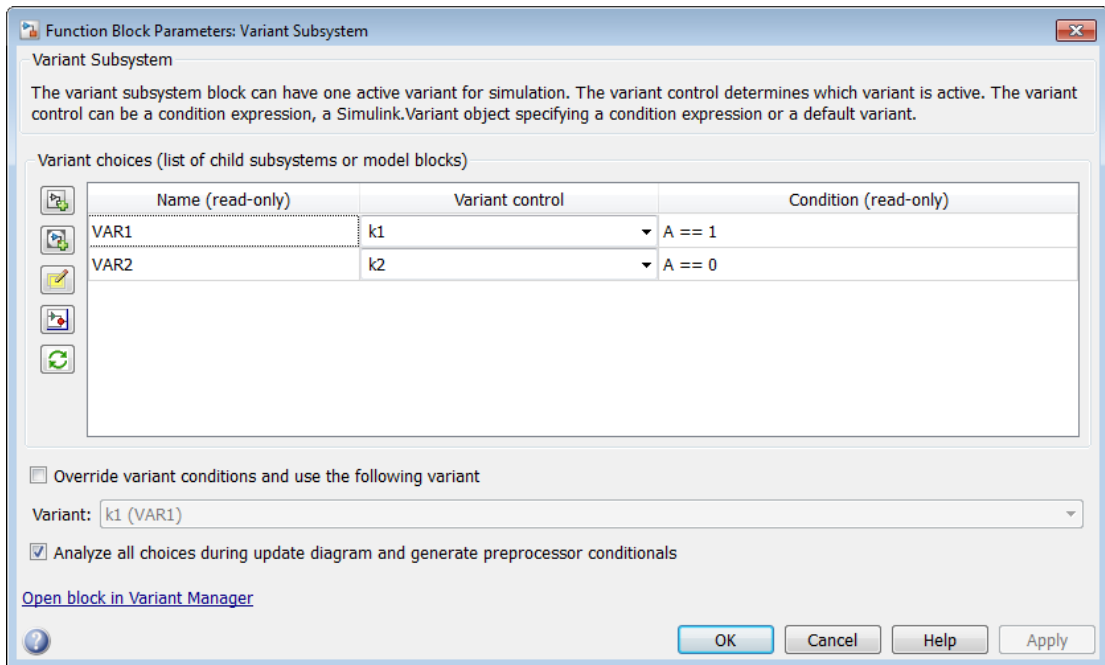
For example, the contents of a Variant Subsystem block are shown below. The variant choices are subsystems VAR1 and VAR2. The blocks are not connected at this level, because connectivity is automatically determined during simulation, based on the active variant.



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, 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. For example:



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
```

# Configure and Map AUTOSAR Component Programmatically

### In this section...

“AUTOSAR Property and Mapping Functions” on page 4-188

“Tree View of AUTOSAR Configuration” on page 4-188

“Properties of AUTOSAR Elements” on page 4-190

“Specify AUTOSAR Element Location” on page 4-193

“Sample Scripts That Use AUTOSAR Functions” on page 4-194

## AUTOSAR Property and Mapping Functions

You can use AUTOSAR property and mapping functions to programmatically configure the Simulink representation of an AUTOSAR software component and map model elements to AUTOSAR component elements. You can programmatically get, set, add, and remove the same component properties and mapping information displayed in the **AUTOSAR Properties** Explorer and **Simulink-AUTOSAR Mapping** Explorer views of the Configure AUTOSAR Interface dialog box. For example:

- Use the AUTOSAR property functions to add properties for AUTOSAR elements, add interfaces, find elements, get and set properties of elements, and delete elements.
- Use the AUTOSAR mapping functions to map model elements to AUTOSAR elements and return AUTOSAR mapping information for model elements.

The AUTOSAR property and mapping functions also validate syntax and semantics for requested AUTOSAR property and mapping changes.

For a complete list of the available property and mapping functions, see the functions listed for “AUTOSAR Component Development”. For example scripts, see “Sample Scripts That Use AUTOSAR Functions” on page 4-194.

---

**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 mapping 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	<ul style="list-style-type: none"> <li>• ReceiverPorts (add/delete)</li> <li>• SenderPorts (add/delete)</li> <li>• SenderReceiverPorts (add/delete)</li> <li>• ModeReceiverPorts (add/delete)</li> <li>• ClientPorts (add/delete)</li> <li>• ServerPorts (add/delete)</li> <li>• NvReceiverPorts (add/delete)</li> <li>• NvSenderPorts (add/delete)</li> <li>• NvSenderReceiverPorts (add/delete)</li> <li>• Behavior (add/delete)</li> <li>• Kind</li> <li>• Name</li> </ul>
ApplicationComponentBehavior	<ul style="list-style-type: none"> <li>• Runnables (add/delete)</li> <li>• Events (add/delete)</li> <li>• PIM (add/delete)</li> <li>• IRV (add/delete)</li> <li>• Parameters (add/delete)</li> <li>• DataTypeMapping</li> <li>• Name</li> </ul>

<b>AUTOSAR Element Class</b>	<b>Properties</b>
DataReceiverPort	<ul style="list-style-type: none"> <li>• Interface</li> <li>• Name</li> </ul>
DataSenderPort	<ul style="list-style-type: none"> <li>• Interface</li> <li>• Name</li> </ul>
DataSenderReceiverPort	<ul style="list-style-type: none"> <li>• Interface</li> <li>• Name</li> </ul>
Runnable	<ul style="list-style-type: none"> <li>• symbol</li> <li>• canBeInvokedConcurrently</li> <li>• SwAddrMethod</li> <li>• Name</li> </ul>
TimingEvent	<ul style="list-style-type: none"> <li>• Period</li> <li>• StartOnEvent</li> <li>• DisabledMode</li> <li>• Name</li> </ul>
DataReceivedEvent	<ul style="list-style-type: none"> <li>• Trigger</li> <li>• StartOnEvent</li> <li>• DisabledMode</li> <li>• Name</li> </ul>
ModeSwitchEvent	<ul style="list-style-type: none"> <li>• Trigger</li> <li>• Activation</li> <li>• StartOnEvent</li> <li>• DisabledMode</li> <li>• Name</li> </ul>
OperationInvokedEvent	<ul style="list-style-type: none"> <li>• Trigger</li> <li>• StartOnEvent</li> <li>• DisabledMode</li> <li>• Name</li> </ul>

AUTOSAR Element Class	Properties
InitEvent	<ul style="list-style-type: none"> <li>• StartOnEvent</li> <li>• Name</li> </ul>
DataReceiveErrorEvent	<ul style="list-style-type: none"> <li>• Trigger</li> <li>• StartOnEvent</li> <li>• DisabledMode</li> <li>• Name</li> </ul>
IrvData	<ul style="list-style-type: none"> <li>• Type</li> <li>• SwAddrMethod</li> <li>• SwCalibrationAccess</li> <li>• DisplayFormat</li> <li>• SwAlignment</li> <li>• Name</li> </ul>
SenderReceiverInterface	<ul style="list-style-type: none"> <li>• DataElements (add/delete)</li> <li>• IsService</li> <li>• Name</li> </ul>
FlowData	<ul style="list-style-type: none"> <li>• Type</li> <li>• SwAddrMethod</li> <li>• InvalidationPolicy</li> <li>• SwCalibrationAccess</li> <li>• DisplayFormat</li> <li>• SwAlignment</li> <li>• Name</li> </ul>
ModeSwitchInterface	<ul style="list-style-type: none"> <li>• ModeGroup (add/delete)</li> <li>• IsService</li> <li>• Name</li> </ul>

AUTOSAR Element Class	Properties
ModeDeclarationGroupElement	<ul style="list-style-type: none"> <li>• ModeGroup</li> <li>• SwCalibrationAccess</li> <li>• DisplayFormat</li> <li>• Name</li> </ul>
ClientServerInterface	<ul style="list-style-type: none"> <li>• Operations (add/delete)</li> <li>• IsService</li> <li>• Name</li> </ul>
NvDataInterface	<ul style="list-style-type: none"> <li>• DataElements (add/delete)</li> <li>• IsService</li> <li>• Name</li> </ul>

## 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:

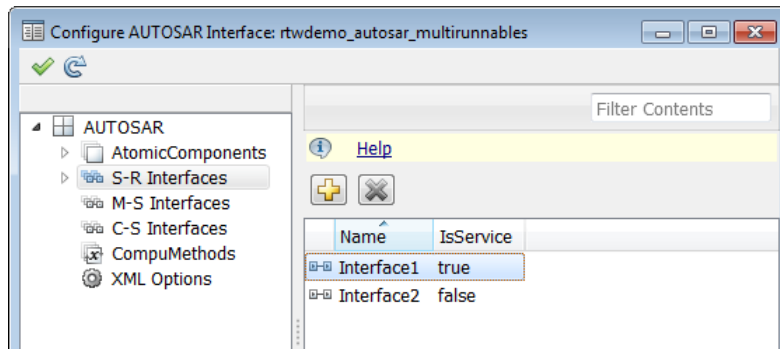
```
/pkgLevel1/pkgLevel2/pkgLevel3/grandParentName/parentName/childName
```

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);
```



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`.

## Sample Scripts That Use AUTOSAR Functions

- “Gather Information for Custom Scheduler Script” on page 4-195
- “Extract SwAddrMethod Information from Runnables” on page 4-196

The example scripts in this topic illustrate combinations of functions that are useful for AUTOSAR-related tasks.

### Gather Information for Custom Scheduler Script

The following script loops through events and runnables in an open model. For each event or runnable, the script uses AUTOSAR property functions to extract 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 hook up a custom scheduler

% Use AUTOSAR properties functions
arProps = autosar.api.getAUTOSARProperties(hModel);

swc = get(arProps,'XmlOptions','ComponentQualifiedName');

% Get internal behavior
ib = get(arProps,swc,'Behavior');

% Get events & 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');
            disp(sprintf('Event %s triggers runnable %s with period %g',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');

    disp(sprintf('Runnable %s has symbol %s',runnableName,runnableSymbol));
end
```

Running this script 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 this script 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

```

### Extract SwAddrMethod Information from Runnables

The following script loops through runnables in an open model and uses AUTOSAR property functions to extract `SwAddrMethod` information. `hModel` specifies the name of an open AUTOSAR model.

```

% Example of how to extract SwAddrMethod information from runnables to attach memory sections

% Use AUTOSAR properties functions
arProps = autosar.api.getAUTOSARProperties(hModel);

swc = get(arProps,'XmlOptions','ComponentQualifiedNames');

% Get internal behavior
ib = get(arProps,swc,'Behavior');

% Get runnables
runnables = get(arProps,ib,'Runnables');

% Loop through runnables
for ii=1:length(runnables)
    runnable = runnables{ii};
    runnableName = get(arProps,runnable,'Name');
    swAddrMethod = get(arProps,runnable,'SwAddrMethod');
    if isempty(swAddrMethod)
        disp(sprintf('Runnable %s has no SwAddrMethod',runnableName));
    else
        disp(sprintf('Runnable %s has SwAddrMethod %s',runnableName,swAddrMethod));
    end
end
end

```



## Limitations and Tips

### In this section...

“Source of Initial Output Value for Function-Call Subsystem Output” on page 4-197

“AUTOSAR Client Block in Referenced Model” on page 4-197

“Use the Merge Block for Inter-Runnable Variables” on page 4-197

### Source of Initial Output Value for Function-Call Subsystem Output

Before exporting a runnable from a function-call subsystem with an output, you must set the Output block parameter `SourceOfInitialOutputValue` to `Dialog`. Otherwise, when you try to export the runnable, the software generates an error indicating this requirement. See the Output reference page.

### AUTOSAR Client Block in Referenced Model

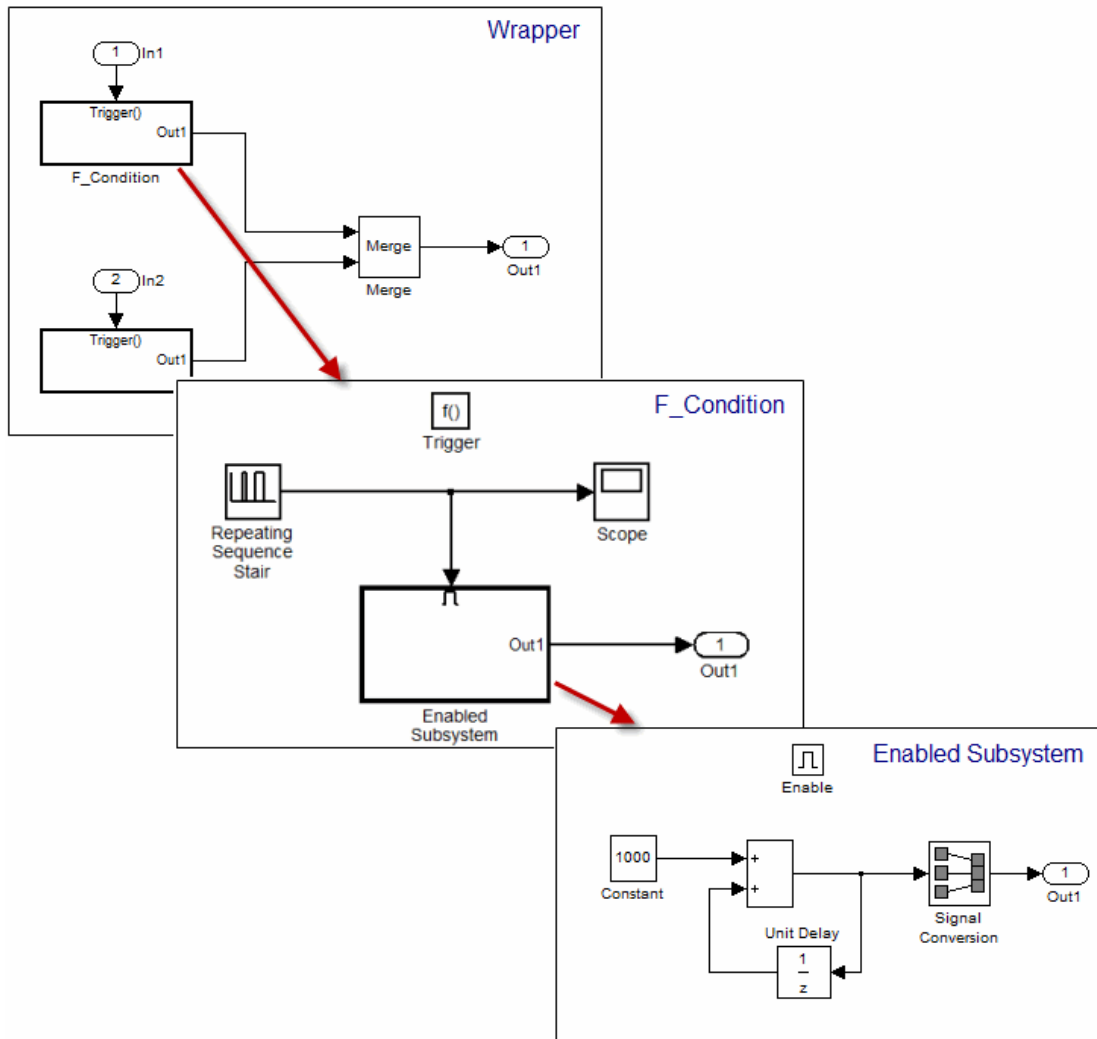
The software does not support the use of an AUTOSAR client block, such as Function Caller or Invoke AUTOSAR Server Operation, in a referenced model.

### 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 output 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 output that enters a Merge block must not be conditionally computed. Consider the following example.



The output from the subsystem `F_condition` is the conditional output from `Enabled Subsystem`. When you try to validate or build the model, the software generates an error.

If you use an S-Function block instead of the `Enabled Subsystem` block, the software generates a *warning* when you validate or build the model.

# AUTOSAR Code Generation

---

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

### 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 5-3

“Configure AUTOSAR Compiler Abstraction Macros” on page 5-4

“Root-Level Matrix I/O” on page 5-5

“Export AUTOSAR Software Component” on page 5-5

### 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	4.2.1
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	3.2.1
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 target 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:

- 1 Open the Configuration Parameters dialog box. In models that use the `autosar.tlc` system target file, **AUTOSAR Code Generation Options** appears in the tree.
- 2 Click **AUTOSAR Code Generation Options** to open the **AUTOSAR Code Generation Options** pane.
- 3 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 SHORT -NAME XML elements must not be greater than 32 characters in length. However, your authoring tool may support the use of longer elements, for example, to name ports and interfaces. The software allows you to specify the maximum length of your SHORT -NAME elements.

Before you build your model, in the **Code Generation > AUTOSAR Code Generation Options** pane of the Configuration Parameters dialog box, in the **Maximum SHORT-NAME length** field, specify the maximum length of your SHORT -NAME elements. You may specify a maximum length of up to 128 characters. The default is 32 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 64 k 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](http://www.autosar.org).

### Configure AUTOSAR Compiler Macro Generation

Before you build your model, 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 the **Use AUTOSAR compiler abstraction macros** option, the software generates 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 the **Use AUTOSAR compiler abstraction macros** option, the software generates 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. Before you build your model, on the **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	<i>modelName.arxml</i>	All AUTOSAR elements.
Modular	<i>modelName_component.arxml</i>	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 4-54.
	<i>modelName_datatype.arxml</i>	Data types and related elements.
	<i>modelName- _implementation.arxml</i>	Software component implementation.
	<i>modelName_interface.arxml</i>	Interfaces, including S-R interfaces, C-S interfaces, and M-S interfaces.
	<i>modelName_behavior.arxml</i>	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, 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 3-10.

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 Replacement for AUTOSAR

### In this section...

“AUTOSAR Code Replacement” 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

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, `Mf1.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.

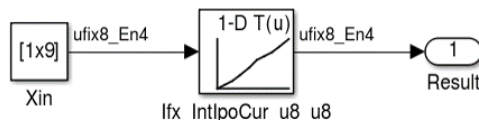
For more information on code replacement and code replacement libraries, see “What Is Code Replacement?” and “Code Replacement Libraries”.

## 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, select **Code Generation > Report > 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()));

    /* Output: '<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
If1_DPSearch Prelookup	Extrapolation method	Clip
	ExtrapMethod	
	Index search method	Linear search or Binary search
	IndexSearchMethod	
	Use last breakpoint for input at or above upper limit	On
	UseLastBreakPoint	

IFL/IFX Function and Block Mapping	Algorithm Property Parameters	Value
	<b>Remove protection against out-of-range input in generated code</b> RemoveProtectionInput	Off
	<b>Integer rounding mode</b> RndMeth	Round or Zero
If1_IpoCur Interpolation Using Prelookup	<b>Interpolation method</b> InterpMethod	Linear
	<b>Extrapolation method</b> ExtrapMethod	Clip
	<b>Valid index input may reach last index</b> ValidIndexMayReachLast	On
	<b>Remove protection against out-of-range input in generated code</b> RemoveProtectionInput	Off
	<b>Integer rounding mode</b> RndMeth	Round or Zero
	If1_IpoMap Interpolation Using Prelookup	<b>Interpolation method</b> InterpMethod
<b>Extrapolation method</b> ExtrapMethod		Clip

IFL/IFX Function and Block Mapping	Algorithm Property Parameters	Value
	<b>Valid index input may reach last index</b> ValidIndexMayReachLast	On
	<b>Remove protection against out-of-range input in generated code</b> RemoveProtectionInput	Off
	<b>Integer rounding mode</b> RndMeth	Round or Zero
If1_IntIpoCur n-D Lookup Table	<b>Interpolation method</b> InterpMethod	Linear
	<b>Extrapolation method</b> ExtrapMethod	Clip
	<b>Index search method</b> IndexSearchMethod	Linear search or Binary search
	<b>Use last table value for inputs at or above last breakpoint</b> UseLastTableValue	On
	<b>Remove protection against out-of-range input in generated code</b> RemoveProtectionInput	Off
	<b>Integer rounding mode</b> RndMeth	Round or Zero

IFL/IFX Function and Block Mapping	Algorithm Property Parameters	Value
If1_IntIpoMap n-D Lookup Table	<b>Interpolation method</b> InterpMethod	Linear
	<b>Extrapolation method</b> ExtrapMethod	Clip
	<b>Index search method</b> IndexSearchMethod	Linear search or Binary search
	<b>Use last table value for inputs at or above last breakpoint</b> UseLastTableValue	On
	<b>Remove protection against out-of-range input in generated code</b> RemoveProtectionInput	Off
	<b>Integer rounding mode</b> RndMeth	Round or Zero
	Ifx_DPSearch Prelookup	<b>Extrapolation method</b> ExtrapMethod
<b>Index search method</b> IndexSearchMethod		Linear search or Binary search
<b>Use last breakpoint for input at or above upper limit</b> UseLastBreakPoint		On

IFL/IFX Function and Block Mapping	Algorithm Property Parameters	Value
	<b>Remove protection against out-of-range input in generated code</b> RemoveProtectionInput	Off
	<b>Integer rounding mode</b> RndMeth	Round or Zero
Ifx_IpoCur Interpolation Using Prelookup	<b>Interpolation method</b> InterpMethod	Linear
	<b>Extrapolation method</b> ExtrapMethod	Clip
	<b>Valid index input may reach last index</b> ValidIndexMayReachLast	On
	<b>Remove protection against out-of-range input in generated code</b> RemoveProtectionInput	Off
	<b>Integer rounding mode</b> RndMeth	Round or Zero
	Ifx_LkUpCur Interpolation Using Prelookup	<b>Interpolation method</b> InterpMethod
<b>Extrapolation method</b> ExtrapMethod		Clip



IFL/IFX Function and Block Mapping	Algorithm Property Parameters	Value
	<b>Remove protection against out-of-range input in generated code</b> RemoveProtectionInput	Off
	<b>Integer rounding mode</b> RndMeth	Round or Zero
	<b>Valid index input may reach last index</b> ValidIndexMayReachLast	On
Ifx_IpoMap Interpolation Using Prelookup	<b>Interpolation method</b> InterpMethod	Linear
	<b>Extrapolation method</b> ExtrapMethod	Clip
	<b>Valid index input may reach last index</b> ValidIndexMayReachLast	On
	<b>Remove protection against out-of-range input in generated code</b> RemoveProtectionInput	Off
	<b>Integer rounding mode</b> RndMeth	Round or Zero
	Ifx_LkUpMap Interpolation Using Prelookup	<b>Interpolation method</b> InterpMethod

IFL/IFX Function and Block Mapping	Algorithm Property Parameters	Value
	<b>Extrapolation method</b>	Clip
	ExtrapMethod	
	<b>Remove protection against out-of-range input in generated code</b>	Off
	RemoveProtectionInput	
	<b>Integer rounding mode</b>	Round or Zero
Ifx_LkUpBaseMap Interpolation Using Prelookup	RndMeth	
	<b>Valid index input may reach last index</b>	On
	ValidIndexMayReachLast	
	<b>Interpolation method</b>	Flat
	InterpMethod	
	<b>Extrapolation method</b>	Clip
	ExtrapMethod	
<b>Remove protection against out-of-range input in generated code</b>	Off	
RemoveProtectionInput		
<b>Integer rounding mode</b>	Round or Zero	
RndMeth		
<b>Valid index input may reach last index</b>	On	
ValidIndexMayReachLast		

<b>IFL/IFX Function and Block Mapping</b>	<b>Algorithm Property Parameters</b>	<b>Value</b>
Ifx_IntIpoCur n-D Lookup Table	<b>Interpolation method</b> InterpMethod	Linear
	<b>Extrapolation method</b> ExtrapMethod	Clip
	<b>Index search method</b> IndexSearchMethod	Linear search or Binary search
	<b>Use last table value for inputs at or above last breakpoint</b> UseLastTableValue	On
	<b>Remove protection against out-of-range input in generated code</b> RemoveProtectionInput	Off
	<b>Integer rounding mode</b> RndMeth	Round or Zero
	Ifx_IntLkUpCur n-D Lookup Table	<b>Interpolation method</b> InterpMethod
<b>Extrapolation method</b> ExtrapMethod		Clip
<b>Index search method</b> IndexSearchMethod		Linear search or Binary search
<b>Remove protection against out-of-range input in generated code</b> RemoveProtectionInput		Off

IFL/IFX Function and Block Mapping	Algorithm Property Parameters	Value
	<b>Integer rounding mode</b> RndMeth	Round or Zero
	<b>Use last table value for inputs at or above last breakpoint</b> UseLastTableValue	On
Ifx_IntIpoFixCur n-D Lookup Table	<b>Interpolation method</b> InterpMethod	Linear
	<b>Extrapolation method</b> ExtrapMethod	Clip
	<b>Index search method</b> IndexSearchMethod	Evenly spaced points
	<b>Use last table value for inputs at or above last breakpoint</b> UseLastTableValue	On
	<b>Remove protection against out-of-range input in generated code</b> RemoveProtectionInput	Off
	<b>Integer rounding mode</b> RndMeth	Round or Zero
	<b>Use last table value for inputs at or above last breakpoint</b> UseLastTableValue	On

IFL/IFX Function and Block Mapping	Algorithm Property Parameters	Value
	Model configuration parameter <b>Optimization &gt; Signals and Parameters &gt; Default parameter behavior</b> DefaultParameterBehavior	Inlined
Ifx_IntLkUpFixCur n-D Lookup Table	<b>Interpolation method</b> InterpMethod	Flat
	<b>Extrapolation method</b> ExtrapMethod	Clip
	<b>Index search method</b> IndexSearchMethod	Evenly spaced points
	<b>Remove protection against out-of-range input in generated code</b> RemoveProtectionInput	Off
	<b>Integer rounding mode</b> RndMeth	Round or Zero
	Model configuration parameter <b>Optimization &gt; Signals and Parameters &gt; Default parameter behavior</b> 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	<b>Interpolation method</b> InterpMethod	Linear	
	<b>Extrapolation method</b> ExtrapMethod	Clip	
	<b>Index search method</b> IndexSearchMethod	Evenly spaced points	
	<b>Use last table value for inputs at or above last breakpoint</b> UseLastTableValue	On	
	<b>Remove protection against out-of-range input in generated code</b> RemoveProtectionInput	Off	
	<b>Integer rounding mode</b> RndMeth	Round or Zero	
	Breakpoint data must not match power 2 spacing.		
	Ifx_IntLkUpFixICur n-D Lookup Table	<b>Interpolation method</b> InterpMethod	Flat
		<b>Extrapolation method</b> ExtrapMethod	Clip
<b>Index search method</b> IndexSearchMethod		Evenly spaced points	

IFL/IFX Function and Block Mapping	Algorithm Property Parameters	Value
	<b>Remove protection against out-of-range input in generated code</b> RemoveProtectionInput	Off
	<b>Integer rounding mode</b> RndMeth	Round or Zero
	<b>Use last table value for inputs at or above last breakpoint</b> UseLastTableValue	On
	Breakpoint data must not match power 2 spacing.	
Ifx_IntIpoMap n-D Lookup Table	<b>Interpolation method</b> InterpMethod	Linear
	<b>Extrapolation method</b> ExtrapMethod	Clip
	<b>Index search method</b> IndexSearchMethod	Linear search or Binary search
	<b>Use last table value for inputs at or above last breakpoint</b> UseLastTableValue	On
	<b>Remove protection against out-of-range input in generated code</b> RemoveProtectionInput	Off
	<b>Integer rounding mode</b> RndMeth	Round or Zero

IFL/IFX Function and Block Mapping	Algorithm Property Parameters	Value
Ifx_IntLkUpMap n-D Lookup Table	<b>Interpolation method</b> InterpMethod	Nearest
	<b>Extrapolation method</b> ExtrapMethod	Clip
	<b>Index search method</b> IndexSearchMethod	Linear search or Binary search
	<b>Remove protection against out-of-range input in generated code</b> RemoveProtectionInput	Off
	<b>Integer rounding mode</b> RndMeth	Round or Zero
	<b>Use last table value for inputs at or above last breakpoint</b> UseLastTableValue	On
	Ifx_IntLkUpBaseMap n-D Lookup Table	<b>Interpolation method</b> InterpMethod
<b>Extrapolation method</b> ExtrapMethod		Clip
<b>Index search method</b> IndexSearchMethod		Linear search or Binary search
<b>Remove protection against out-of-range input in generated code</b> RemoveProtectionInput		Off



IFL/IFX Function and Block Mapping	Algorithm Property Parameters	Value
	<b>Integer rounding mode</b> RndMeth	Round or Zero
	<b>Use last table value for inputs at or above last breakpoint</b> UseLastTableValue	On
Ifx_IntIpoFixMap n-D Lookup Table	<b>Interpolation method</b> InterpMethod	Linear
	<b>Extrapolation method</b> ExtrapMethod	Clip
	<b>Index search method</b> IndexSearchMethod	Evenly spaced points
	<b>Use last table value for inputs at or above last breakpoint</b> UseLastTableValue	On
	<b>Remove protection against out-of-range input in generated code</b> RemoveProtectionInput	Off
	<b>Integer rounding mode</b> RndMeth	Round or Zero
	Model configuration parameter <b>Optimization &gt; Signals and Parameters &gt; Default parameter behavior</b> 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	<b>Interpolation method</b> InterpMethod	Nearest
	<b>Extrapolation method</b> ExtrapMethod	Clip
	<b>Index search method</b> IndexSearchMethod	Evenly spaced points
	<b>Remove protection against out-of-range input in generated code</b> RemoveProtectionInput	Off
	<b>Integer rounding mode</b> RndMeth	Round or Zero
	<b>Use last table value for inputs at or above last breakpoint</b> UseLastTableValue	On
	Model configuration parameter <b>Optimization &gt; Signals and Parameters &gt; Default parameter behavior</b> DefaultParameterBehavior	Inlined
	Breakpoint data must match power 2 spacing.	
	Ifx_IntLkUpFixBaseMap n-D Lookup Table	<b>Interpolation method</b> InterpMethod

IFL/IFX Function and Block Mapping	Algorithm Property Parameters	Value
	<b>Extrapolation method</b> ExtrapMethod	Clip
	<b>Index search method</b> IndexSearchMethod	Evenly spaced points
	<b>Remove protection against out-of-range input in generated code</b> RemoveProtectionInput	Off
	<b>Integer rounding mode</b> RndMeth	Round or Zero
	<b>Use last table value for inputs at or above last breakpoint</b> UseLastTableValue	On
	Model configuration parameter <b>Optimization &gt; Signals and Parameters &gt; Default parameter behavior</b> DefaultParameterBehavior	Inlined
	Breakpoint data must match power 2 spacing.	
	Ifx_IntIpoFixIMap n-D Lookup Table	<b>Interpolation method</b> InterpMethod
<b>Extrapolation method</b> ExtrapMethod		Linear
<b>Index search method</b> IndexSearchMethod		Evenly spaced points

IFL/IFX Function and Block Mapping	Algorithm Property Parameters	Value
	<b>Use last table value for inputs at or above last breakpoint</b> UseLastTableValue	On
	<b>Remove protection against out-of-range input in generated code</b> RemoveProtectionInput	Off
	<b>Integer rounding mode</b> RndMeth	Round or Zero
	Breakpoint data must not match power 2 spacing.	
Ifx_IntLkUpFixIMap n-D Lookup Table	<b>Interpolation method</b> InterpMethod	Nearest
	<b>Extrapolation method</b> ExtrapMethod	Clip
	<b>Index search method</b> IndexSearchMethod	Evenly spaced points
	<b>Remove protection against out-of-range input in generated code</b> RemoveProtectionInput	Off
	<b>Integer rounding mode</b> RndMeth	Round or Zero
	<b>Use last table value for inputs at or above last breakpoint</b> 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	<b>Interpolation method</b> InterpMethod	Flat
	<b>Extrapolation method</b> ExtrapMethod	Clip
	<b>Index search method</b> IndexSearchMethod	Evenly spaced points
	<b>Remove protection against out-of-range input in generated code</b> RemoveProtectionInput	Off
	<b>Integer rounding mode</b> RndMeth	Round or Zero
	<b>Use last table value for inputs at or above last breakpoint</b> 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	<ul style="list-style-type: none"> <li>Input and corresponding breakpoint arguments have the same data type.</li> </ul>

Function Type	Match Process Checks Whether
	<ul style="list-style-type: none"><li>• Output and the table data argument have the same data type.</li></ul>
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. For example:

- You can run a top model that is configured for the AUTOSAR target (`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 “Top-Model SIL or PIL Simulation” and “Model Block SIL or PIL Simulation”.

---

**Note:** You can create a SIL or PIL block for a component configured for the AUTOSAR target. For more information about configuring and running simulations with SIL or PIL blocks, see “Use a SIL or PIL Block”. 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

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” on page 5-30

### 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 target 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

When you build a Simulink model for an AUTOSAR target, the software generates 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.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  
getOutport  
getSensorActuatorComponentNames  
mapDataTransfer  
mapFunction

mapFunctionCaller  
mapInport  
mapOutport  
set  
setDependencies  
setFile  
updateModel

# 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 *Support Package Installer*, a wizard that guides you through the installation process. On the MATLAB **Home** tab, 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 4-188
- “Configure the AUTOSAR Interface” 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 string representing the model name.

Example: `arProps`

### **parentPath** — Path to a parent AUTOSAR element

string

Path to a parent AUTOSAR element to which to add a specified child property element.

Example: `'Input'`

### **property** — Type of property

string

Type of property to add, among valid properties for the AUTOSAR element.

Example: 'DataElements'

**name — Name of child property element**

string

Name of the child property element to add.

Example: 'DE1'

**childproperty, value — Child property and value**

name string, value

Child property to set, and its value. Table “Properties of AUTOSAR Elements” on page 4-190 lists properties that are associated with AUTOSAR elements.

Example: 'Name', 'event1'

**See Also**

delete

# 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 *Support Package Installer*, a wizard that guides you through the installation process. On the MATLAB **Home** tab, 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 4-188
- “Configure the AUTOSAR Interface” 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 string representing the model name.

Example: `arProps`

### **category** — Element category

string

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

string

Fully-qualified path to the element package.

Example: '/pkg/if'

### **name** — Element name

string

Name of the element to add.

Example: 'Interface3'

### **property, value** — Element property and value

name string, value

Property/value pairs for setting values of element properties. Table “Properties of AUTOSAR Elements” on page 4-190 lists properties that are associated with AUTOSAR elements.

Example: `'IsService',true`

### **See Also**

delete

# 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 *Support Package Installer*, a wizard that guides you through the installation process. On the MATLAB **Home** tab, click **Add-Ons > Get Hardware Support Packages**. For more information, see “Support Package for AUTOSAR Standard” on page 1-3.

---

## Construction

<code>arxml.importer</code>	Construct <code>arxml.importer</code> object
-----------------------------	--

## Methods

<code>createCalibrationComponentObjects</code>	Create Simulink calibration objects from AUTOSAR calibration component
<code>createComponentAsModel</code>	Create AUTOSAR atomic software component as Simulink model
<code>getApplicationComponentNames</code>	Get list of application software component names

<code>getCalibrationComponentNames</code>	Get calibration component names
<code>getClientServerInterfaceNames</code>	Get list of client-server interfaces
<code>getComponentNames</code>	Get application and sensor/actuator software component names
<code>getDependencies</code>	Get list of XML dependency files
<code>getFile</code>	Return software component XML file name
<code>getSensorActuatorComponentNames</code>	Get list of sensor/actuator software component names
<code>setDependencies</code>	Set XML file dependencies
<code>setFile</code>	Set software component XML file name
<code>updateModel</code>	Merge AUTOSAR XML changes into associated mapped Simulink model

## Copy Semantics

Handle. To learn how this affects your use of the class, see [Copying Objects](#) in the [MATLAB Programming Fundamentals](#) documentation.

# 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 *Support Package Installer*, a wizard that guides you through the installation process. On the MATLAB **Home** tab, click **Add-Ons > Get Hardware Support Packages**. For more information, see “Support Package for AUTOSAR Standard” on page 1-3.

---

## 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 3-2

# 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 *Support Package Installer*, a wizard that guides you through the installation process. On the **MATLAB Home** tab, 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 4-188
- “Configure the AUTOSAR Interface” on page 4-3

## Input Arguments

### **model** — Model for which to create AUTOSAR properties and mapping

`handle` | `string`

Model for which to create AUTOSAR properties and Simulink to AUTOSAR mapping, specified as a handle or string 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`



# 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 computational method that you provide. Given one scaled value, the `AUTOSAR.DualScaledParameter` can compute the other scaled value using the computational 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 computational method. This offline computation results in leaner generated code.

If you provide the calibration value, the parameter computes the main value using the computational 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

The screenshot shows a dialog box titled "AUTOSAR.DualScaledParameter: param" with two tabs: "Calibration Attributes" and "Main Attributes". The "Main Attributes" tab is active. The dialog contains the following fields and controls:

- Value:** A text input field containing "[ ]".
- Data type:** A dropdown menu set to "auto" with a ">>" button to its right.
- Dimensions:** A text input field containing "[0 0]".
- Complexity:** A text input field containing "real".
- Minimum:** A text input field containing "[ ]".
- Maximum:** A text input field containing "[ ]".
- Units:** An empty text input field.
- Code generation options:** A section containing:
  - Storage class:** A dropdown menu set to "Auto".
  - Alignment:** A text input field containing "-1".
- Description:** A large empty text area.

At the bottom of the dialog are four buttons: "OK", "Cancel", "Help", and "Apply".

This tab shows the properties inherited from the `AUTOSAR.Parameter` class. For more information, see `AUTOSAR.Parameter`.

## Calibration Attributes Tab

The screenshot shows a dialog box titled "AUTOSAR.DualScaledParameter: param" with a close button (X) in the top right corner. The dialog has two tabs: "Calibration Attributes" (selected) and "Main Attributes". The "Calibration Attributes" tab contains the following fields and controls:

- CompuMethod name:
- Calibration value:
- Calibration minimum:  Calibration maximum:
- CalToMain compute numerator:
- CalToMain compute denominator:
- Calibration name:
- Calibration units:
- SwCalibrationAccess:
- DisplayFormat:
- Parameter validation section:
  - Is configuration valid:
  - Diagnostic message:

At the bottom of the dialog, there are four buttons: "OK", "Cancel", "Help", and "Apply".

### CompuMethod name

Name of the AUTOSAR computation method (**CompuMethod**) to generate for the parameter in arxml code. For an AUTOSAR dual-scaled parameter, the software generates the **CompuMethod** category RAT\_FUNC. For an example, see “Configure Rational Function CompuMethod for Dual-Scaled Parameter” on page 4-157.

### 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 computational method. The parameter uses the computational 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 computational method. The parameter uses the computational 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 computational method. The parameter uses the computational 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 **d** 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 string 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 string 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 AUTOSAR Display Format for Measurement and Calibration” on page 4-143.

### 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 <b>arxml</b> 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 computational method. (CalToMain compute numerator)  Once you have applied <code>CalToMainCompuNumerator</code> , you cannot change it.
CalToMainCompuDenominator	RW	Denominator coefficients of the computational method. (CalToMain compute denominator)  Once you have applied <code>CalToMainCompuDenominator</code> , 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 — <code>ReadOnly</code> , <code>ReadWrite</code> , or <code>NotAccessible</code> . (SwCalibrationAccess)
DisplayFormat	RW	Format to be used by measurement and calibration tools to display the data. Format string uses the general form <code>%[flags][width][.precision]type</code> — for example, <code>%2.1d</code> 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 computational method.

```
T1Rec.CalToMainCompuNumerator = [1];
T1Rec.CalToMainCompuDenominator = [1 0];
```

This computational 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: []

```

```
DocUnits: ''  
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.DocUnits = 'Hertz';
```

Open the `AUTOSAR.DualScaledParameter` dialog box.

```
open T1Rec
```

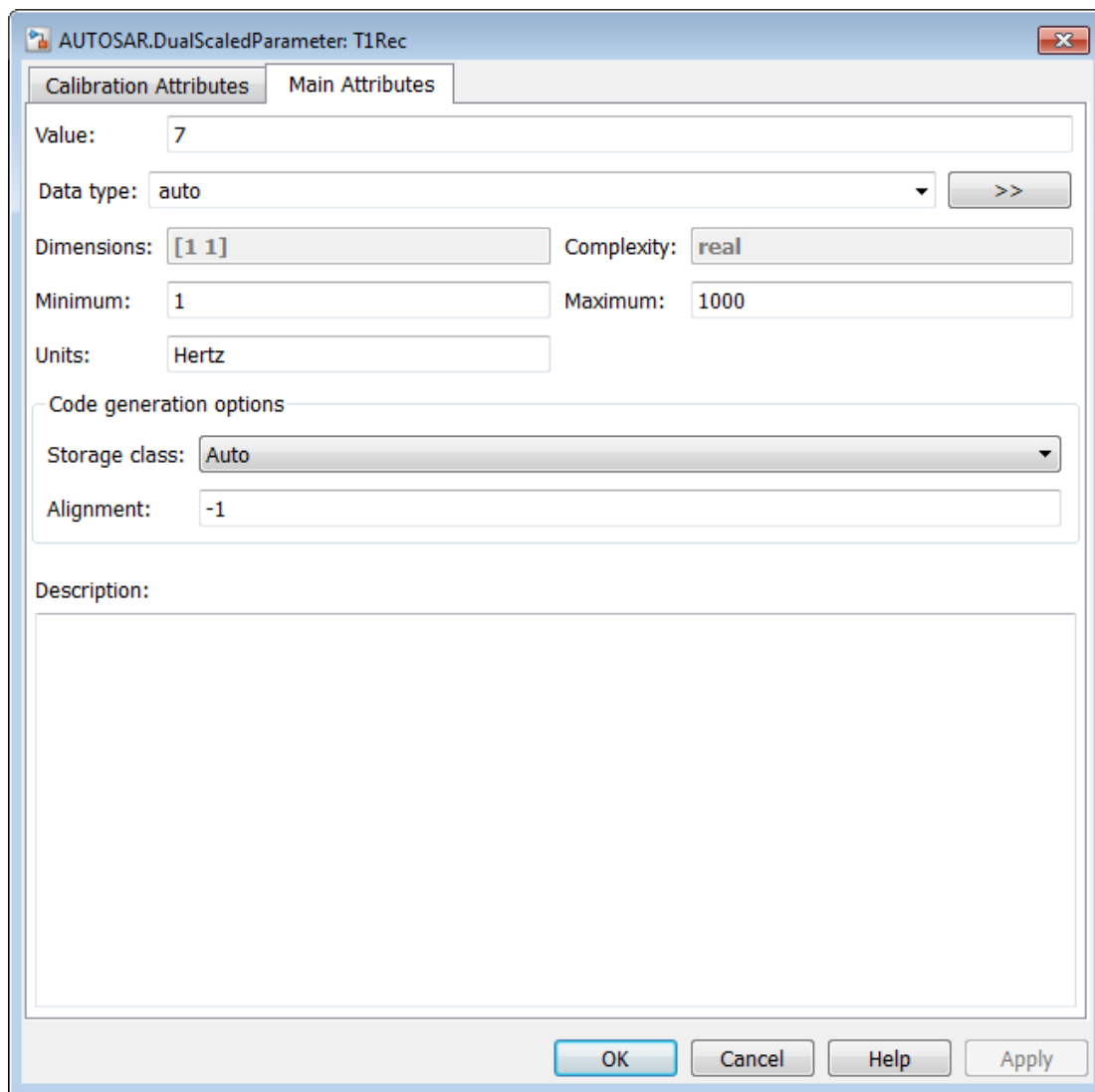
The screenshot shows a dialog box titled "AUTOSAR.DualScaledParameter: T1Rec" with two tabs: "Calibration Attributes" (selected) and "Main Attributes". The "Calibration Attributes" tab contains the following fields and controls:

- CompuMethod name:
- Calibration value:
- Calibration minimum:  Calibration maximum:
- CalToMain compute numerator:
- CalToMain compute denominator:
- Calibration name:
- Calibration units:
- SwCalibrationAccess:
- DisplayFormat:
- Parameter validation section:
  - Is configuration valid:
  - Diagnostic message:

At the bottom of the dialog are four buttons: "OK", "Cancel", "Help", and "Apply".

The **Calibration Attributes** tab displays the calibration value and the computational 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
    DocUnits: ''
    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, Embedded Coder generates 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 “Tunable Block Parameters”).

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

AUTOSAR.Parameter: arParam

Standard attributes Additional attributes

Value: [ ]

Data type: auto >>

Dimensions: [0 0] Complexity: real

Minimum: [ ] Maximum: [ ]

Units: [ ]

Code generation options

Storage class: CalPrm (Custom)

Custom attributes

HeaderFile: [ ]

ElementName: UNDEFINED

PortName: UNDEFINED

InterfacePath: UNDEFINED

CalibrationComponent: [ ]

ProviderPortName: [ ]

Alias: [ ]

Alignment: -1

Description:

[ ]

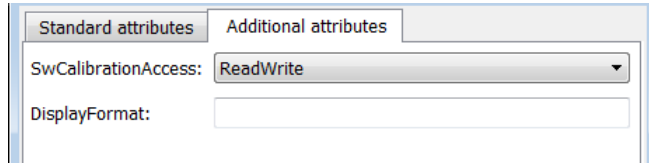
OK Cancel Help Apply



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_sw/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 AUTOSAR Display Format for Measurement and Calibration” on page 4-143.

For more information, see:

- “Tunable Parameter Storage”
- “Model AUTOSAR Calibration Parameters” on page 2-19
- “Configure AUTOSAR Calibration Parameters” on page 4-123
- “Configure AUTOSAR Calibration Component” on page 4-125
- “Variation Point Proxies” on page 2-36
- “Configure AUTOSAR Variation Point Proxies” on page 4-185
- “Configure AUTOSAR Data for Measurement and Calibration” on page 4-135

## See Also

- `Simulink.Parameter`
- `AUTOSAR.DualScaledParameter`
- `AUTOSAR4.Parameter`

# AUTOSAR.Signal

Specify data type, code generation options, other attributes of signal

## Description

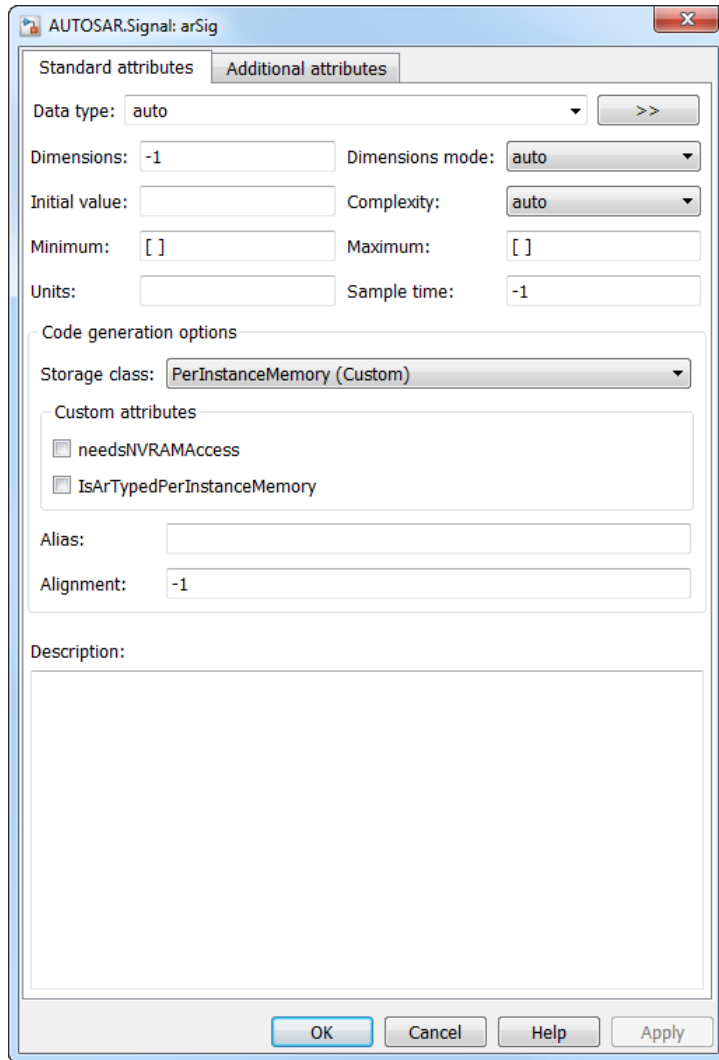
With this class, you can create workspace objects for modeling per-instance memory for AUTOSAR runnables. You can create an `AUTOSAR.Signal` object in the base MATLAB workspace.

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**.

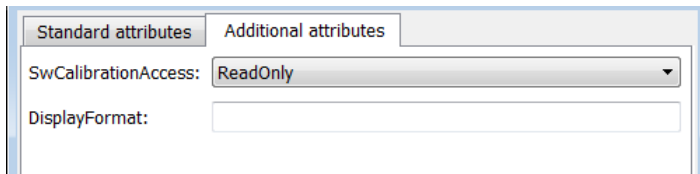
## Property Dialog Box



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 AUTOSAR Display Format for Measurement and Calibration” on page 4-143.

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 2-37
- “Configure AUTOSAR Per-Instance Memory” on page 4-179

- “Configure AUTOSAR Data for Measurement and Calibration” on page 4-135

## **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')
```

- “Configure the AUTOSAR Interface” on page 4-3

## Input Arguments

**model** — Model for which to close the Configure AUTOSAR Interface dialog box

handle | string

Model for which to close the Configure AUTOSAR Interface dialog box, specified as a handle or string representing the model name.

Example: 'rtwdemo\_autosar\_multirunnables'

**See Also**

`autosar.api.create` | `autosar_ui_launch`



# 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')
```

- “Configure the AUTOSAR Interface” on page 4-3

## Input Arguments

**model** — Model for which to display AUTOSAR interface configuration settings

handle | string

Model for which to display AUTOSAR interface configuration settings, specified as a handle or string representing the model name.

Example: 'rtwdemo\_autosar\_multirunnables'

**See Also**

autosar.api.create | autosar\_ui\_close

# 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 “Tunable Block Parameters”).

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 screenshot shows a dialog box titled "AUTOSAR4.Parameter: ar4Param". It has two tabs: "Standard attributes" and "Additional attributes". The "Standard attributes" tab is selected. The dialog contains the following fields and controls:

- Value: [ ]
- Data type: auto (dropdown menu) with a ">>" button to its right.
- Dimensions: [0 0] (text input)
- Complexity: real (text input)
- Minimum: [ ] (text input)
- Maximum: [ ] (text input)
- Units: [ ] (text input)
- Code generation options section:
  - Storage class: Global (Custom) (dropdown menu)
  - Custom attributes section:
    - MemorySection: Default (dropdown menu)
- Alias: [ ] (text input)
- Alignment: -1 (text input)
- Description: [ ] (text area)

At the bottom of the dialog are four buttons: OK, Cancel, Help, and Apply.

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:

The screenshot shows a software interface with two tabs: 'Standard attributes' and 'Additional attributes'. The 'Additional attributes' tab is active. It contains two fields: 'SwCalibrationAccess' with a dropdown menu showing 'ReadWrite', and 'DisplayFormat' with an empty text input box.

- **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 AUTOSAR Display Format for Measurement and Calibration” on page 4-143.

For more information, see

- “Static and Constant Memory” on page 2-38
- “Configure AUTOSAR Static or Constant Memory” on page 4-182
- “Configure AUTOSAR Data for Measurement and Calibration” on page 4-135

## See Also

- `Simulink.Parameter`
- `AUTOSAR.Parameter`
- `AUTOSAR.DualScaledParameter`

## AUTOSAR4.Signal

Specify data type, code generation options, other attributes of signal

### Description

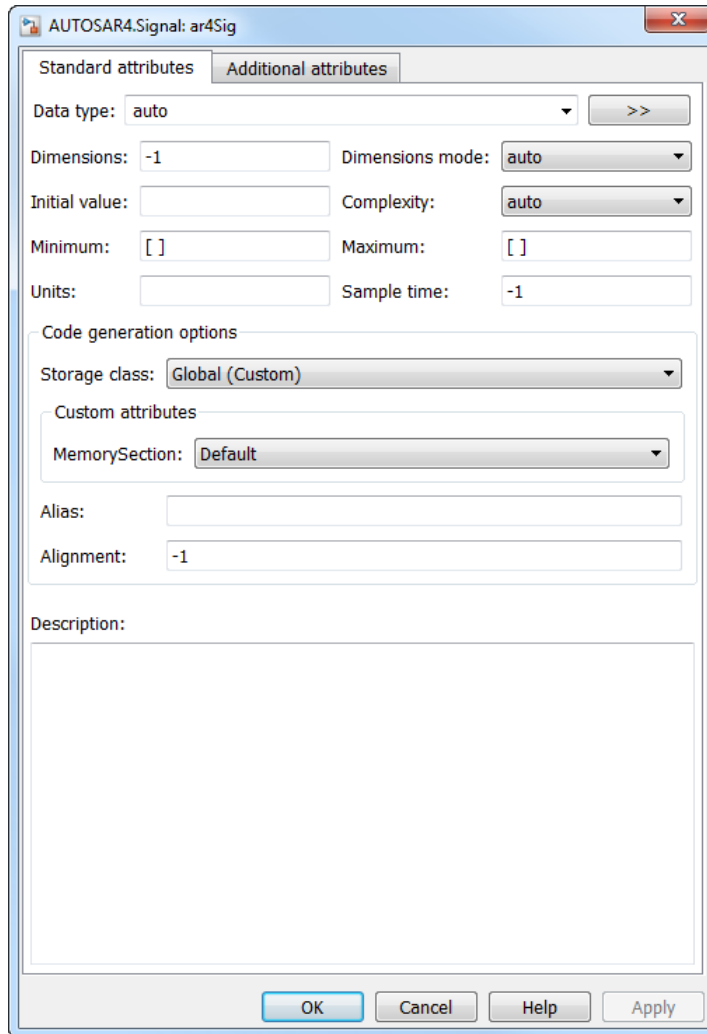
With this class, you can create workspace objects for mapping internal global signals to AUTOSAR memory sections. You can create an `AUTOSAR4.Signal` object in the base MATLAB workspace.

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

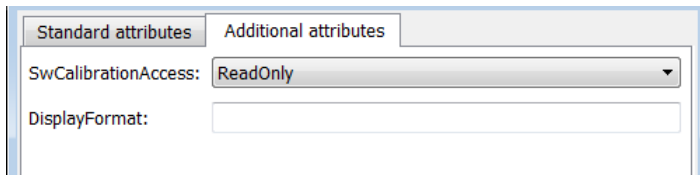


The `Simulink.Signal` reference page describes the signal attributes in detail. The `AUTOSAR4.Signal` class extends the `Simulink.Signal` class with the following additional selection for the **Storage class** attribute:

- **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 AUTOSAR Display Format for Measurement and Calibration” on page 4-143.

For more information, see

- “Static and Constant Memory” on page 2-38
- “Configure AUTOSAR Static or Constant Memory” on page 4-182
- “Configure AUTOSAR Data for Measurement and Calibration” on page 4-135

## 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 *Support Package Installer*, a wizard that guides you through the installation process. On the MATLAB **Home** tab, click **Add-Ons > Get Hardware Support Packages**. For more information, see “Support Package for AUTOSAR Standard” on page 1-3.

---

## Input Arguments

<i>importerObj</i>	Handle to imported AUTOSAR information created in a previous call to <code>arxml.importer</code> .
--------------------	--

*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'`

String specifying an existing Simulink data dictionary into which to import calibration objects corresponding to AUTOSAR data types in the XML file. 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 the existing 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 3-2

# 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 *Support Package Installer*, a wizard that guides you through the installation process. On the MATLAB **Home** tab, click **Add-Ons > Get Hardware Support Packages**. For more information, see “Support Package for AUTOSAR Standard” on page 1-3.

---

## Tip

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

<i>importerObj</i>	Handle to imported AUTOSAR information created in a previous call to <code>arxml.importer</code> .
<i>ComponentName</i>	Absolute short name path of the atomic software component.
<i>PropertyN, ValueN</i>	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 <code>arxml</code> 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 interrunnable 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 3-5 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.

'InitializationRunnable'

String specifying the name of an existing runnable as the initialization runnable for the component.

'DataDictionary'

String specifying an existing Simulink data dictionary into which to import data objects corresponding to AUTOSAR data types in the XML file. The model becomes associated with the specified data dictionary.

'NameConflictAction'

'overwrite' (default) or 'makenameunique' or 'error'. Use this property to determine the action 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 the existing 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 3-2

# 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 *Support Package Installer*, a wizard that guides you through the installation process. On the MATLAB **Home** tab, 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 4-188

- “Configure the AUTOSAR Interface” 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 string representing the model name.

Example: `arProps`

### **elementPath** — Path to AUTOSAR element

string

Path to the AUTOSAR element to delete.

Example: `'Input'`

## See Also

`add`



# 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 *Support Package Installer*, a wizard that guides you through the installation process. On the MATLAB **Home** tab, 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 4-188
- “Import AUTOSAR Software Component” on page 3-2
- “Configure the AUTOSAR Interface” 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 string representing the model name.

Example: arProps

## See Also

arxml.importer

---

# 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 *Support Package Installer*, a wizard that guides you through the installation process. On the MATLAB **Home** tab, 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 `IService` 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 string representing the model name.

Example: `arProps`

### **rootPath** — Starting point of the search

string

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

string

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 string, 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 4-190 lists properties that are associated with AUTOSAR elements.

Example: 'IsService', true

## Output Arguments

**paths** — Return structure

cell array of strings

Structure to which paths are returned.

Example: ifPaths

## See Also

add | delete | get | set

# 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 *Support Package Installer*, a wizard that guides you through the installation process. On the MATLAB **Home** tab, 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 4-188
- “Configure the AUTOSAR Interface” 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 string representing the model name.

Example: `arProps`

### **elementPath** — Path to AUTOSAR element

string

Path to the AUTOSAR element for which to return the value of a property.

Example: `'Input'`

### **property** — Type of property

string

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

# 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 *Support Package Installer*, a wizard that guides you through the installation process. On the MATLAB **Home** tab, click **Add-Ons > Get Hardware Support Packages**. For more information, see “Support Package for AUTOSAR Standard” on page 1-3.

---

## Input Arguments

<i>importerObj</i>	Handle to imported AUTOSAR information created in a previous call to <code>arxml.importer</code> .
--------------------	--

## Output Arguments

<i>applicationSoftwareComponentNames</i>	Cell array of strings. Each element is absolute short-name path of corresponding application software component: <code>'/root_package_name[/sub_package_name]/component_short_name'</code>
--	---



## **See Also**

arxml.importer.getSensorActuatorComponentNames |  
arxml.importer.getComponentNames

## **How To**

- “Import AUTOSAR Software Component” on page 3-2

## 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 *Support Package Installer*, a wizard that guides you through the installation process. On the MATLAB **Home** tab, 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 strings 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 3-2

## 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 *Support Package Installer*, a wizard that guides you through the installation process. On the **MATLAB Home** tab, 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 strings. 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 2-10
- “Import AUTOSAR Software Component” on page 3-2
- “Configure AUTOSAR Client-Server Communication” on page 4-92

## 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 *Support Package Installer*, a wizard that guides you through the installation process. On the MATLAB **Home** tab, click **Add-Ons** > **Get Hardware Support Packages**. For more information, see “Support Package for AUTOSAR Standard” on page 1-3.

---

### Input Arguments

<i>importerObj</i>	Handle to imported AUTOSAR information created in a previous call to <code>arxml.importer</code> .
--------------------	--

## Output Arguments

*componentNames*      Cell array of strings 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 3-2

## getDataTransfer

Get AUTOSAR mapping information for Simulink data transfer line

### Syntax

```
[arIrvName, arDataAccessMode]=getDataTransfer(s1Map,  
s1DataTransferName)
```

### Description

[arIrvName, arDataAccessMode]=getDataTransfer(s1Map, s1DataTransferName) returns the values of the AUTOSAR inter-runnable variable arIrvName and AUTOSAR data access mode arDataAccessMode that are mapped to Simulink data transfer line s1DataTransferName.

---

**Note:** This function requires the Embedded Coder Support Package for AUTOSAR Standard. You install support packages using *Support Package Installer*, a wizard that guides you through the installation process. On the MATLAB **Home** tab, 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 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  
s1Map=autosar.api.getSimulinkMapping('rtwdemo_autosar_multirunnables');  
[arIrvName, arDataAccessMode]=getDataTransfer(s1Map, 'irv4')  
  
arIrvName =  
IRV4
```



```
arDataAccessMode =
Implicit
```

- “Configure and Map AUTOSAR Component Programmatically” on page 4-188
- “Configure the AUTOSAR Interface” on page 4-3

## Input Arguments

**s1Map** — Simulink to AUTOSAR mapping information for a model  
handle

Simulink to AUTOSAR mapping information for a model, previously returned by `s1Map = autosar.api.getSimulinkMapping(model)`. `model` is a handle or string representing the model name.

Example: `s1Map`

**s1DataTransferName** — Name of model data transfer line  
string

Name of the model data transfer line for which to return AUTOSAR mapping information.

Example: `'irv4'`

## Output Arguments

**arIrvName** — Name of AUTOSAR inter-runnable variable  
string

Variable that returns the name of AUTOSAR inter-runnable variable mapped to the specified Simulink data transfer line.

Example: `arIrvName`

**arDataAccessMode** — Value of AUTOSAR data access mode  
string

Variable that returns the value of the AUTOSAR data access mode mapped to the specified Simulink data transfer line. The value is `Implicit` or `Explicit`.

Example: `arDataAccessMode`

**See Also**

`mapDataTransfer`

# 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 *Support Package Installer*, a wizard that guides you through the installation process. On the MATLAB **Home** tab, 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 strings.

## How To

- “Import AUTOSAR Software Component” on page 3-2

## 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 *Support Package Installer*, a wizard that guides you through the installation process. On the MATLAB **Home** tab, 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 3-2

## 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 *Support Package Installer*, a wizard that guides you through the installation process. On the MATLAB **Home** tab, 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 4-188

- “Configure the AUTOSAR Interface” on page 4-3

## Input Arguments

### **s1Map** — Simulink to AUTOSAR mapping information for a model

handle

Simulink to AUTOSAR mapping information for a model, previously returned by `s1Map = autosar.api.getSimulinkMapping(model)`. `model` is a handle or string representing the model name.

Example: `s1Map`

### **s1FcnName** — Name of Simulink entry point function

string

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 'StepFunctionN' for a sub-rate task, where *N* is the task identifier.
- For an exported function, 'FcnCallName', where *FcnCallName* 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`.
- For a server function, 'ServerFcnName', where *ServerFcnName* is the name of the Simulink function. For example, 'readData' in the example model used in “Configure AUTOSAR Server” on page 4-92.

Example: 'StepFunction2'

## Output Arguments

### **arRunnableName** — Name of AUTOSAR runnable

string

Variable that returns the name of the AUTOSAR runnable mapped to the specified Simulink entry-point function.

Example: `arRunnableName`

**See Also**

`mapFunction`



# getFunctionCaller

Get AUTOSAR mapping information for Simulink function-caller block

## Syntax

```
[arPortName,arOperationName] = getFunctionCaller(s1Map,s1FcnName)
```

## Description

[arPortName,arOperationName] = getFunctionCaller(s1Map,s1FcnName) returns the value of the AUTOSAR client port arPortName and AUTOSAR operation arOperationName mapped to the Simulink function caller block for Simulink function s1FcnName.

---

**Note:** This function requires the Embedded Coder Support Package for AUTOSAR Standard. You install support packages using *Support Package Installer*, a wizard that guides you through the installation process. On the MATLAB **Home** tab, 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')
s1MapC = autosar.api.getSimulinkMapping('mControllerWithInterface_client');
mapFunctionCaller(s1MapC,'readData','cPort','readData');
[arPort,arOp] = getFunctionCaller(s1MapC,'readData')

arPort =
cPort

arOp =
```

`readData`

- “Configure and Map AUTOSAR Component Programmatically” on page 4-188
- “Configure the AUTOSAR Interface” on page 4-3

## Input Arguments

**s1Map** — Simulink to AUTOSAR mapping information for a model

handle

Simulink to AUTOSAR mapping information for a model, previously returned by `s1Map = autosar.api.getSimulinkMapping(model)`. `model` is a handle or string representing the model name.

Example: `s1Map`

**s1FcnName** — Name of Simulink function

string

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

string

Variable that returns the name of the AUTOSAR client port mapped to the specified function-caller block.

Example: `arPort`

**arOperationName** — Name of AUTOSAR operation

string

Variable that returns the name of the AUTOSAR operation mapped to the specified function-caller block.

Example: `arOp`

**See Also**

mapFunctionCaller

## getInport

Get AUTOSAR mapping information for Simulink inport

### Syntax

```
[arPortName,arDataElementName,arDataAccessMode]=getInport(s1Map,  
s1PortName)
```

### Description

[arPortName,arDataElementName,arDataAccessMode]=getInport(s1Map, s1PortName) returns the values of the AUTOSAR port arPortName, AUTOSAR data element arDataElementName, and AUTOSAR data access mode arDataAccessMode mapped to Simulink inport s1PortName.

---

**Note:** This function requires the Embedded Coder Support Package for AUTOSAR Standard. You install support packages using *Support Package Installer*, a wizard that guides you through the installation process. On the MATLAB **Home** tab, 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  
s1Map=autosar.api.getSimulinkMapping('rtwdemo_autosar_multirunnables');  
[arPortName,arDataElementName,arDataAccessMode]=getInport(s1Map, 'RPort_DE1')  
  
arPortName =  
RPort  
  
arDataElementName =
```

DE1

```
arDataAccessMode =  
ImplicitReceive
```

- “Configure and Map AUTOSAR Component Programmatically” on page 4-188
- “Configure the AUTOSAR Interface” on page 4-3

## Input Arguments

### **s1Map** — Simulink to AUTOSAR mapping information for a model

handle

Simulink to AUTOSAR mapping information for a model, previously returned by `s1Map = autosar.api.getSimulinkMapping(model)`. `model` is a handle or string representing the model name.

Example: `s1Map`

### **s1PortName** — Name of model inport

string

Name of the model inport for which to return AUTOSAR mapping information.

Example: `'Input'`

## Output Arguments

### **arPortName** — Name of AUTOSAR port

string

Variable that returns the name of the AUTOSAR port mapped to the specified Simulink inport.

Example: `arPortName`

### **arDataElementName** — Name of AUTOSAR data element

string

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**

string

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`, or `EndToEndRead`.

Example: `arDataAccessMode`

**See Also**

`mapInport`

# getOutput

Get AUTOSAR mapping information for Simulink output

## Syntax

```
[arPortName,arDataElementName,arDataAccessMode]=getOutput(slMap,slPortName)
```

## Description

[arPortName,arDataElementName,arDataAccessMode]=getOutput(slMap,slPortName) returns the values of the AUTOSAR provider port arPortName, AUTOSAR data element arDataElementName, and AUTOSAR data access mode arDataAccessMode mapped to Simulink output slPortName.

---

**Note:** This function requires the Embedded Coder Support Package for AUTOSAR Standard. You install support packages using *Support Package Installer*, a wizard that guides you through the installation process. On the MATLAB **Home** tab, 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 Output

Get AUTOSAR mapping information for a model output in the example model `rtwdemo_autosar_multirunnables`. The model has an output named `PPort_DE1`.

```
rtwdemo_autosar_multirunnables
slMap=autosar.api.getSimulinkMapping('rtwdemo_autosar_multirunnables');
[arPortName,arDataElementName,arDataAccessMode]=getOutput(slMap,'PPort_DE1')

arPortName =
PPort

arDataElementName =
```

DE1

```
arDataAccessMode =  
ImplicitSend
```

- “Configure and Map AUTOSAR Component Programmatically” on page 4-188
- “Configure the AUTOSAR Interface” on page 4-3

## Input Arguments

### **s1Map** — Simulink to AUTOSAR mapping information for a model

handle

Simulink to AUTOSAR mapping information for a model, previously returned by `s1Map = autosar.api.getSimulinkMapping(model)`. `model` is a handle or string representing the model name.

Example: `s1Map`

### **s1PortName** — Name of model output

string

Name of the model output for which to return AUTOSAR mapping information.

Example: `'Output'`

## Output Arguments

### **arPortName** — Name of AUTOSAR port

string

Variable that returns the name of the AUTOSAR port mapped to the specified Simulink output.

Example: `arPortName`

### **arDataElementName** — Name of AUTOSAR data element

string

Variable that returns the name of the AUTOSAR data element mapped to the specified Simulink output.



Example: arDataElementName

**arDataAccessMode** — Value of AUTOSAR data access mode

string

Variable that returns the value of the AUTOSAR data access mode mapped to the specified Simulink outport. The value can be `ImplicitSend`, `ImplicitSendByRef`, `ExplicitSend`, or `EndToEndWrite`.

Example: arDataAccessMode

**See Also**

mapOutport

## 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 *Support Package Installer*, a wizard that guides you through the installation process. On the MATLAB **Home** tab, 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

*sensoractuatorSoftwareComponentNames*      Cell array of strings. 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 3-2

## mapDataTransfer

Map Simulink data transfer line to AUTOSAR inter-runnable variable

### Syntax

```
mapDataTransfer(s1Map, s1DataTransferName, arIrvName, arDataAccessMode)
```

### Description

`mapDataTransfer(s1Map, s1DataTransferName, arIrvName, arDataAccessMode)` maps the Simulink data transfer line `s1DataTransferName` 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 *Support Package Installer*, a wizard that guides you through the installation process. On the MATLAB **Home** tab, 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 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
s1Map=autosar.api.getSimulinkMapping('rtwdemo_autosar_multirunnables');
mapDataTransfer(s1Map, 'irv4', 'IRV4', 'Explicit');
[arIrvName, arDataAccessMode]=getDataTransfer(s1Map, 'irv4')

arIrvName =
IRV4

arDataAccessMode =
```

Explicit

- “Configure and Map AUTOSAR Component Programmatically” on page 4-188
- “Configure the AUTOSAR Interface” on page 4-3

## Input Arguments

### **s1Map** — Simulink to AUTOSAR mapping information for a model

handle

Simulink to AUTOSAR mapping information for a model, previously returned by `s1Map = autosar.api.getSimulinkMapping(model)`. `model` is a handle or string representing the model name.

Example: `s1Map`

### **s1DataTransferName** — Name of model data transfer line

string

Name of the model data transfer line for which to set AUTOSAR mapping information.

Example: `'irv4'`

### **arIrvName** — Name of AUTOSAR inter-runnable variable

string

Name of the AUTOSAR inter-runnable variable to which to map the specified Simulink data transfer line.

Example: `'IRV4'`

### **arDataAccessMode** — Value of AUTOSAR data access mode

string

Value of the AUTOSAR data access mode to which to map the specified Simulink data transfer line. The value can be `Implicit` or `Explicit`.

Example: `'Explicit'`

## See Also

`getDataTransfer`

## mapFunction

Map Simulink entry-point function to AUTOSAR runnable

### Syntax

```
mapFunction(s1Map,s1FcnName,arRunnableName)
```

### Description

`mapFunction(s1Map,s1FcnName,arRunnableName)` maps the Simulink entry-point function `s1FcnName` to the AUTOSAR runnable `arRunnableName`.

---

**Note:** This function requires the Embedded Coder Support Package for AUTOSAR Standard. You install support packages using *Support Package Installer*, a wizard that guides you through the installation process. On the MATLAB **Home** tab, 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')
s1Map=autosar.api.getSimulinkMapping('rtwdemo_autosar_multirunnables');
mapFunction(s1Map,'InitializeFunction','Runnable_Init');
arRunnableName=getFunction(s1Map,'InitializeFunction')

arRunnableName =
Runnable_Init
```

- “Configure and Map AUTOSAR Component Programmatically” on page 4-188
- “Configure the AUTOSAR Interface” on page 4-3

## Input Arguments

### **s1Map** — Simulink to AUTOSAR mapping information for a model

handle

Simulink to AUTOSAR mapping information for a model, previously returned by `s1Map = autosar.api.getSimulinkMapping(model)`. `model` is a handle or string representing the model name.

Example: `s1Map`

### **s1FcnName** — Name of Simulink entry point function

string

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 `'StepFunctionN'` for a sub-rate task, where `N` is the task identifier.
- For an exported function, `'FcnCallName'`, where `FcnCallName` 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`.
- For a server function, `'ServerFcnName'`, where `ServerFcnName` is the name of the Simulink function. For example, `'readData'` in the example model used in “Configure AUTOSAR Server” on page 4-92.

Example: `'StepFunction2'`

### **arRunnableName** — Name of AUTOSAR runnable

string

Name of the AUTOSAR runnable to which to map the specified Simulink entry-point function.

Example: `'Runnable2'`

## See Also

`getFunction`

## mapFunctionCaller

Map Simulink function-caller block to AUTOSAR client port and operation

### Syntax

```
mapFunctionCaller(s1Map,s1FcnName,arPortName,arOperationName)
```

### Description

`mapFunctionCaller(s1Map,s1FcnName,arPortName,arOperationName)` maps the Simulink function-caller block for Simulink function `s1FcnName` to AUTOSAR client port `arPortName` and AUTOSAR operation `arOperationName`.

If your model has multiple callers of Simulink function `s1FcnName`, 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 *Support Package Installer*, a wizard that guides you through the installation process. On the MATLAB **Home** tab, 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')
s1MapC = autosar.api.getSimulinkMapping('mControllerWithInterface_client');
mapFunctionCaller(s1MapC,'readData','cPort','readData');
[arPort,arOp] = getFunctionCaller(s1MapC,'readData')

arPort =
```



cPort  
arOp =  
readData

- “Configure and Map AUTOSAR Component Programmatically” on page 4-188
- “Configure the AUTOSAR Interface” on page 4-3

## Input Arguments

**s1Map** — Simulink to AUTOSAR mapping information for a model  
handle

Simulink to AUTOSAR mapping information for a model, previously returned by `s1Map = autosar.api.getSimulinkMapping(model)`. `model` is a handle or string representing the model name.

Example: `s1Map`

**s1FcnName** — Name of Simulink function  
string

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  
string

Name of the AUTOSAR client port to which to map the specified function-caller block.

Example: `'cPort'`

**arOperationName** — Name of AUTOSAR operation  
string

Name of the AUTOSAR operation to which to map the specified function-caller block.

Example: `'readData'`

## See Also

`getFunctionCaller`

## mapInport

Map Simulink inport to AUTOSAR port

### Syntax

```
mapInport(sIMap, sIPortName, arPortName, arDataElementName,  
arDataAccessMode)
```

### Description

`mapInport(sIMap, sIPortName, arPortName, arDataElementName, arDataAccessMode)` maps the Simulink inport `sIPortName` 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 *Support Package Installer*, a wizard that guides you through the installation process. On the MATLAB **Home** tab, 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  
sIMap=autosar.api.getSimulinkMapping('rtwdemo_autosar_multirunnables');  
mapInport(sIMap, 'RPort_DE1', 'RPort', 'DE1', 'ExplicitReceive');  
[arPortName, arDataElementName, arDataAccessMode]=getInport(sIMap, 'RPort_DE1')  
  
arPortName =  
RPort
```

```
arDataElementName =
DE1
```

```
arDataAccessMode =
ExplicitReceive
```

- “Configure and Map AUTOSAR Component Programmatically” on page 4-188
- “Configure the AUTOSAR Interface” on page 4-3

## Input Arguments

### **s1Map** — Simulink to AUTOSAR mapping information for a model

handle

Simulink to AUTOSAR mapping information for a model, previously returned by `s1Map = autosar.api.getSimulinkMapping(model)`. `model` is a handle or string representing the model name.

Example: `s1Map`

### **s1PortName** — Name of model inport

string

Name of the model inport for which to set AUTOSAR mapping information.

Example: `'Input'`

### **arPortName** — Name of AUTOSAR port

string

Name of the AUTOSAR port to which to map the specified Simulink inport.

Example: `'Input'`

### **arDataElementName** — Name of AUTOSAR data element

string

Name of the AUTOSAR data element to which to map the specified Simulink inport.

Example: `'Input'`

### **arDataAccessMode** — Value of AUTOSAR data access mode

string

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`, or `EndToEndRead`.

Example: `'ExplicitReceive'`

### **See Also**

`getInport`

# mapOutput

Map Simulink outport to AUTOSAR port

## Syntax

```
mapOutput(s1Map,s1PortName,arPortName,arDataElementName,
arDataAccessMode)
```

## Description

mapOutput(s1Map,s1PortName,arPortName,arDataElementName,arDataAccessMode) maps the Simulink outport s1PortName 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 *Support Package Installer*, a wizard that guides you through the installation process. On the MATLAB **Home** tab, 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
s1Map=autosar.api.getSimulinkMapping('rtwdemo_autosar_multirunnables');
mapOutput(s1Map,'PPort_DE1','PPort','DE1','ExplicitSend');
[arPortName,arDataElementName,arDataAccessMode]=getOutport(s1Map,'PPort_DE1')

arPortName =
PPort
```

```
arDataElementName =  
DE1
```

```
arDataAccessMode =  
ExplicitSend
```

- “Configure and Map AUTOSAR Component Programmatically” on page 4-188
- “Configure the AUTOSAR Interface” on page 4-3

## Input Arguments

### **s1Map** — Simulink to AUTOSAR mapping information for a model

handle

Simulink to AUTOSAR mapping information for a model, previously returned by `s1Map = autosar.api.getSimulinkMapping(model)`. `model` is a handle or string representing the model name.

Example: `s1Map`

### **s1PortName** — Name of model output

string

Name of the model output for which to set AUTOSAR mapping information.

Example: `'Output'`

### **arPortName** — Name of AUTOSAR port

string

Name of the AUTOSAR port to which to map the specified Simulink output.

Example: `'Output'`

### **arDataElementName** — Name of AUTOSAR data element

string

Name of the AUTOSAR data element to which to map the specified Simulink output.

Example: `'Output'`

### **arDataAccessMode** — Value of AUTOSAR data access mode

string

Value of the AUTOSAR data access mode to which to map the specified Simulink output. The value can be `ImplicitSend`, `ImplicitSendByRef`, `ExplicitSend`, or `EndToEndWrite`.

Example: `'ExplicitSend'`

### **See Also**

`getOutputport`

## 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 a 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 *Support Package Installer*, a wizard that guides you through the installation process. On the **MATLAB Home** tab, 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', 'ComponentQualifiedNames');
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 4-188
- “Configure the AUTOSAR Interface” 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 string representing the model name.

Example: *arProps*

### **elementPath** — Path to an AUTOSAR element

string

Path to an AUTOSAR element for which to set a property. To set XML packaging options, specify *XmlOptions*.

Example: 'Input'

### **property** — Type of property

string

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

# 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 *Support Package Installer*, a wizard that guides you through the installation process. On the MATLAB **Home** tab, click **Add-Ons > Get Hardware Support Packages**. For more information, see “Support Package for AUTOSAR Standard” on page 1-3.

---

## Input Arguments

<i>importerObj</i>	Handle to imported AUTOSAR information created in a previous call to <code>arxml.importer</code> .
<i>dependencies</i>	Can be: <ul style="list-style-type: none"><li>• a cell array of strings (for a list of dependencies)</li><li>• a char array (for a single dependency)</li><li>• or the empty array <code>[]</code> (for removing a dependency)</li></ul>

---

**Note:** The atomic software components described in the XML file dependencies are ignored.

---

## **How To**

- “Import AUTOSAR Software Component” on page 3-2

---

## 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 *Support Package Installer*, a wizard that guides you through the installation process. On the MATLAB **Home** tab, click **Add-Ons > Get Hardware Support Packages**. For more information, see “Support Package for AUTOSAR Standard” on page 1-3.

---

## Input Arguments

<i>importerObj</i>	Handle to imported AUTOSAR information created in a previous call to <code>arxml.importer</code> .
<i>filename</i>	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 3-2

# updateModel

**Class:** arxml.importer

**Package:** arxml

Merge AUTOSAR XML changes into associated mapped Simulink model

## 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/merge 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 *Support Package Installer*, a wizard that guides you through the installation process. On the MATLAB **Home** tab, click **Add-Ons > Get Hardware Support Packages**. For more information, see “Support Package for AUTOSAR Standard” on page 1-3.

---

## Input Arguments

<i>importerObj</i>	Handle to imported AUTOSAR information created in a previous call to <code>arxml.importer</code> .
<i>modelName</i>	Name of the model to be updated with changes in the imported AUTOSAR description represented by <i>importerObj</i> . 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
### Creating HTML report mySWC_update_report.html
>>
```

### How To

- “Merge AUTOSAR Authoring Tool Changes Into Model” on page 3-6



# Blocks — Alphabetical List

---

Signal Invalidation

# Signal Invalidation

Control AUTOSAR root output 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 output block.

The following modeling rules apply to using a Signal Invalidation block.

- The block output:
  - Destination should be a root output 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 output, 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 4-74

