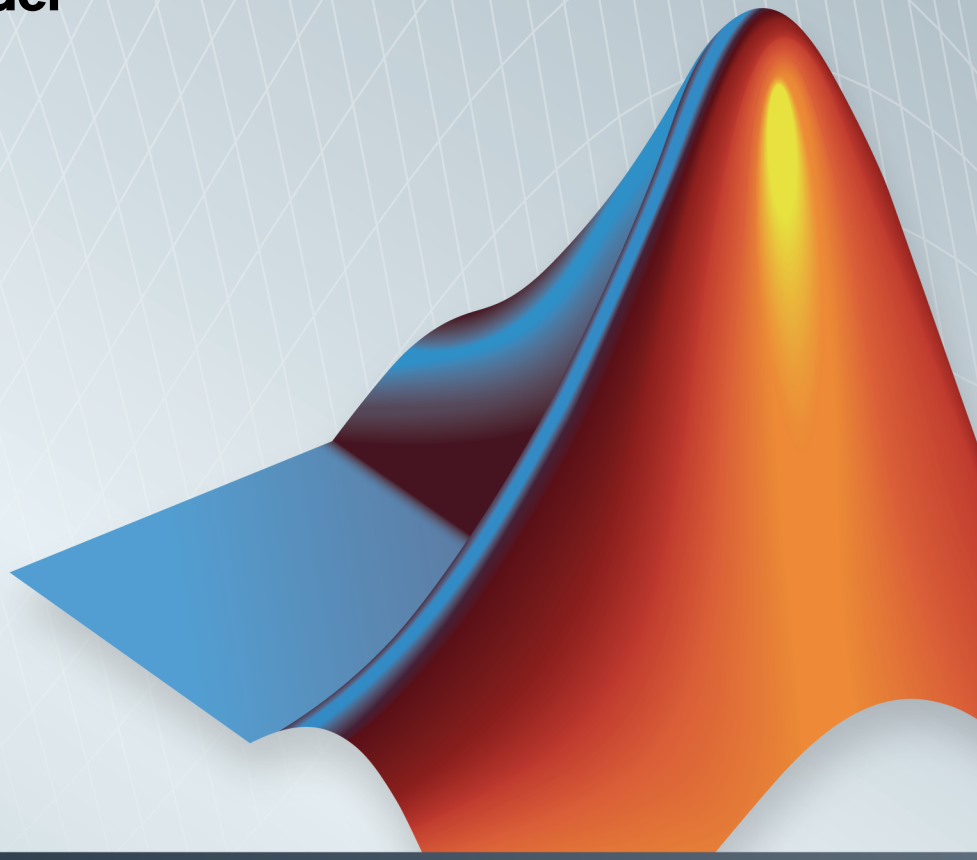
Embedded Coder®

AUTOSAR

R2014**b**



MATLAB® & SIMULINK®

MathWorks®

How to Contact MathWorks

Latest news: www.mathworks.com

Sales and services: www.mathworks.com/sales_and_services

User community: www.mathworks.com/matlabcentral

Technical support: www.mathworks.com/support/contact_us

Phone: 508-647-7000

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

Embedded Coder® AUTOSAR

Revision History

October 2014	Online only	New for Version 6.7 (Release 2014b)

Check Bug Reports for Issues and Fixes

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

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

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

Contents

1 Overview of AUTOSAR Support

2 Modeling Patterns for AUTOSAR

3 AUTOSAR Component Creation

4 AUTOSAR Component Development

5 AUTOSAR Code Generation

6 Functions — Alphabetical List

1

Overview of AUTOSAR Support

AUTOSAR Standard

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

The AUTOSAR standard addresses:

- Architecture – Three layers, *Application*, *Runtime Environment* (RTE), and *Basic Software*, enable decoupling of AUTOSAR Software Components from the execution platform. Standard interfaces between AUTOSAR Software Components and the Runtime Environment allow reuse or relocation of components within the Electronic Control Unit (ECU) topology of a vehicle.
- Methodology – Specification of code formats and description file templates, for example.
- Application Interfaces – Specification of interfaces for typical automotive applications.

For more information, see:

- `www.autosar.org` for details on the AUTOSAR standard.
- "Simulink Modeling Patterns for AUTOSAR" on page 2-2 to model AUTOSAR Software Components and related concepts in Simulink®.
- "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® 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® Toolstrip, 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”.

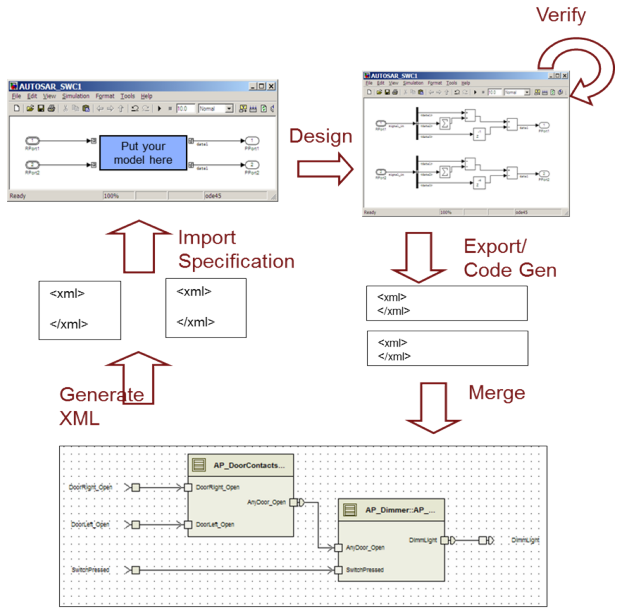
Workflows for AUTOSAR

This section 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 4-74

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 4-2
   - "Configure AUTOSAR Client-Server Communication" on page 4-59
   - "Configure and Map AUTOSAR Component Programmatically" on page 4-116

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 4-42

   You can also verify your generated code in a simulation. See "Verify AUTOSAR C Code with SIL and PIL" on page 5-27.

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

   You can use the authoring tool to export specifications, which can be imported back into Simulink. During the import, you can request that `arxml` changes be detected and merged into the model. See "Merge AUTOSAR Authoring Tool Changes Into Model" on page 3-5.

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 3-10. Subsequent tasks in the workflow are as listed above.

Sample Workflows

Embedded Coder provides the following scripts and models to demonstrate AUTOSAR workflows.

Example	How to ...
AUTOSAR Code Generation: `rtwdemo_autosar_legacy_script`	Generate AUTOSAR-compliant code and export AUTOSAR Software Component description XML files from a Simulink model.
AUTOSAR Code Generation for Multiple Runnable Entities: `rtwdemo_autosar_multirunnables_script`	Configure and generate AUTOSAR-compliant code and export AUTOSAR Software Component description XML files for a Simulink model with multiple runnables.
Import and Export an AUTOSAR Software Component: `rtwdemo_autosar_roundtrip_script`	Use an AUTOSAR authoring tool with Simulink to develop AUTOSAR Software Components. Learn how to import software component interfaces into Simulink, modify and export them, and merge the completed software component back into an AUTOSAR authoring tool.
Using Data Stores to Access Per-Instance Memory: `rtwdemo_autosar_PIM_script`	Publish an AUTOSAR Software Component with per-instance memory.

AUTOSAR Terminology

Term	Notes

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

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

Term	Notes
ErrorStatus	Indicates errors detected by communication system. Runtime Environment defines the following macros for sender-receiver communication: • `RTE_E_OK: no errors` • `RTE_E_INVALID: data element invalid` • `RTE_E_MAX_AGE_EXCEEDED: data element outdated`
OperationPrototype (operation)	• Invoked by a client • Provides value for each argument with direction `in` or `inout`, which must be of the corresponding data type • Client expects to receive a response to the invoked operation, part of which is a value with direction `out` or `inout`
PortInterface	• Characterizes information provided or required by a port • Can be either Sender-Receiver Interface or Client-Server Interface
Primitive data types	Category of data types that allow a direct mapping to C intrinsic types.
Provide port (PPort)	Port providing data or service of a server.
Require port (RPort)	Port requiring data or service of a server.
RTEEvent	Event or situation that triggers execution of a runnable by the Runtime Environment (RTE). The software supports the following RTEEvents: • OperationInvokedEvent (applicable to server operations) • TimingEvent • DataReceivedEvent
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	• PortInterface for sender-receiver communication • Defines data elements sent by sending component (with `Provide` port providing Sender-Receiver Interface) or received by receiving component (with `Require` requiring Sender-Receiver Interface)
Sender-Receiver Annotation	Annotation of data elements in a port that implements Sender-Receiver Interface.
Sensor Actuator Software Component	AUTOSAR Software Component dedicated to the control of a sensor or actuator.
Service	Logical entity of Basic Software that offers functionality, which is used by various AUTOSAR Software Components.

Modeling Patterns for AUTOSAR

Simulink Modeling Patterns for AUTOSAR

The following sections 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

In this section...

About AUTOSAR Software Components

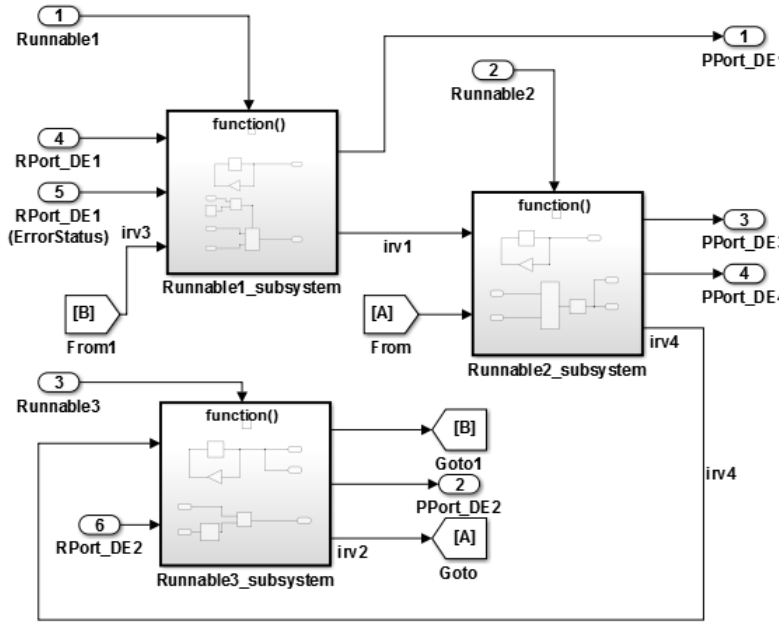
In AUTOSAR, application software consists of separate units, *AUTOSAR Software Components* (ASWCs).

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 2-14

An AUTOSAR Software Component interacts with other 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).

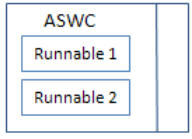
In Simulink, you represent an AUTOSAR Software Component using a model. For example, in the example model rtwdemo_autosar_multirunnables, shown below, the model represents an AUTOSAR Software Component, and function-call subsystems represent runnables that implement its behavior. The function-call subsystems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



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 Runtime Environment (RTE). For example, TimingEvent is an RTEEvent that is generated periodically.

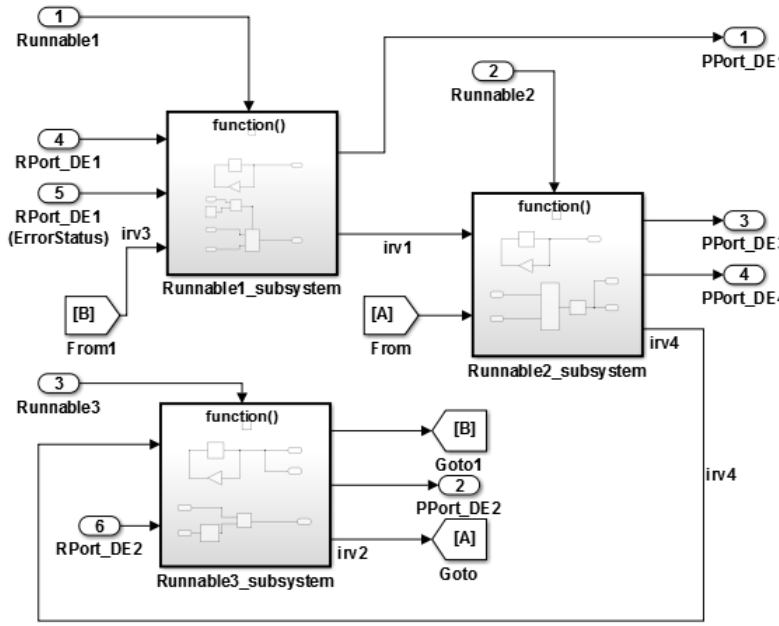


A component also can contain a single runnable, represented by a model, and can be single-rate or multirate. However, the software implements each component as a single-tasking operation.

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 4-105.

Multi-Instance Atomic 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

Model AUTOSAR Communication

In this section...

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

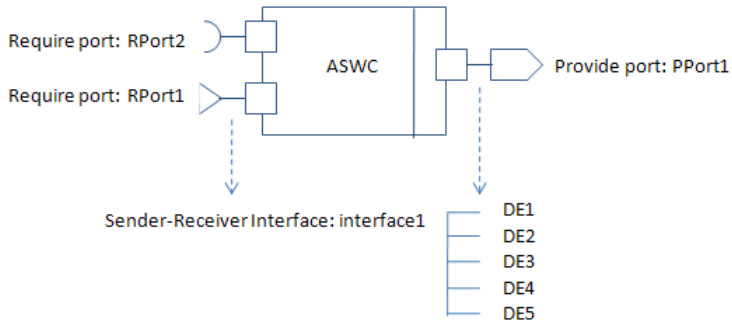
The following figure shows an AUTOSAR Software Component with four ports representing the port and interface combinations for Sender-Receiver and Client-Server interfaces.

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

Sender-Receiver Interface

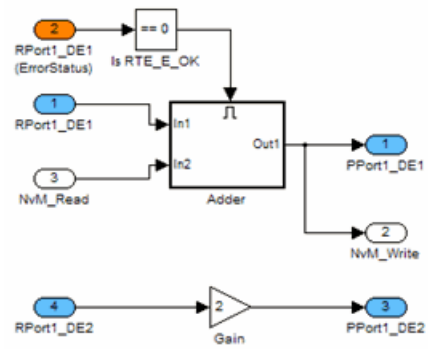
A Sender-Receiver Interface consists of one or more data elements. Although a `Require`, `Provide`, or `Provide-Require` port may reference a Sender-Receiver Interface, the

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 4-48.

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

Use the 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 4-2.

Client-Server Interface

AUTOSAR allows client-server communication between:

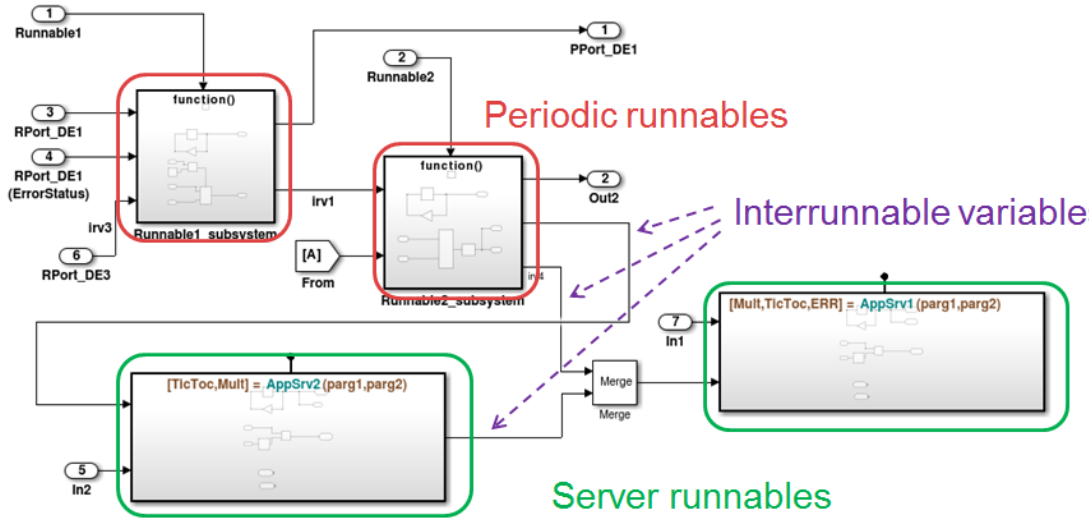
- Application software components
- An application software component and Basic Software

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 top-model export-functions modeling style to create interconnected Simulink functions, function-calls, and root model inports and outports.

The following diagram illustrates an export-functions framework in which Simulink Function blocks model AUTOSAR server runnables, Function Caller blocks model AUTOSAR client invocations, and Simulink data transfer lines model AUTOSAR inter-runnable variables (IRVs).



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

1. Model server functions and caller blocks in Simulink. For example, create Simulink Function blocks at the root level of a model, with corresponding Function Caller blocks that call the functions. Use the Simulink toolset to simulate and develop the blocks.
2. In the context of a model configured for AUTOSAR, map and configure the Simulink functions to AUTOSAR server runnables. Validate the configuration, simulate, and generate C code and `arxml` from the model.
3. In the context of another model configured for AUTOSAR, map and configure function caller blocks to AUTOSAR client ports and AUTOSAR operations. Validate the configuration, simulate, and generate C code and `arxml` from the model.
4. Integrate the generated C code into a test framework for testing, for example, with SIL simulation. (Ultimately, the generated C code and `arxml` are integrated into the AUTOSAR run-time environment (RTE).)

For a detailed example, see “Configure AUTOSAR Client-Server Communication” on page 4-59.

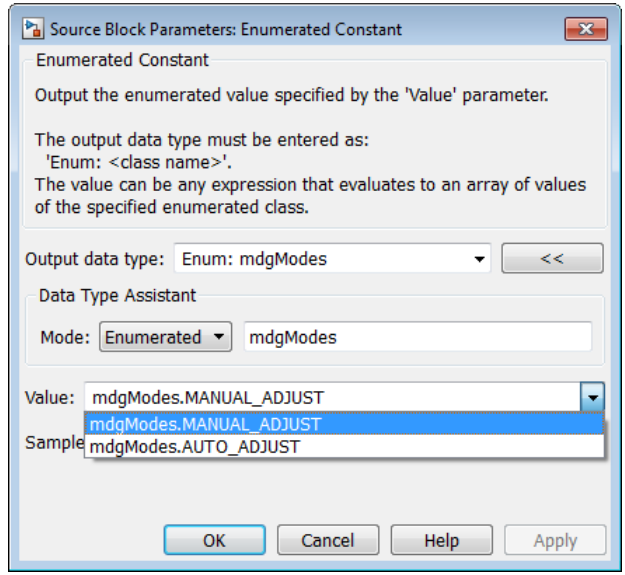
Mode-Switch Interface

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

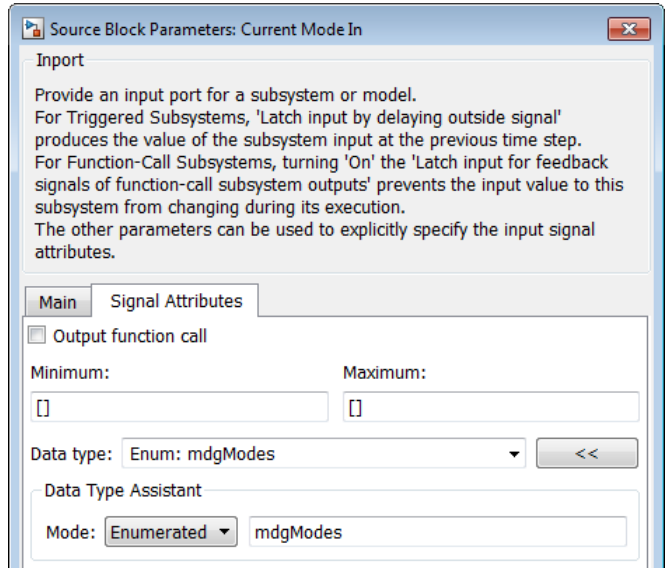
- Model the mode receiver port for an AUTOSAR software component using a Simulink inport.
- Specify a mode-switch event to trigger an initialize runnable or other runnable.

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

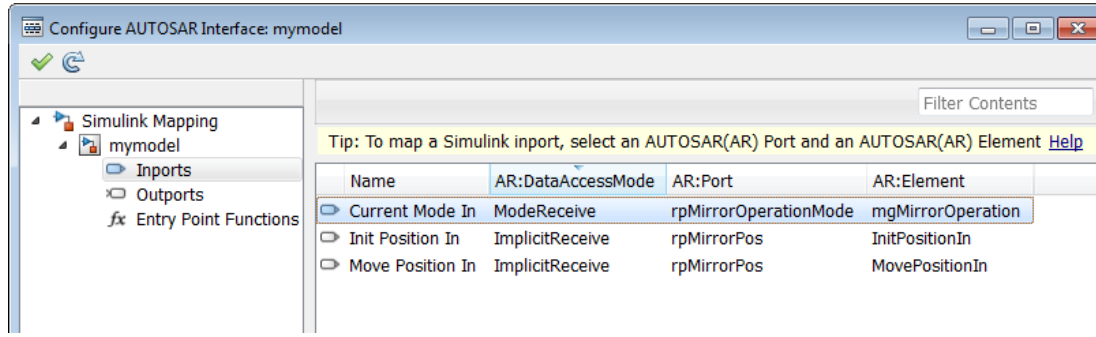
1. Declare a mode declaration group — a group of mode values — using Simulink enumeration. For example, you could create an enumerated type `mdgModes`, with enumerated values `MANUAL_ADJUST` and `AUTO_ADJUST`, as displayed in the Enumerated Constant block properties dialog box below.



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

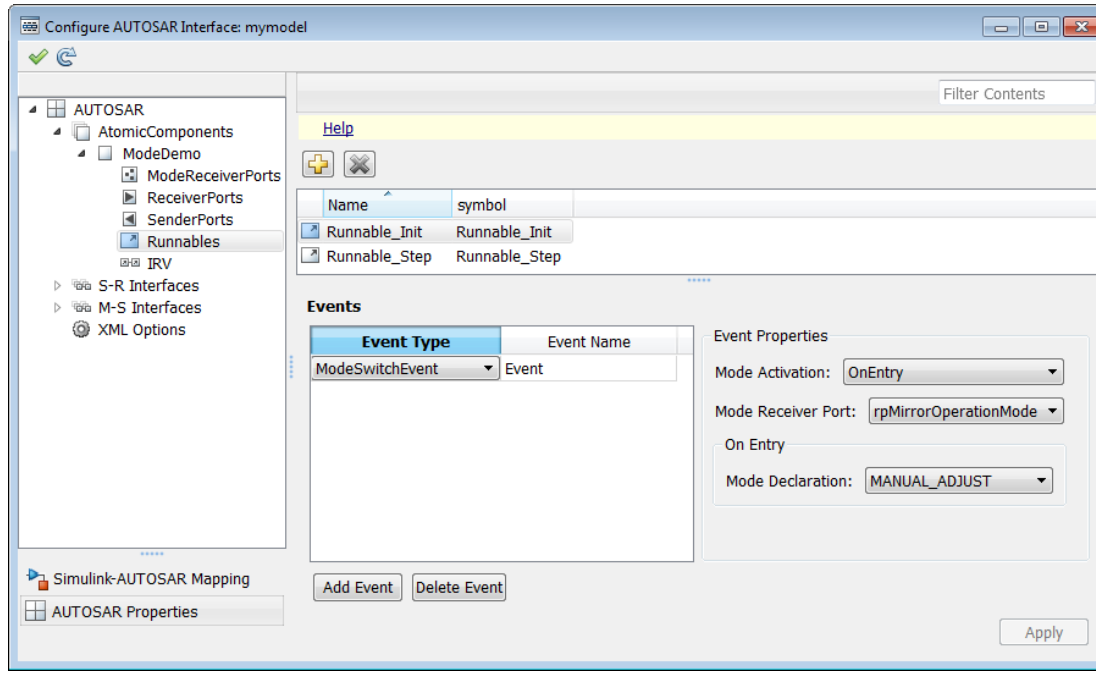


3 Use the Configure AUTOSAR Interface dialog box in the **Simulink-AUTOSAR Mapping** view (or equivalent API commands) to specify the mapping of the Simulink inport to the AUTOSAR mode-receiver port. In the following example, Simulink inport `Current Mode In` is mapped to AUTOSAR mode-receiver port `rpMirrorOperationMode` 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 was added for `Runnable_Init`, and configured to activate on entry (versus on exit or on transition). It was 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 4-52.

Model AUTOSAR Calibration Parameters

In this section...

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` into Simulink or create a calibration component in Simulink. See "Calibration Components" on page 2-15.
- Internal calibration parameters, which are defined and accessed by only one AUTOSAR Software Component.

The software supports import, export, and 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.

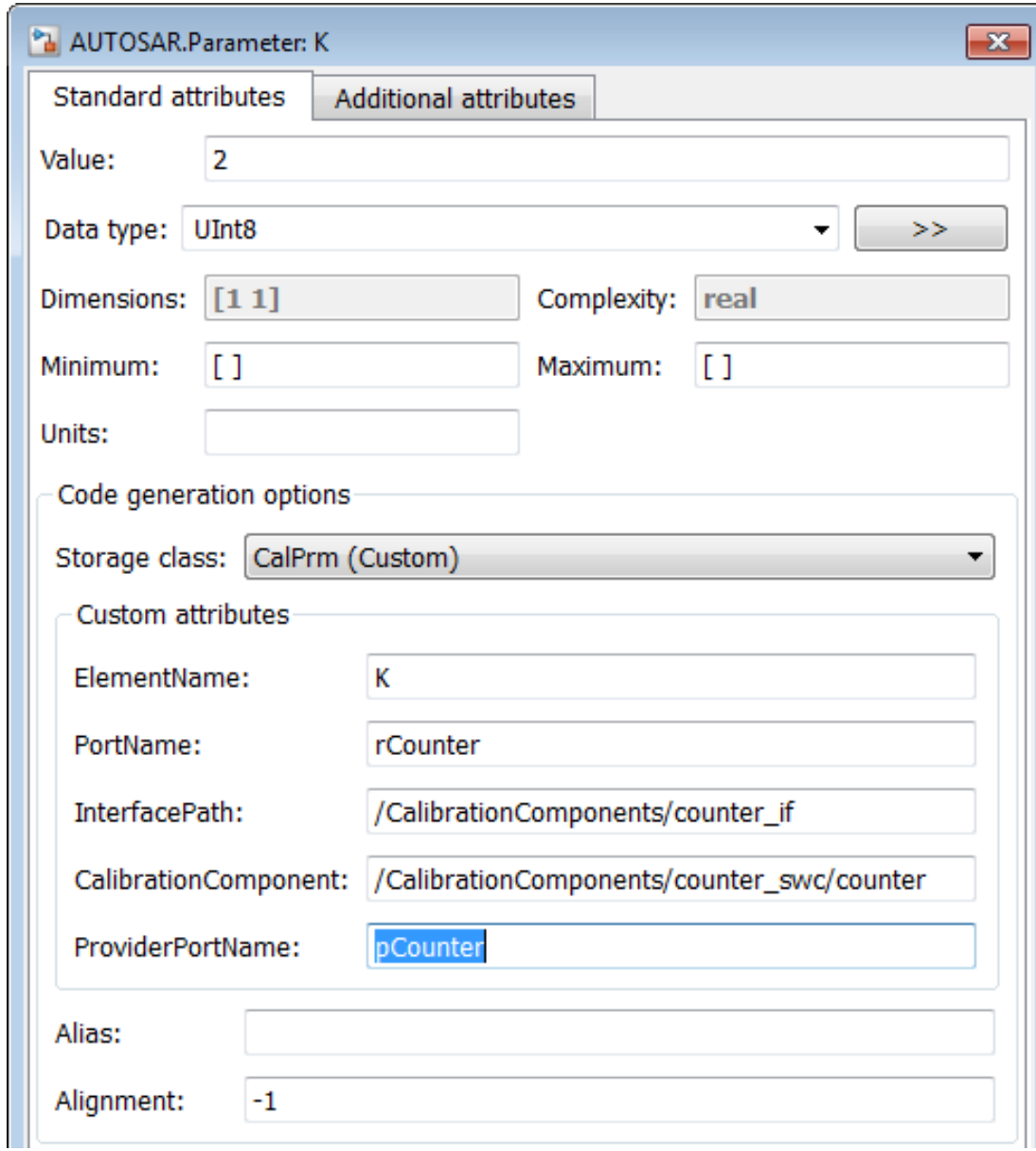
For more information, see "Import AUTOSAR Software Component" on page 3-2.

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 4-74 and rtwdemo_autosar_legacy_script.

Calibration Components

An AUTOSAR calibration component (`ParameterSwComponent`) contains calibration parameters that can be accessed by ASWCs 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
ElementName:
K
PortName:
rCounter
InterfacePath:
/CalibrationComponents/counter_if
CalibrationComponent:
/CalibrationComponents/counter_swc/counter
ProviderPortName:
pCounter
Alias:
Alignment:
-1

For a detailed example, see “Configure AUTOSAR Calibration Component” on page 4-76.

Model AUTOSAR Data Types

In this section...

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.

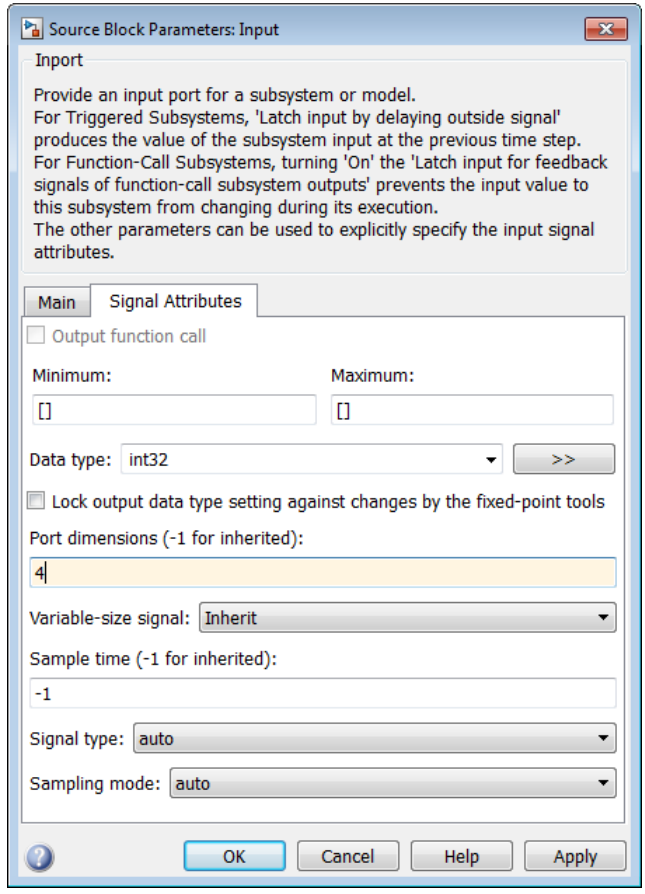
You can use Simulink data types to define AUTOSAR primitive types.

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

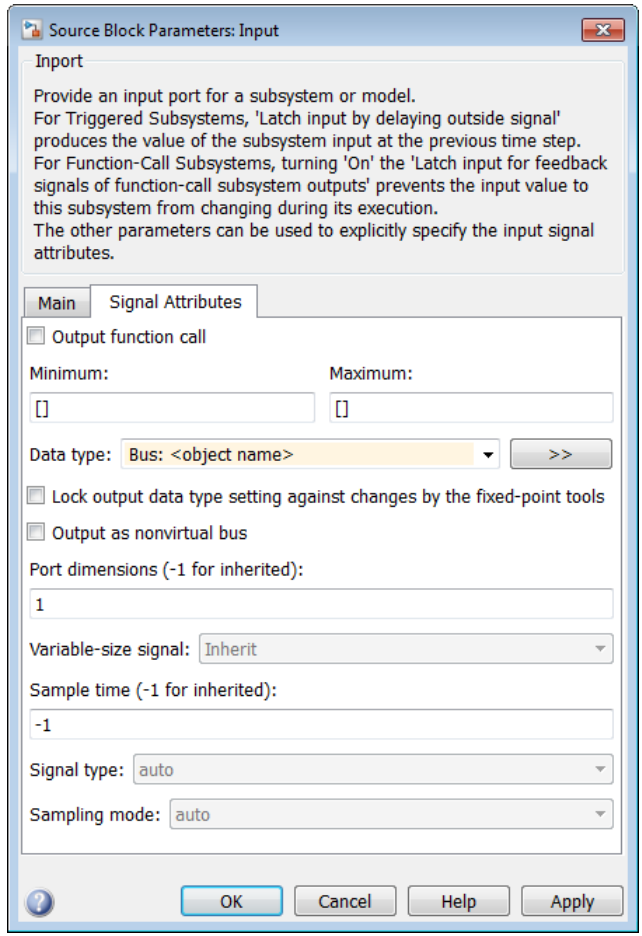
AUTOSAR Data Type	Simulink Data Type
`uint16`	`uint16`
`uint32`	`uint32`

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.

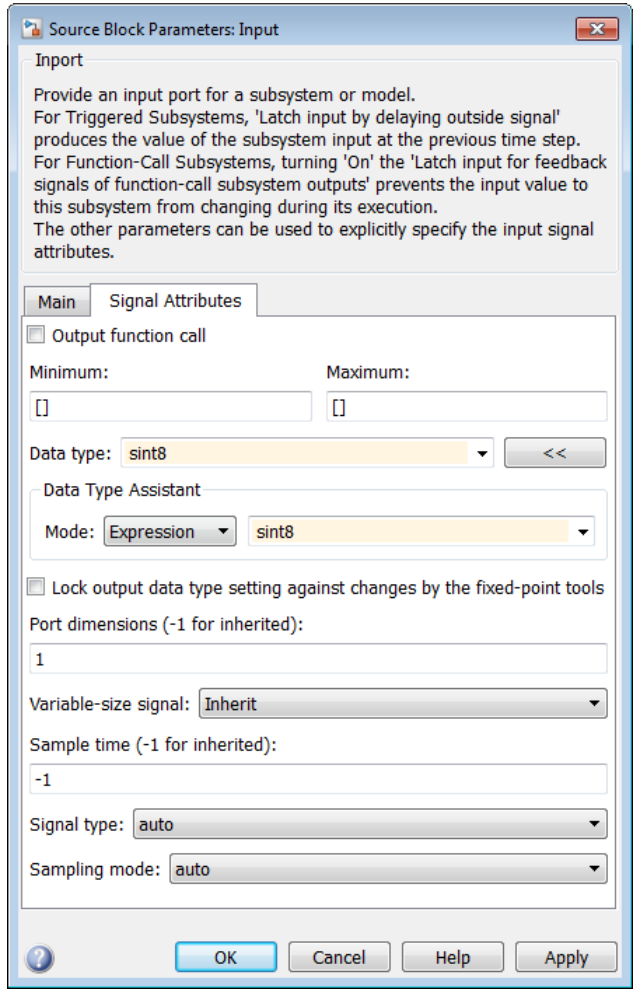
The following figure shows how to specify a wide signal, which corresponds to an AUTOSAR composite array.



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 2-23.

AUTOSAR	Simulink

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

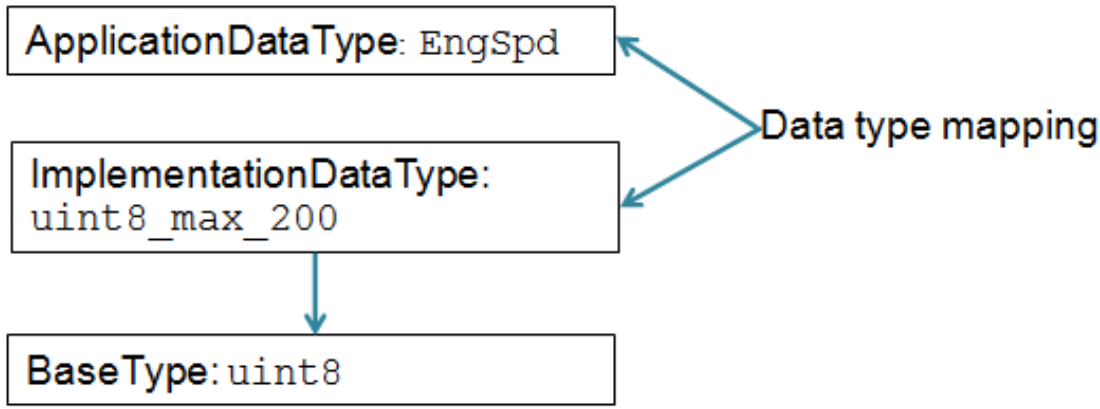
Primitive type (fixed point), for example, `myFixPt`	`>> myFixPt = Simulink.NumericType;`
	`>> myFixPt.DataTypeMode = ...`
	`>> myFixPt.IsAlias = true;`
	`>> myFixPt.HeaderFile = 'Rte_Type.h';`

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

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

Release 4.x Data Types

AUTOSAR Release 4.0 introduced a new approach to AUTOSAR data types, in which base data types are mapped to implementation data types and application data types. Application and implementation data types separate application-level physical attributes, such as real-world range of values, data structure, and physical semantics, from implementation-level attributes, such as stored-integer minimum and maximum and specification of a primitive-type (integer, Boolean, real, and so on).



The software supports AUTOSAR R4.x compliant data types in Simulink originated and round-trip workflows:

- For AUTOSAR components originated in Simulink, the software generates AUTOSAR application, implementation, and base types to preserve the information contained within Simulink data types.
- For round-trip workflows involving AUTOSAR components originated outside MATLAB, the `arxml` importer and exporter preserve data type information and mapping for each imported AUTOSAR data type.

For more details about workflow support, see “R4.x Data Types in Simulink Originated Workflow” on page 2-25 and “R4.x Data Types in Round-Trip Workflow” on page 2-26.

For information about mapping value constraints between AUTOSAR application data types and Simulink data types, see “Application Data Type Physical Constraint Mapping” on page 2-27.

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”.

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`
 	
`>> myInt16 = Simulink.AliasType;`
 	
`>> myInt16.BaseType = 'int16';`
 	
`>> myInt16.IsAlias = 'true'`
 	
Covers Boolean, integer, real
	`<IMPLEMENTATION-DATA-TYPE>`
	
	`<SHORT-NAME> myInt16</SHORT-NAME>`
	
	`<CATEGORY>VALUE </CATEGORY>`
	
	...
		Not generated

Primitive (fixed point), for example, `myFixPt`	`<IMPLEMENTATION-DATA-TYPE>`
	
	`<SHORT-NAME>`
		`<APPLICATION-PRIMITIVE-DATA-TYPE>`
		
		`<SHORT-NAME>`

Simulink Data Type	AUTOSAR XML	
	Implementation Type	Application Type

`>> myFixPt = Simulink.NumericType;`
 	
`>> myFixPt.DataTypeMode = ...`
 	
`>> myFixPt.IsAlias = 'true';`
	`myFixPt></SHORT-NAME>`
	
	`<CATEGORY>VALUE </CATEGORY>` ...
		`myFixPt` ...
		
		`<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:

```
ApplDT1
```

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

```
ImplDT1
```

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

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

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

Application Data Type Physical Constraint Mapping

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

- If you import ARXML files, `PhysConstr` values on `ApplicationDataType`s in the ARXML files are imported to `Min` and `Max` values on the corresponding Simulink data objects and root-level I/O signals.

- When you export ARXML from a model, the `Min` and `Max` values specified on Simulink data objects and root-level I/O signals are exported to the corresponding `ApplicationDataType` `PhysConstr`s in the ARXML files.

CompuMethod Categories for Data Types

AUTOSAR computation methods (`CompuMethods`) are used for the conversion of internal values into their physical representation, and vice versa. The `category` attribute of a `CompuMethod` represents a specialization of the `CompuMethod`, which may impose semantic constraints. The `CompuMethod` categories 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 `CompuMethod`s for every primitive application type, allowing measurement and calibration tools to monitor and interact with the application data. The following table shows the `CompuMethod` categories that Embedded Coder software generates for data types in a model that is configured for AUTOSAR.

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

Data Type	CompuMethod Category	CompuMethod on Application Type	CompuMethod on Implementation Type
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 `Identcl`.

For information about configuring `CompuMethod`s for code generation, see "Configure AUTOSAR CompuMethods" on page 4-93.

Model AUTOSAR 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.

In Simulink, you can model per-instance memory through the use of Data Store Memory and Data Store Read/Write blocks together with an `AUTOSAR.Signal` data object that specifies, for example, the `PerInstanceMemory` custom storage class.

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`, `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), which enables you to access and use NVRAM in your AUTOSAR application.

To model AUTOSAR-typed per-instance memory, open an `AUTOSAR.Signal` data object and set the **Storage class** parameter to `PerInstanceMemory (Custom)`. You can then configure the following attributes:

- **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; clear for C-typed per-instance memory.
- **SwCalibrationAccess** — Configure software calibration access to the data as `NotAccessible`, `ReadOnly`, or `ReadWrite`.
- **Initial value** — Configure an initial value for the data.

For detailed information about how you model per-instance memory, see the example rtwdemo_autosar_PIM_script or the example model `rtwdemo_autosar_PIM`. For an outline, see “Configure AUTOSAR Per-Instance Memory” on page 4-100.

Model AUTOSAR 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` with Static and Constant Memory. When exported in `arxml`, 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, you must 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 4-102 and “Configure AUTOSAR Data for Measurement and Calibration” on page 4-80.

AUTOSAR Component Creation

Import AUTOSAR Software Component

In this section...

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 3-9.
- 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 3-5.

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:

     ```
     obj.createComponentAsModel('/pkg/swc')
     ```

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

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

For a multi-runnable AUTOSAR software component, you can set the `CreateInternalBahavior` property on the model creation to automatically import the component internal behavior. For more information, see "Import Internal Behavior of AUTOSAR Software Component" on page 3-5.

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

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

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

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

  For example:

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

  For more information on updating a model with `arxml` changes, see "Merge AUTOSAR Authoring Tool Changes Into Model" on page 3-5.

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 4-2
- "Export AUTOSAR Component XML and C Code" on page 5-2
- "Configure and Map AUTOSAR Component Programmatically" on page 4-116

To see how to import, modify, and export an AUTOSAR Software Component, view the example Import and Export an AUTOSAR Software Component. (Go to Simulink Coder Examples in the help browser or enter the command `rtwdemos`.)

Import Internal Behavior of AUTOSAR Software Component

The AUTOSAR `arxml` importer tool can automatically import the internal behavior of a multi-runnable AUTOSAR software component into a Simulink model. You can use the `createComponentAsModel` method of the class `arxml.importer` to specify that internal behavior be imported. For example:

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

The importer:

- Adds subsystem blocks in the model and maps them to corresponding runnables imported from the AUTOSAR software component.
- Adds signal lines in the model and maps them to corresponding inter-runnable variables imported from the AUTOSAR software component.

Merge AUTOSAR Authoring Tool Changes Into Model

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

Given a model into which you have imported `arxml` code or from which you have exported `arxml` code, suppose that changes have been 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')
```

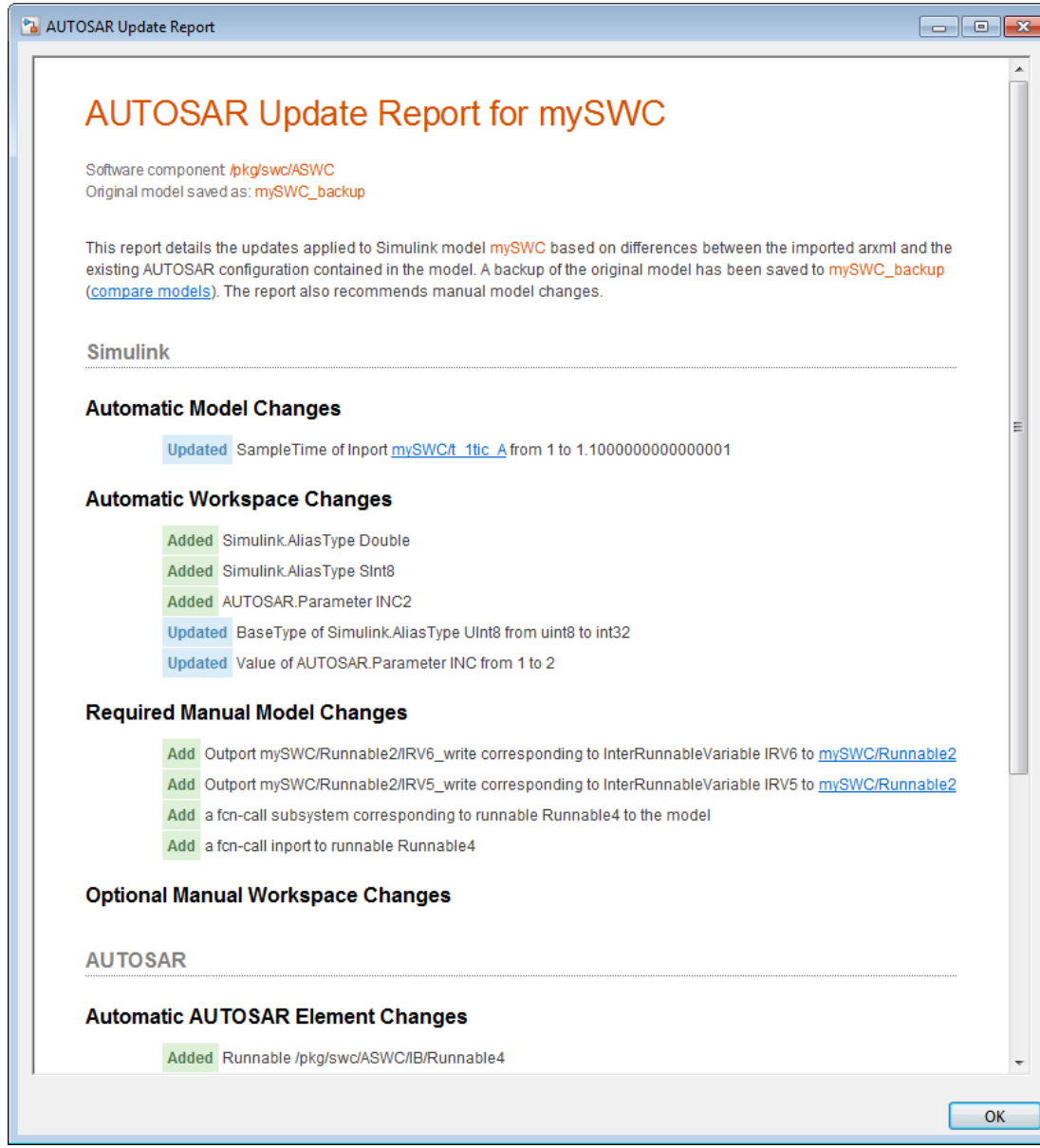
2. Issue MATLAB commands to import `arxml` into the model and update the model with changes.

Note: The imported `arxml` 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');
>> obj.updateModel('mySWC');
### Updating model mySWC
### Saving original model as mySWC_backup
### Creating HTML report mySWC_update_report.html
>>
```

3. In the Command Window, the report name that follows the output text `Creating HTML report` is a hyperlink. Click the report name to open the AUTOSAR Update Report.


AUTOSAR Update Report
AUTOSAR Update Report for mySWC
Software component /pkg/swc/ASWC
Original model saved as: mySWC_backup
This report details the updates applied to Simulink model mySWC based on differences between the imported arxml and the existing AUTOSAR configuration contained in the model. A backup of the original model has been saved to mySWC_backup (compare models). The report also recommends manual model changes.
Simulink
Automatic Model Changes
Updated SampleTime of Inport mySWC/t_1tic_A from 1 to 1.1000000000000001
Automatic Workspace Changes
Added Simulink.AliasType Double
Added Simulink.AliasType SInt8
Added AUTOSAR.Parameter INC2
Updated BaseType of Simulink.AliasType UInt8 from uint8 to int32
Updated Value of AUTOSAR.Parameter INC from 1 to 2
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
OK

4 Examine the report. For example, you can verify the reported changes and perform the required manual changes.

Round-Trip Preservation of AUTOSAR Elements and UUIDs

To help 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, as follows:

- 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` with their UUIDs and other information preserved.

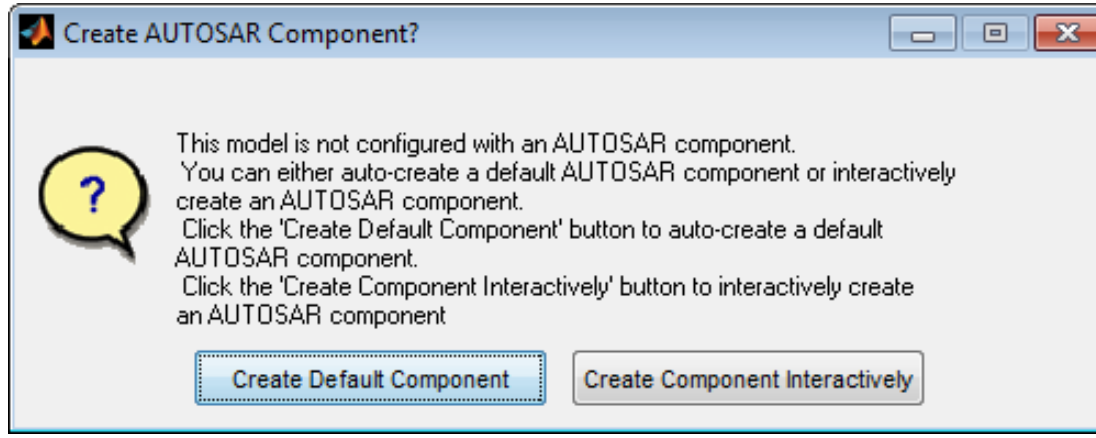
As a result, the `arxml` files exported from Simulink can more easily be merged back into the AAT environment. Existing elements retain their UUIDs, while new elements created in Simulink get new UUIDs.

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:

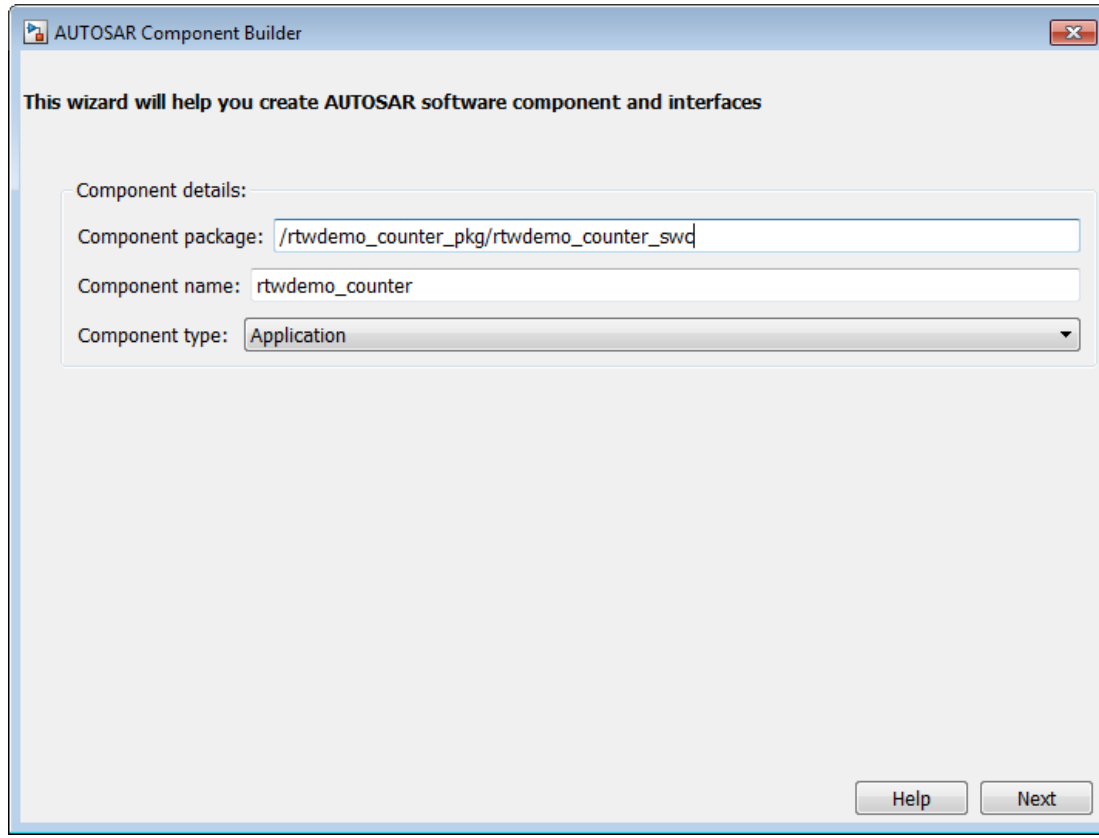
   - **Create Default Component** — Automatically create an AUTOSAR component with default settings and open it in the Configure AUTOSAR Interface dialog box.
   - **Create Component Interactively** — Interactively create an AUTOSAR component using the AUTOSAR Component Builder dialog box.



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 4-2. 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:

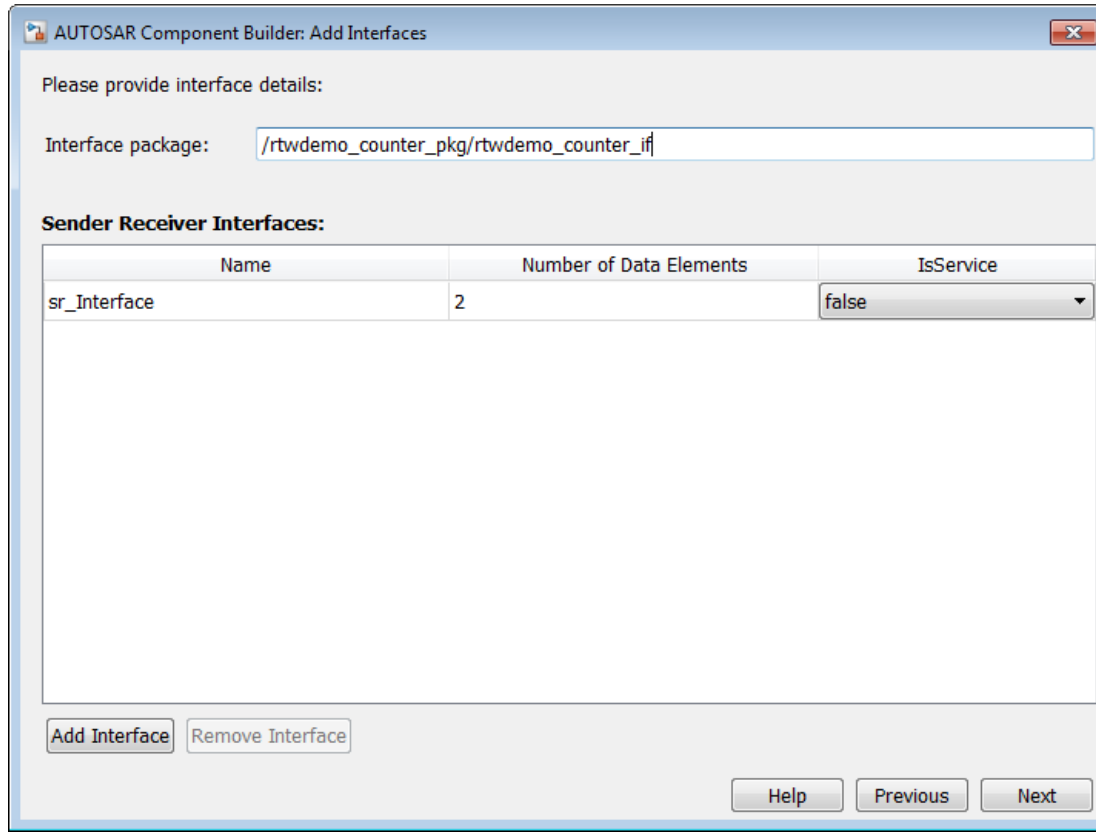
- Path for the AUTOSAR component package.
- Name for the AUTOSAR component.
- AUTOSAR component type: `Application` for an AUTOSAR application software component, or `Sensor Accuator` for an AUTOSAR sensor/actuator software component.



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:

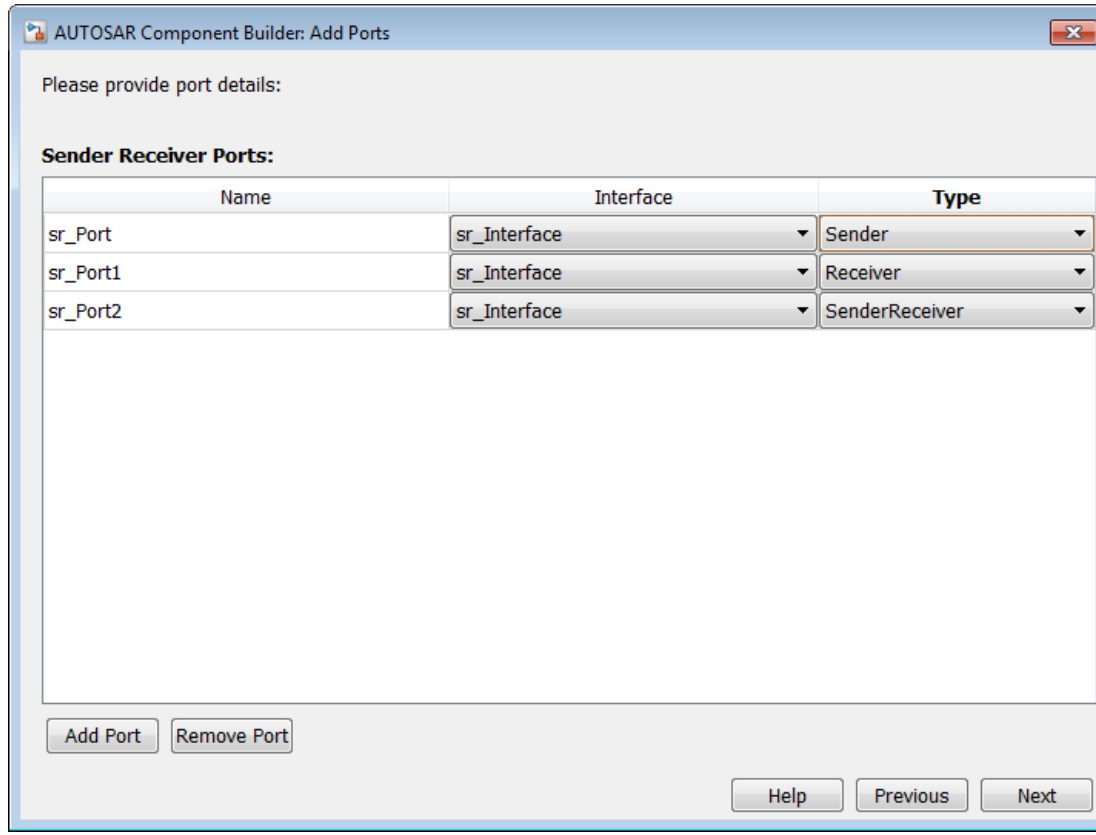
- Modify the name of the Interface package.
- Click **Add Interface** to add more interfaces to the displayed list.
- Click **Remove Interface** to remove a selected interface.
- For each listed interface, edit the name and the number of data elements it contains, and select whether the interface is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

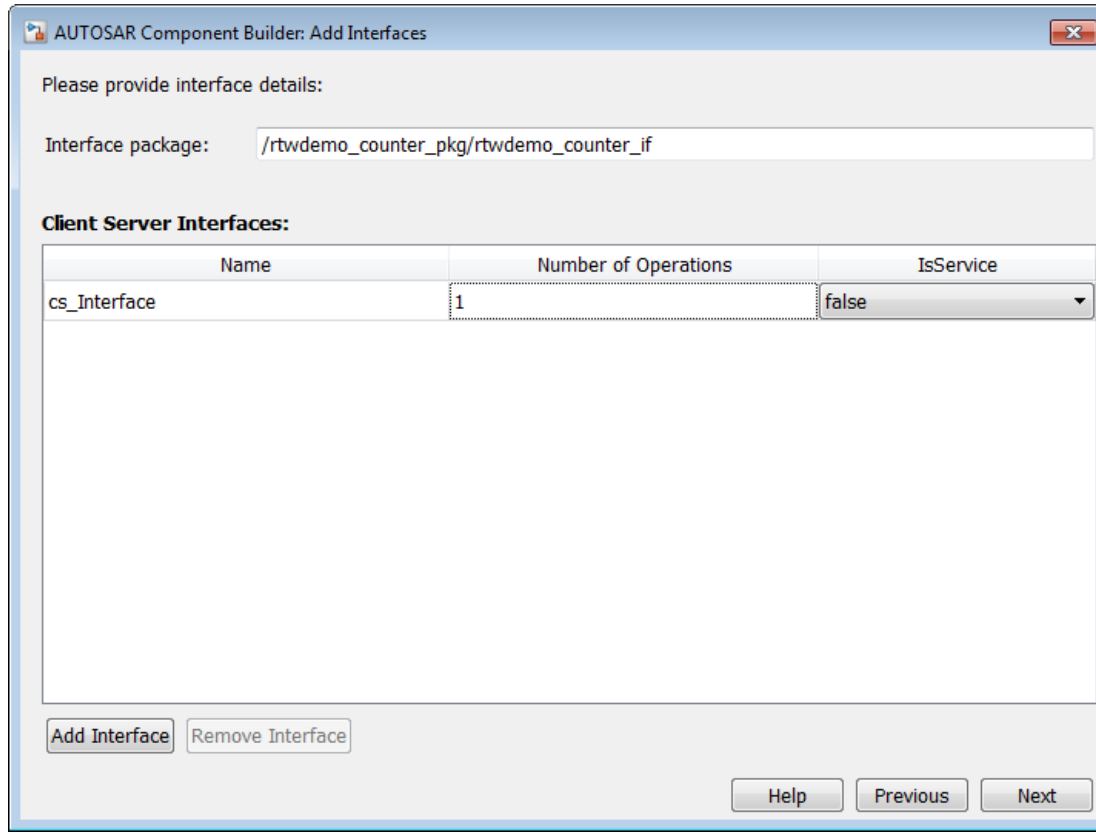


Click **Next** to go to the Add Client Server Interfaces view.

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.

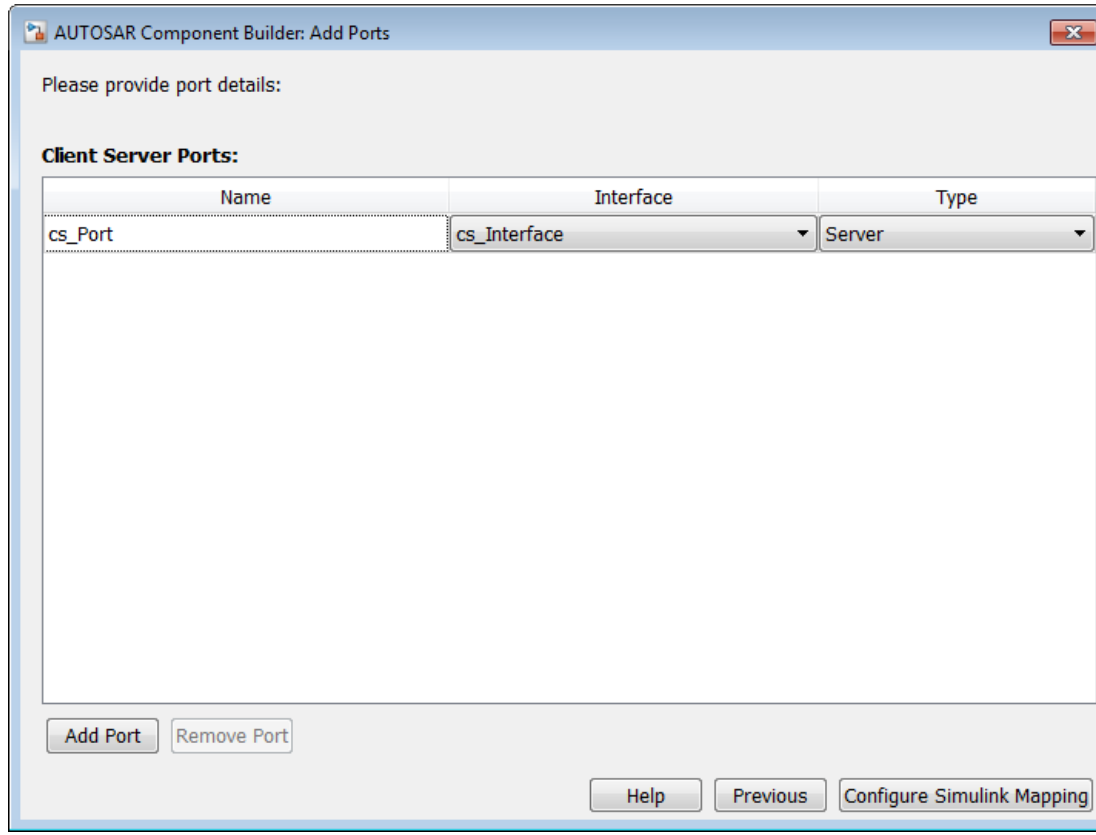


Click **Next** to go to the Add Client Server Ports view.

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 4-2.

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® objects that compose the `arxml.importer` object.

AUTOSAR Component Development

Configure the AUTOSAR Interface

In this section...

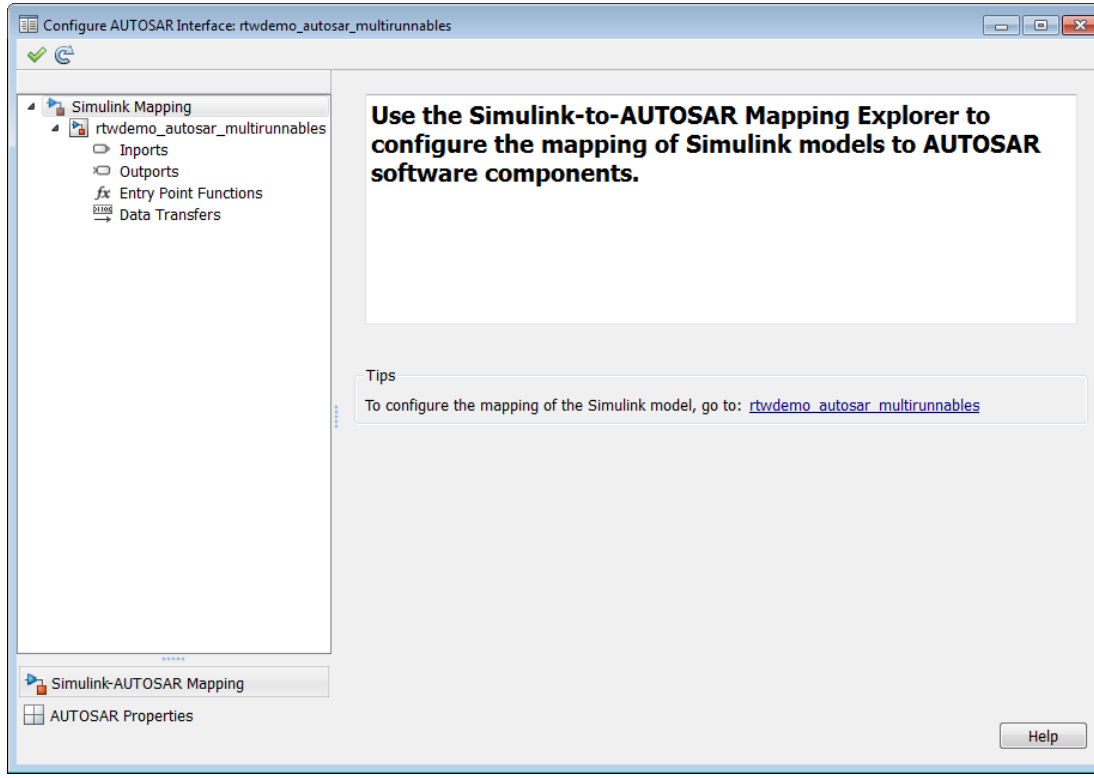
Overview of AUTOSAR Interface Configuration

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, 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 4-116.

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, or both 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 4-4. For more information on the AUTOSAR Properties Explorer, see "Configure AUTOSAR Component Using AUTOSAR Properties Explorer" on page 4-9.

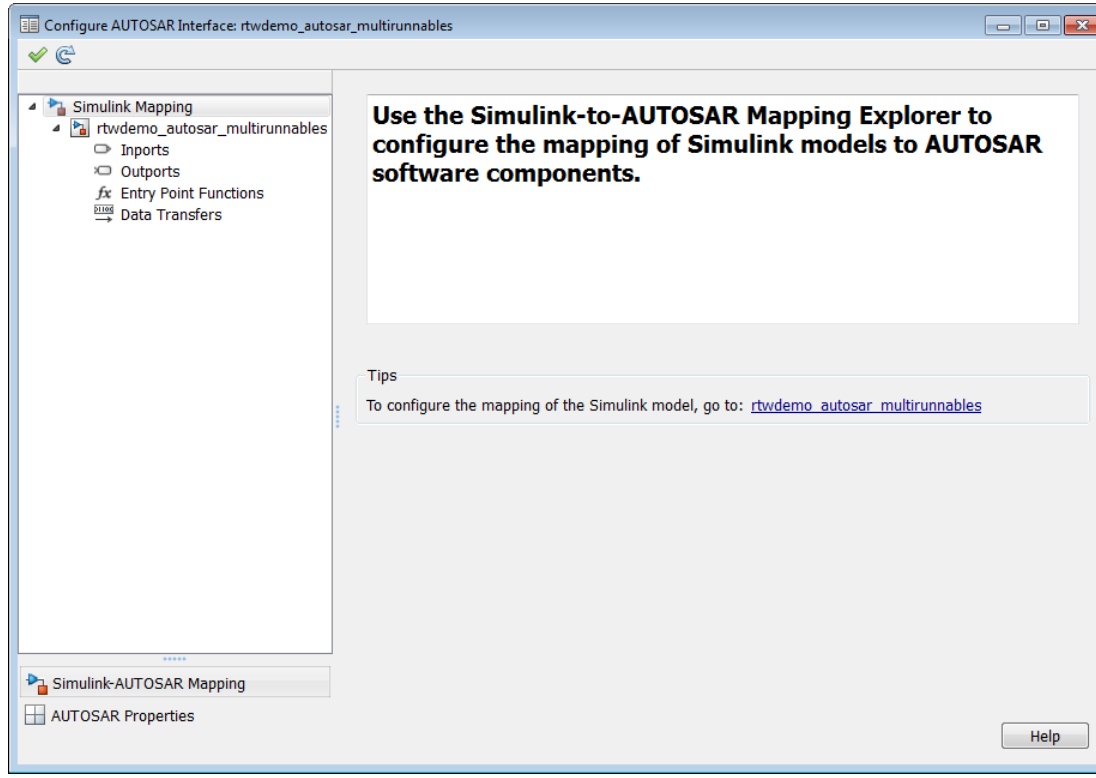
Map Model Elements Using Simulink-AUTOSAR Mapping Explorer

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:

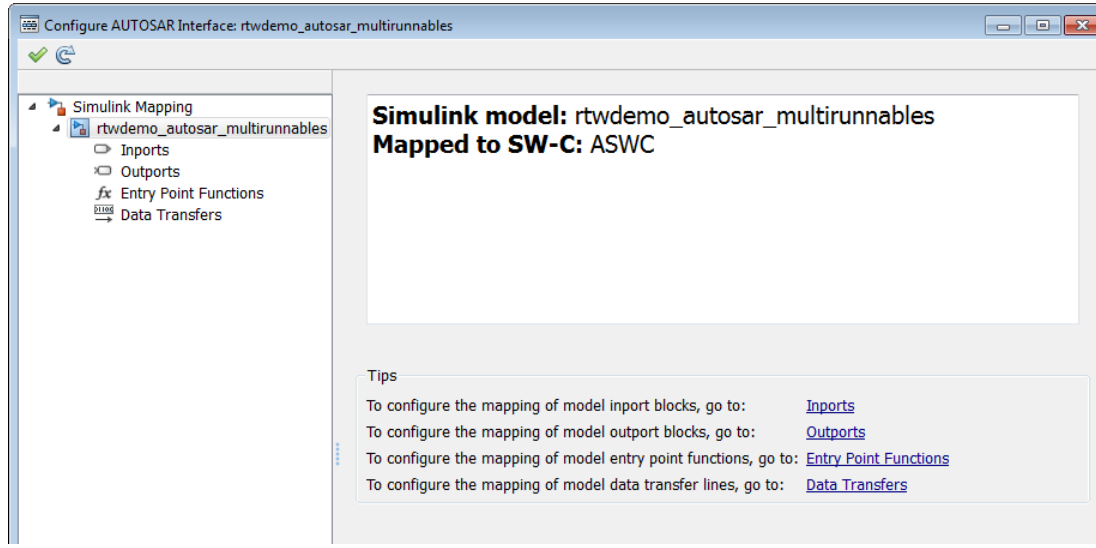
   - A Simulink model can be mapped to an AUTOSAR component.
   - A Simulink inport or outport can be mapped to a data element of an AUTOSAR port, with a specific data access mode.
   - A Simulink entry-point function can be mapped to an AUTOSAR runnable.
   - A Simulink data transfer line can be mapped 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.

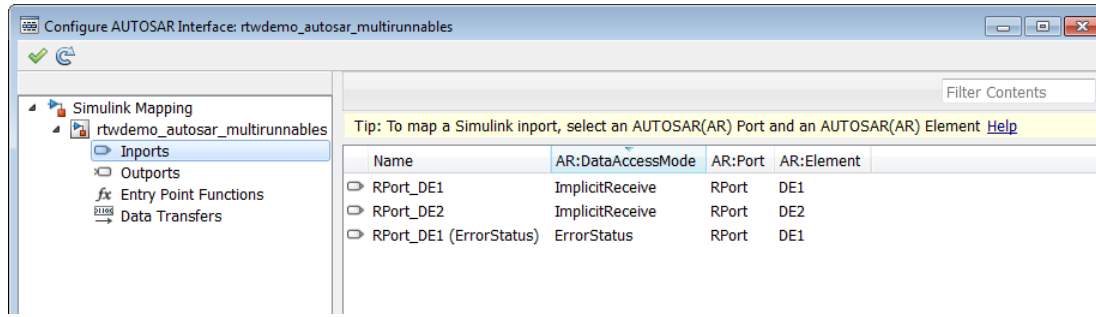
The model view of the Simulink-AUTOSAR Mapping Explorer displays the model name and the name of the AUTOSAR component to which the model is being mapped.



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:

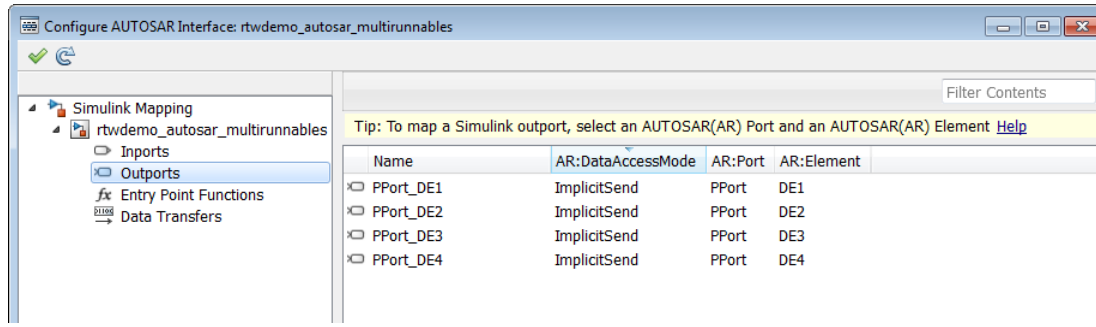
- Map a Simulink inport by selecting the inport and then selecting menu values for an AUTOSAR Port and an AUTOSAR Element.
- Select a port and then select a menu value for its AUTOSAR port data access mode: `ImplicitReceive`, `ExplicitReceive`, `QueuedExplicitReceive`, `ErrorStatus`, or `ModeReceive`.



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

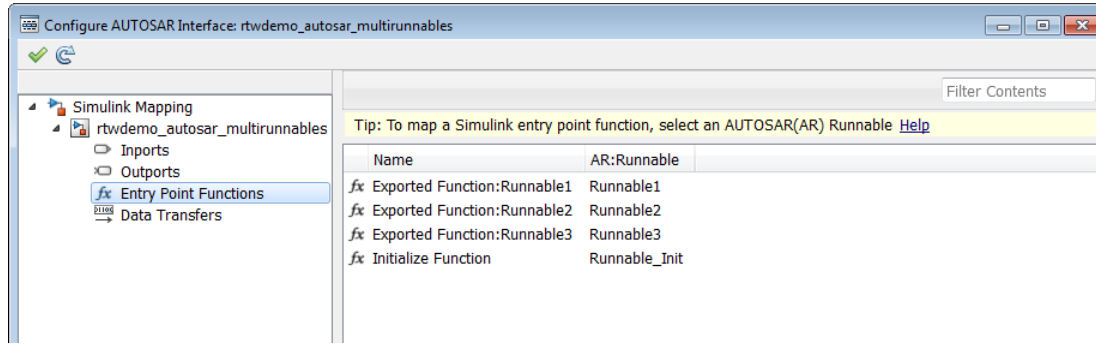
The Outports view of the Simulink-AUTOSAR Mapping Explorer maps each Simulink outport to a data element of an AUTOSAR port. You can:

- 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` or `ExplicitSend`.



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.



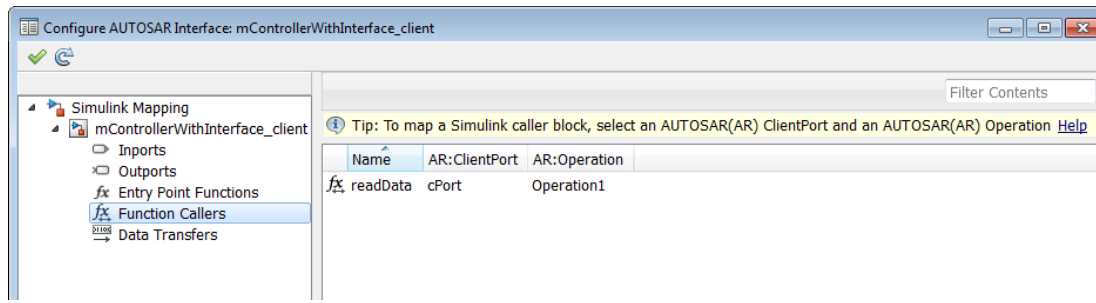
7 In the left-hand pane of the Configure AUTOSAR Interface dialog box, select **Function Callers**.

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.

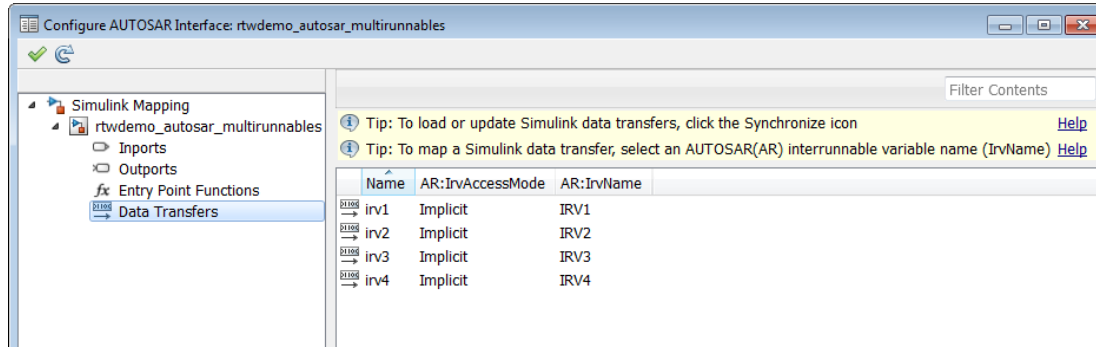
You can map a Simulink function caller by selecting the function name and then selecting menu values for a client port and an operation, among those listed for the AUTOSAR component.



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.



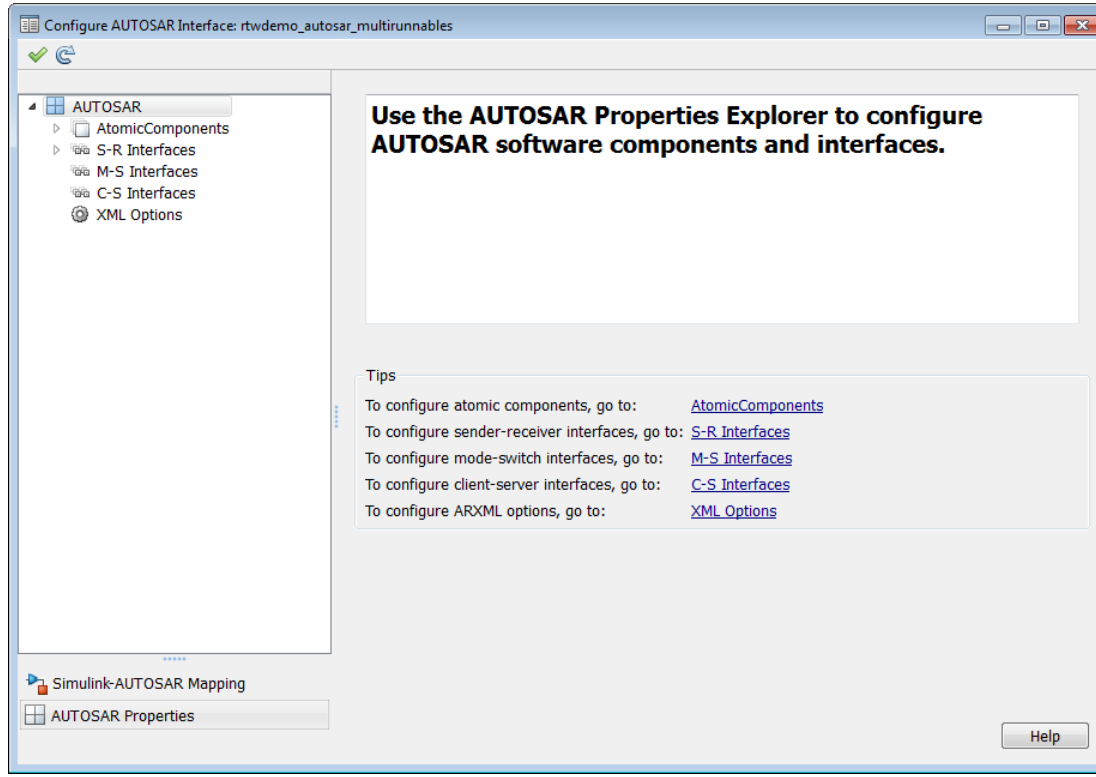
9 Click the Validate icon to validate the AUTOSAR interface configuration. If errors are reported, address them and then retry validation.

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.

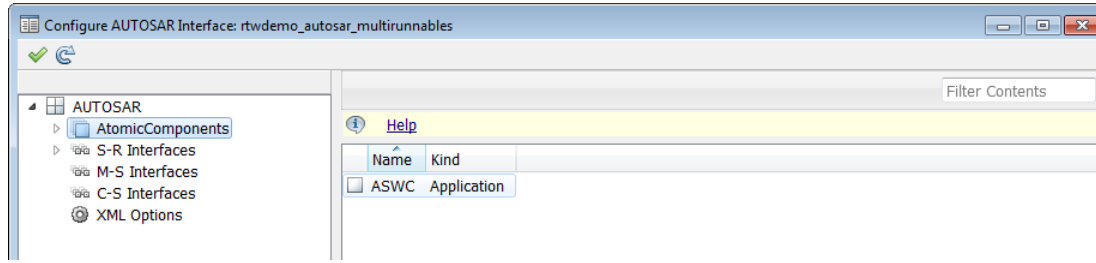
2 Open the Configure AUTOSAR Interface dialog box by selecting **Code > C/C++ Code > Configure Model as AUTOSAR Component**. If the AUTOSAR Properties Explorer is not already selected, click **AUTOSAR Properties**. The top-level view shows the types of AUTOSAR elements for which properties can be configured — atomic software components, S-R interfaces, M-S interfaces, C-S interfaces, and XML options.



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

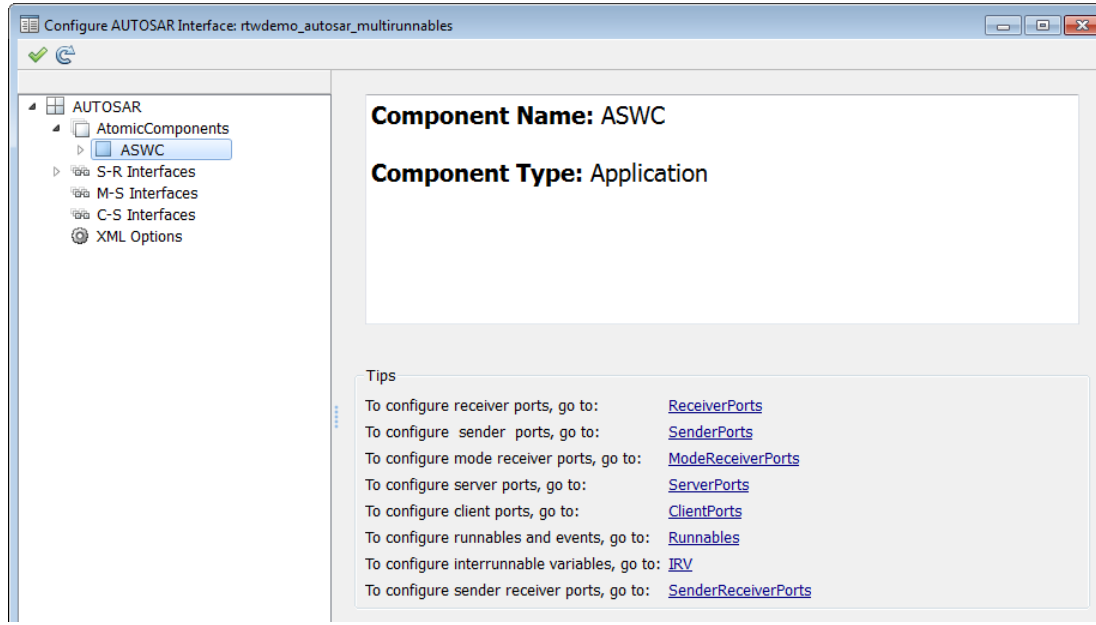
The Atomic Components view of the AUTOSAR Properties Explorer displays atomic components and their types. You can:

- 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.

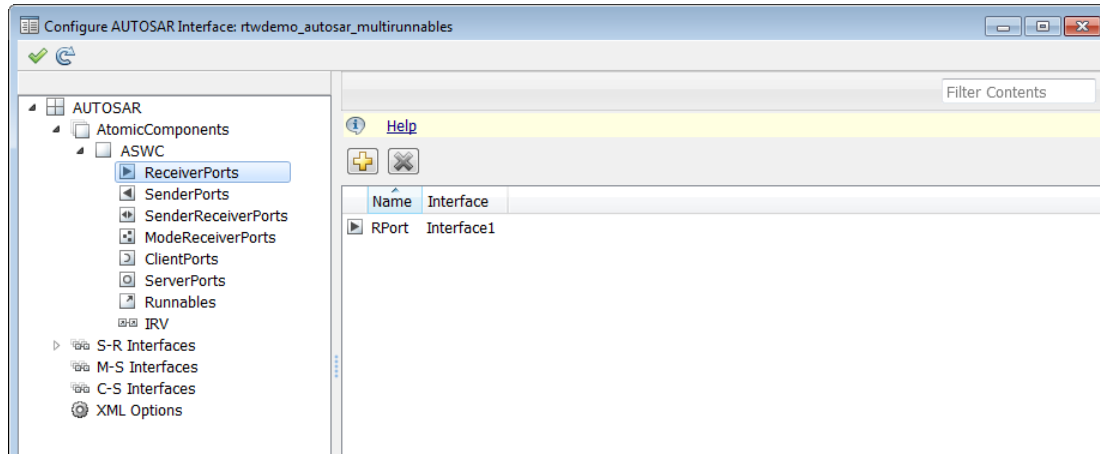
The component view of the AUTOSAR Properties Explorer displays the name and type of the selected component, and lists the types of AUTOSAR component elements for which properties can be configured — receiver ports, sender ports, sender-receiver ports, mode receiver ports, server ports, client ports,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



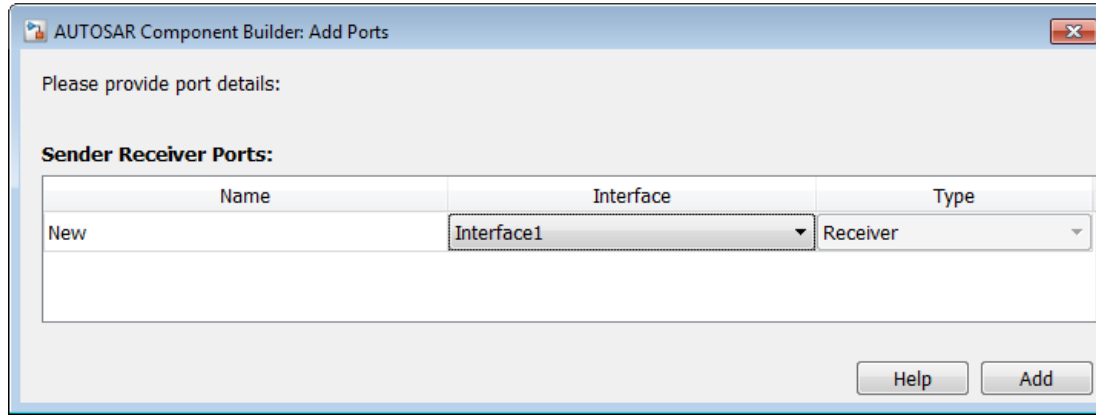
5 In the left-hand pane of the Configure AUTOSAR Interface dialog box, expand the component and select **ReceiverPorts**.

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.



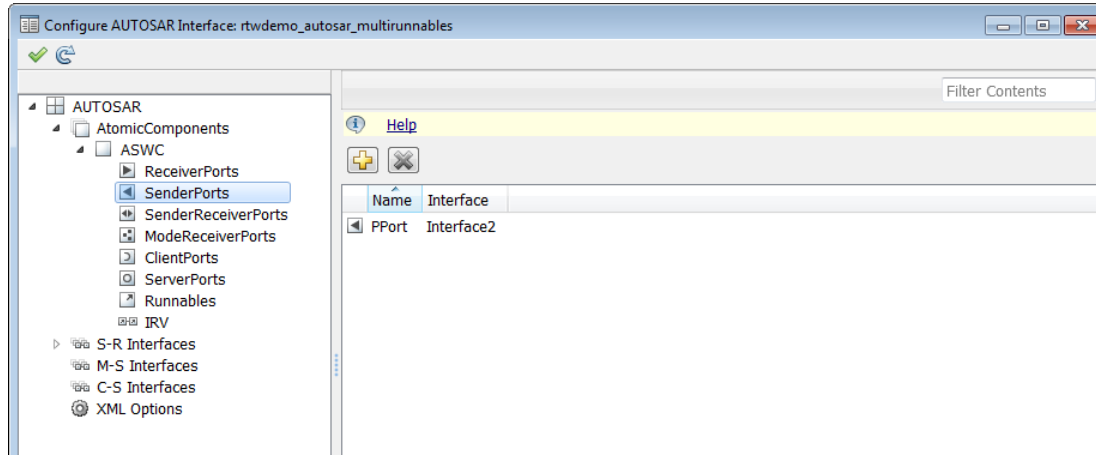
The Add Ports dialog box lets you add a receiver port and associate it with an existing S-R interface. Click **Add** to add the port and return to the Receiver Ports view.



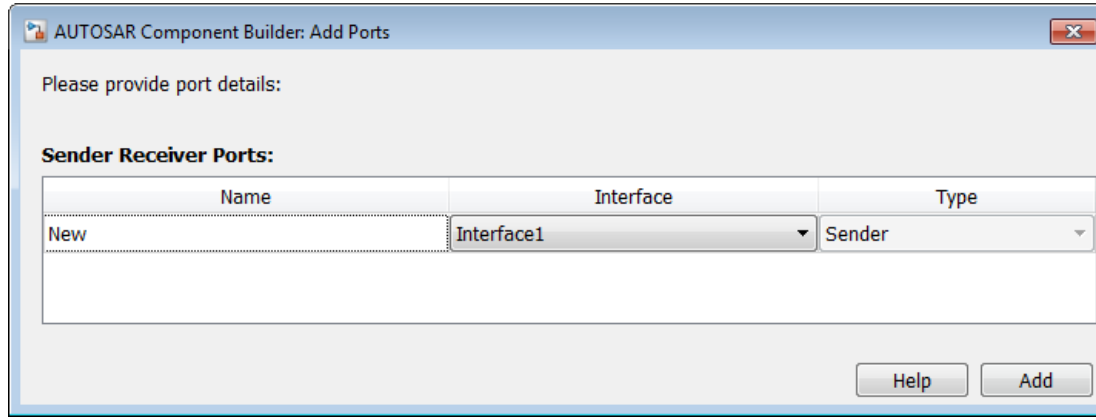
6 In the left-hand pane of the Configure AUTOSAR Interface dialog box, select **SenderPorts**.

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 AUTOSAR Component Builder 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.



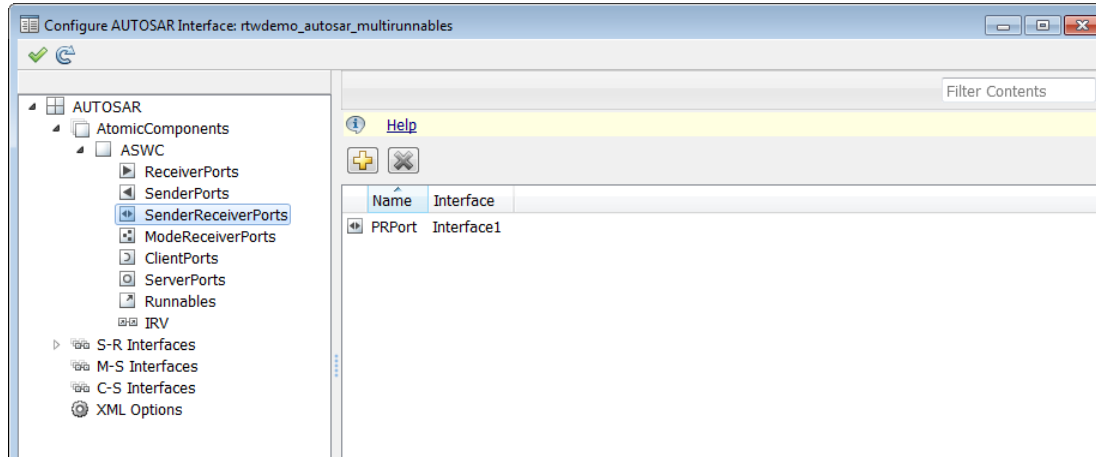
7 In the left-hand pane of the Configure AUTOSAR Interface dialog box, select **SenderReceiverPorts**.

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

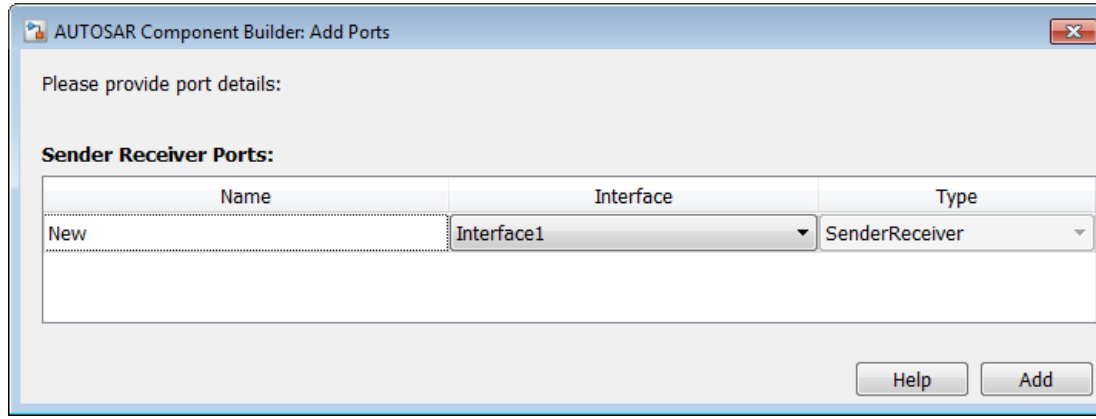
- Select an AUTOSAR sender-receiver port, and view and optionally reselect its associated S-R interface.

- Rename an AUTOSAR sender-receiver port by clicking its name and then editing the name string.
- Click the Add icon to open an AUTOSAR Component Builder 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.



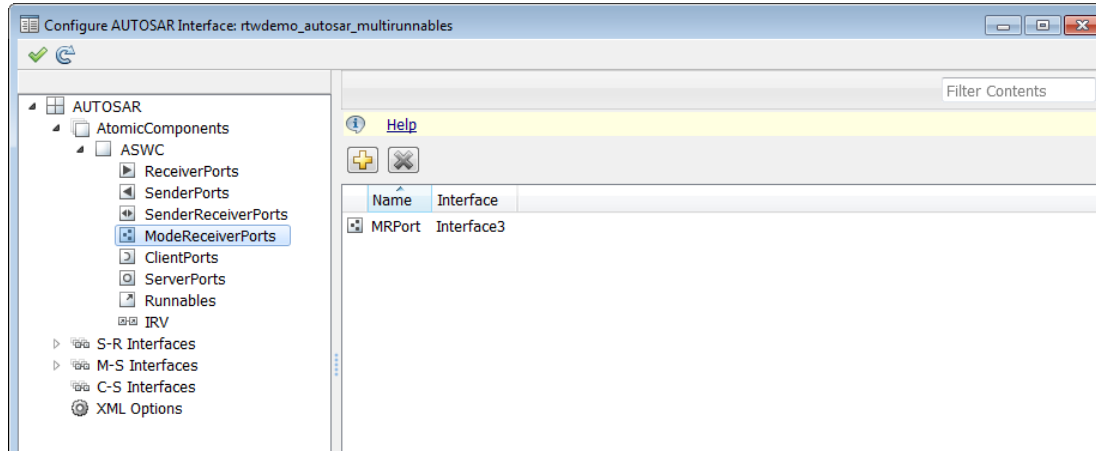
The Add Ports dialog box lets you add a sender-receiver port and associate it with an existing S-R interface. Click **Add** to add the port and return to the Sender-Receiver Ports view.



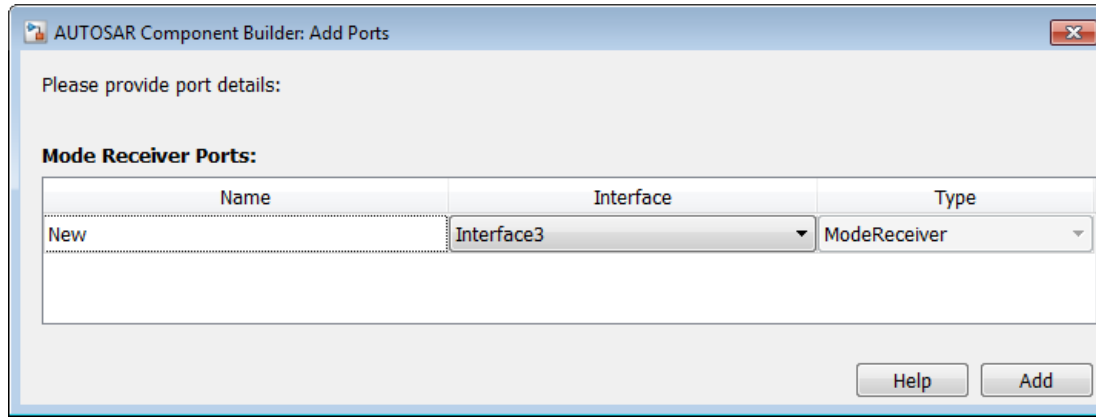
8 In the left-hand pane of the Configure AUTOSAR Interface dialog box, select **ModeReceiverPorts**.

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.



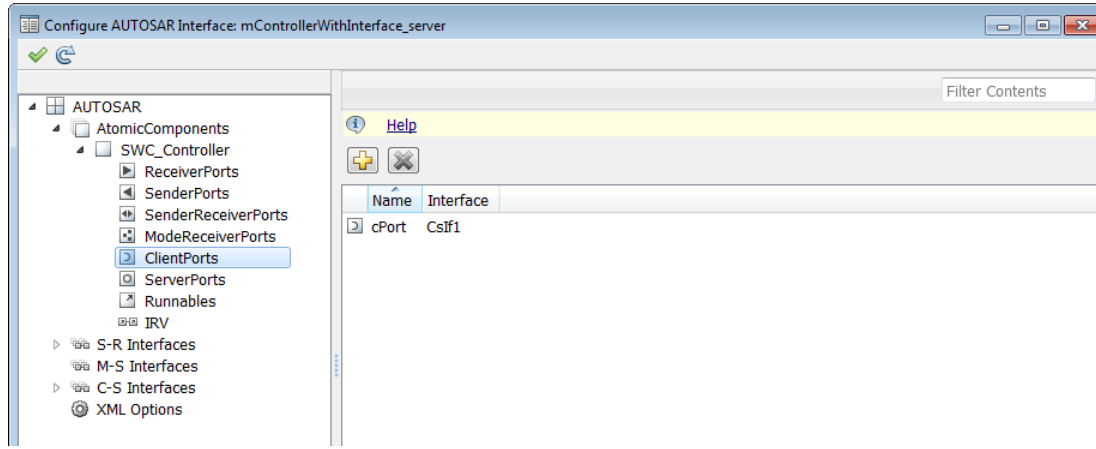
For more information about adding and configuring mode-receiver ports, see “Configure AUTOSAR Mode Receiver Ports and Mode-Switch Events” on page 4-52.

9

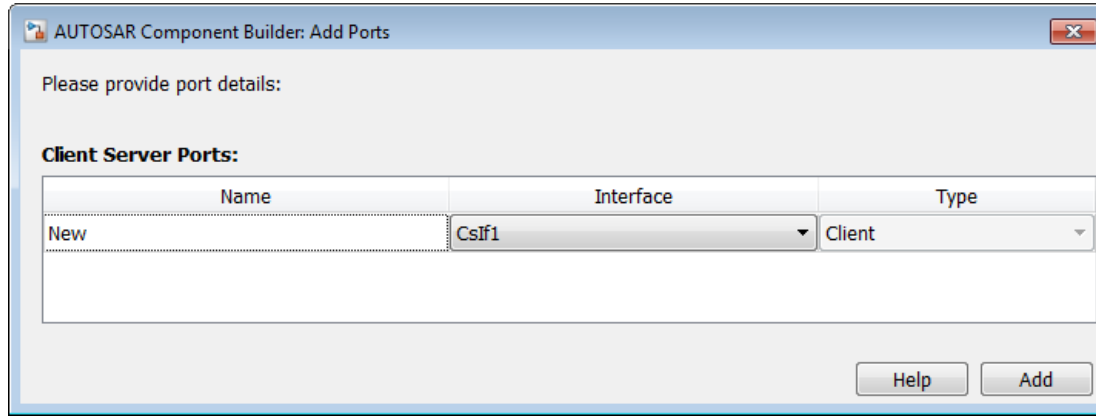
In the left-hand pane of the Configure AUTOSAR Interface dialog box, select **ClientPorts**.

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.



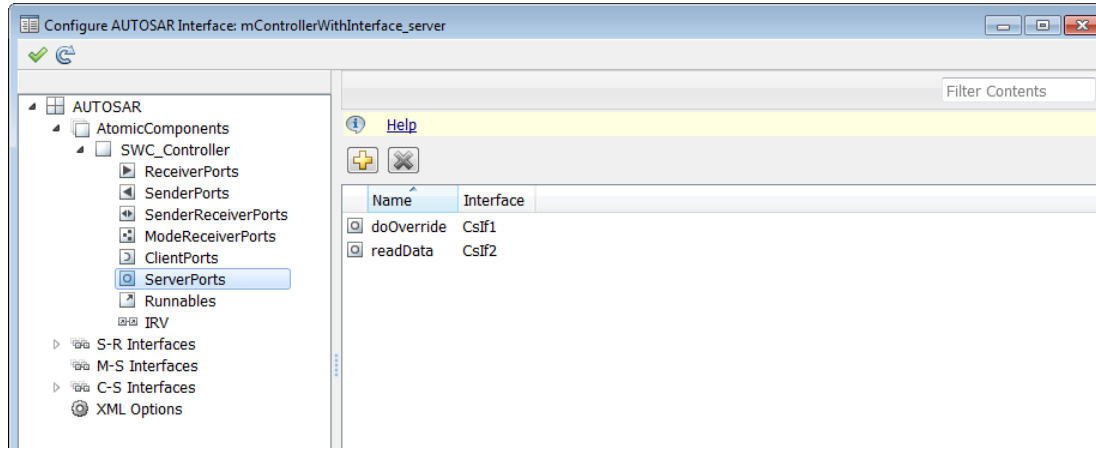
The Add Ports dialog box lets you add a client port and associate it with an existing C-S interface. If a C-S interface does not exist in the component, you must create one before adding the port. Click **Add** to add the port and return to the Client Ports view.



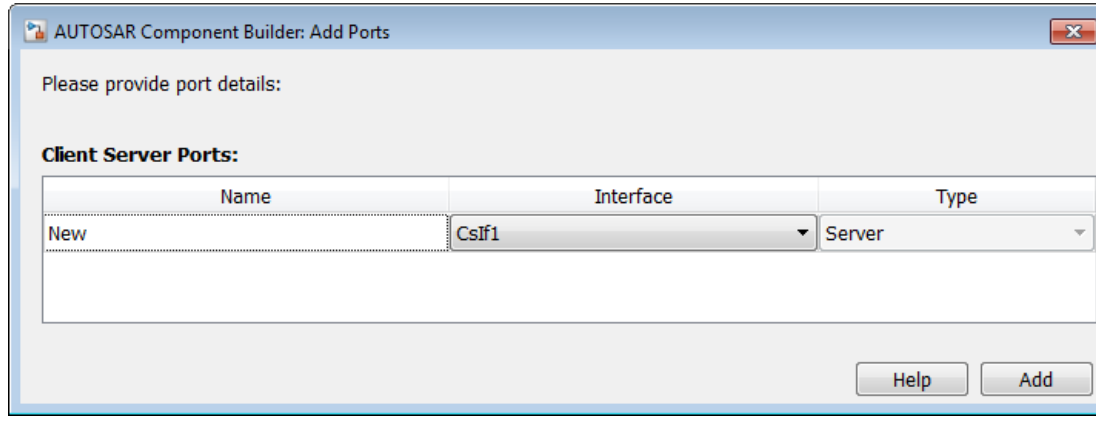
10 In the left-hand pane of the Configure AUTOSAR Interface dialog box, select **ServerPorts**.

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.
- 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. 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.



11 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

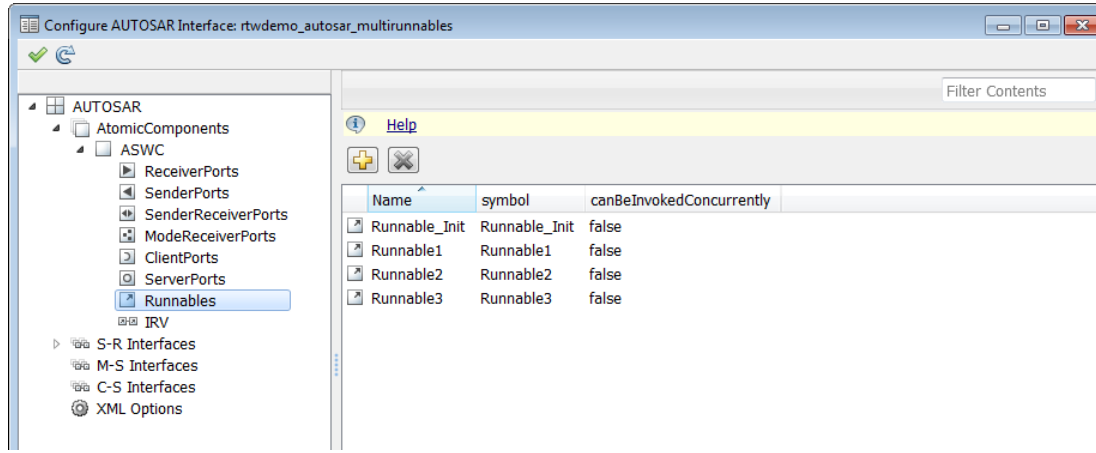
```
<RUNNABLE-ENTITY UUID="65432c3e-34c7-5e82-4229-f6d04927eb78">
    <SHORT-NAME>Runnable1</SHORT-NAME>
...
    <SYMBOL>test_symbol</SYMBOL>
...
</RUNNABLE-ENTITY>
```

  rtwdemo_autosar_multirunnables.c

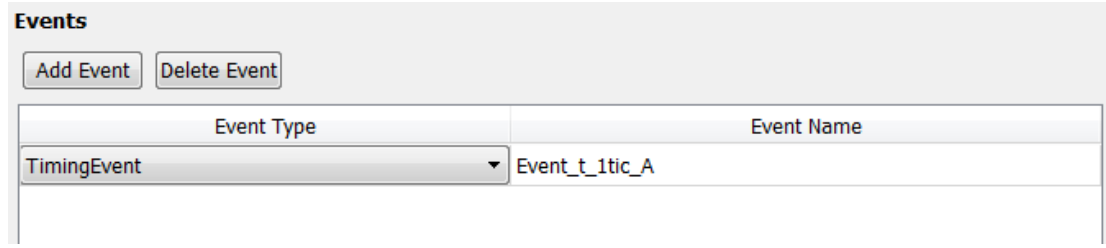
```
/* Output function for RootInportFunctionCallGenerator:
   '<Root>/RootFcnCall_InsertedFor_Runnable1_at_outport_1' */
void test_symbol(void)
{
...
}
```

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

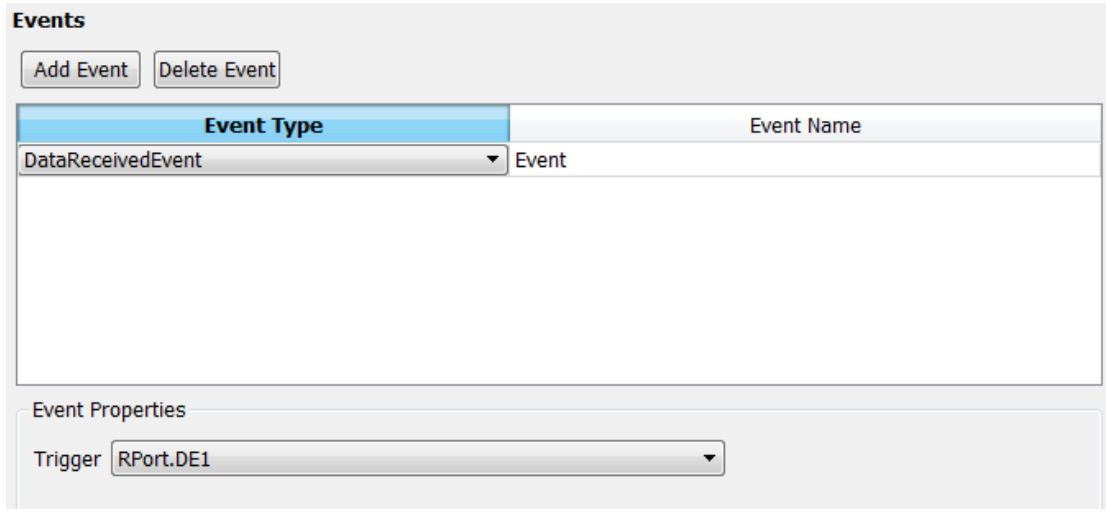
- For an AUTOSAR server runnable, set the runnable property `canBeInvokedConcurrently` to designate whether to enforce concurrency constraints. For nonserver runnables, leave `canBeInvokedConcurrently` set to `false`. For more information, see “Concurrency Constraints for AUTOSAR Server Runnables”.
-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



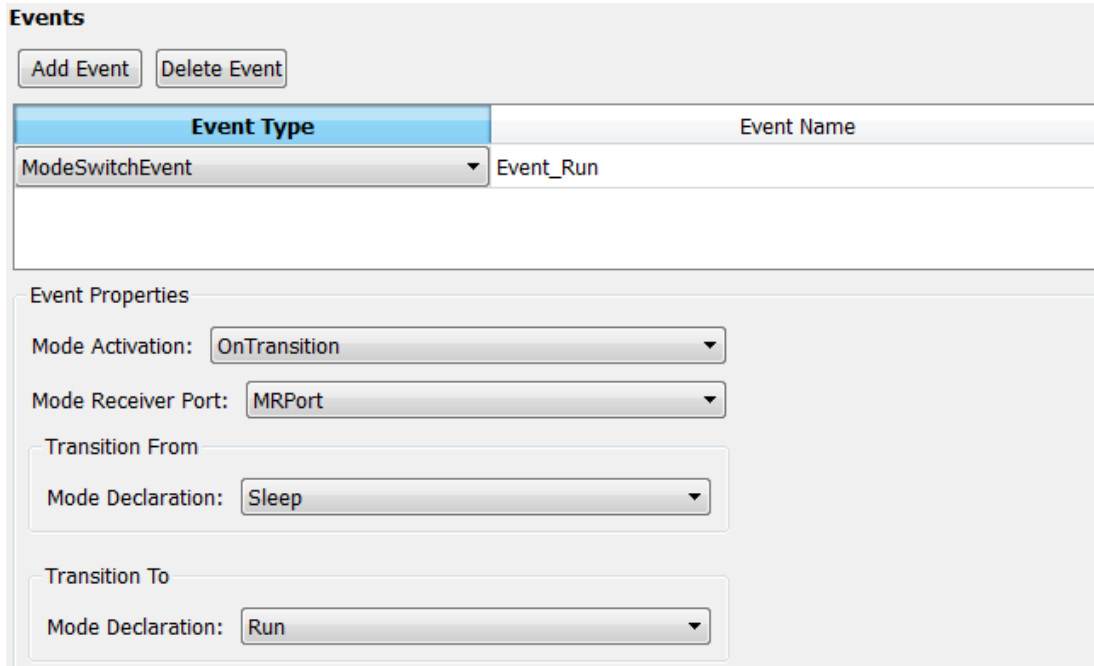
Select a runnable to see its list of associated events. The **Events** pane lists each AUTOSAR event with its type — `TimingEvent`, `DataReceivedEvent`, `ModeSwitchEvent`, `OperationInvokedEvent`, or `InitEvent` — and name. You can rename an AUTOSAR event by clicking its name and then editing the name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



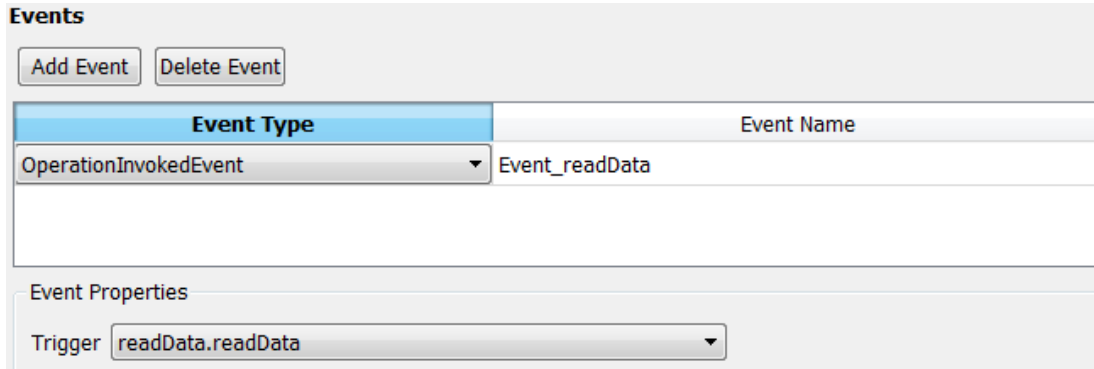
If you select an event of type `DataReceivedEvent`, the **Trigger** property is displayed. Select a trigger for the event from the list of available receiver ports.



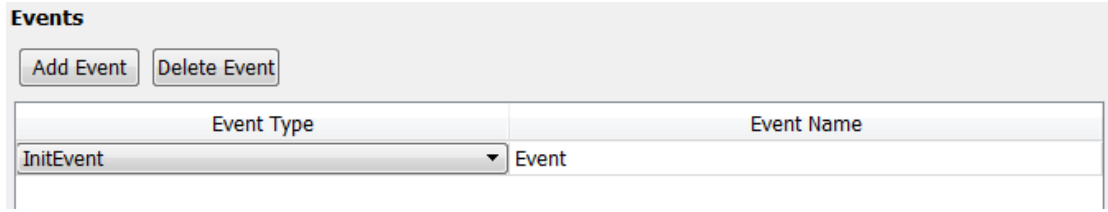
If you select an event of type `ModeSwitchEvent`, the **Mode Activation** and **Mode Receiver Port** properties are displayed. Select a mode receiver port for the event from the list of configured mode-receiver ports. Select a mode activation value for the event from the list of values (`OnEntry`, `OnExit`, 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 4-52.)



If you select an event of type `OperationInvokedEvent`, the runnable becomes an AUTOSAR server runnable. The **Trigger** property is displayed. Select a trigger for the event from the list of available trigger ports. (For more information on using an `OperationInvokedEvent`, see "Configure AUTOSAR Client-Server Communication" on page 4-59.)



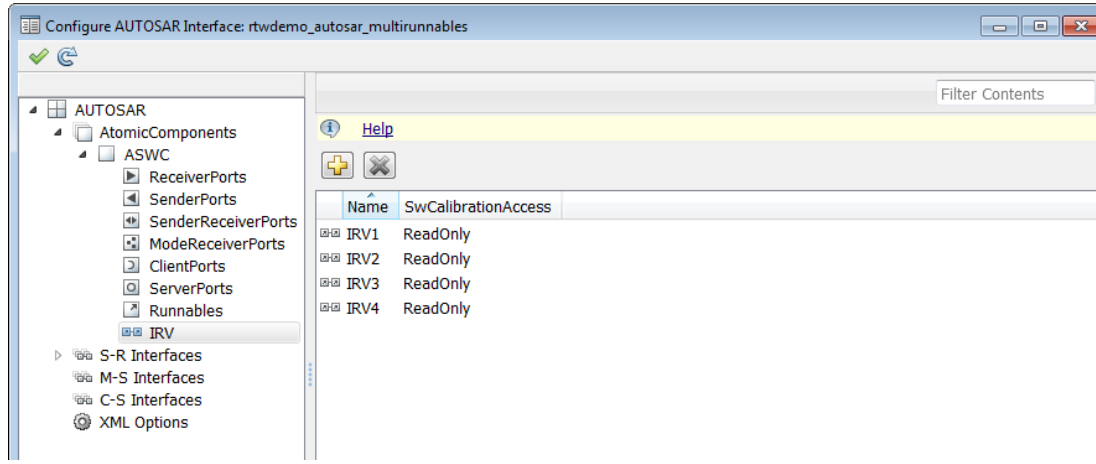
If you select an event of type `InitEvent`, you can rename the event by clicking its name and then editing the name string. (For more information on using an `InitEvent`, see "Configure AUTOSAR Initialization Runnable" on page 4-45.)



12 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:

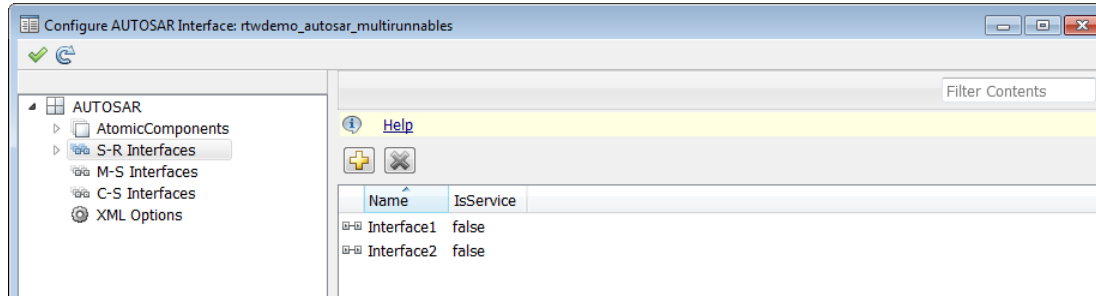
- Rename an AUTOSAR IRV by clicking its name and then editing the name string.
- Specify the level of measurement and calibration tool access to an IRV, by setting its **SwCalibrationAccess** value to `ReadOnly`, `ReadWrite`, or `NotAccessible`.
- Click the Add icon to add an AUTOSAR IRV.
- Select an AUTOSAR IRV and then click the Delete icon to remove it.



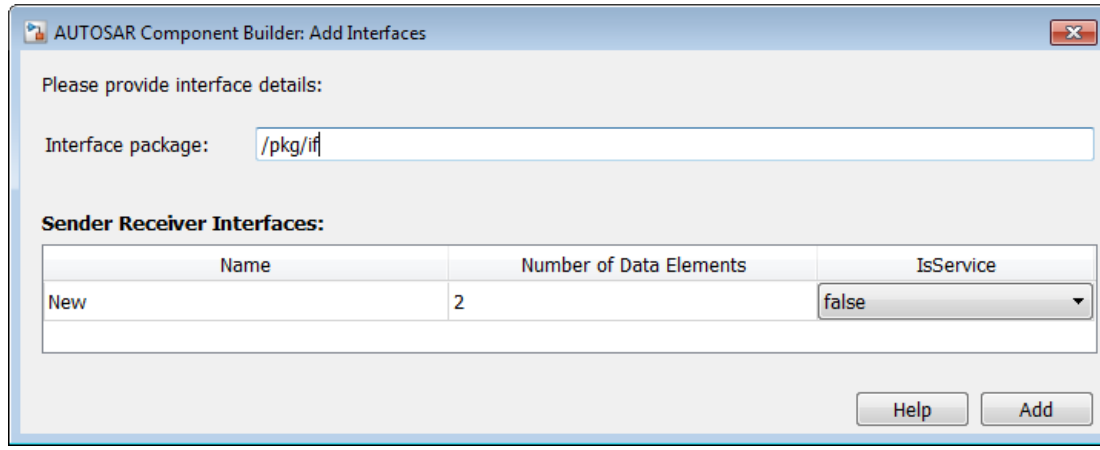
13 In the left-hand pane of the Configure AUTOSAR Interface dialog box, select **S-R Interfaces**.

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 AUTOSAR Component Builder 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.

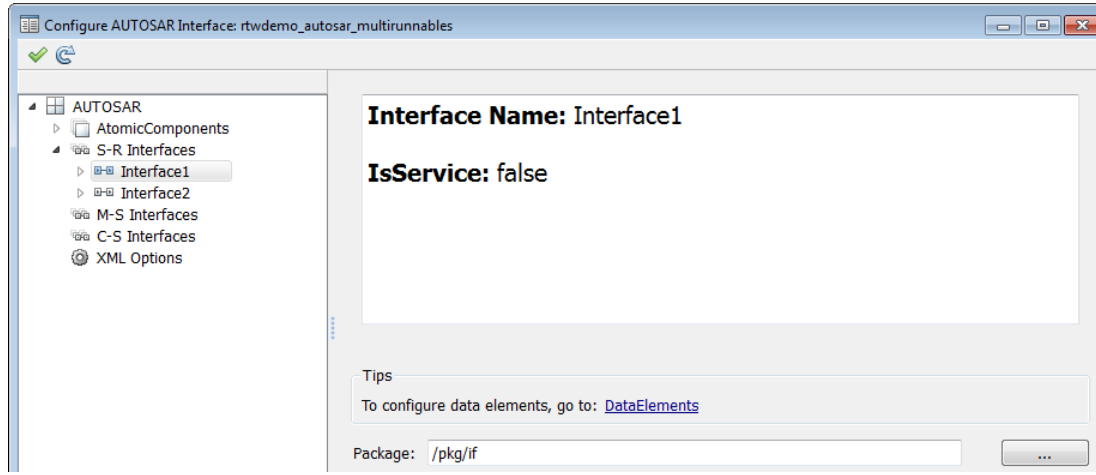


14 In the left-hand pane of the Configure AUTOSAR Interface dialog box, expand **S-R Interfaces** and select an S-R interface from the list.

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 for the interface.

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

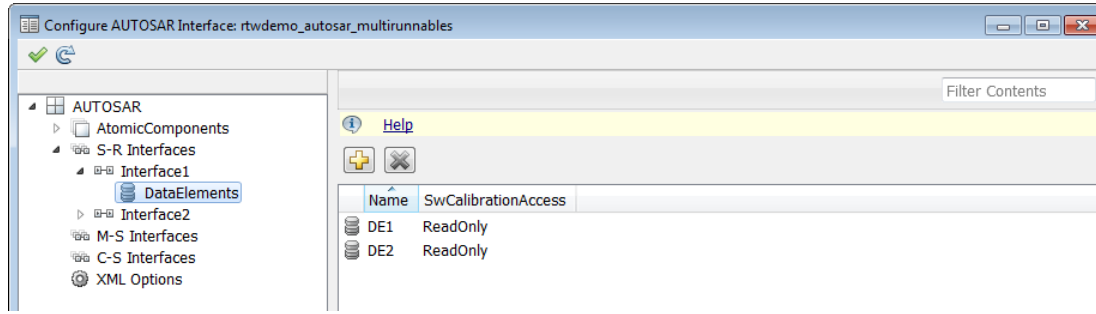
- Enter a package path in the **Package** parameter field.
- Click the button to the right of the **Package** field to open the AUTOSAR Package Browser. Use the browser to navigate to an existing package or create a new package. When you select a package in the browser and click **Apply**, the interface **Package** parameter value is updated with your selection. For more information about the AUTOSAR Package Browser, see “Configure AUTOSAR Package for Interface” on page 4-40.



15 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

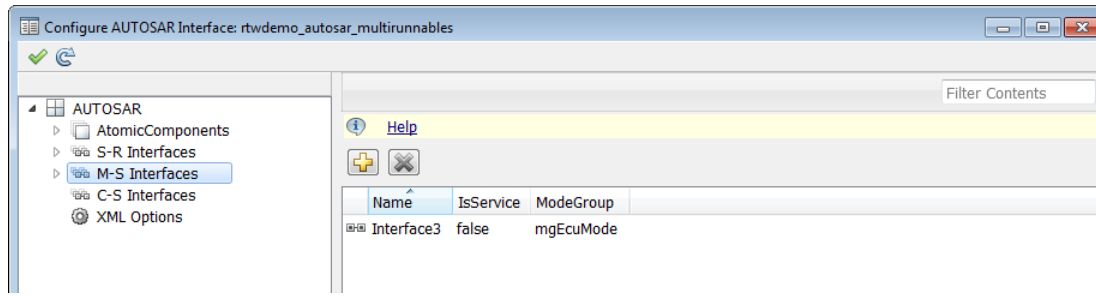
- Select an S-R interface data element and edit the name value.
- Specify the level of measurement and calibration tool access to an S-R interface data element, by setting its **SwCalibrationAccess** value to `ReadOnly`, `ReadWrite`, or `NotAccessible`.
- Click the Add icon to add a data element.
- Select a data element and then click the Delete icon to remove it.



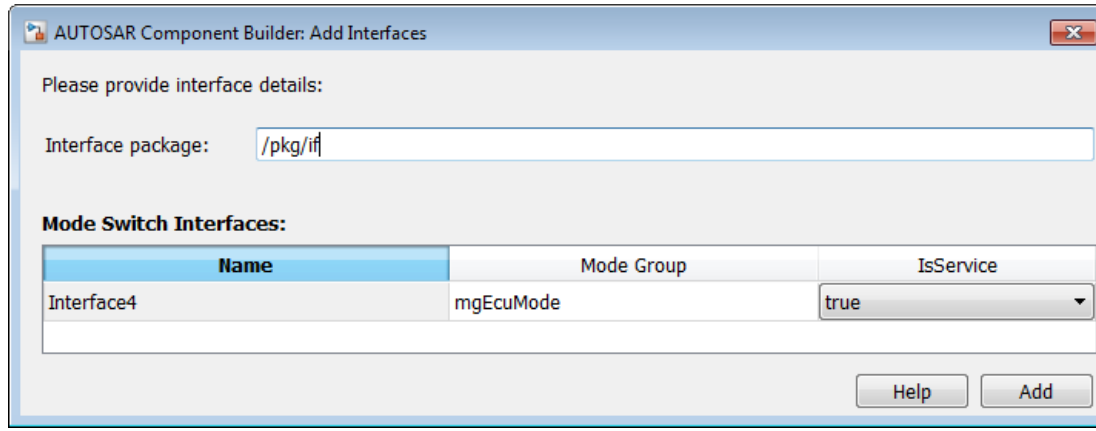
16 In the left-hand pane of the Configure AUTOSAR Interface dialog box, select **M-S Interfaces**.

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, select a menu value to specify whether or not it is a service, and modify the name of its associated mode group. 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 4-52.
- 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.

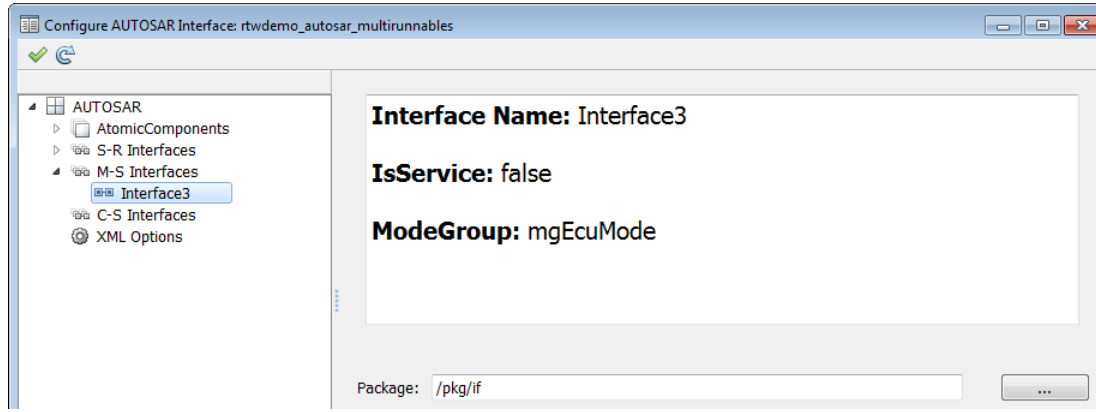


17 In the left-hand pane of the Configure AUTOSAR Interface dialog box, expand **M-S Interfaces** and select an M-S interface from the list.

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:

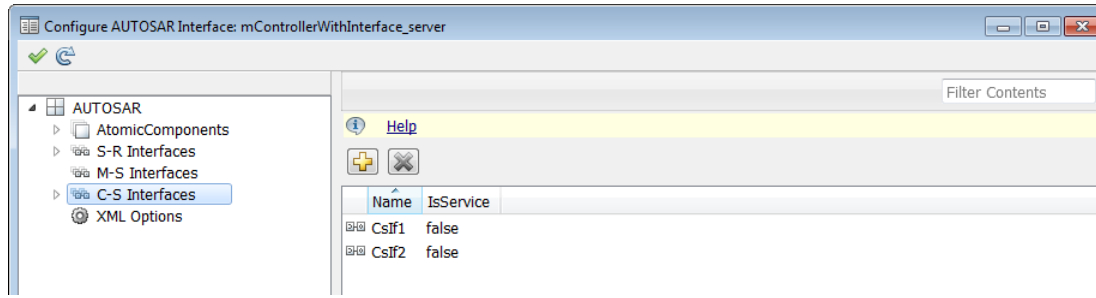
- Enter a package path in the **Package** parameter field.
- Click the button to the right of the **Package** field to open the AUTOSAR Package Browser. Use the browser to navigate to an existing package or create a new package. When you select a package in the browser and click **Apply**, the interface **Package** parameter value is updated with your selection. For more information about the AUTOSAR Package Browser, see “Configure AUTOSAR Package for Interface” on page 4-40.



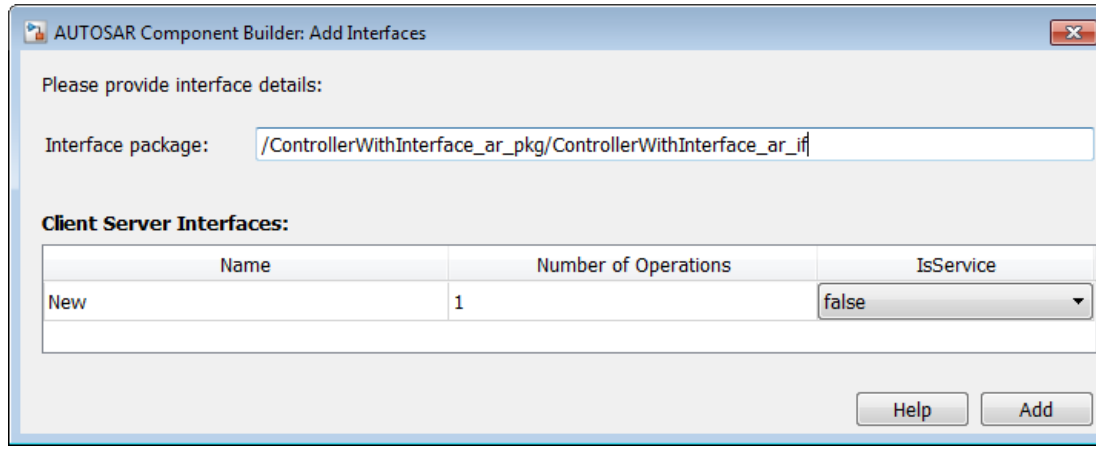
18 In the left-hand pane of the Configure AUTOSAR Interface dialog box, select **C-S Interfaces**.

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.

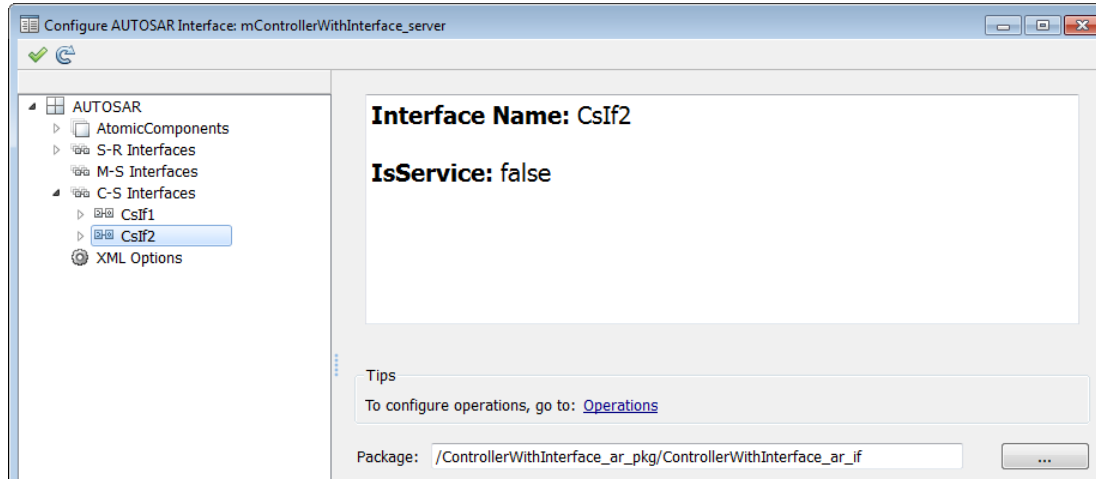


19 In the left-hand pane of the Configure AUTOSAR Interface dialog box, expand **C-S Interfaces** and select a C-S interface from the list.

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:

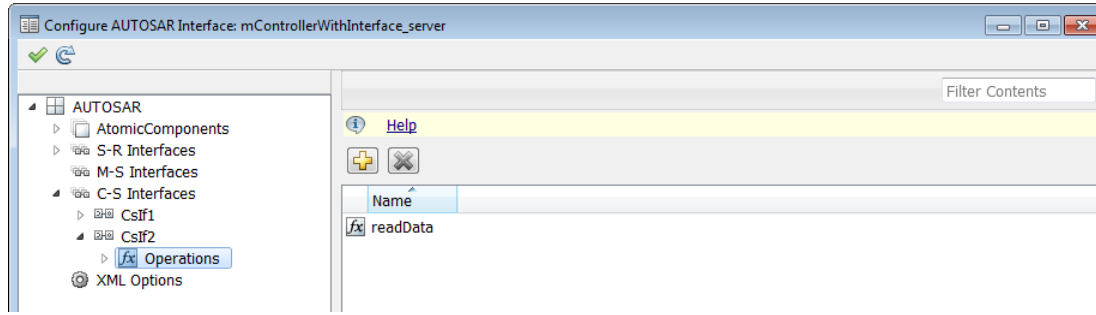
- Enter a package path in the **Package** parameter field.
- Click the button to the right of the **Package** field to open the AUTOSAR Package Browser. Use the browser to navigate to an existing package or create a new package. When you select a package in the browser and click **Apply**, the interface **Package** parameter value is updated with your selection. For more information about the AUTOSAR Package Browser, see “Configure AUTOSAR Package for Interface” on page 4-40.



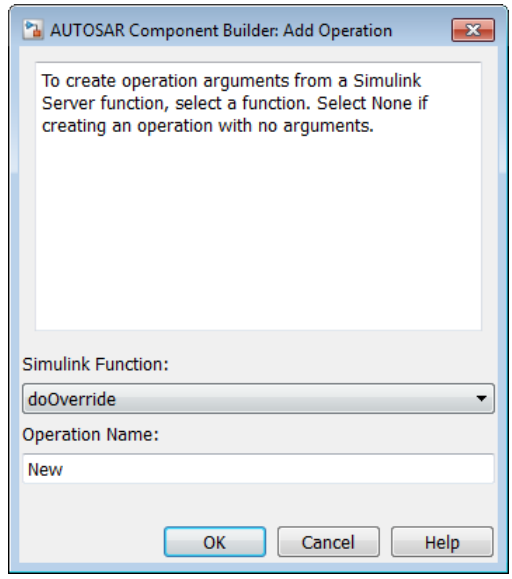
20 In the left-hand pane of the Configure AUTOSAR Interface dialog box, expand the selected interface and select **Operations**.

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.



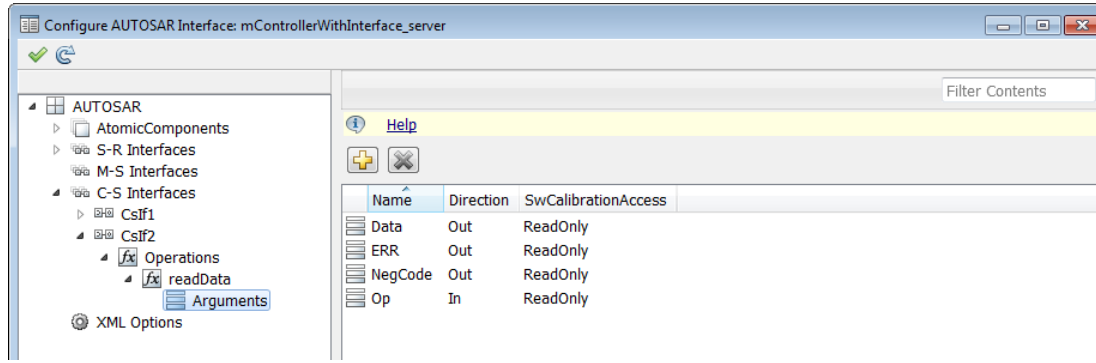
21 In the left-hand pane of the Configure AUTOSAR Interface dialog box, expand **Operations** and select an operation from the list.

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

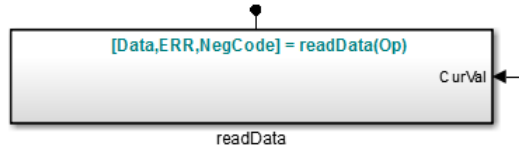
22 In the left-hand pane of the Configure AUTOSAR Interface dialog box, expand the selected operation and select **Arguments**.

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, by setting its **Direction** value to `In`, `Out`, or `InOut`.
- Specify the level of measurement and calibration tool access to a C-S operation argument, by setting its **SwCalibrationAccess** value to `ReadOnly`, `ReadWrite`, or `NotAccessible`.
- 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.



23 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 Packages for...
`modelname``_component.arxml`	Software components, including calibration components. This is the main `arxml` file exported for the Simulink model. In addition to component packages, the file includes elements

Exported File Name	By Default Contains Packages for...
	for which packages have not been configured using the XML Options view of the AUTOSAR Properties Explorer.
*modelname*_datatype.arxml	Data types and related elements, including • 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) **Note:** Elements for which packages are not configured are placed in the main `arxml` file, *modelname*_component.arxml. For more information about configuring packages, see "Configure AUTOSAR Package Structure".
*modelname*_implementation.arxml	Software component implementation.
*modelname*_interface.arxml	Interfaces, including S-R interfaces, C-S interfaces, and M-S interfaces.
*modelname*_behavior.arxml	Software component internal behavior (generated only for schema 3.x or earlier).

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

- Configure the AUTOSAR packages into which AUTOSAR elements are exported. (Grouping AUTOSAR elements into AUTOSAR packages is logically distinct from the output file packaging controlled by **Exported XML file packaging**.) Inspect

and modify the AUTOSAR package parameters grouped under the headings **Package Paths**, **Qualified Names**, and **Additional Packages**. For more information, see "Configure AUTOSAR Package Structure" on page 4-108.

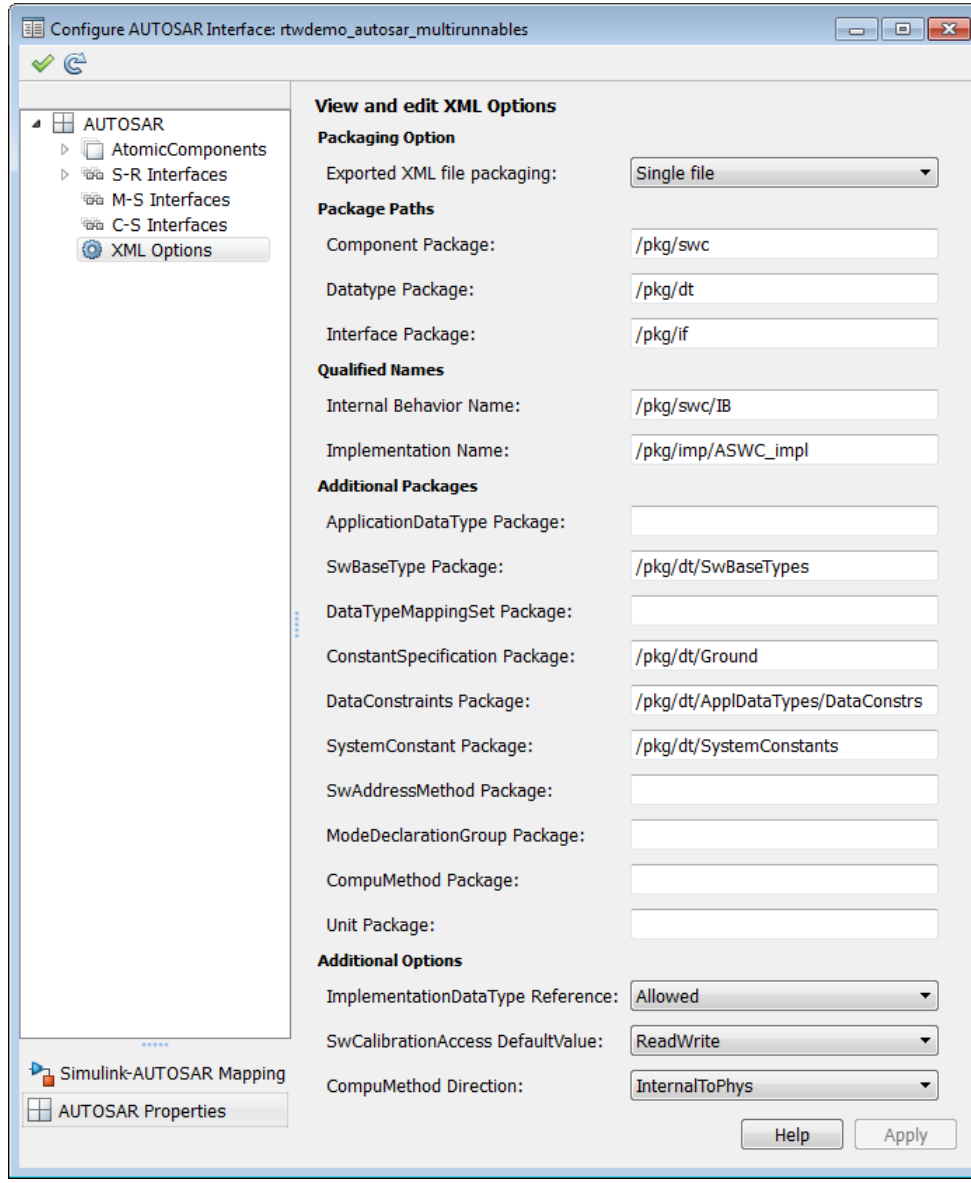
- Optionally override the default behavior for generating AUTOSAR application data types in XML. To force generation of an application data type for each AUTOSAR data type, set **ImplementationDataType Reference** to `NotAllowed`. For more information, see "Control Application Data Type Generation" on page 4-89.
- 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 4-80.
- Control the direction of `CompuMethod` conversion for linear-function `CompuMethod`s. Select one of the following values for **CompuMethod Direction**:

  - `InternalToPhys` (default) — Generate `CompuMethod` sections for conversion of internal values into their physical representations.
  - `PhysToInternal` — Generate `CompuMethod` sections for conversion of physical values into their internal representations.
  - `Bidirectional` — Generate `CompuMethod` sections for both internal-to-physical and physical-to-internal conversion directions.

  For more information, see "CompuMethod Direction for Linear Functions" on page 4-93.

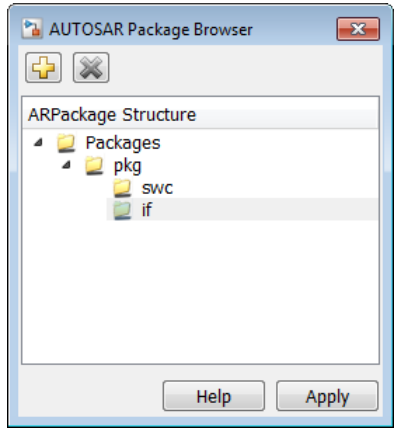

Configure AUTOSAR Interface: rtwdemo_autosar_multirunnables
AUTOSAR
AtomicComponents
S-R Interfaces
M-S Interfaces
C-S Interfaces
XML Options
Simulink-AUTOSAR Mapping
AUTOSAR Properties
View and edit XML Options
Packaging Option
Exported XML file packaging:
Single file
Package Paths
Component Package:
/pkg/swc
Datatype Package:
/pkg/dt
Interface Package:
/pkg/if
Qualified Names
Internal Behavior Name:
/pkg/swc/IB
Implementation Name:
/pkg/imp/ASWC_impl
Additional Packages
ApplicationDataType Package:
SwBaseType Package:
/pkg/dt/SwBaseTypes
DataTypeMappingSet Package:
ConstantSpecification Package:
/pkg/dt/Ground
DataConstraints Package:
/pkg/dt/ApplDataTypes/DataConstrs
SystemConstant Package:
/pkg/dt/SystemConstants
SwAddressMethod Package:
ModeDeclarationGroup Package:
CompuMethod Package:
Unit Package:
Additional Options
ImplementationDataType Reference:
Allowed
SwCalibrationAccess DefaultValue:
ReadWrite
CompuMethod Direction:
InternalToPhys
Help
Apply

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

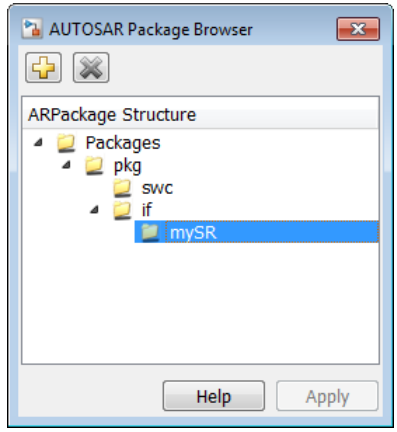
Configure AUTOSAR Package for Interface

As part of configuring AUTOSAR interfaces, you specify the AUTOSAR package for individual S-R, M-S, and C-S interfaces 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.

Configure AUTOSAR Multiple Runnables

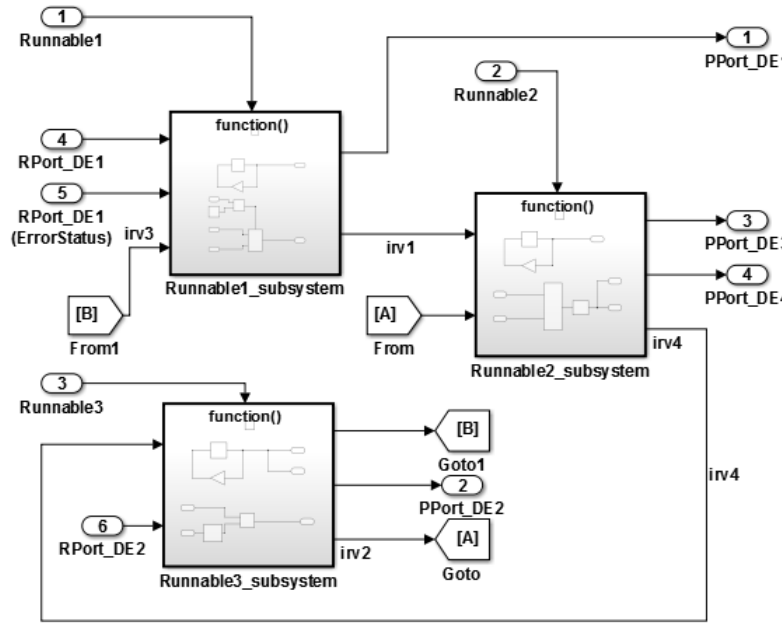
You model AUTOSAR multi-runnables as function-call subsystems at the top level of a model. When you generate code for the model, each function-call subsystem representing a runnable appears in the model C code as a callable model entry-point function. For example, here is an excerpt of the generated C code for function-call subsystem `Runnable3_subsystem` in the example model rtwdemo_autosar_multirunnables:

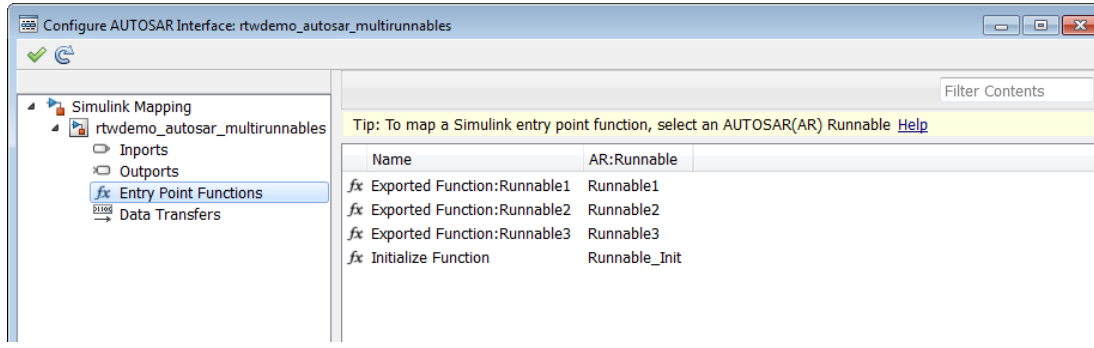
```
void Runnable3(void)
{
  /* local block i/o variables */
  real_T *rtb_TmpSignalConversionAtIn2Out;
  int8_T rtb_Delay;
  int8_T rtb_Gain;

  /* RootInportFunctionCallGenerator:
   * '<Root>/RootFcnCall_InsertedFor_Runnable3_at_outport_1'
   * incorporates:  SubSystem: '<Root>/Runnable3_subsystem'
   */
  /* UnitDelay: '<S3>/Delay' */
  rtb_Delay = rtDWork.Delay_DSTATE;

  /* Gain: '<S3>/Gain' incorporates:
   *  UnitDelay: '<S3>/Delay'
   */
  rtb_Gain = (int8_T)-rtDWork.Delay_DSTATE;
...
  /* Update for UnitDelay: '<S3>/Delay' */
  rtDWork.Delay_DSTATE = rtb_Gain;
}
```

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 reestablished 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.

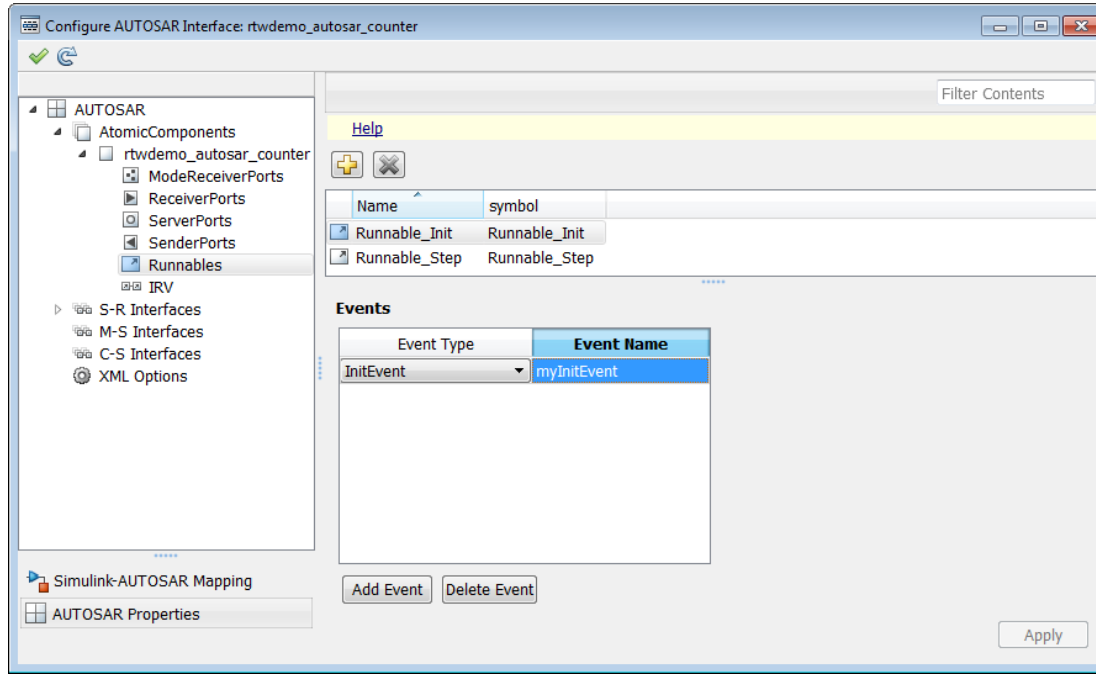
Configure AUTOSAR Initialization Runnable

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 4-52.)

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,

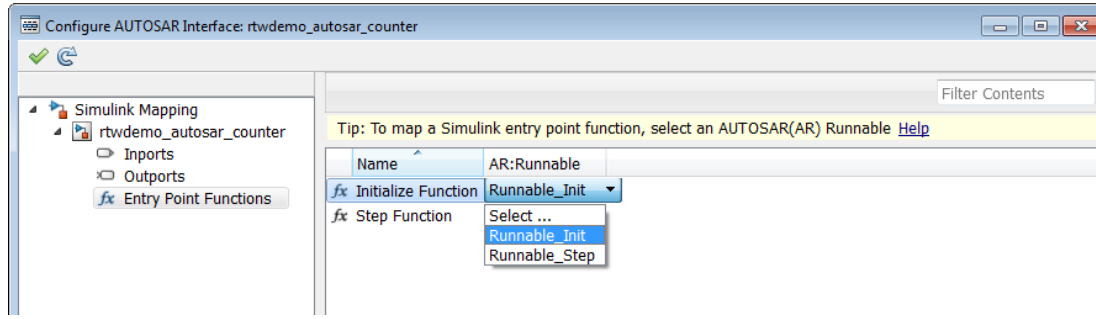
1. Open a model configured for AUTOSAR.
2. Open the Configuration Parameters dialog box, go to **Code Generation** > **AUTOSAR Code Generation Options**, and verify that the selected AUTOSAR schema version is `4.1`.
3. Open the Configure AUTOSAR Interface dialog box, and select the AUTOSAR Properties 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`.



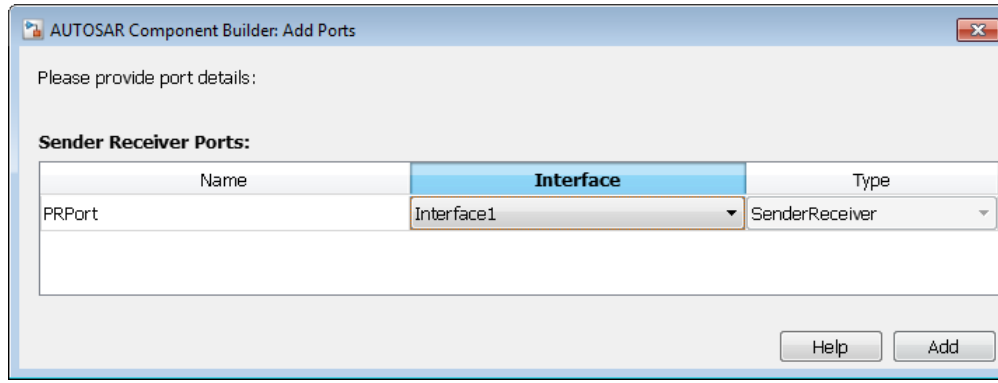
When you export `arxml` code from a model containing an initialization runnable, the `arxml` exporter generates an `InitEvent` that is mapped to the initialization runnable and function. For example:

```
<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>
```

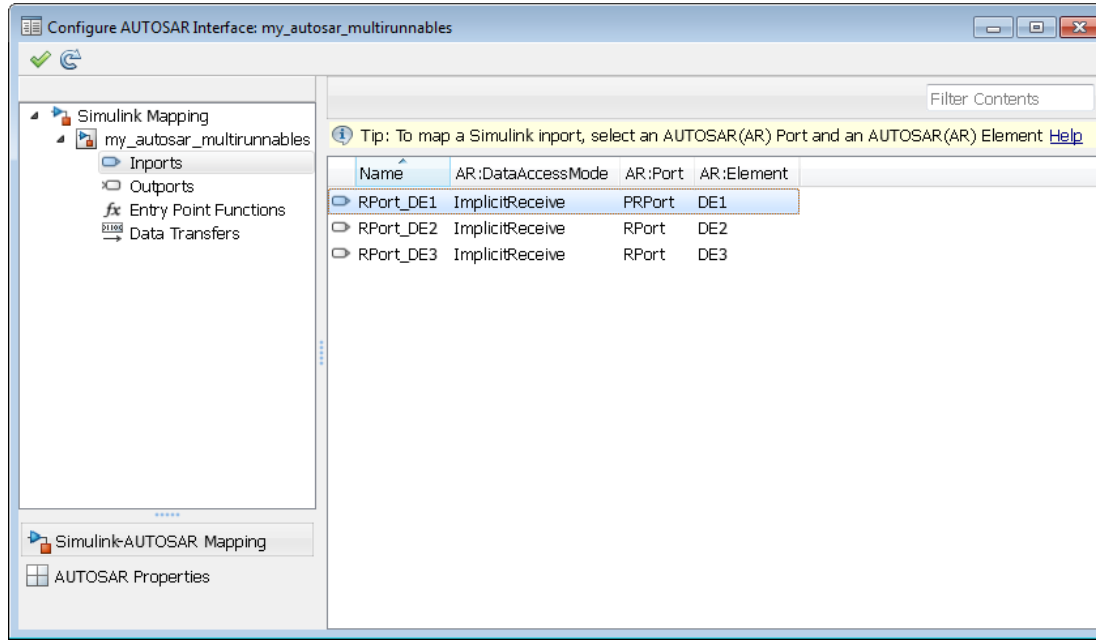
Configure AUTOSAR Provide-Require Port

AUTOSAR Release 4.1 introduced the AUTOSAR provide-require ports (`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:

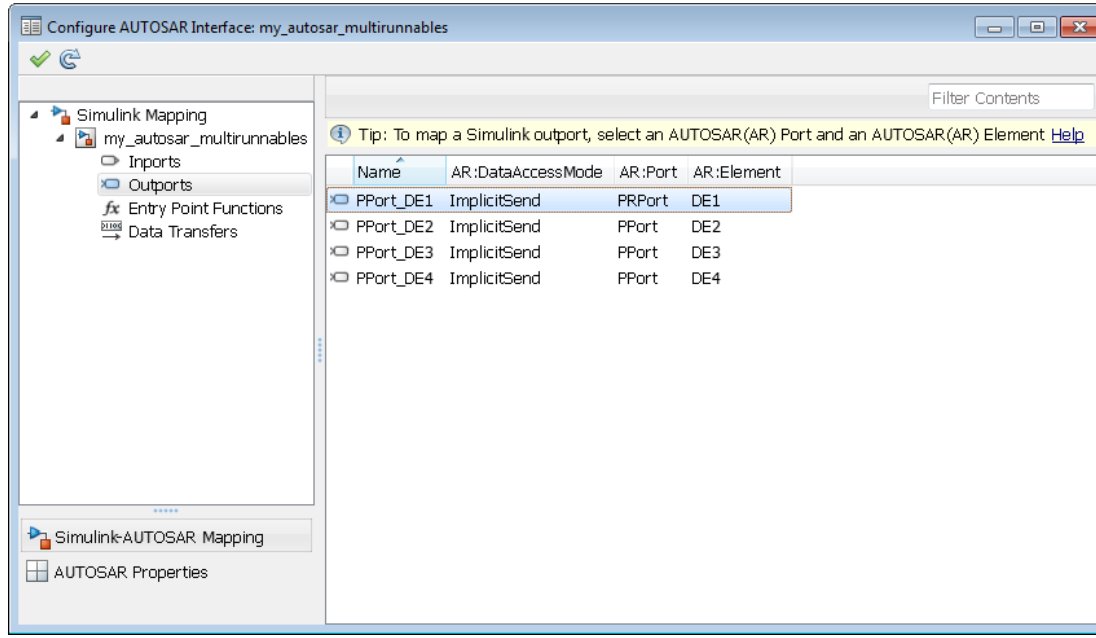
1. Open a model that is configured for AUTOSAR, and in which a runnable has an inport and an outport suitable for pairing into an AUTOSAR `PRPort`. In this example, the `RPort_DE1` inport and `PPort_DE1` outport both use data type `int8`, port dimension 1, and signal type `real`.
2. Open the Configure AUTOSAR Interface dialog box, and navigate to the SendReceiverPorts 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 outport to the AUTOSAR sender-receiver port you created, select the outport, set **AR:Port** to the value `PRPort`, and set **AR:Element** to the same data element selected in the previous step.



6 Click the Validate icon to validate the updated AUTOSAR interface configuration. If errors are reported, address them and then retry validation. A common error flagged by validation is mismatched properties between the inport and outport that are mapped to the AUTOSAR `PRPort`.

Alternatively , you can programmatically add and map a `PRPort` port using AUTOSAR property and map functions. The following example adds an AUTOSAR `PRPort` (sender-receiver port) and then maps it to a Simulink inport and outport pair.

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

swcPath =
    'ASWC

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

prportPath =
    'ASWC/PRPort'

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

```
mapInport(slMap,'RPort_DE1','PRPort','DE1','ImplicitReceive')
mapOutport(slMap,'PPort_DE1','PRPort','DE1','ImplicitSend')
[arPortName,arDataElementName,arDataAccessMode] = getOutport(slMap,'PPort_DE1')

arPortName =
PRPort

arDataElementName =
DE1

arDataAccessMode =
ImplicitSend'
```

Configure AUTOSAR Mode Receiver Ports and Mode-Switch Events

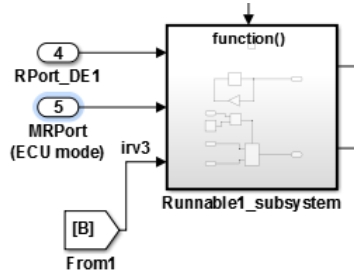
This example shows how to configure an AUTOSAR mode-receiver port and an AUTOSAR mode-switch event, using the example model `rtwdemo_autosar_multirunnables` as a starting point.

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

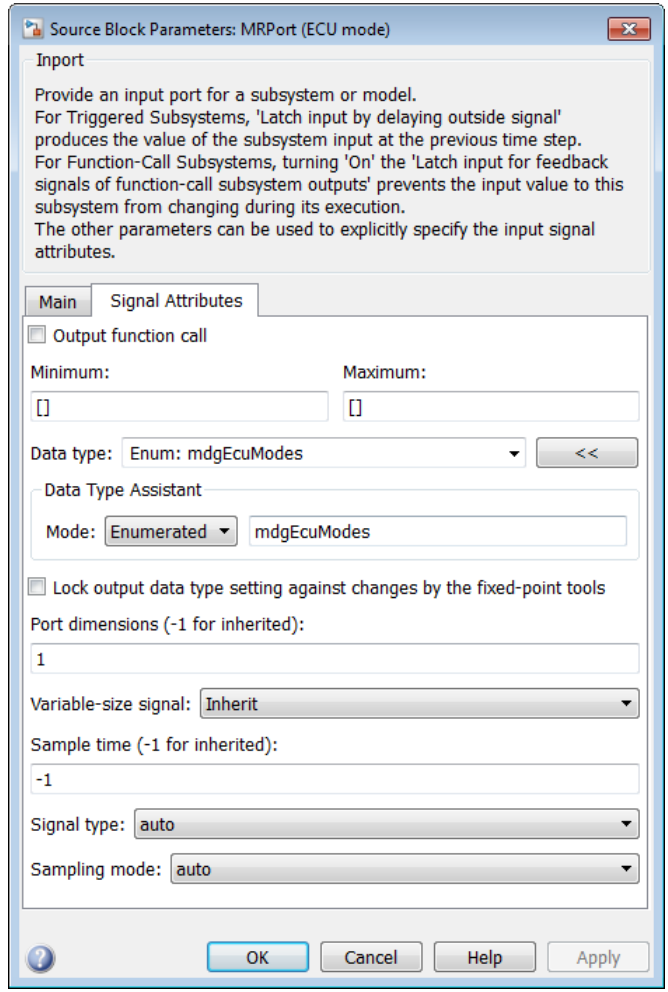
1. Open the example model rtwdemo_autosar_multirunnables. Save a copy to a writable work folder.
2. Declare a mode declaration group — a group of mode values — using Simulink enumeration. Enter the following command in the MATLAB Command Window:

```
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**.

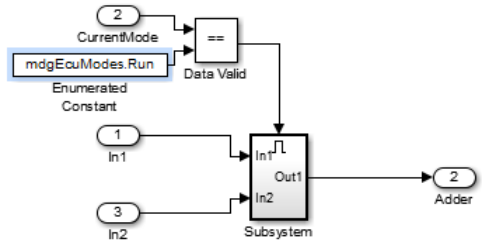


5 In the model window, open the function-call subsystem named `Runnable1_subsystem` and make the following changes:

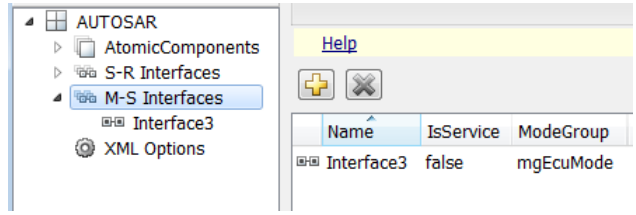
   a Rename inport `ErrorStatus` to `CurrentMode`.

   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**.

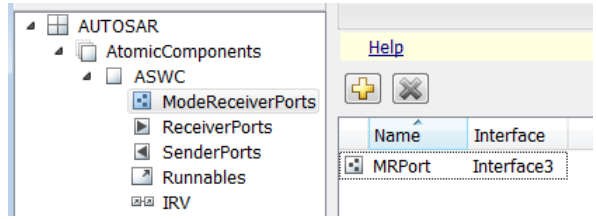


6 Add an AUTOSAR mode-switch interface to the model. Open the Configure AUTOSAR Interface dialog box using **Code > C/C++ Code > Configure Model as AUTOSAR Component**. Select the AUTOSAR Properties Explorer, and select **M-S Interfaces**. To open the Add Interfaces dialog box, click the Add icon . In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

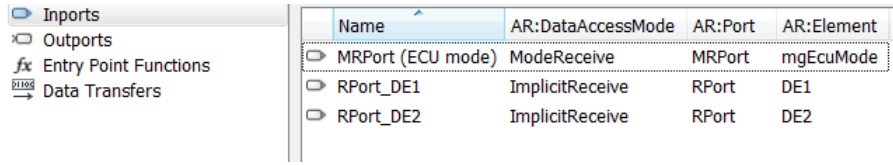


The value you specify for the AUTOSAR mode group is used in a later step, when you map a Simulink inport to an AUTOSAR mode-receiver port and element.

7 Add an AUTOSAR mode-receiver port to the model. In the AUTOSAR Properties Explorer, expand component `ASWC` and select **ModeReceiverPorts**. To open the Add Ports dialog box, click the Add icon . In the Add Ports dialog box, specify **Name** as `MRPort`. **Interface** is already set to `Interface3` (the only available value in this configuration), and **Type** is already set to `ModeReceiver`. Click **Add**.



8 Map the Simulink inport `MRPort (ECU mode)` to the AUTOSAR mode-receiver port and element. Select the Simulink-AUTOSAR Mapping Explorer, and select **Inports**. In the row for inport `MRPort (ECU mode)`, set **AR:DataAccessMode** to `ModeReceive`, set **AR:Port** to `MRPort`, and set **AR:Element** to `mgEcuMode`. (The AUTOSAR element value matches the **ModeGroup** value you specified when you added AUTOSAR mode-switch interface `Interface3`.)



This step completes the AUTOSAR mode-receiver port configuration. Click the Validate icon  to validate the AUTOSAR interface configuration. If errors are reported, address them and then retry validation. When the model passes validation, save the model.

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

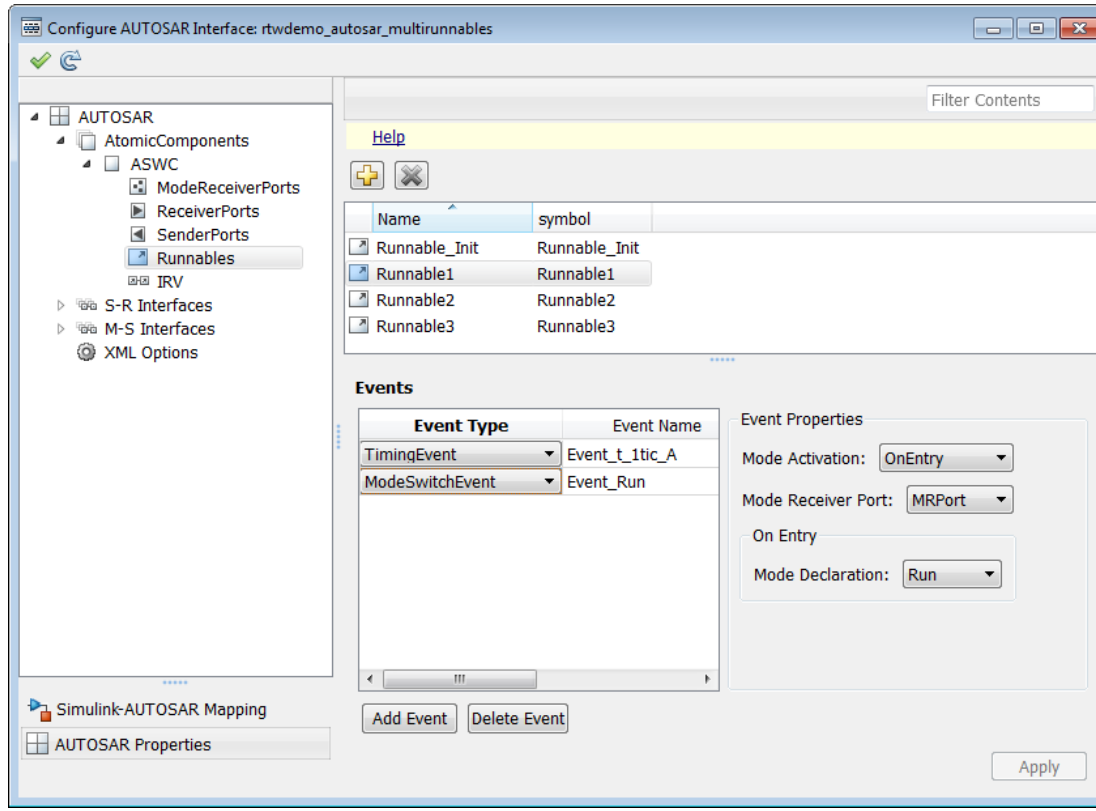
---

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

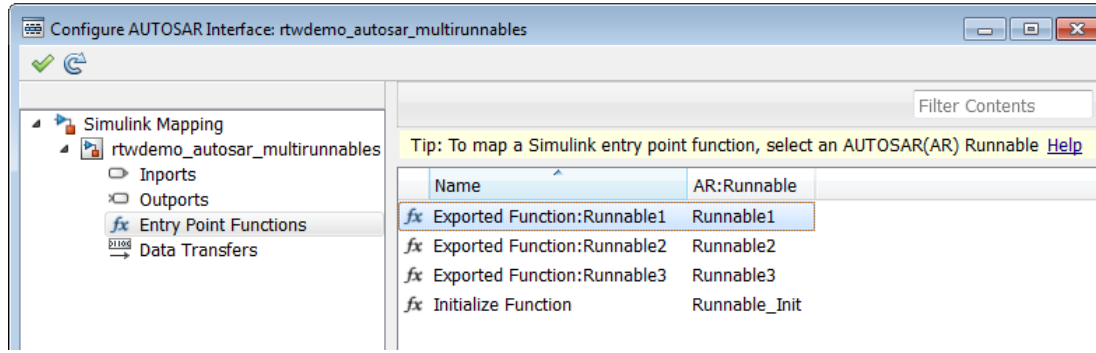
a Open the Configure AUTOSAR Interface dialog box, if it is not already open. Select the AUTOSAR Properties Explorer, expand the `ASWC` component, and select **Runnables**. In the list of runnables, select `Runnable1`. This selection activates an **Events** configuration pane for the runnable.

b To add an event to the list of events for `Runnable1`, click **Add Event**. For the new event, set **Event Type** to `ModeSwitchEvent`. (This activates an **Event Properties** subpane.) Specify **Event Name** as `Event_Run`.

c In the **Event Properties** subpane, set **Mode Activation** to `OnEntry`, set **Mode Receiver Port** to `MRPort`, and set **Mode Declaration** to `Run`. Click **Apply**.



10 Select the Simulink-AUTOSAR Explorer, and select **Entry Point Functions**. In this example model, Simulink entry-point functions have already been mapped to AUTOSAR runnables, including the runnable `Runnable1`, to which you just added a mode-switch event.



11 This completes the AUTOSAR mode-switch event configuration. Click the Validate icon  to validate the AUTOSAR interface configuration. If errors are reported, address them and then retry validation. When the model passes validation, save the model. Optionally, you can generate XML and C code from the model and inspect the results.

Configure Disabled Mode for 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`, or `OperationInvokedEvent`. 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 properties 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 Client-Server Communication

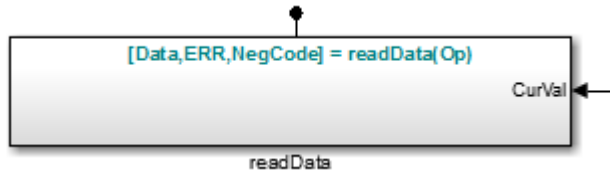
In this section...

You can model AUTOSAR clients and servers 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 2-9. This section provides detailed examples of AUTOSAR client and server configuration.

Configure AUTOSAR Server

This example shows how to configure a Simulink Function block as an AUTOSAR server. The example uses the files `mControllerWithInterface_server.slx` and `ExampleApplicationErrorType.m` in *`matlabroot`*`/help/toolbox/ecoder/examples/client_server`.

1. Add a Simulink Function block to a model. In the Simulink Library Browser, the Simulink Function block is in **User-Defined Functions**.
2. Configure the Simulink Function block to implement a server function. Configure a function prototype and implement the server function algorithm. For example, here is a Simulink Function block for a `readData` function.

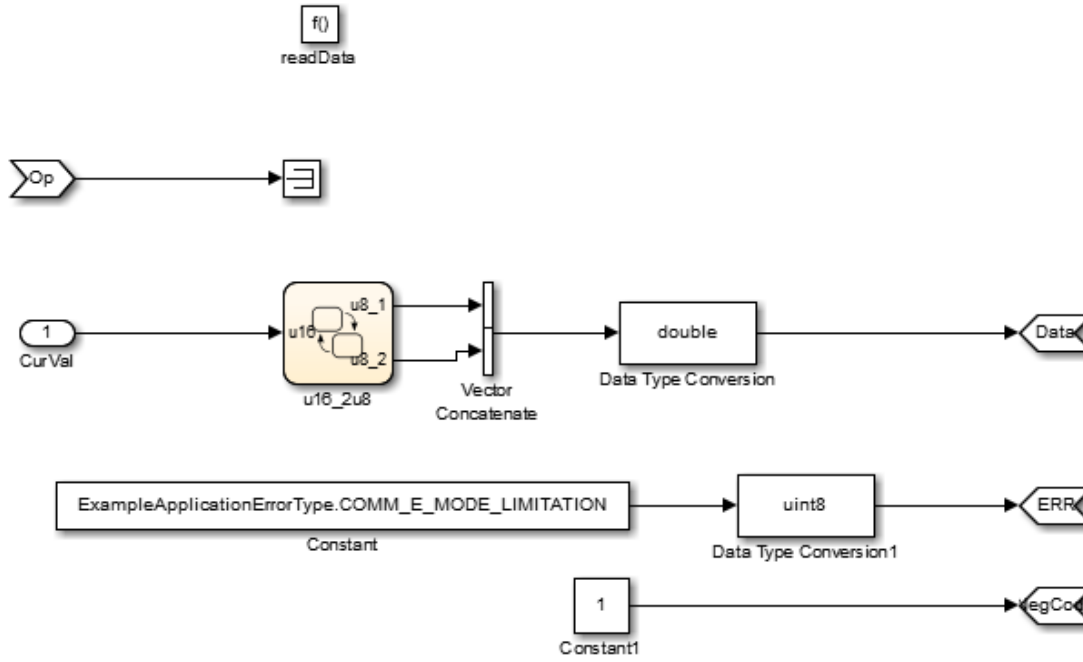


Here are the contents of the Simulink Function block, which implement the server function `readData`. The contents include:

- Argument Inport block `Op` and Argument Outport blocks `Data`, `ERR`, and `NegCode`, corresponding to the function prototype `[Data,ERR,NegCode]=readData(Op)`.

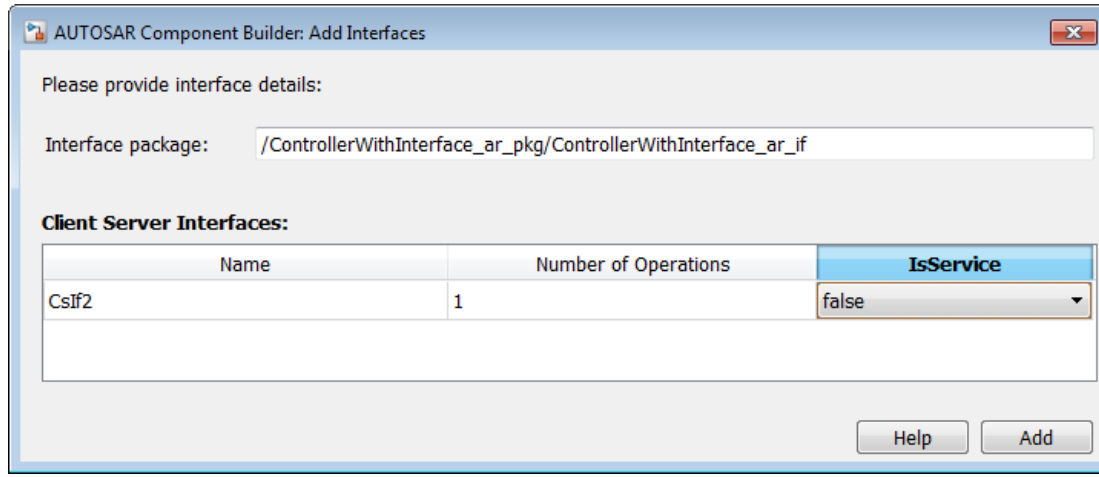
Note: When configuring server function arguments, you must specify signal data type, port dimensions, and signal type on the **Signal Attributes** tab of the inport and outport blocks. The AUTOSAR configuration fails validation if signal attributes are absent for server function arguments.

- Blocks implementing the `readData` function algorithm.
- A Constant block representing the value of an application error defined for the server function.
- 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.



3 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.

4 If an AUTOSAR client-service interface does not yet exist in the AUTOSAR configuration, create one. Open the Configure AUTOSAR Interface dialog box, and select the AUTOSAR Properties Explorer. In the C-S Interfaces view, and click the Add icon  . In the Add Client Server Interfaces dialog box, name and configure the new C-S Interface, and then click **Add**.

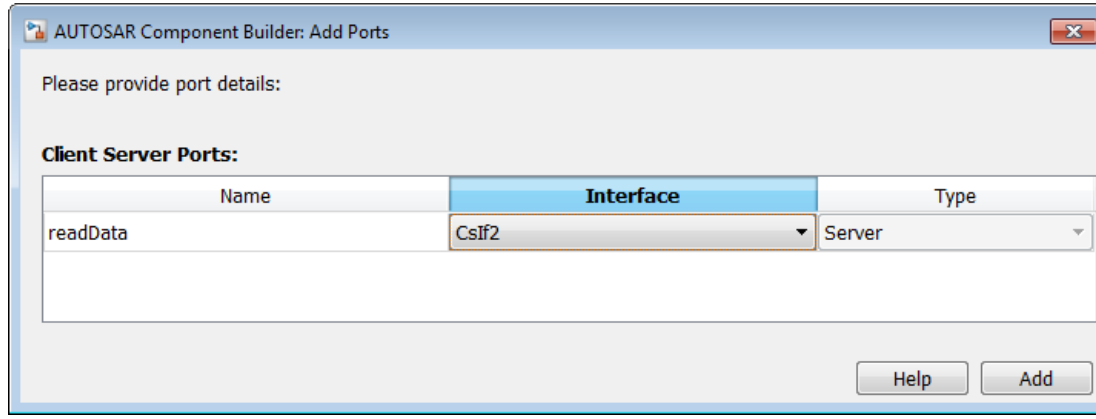


5 If an AUTOSAR server port does not yet exist in the AUTOSAR configuration, create one. In the AUTOSAR Properties Explorer, expand **AtomicComponents** and expand the individual component that you are configuring. In the ServerPorts view, click the Add icon  . In the Add Client Server Ports dialog box:

a Set **Name** to the name of the server function, `readData`.

b In the **Interface** list, select a C-S interface.

c Click **Add**.

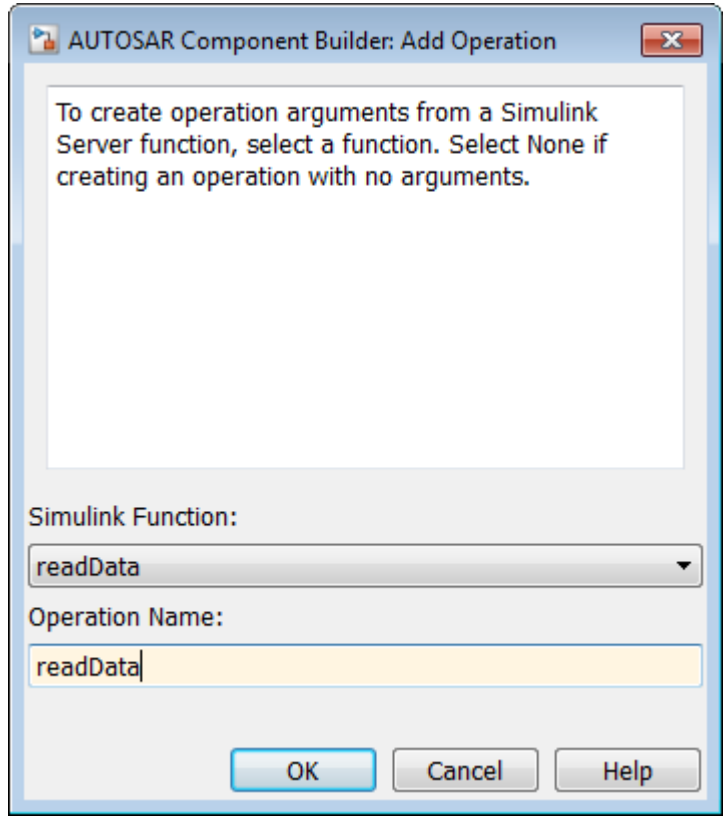


6 Create an AUTOSAR server operation corresponding to the Simulink server function. In the AUTOSAR Properties Explorer, expand **C-S Interfaces**, and expand the individual C-S interface that you selected for the server port. In the Operations view, click the Add icon . In the Add Operation dialog box:

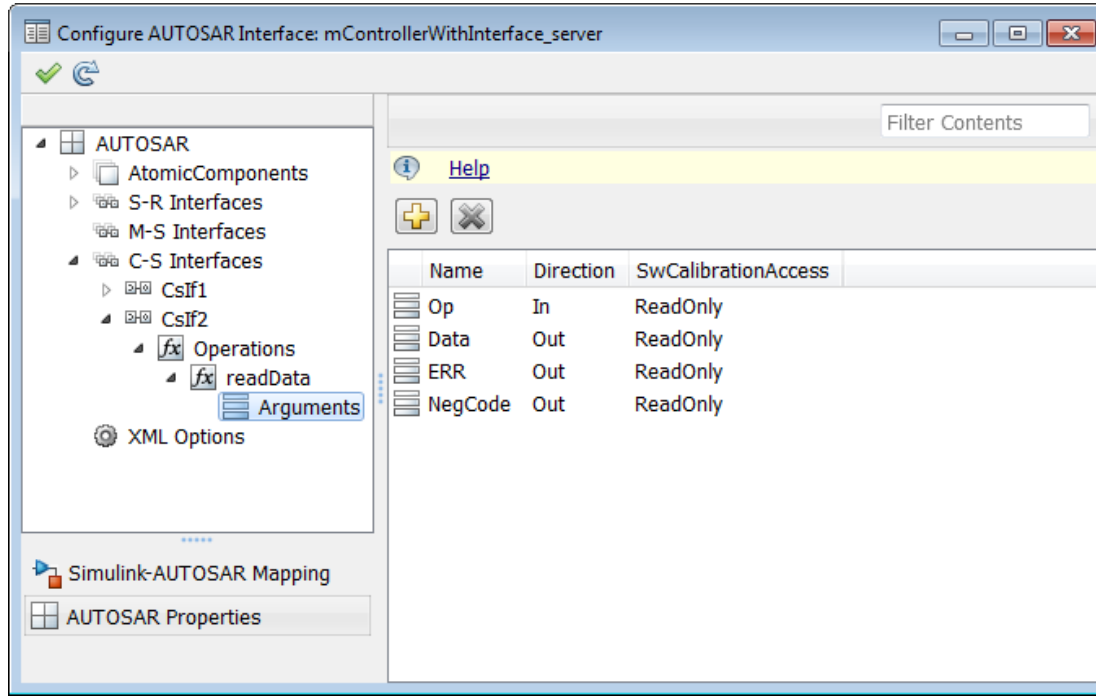
a Enter the **Operation Name**.

b In the **Simulink Function** list, select the server function `readData`. When you click **OK**, operation arguments are created based on the arguments of the server function.

c Click **OK**.



7 Examine the server operation arguments that you created. Expand **Operations**, expand the individual operation that you created in the previous step, and select **Arguments**. The listed arguments correspond to the Simulink server function prototype.



8 Configure an AUTOSAR server runnable for the server function `readData`. (This step requires that the AUTOSAR C-S interface, server port, and operation for `readData` exist in the model.) In the AUTOSAR Properties Explorer, expand **AtomicComponents** and expand the individual component that you are configuring. In the Runnables view:

a If a suitable runnable does not already exist in the AUTOSAR configuration, use the Add icon  to add a new runnable.

b Select the runnable and configure its basic properties. See `Runnable_readdata` properties in the figure.

Note: The **symbol** name specified for the runnable must match the Simulink server function name.

---

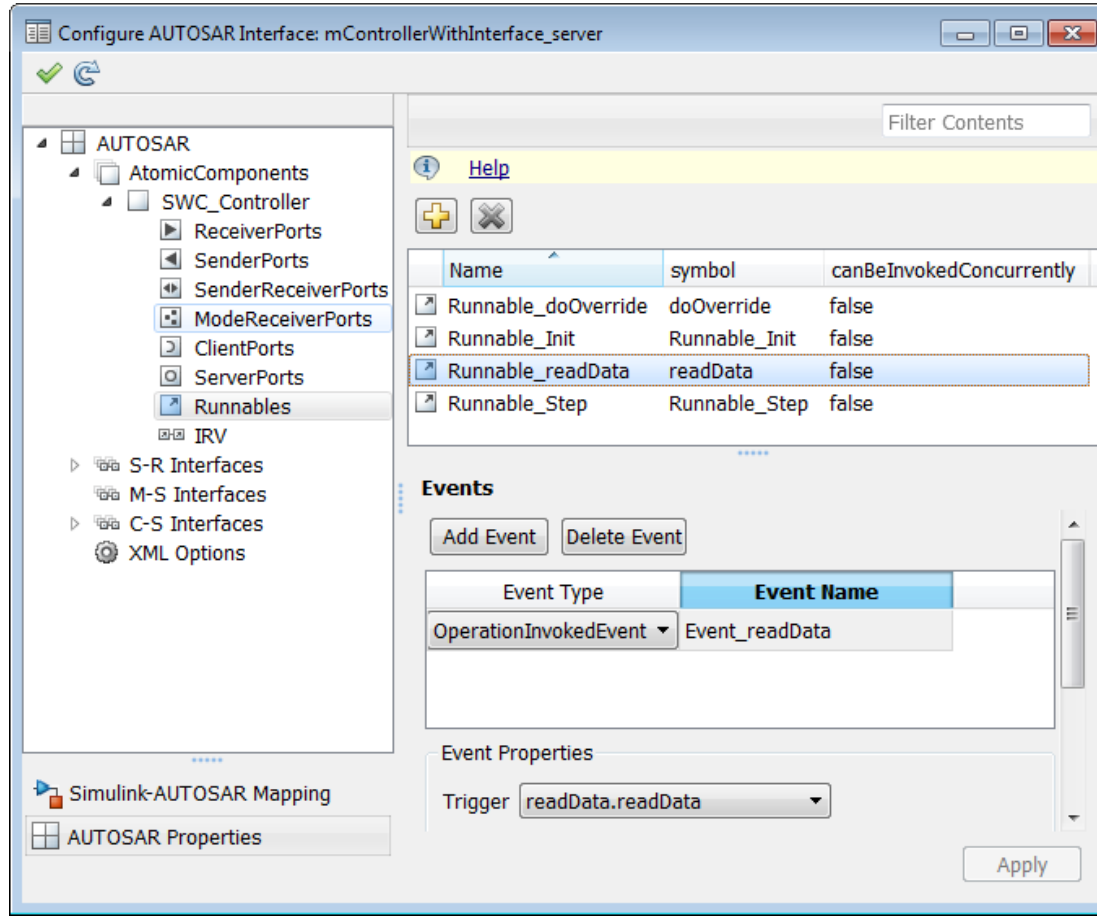
c To make the runnable a server runnable, create an operation-invoked event to trigger the runnable.

i Under **Events**, click **Add Event**. Select the new event.

ii Enter the **Event Name**.

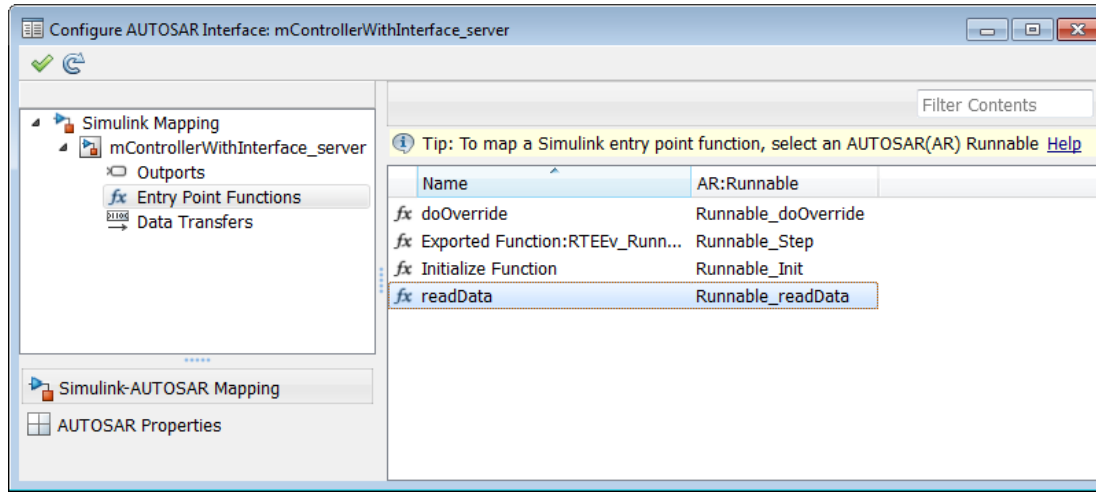
iii For **Event Type**, select `OperationInvokedEvent`.

iv Under **Event Properties**, set **Trigger** to the value that corresponds to the server port and C-S operation previously created for `readData`. Click **Apply**.



9 Map the Simulink server function to the AUTOSAR server runnable. In the Configure AUTOSAR Interface dialog box, switch to the Simulink-AUTOSAR

Mapping Explorer. In the Entry Point Functions view, select the `readData` function. In the **AR:Runnables** list, select the server runnable configured in the previous step.



10 Validate the configuration, and fix any errors that are flagged, until validation succeeds.

11 Generate C code and `arxml` for the model.

After you configure an AUTOSAR server, configure a corresponding AUTOSAR client invocation, as described in “Configure AUTOSAR Client” on page 4-66.

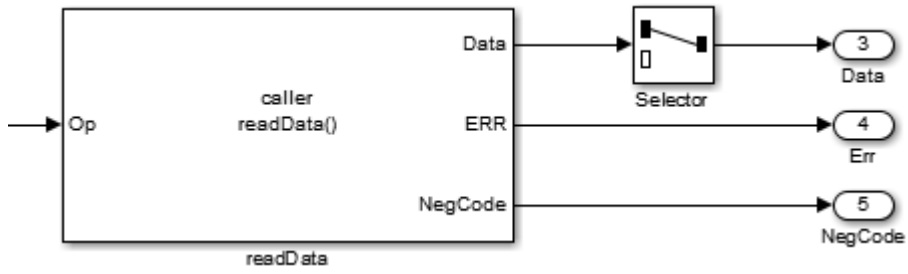
Configure AUTOSAR Client

After you configure an AUTOSAR server, as described in “Configure AUTOSAR Server” on page 4-59, 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 `mControllerWithInterface_client.slx` in *`matlabroot`*`/help/toolbox/ecoder/examples/client_server`.

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

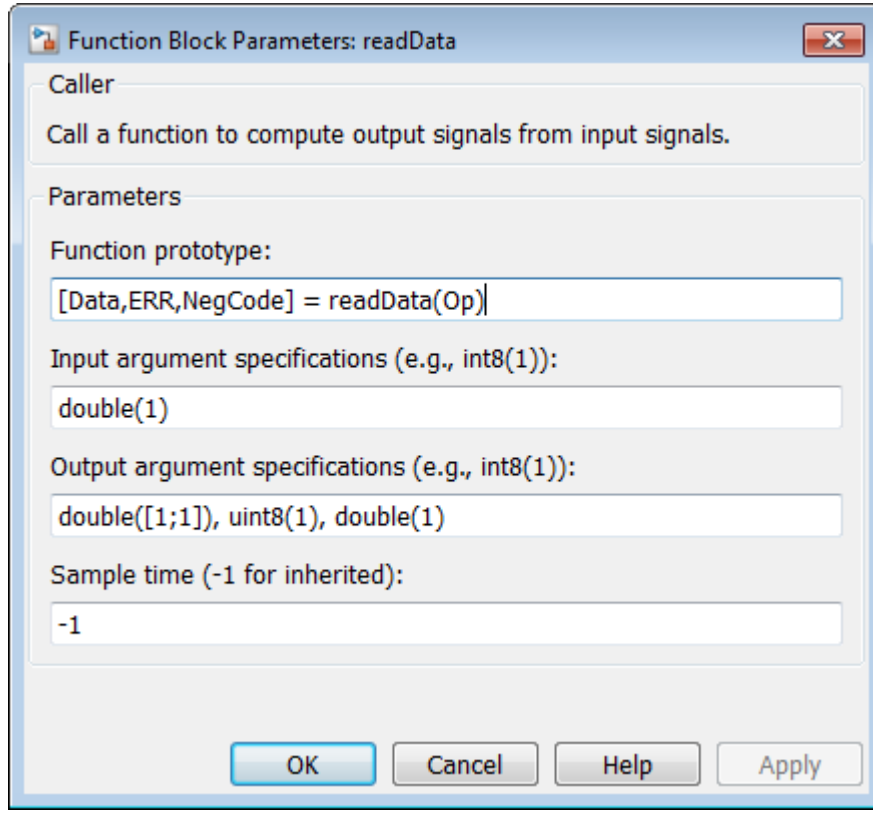
2 Configure the Function Caller block to call a corresponding Simulink Function block. Edit the block attributes to specify the server function prototype. For example, here

is a Function Caller block for the `readData` server function defined in the previous section.



Note: The function name specified in the Function Caller block must match the **Operation Name** specified when creating an AUTOSAR operation on the C-S interface for the server port. See the operation creation step in "Configure AUTOSAR Server" on page 4-59.

Shown in this graphic are the block attributes for the Function Caller block for `readData`. The attributes include the function prototype and argument specifications, which must match the function prototype and arguments previously specified for the `readData` server function.

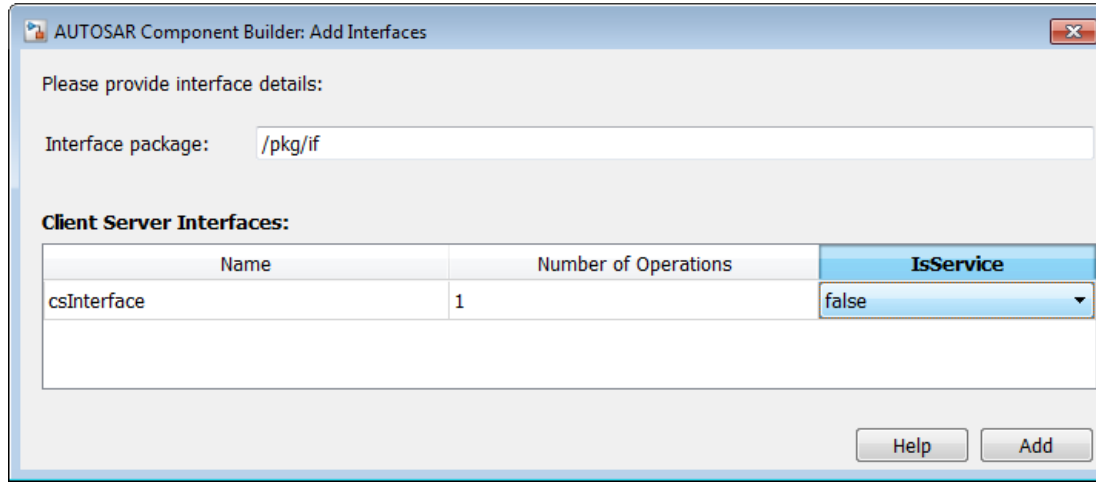


3 When the function invocation is working in Simulink, set up the Function Caller block in a model configured for AUTOSAR. For example, configure the current model for AUTOSAR or copy the block into an AUTOSAR model.

If you copy the Function Caller block into an AUTOSAR model, open the Configure AUTOSAR Interface dialog box and click the Synchronize icon  . This action synchronizes Simulink data transfers and function callers in your model. After synchronizing, the function caller you added appears in the Function Callers view of the Simulink-AUTOSAR Mapping Explorer.

4 If an AUTOSAR client-service interface does not yet exist in the AUTOSAR configuration, create one. Open the Configure AUTOSAR Interface dialog box, and select the AUTOSAR Properties Explorer. In the C-S Interfaces view, and click the

Add icon . In the Add Client Server Interfaces dialog box, name and configure the new C-S Interface, and then click **Add**.

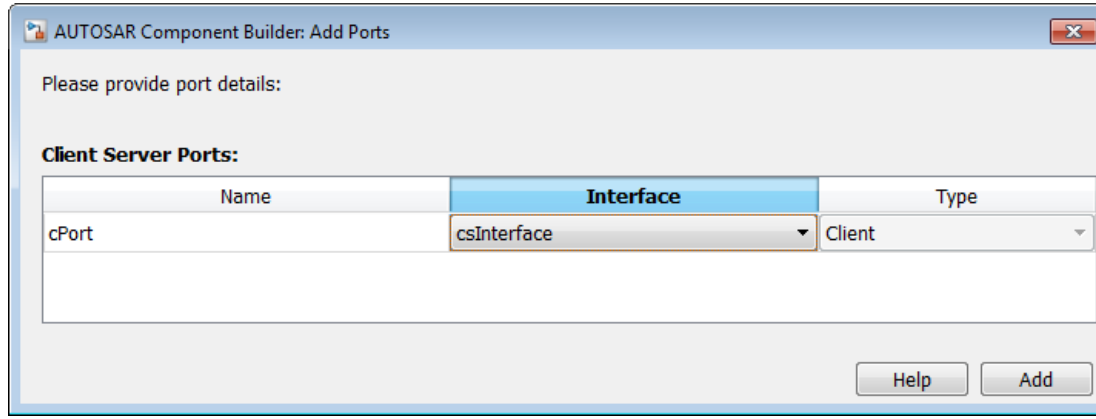


5 If an AUTOSAR client port does not yet exist in the AUTOSAR configuration, create one. In the AUTOSAR Properties Explorer, expand **AtomicComponents** and expand the individual component that you are configuring. In the ClientPorts view, click the Add icon . In the Add Client Server Ports dialog box:

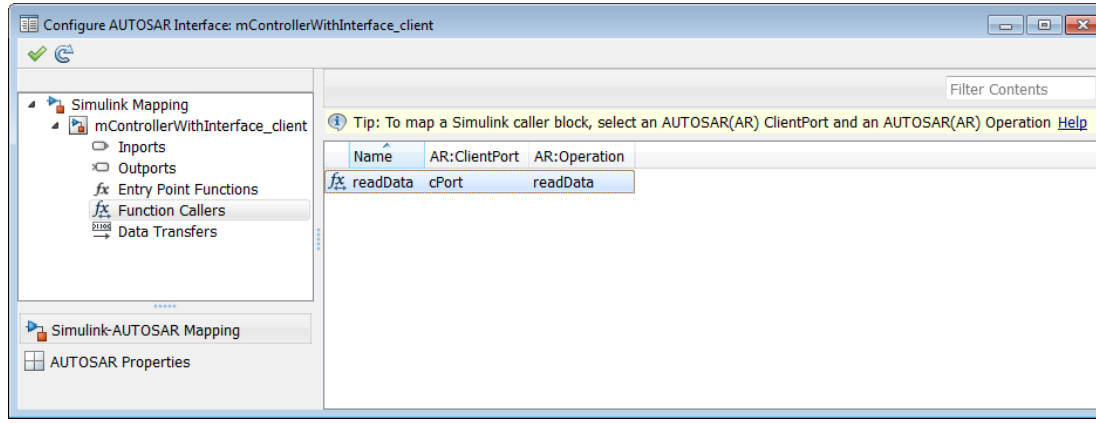
a Enter a port name in the **Name** field.

b In the **Interface** list, select a C-S interface.

c Click **Add**.



6 Map the Simulink function caller to an AUTOSAR client port and operation. In the Configure AUTOSAR Interface dialog box, switch to the Simulink-AUTOSAR Mapping Explorer. In the Function Callers view, select the `readData` function. From the **AR:ClientPort** and **AR:Operation** lists, select the corresponding client port and C-S operation from the ports and operations available in the model.



7 Validate the configuration and fix any errors that are flagged, until validation succeeds.

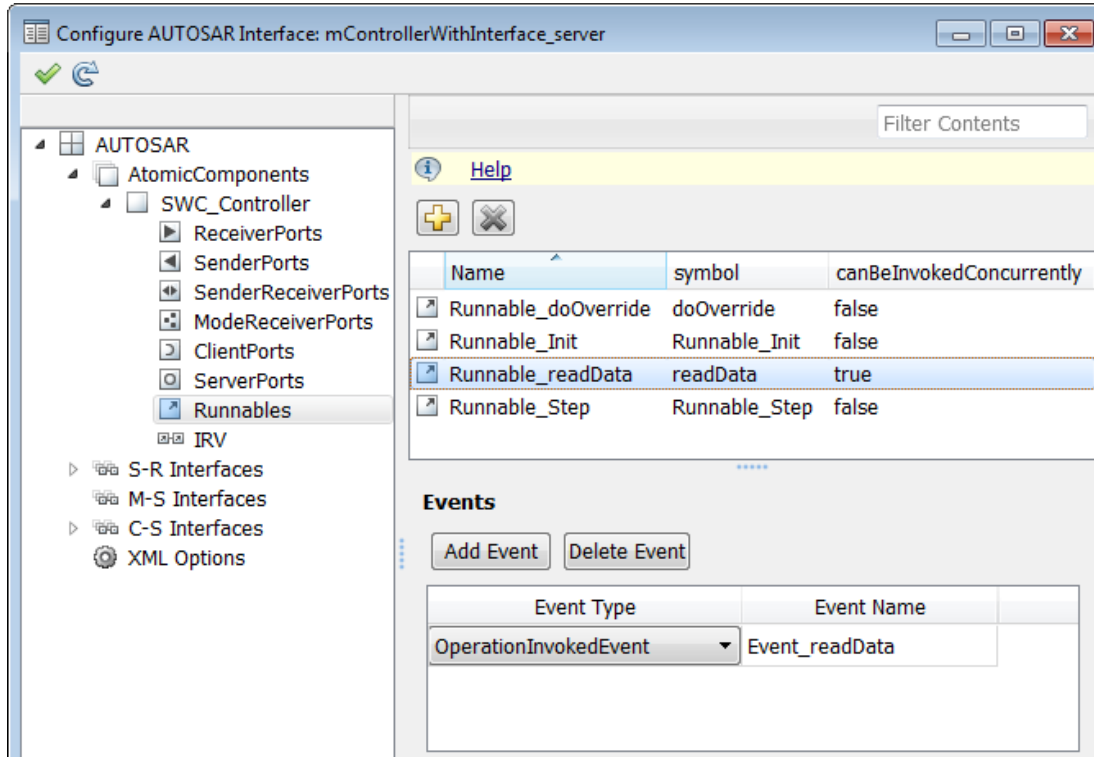
8 Generate C code and `arxml` for the model.

Concurrency Constraints for AUTOSAR Server Runnables

The following blocks and modeling patterns are incompatible with concurrent execution of an AUTOSAR server runnable.

- Blocks inside a Simulink function:
  - Blocks with state, such as Unit Delay.
  - Blocks with zero-crossing logic, such as Triggered Subsystem and Enabled Subsystem.
  - Stateflow® charts.
  - Other Simulink Function blocks.
  - Noninlined subsystems.
  - Legacy C function calls with side effects.
- Modeling patterns inside a Simulink function:
  - Writing to a data store memory (per-instance-memory).
  - Writing to a global block signal (for example, static memory).

To enforce concurrency constraints for AUTOSAR server runnables, use the runnable property `canBeInvokedConcurrently`. The property is located in the Runnables view of the AUTOSAR Properties Explorer.



When `canBeInvokedConcurrently` is set to `true` for a server runnable, AUTOSAR validation checks for blocks and modeling patterns that are incompatible with concurrent execution of a server runnable. If a Simulink function contains an incompatible block or modeling pattern, validation reports errors. If `canBeInvokedConcurrently` is set to `false`, validation does not check for blocks and modeling patterns that are incompatible with concurrent execution of a server runnable.

You can set the property `canBeInvokedConcurrently` to `true` only for an AUTOSAR server runnable — that is, a runnable with an `OperationInitiatedEvent`. 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 properties function `set`. The following example sets the runnable property

`canBeInvokedConcurrently` to `true` for an AUTOSAR sever 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'
```

MATLAB APIs for Client and Server Configuration

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 properties functions such as `set` and `get`.

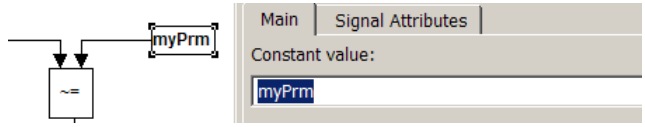
Configure AUTOSAR Calibration Parameters

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

For example, to configure an *internal calibration parameter* for your AUTOSAR model:

1. Create an `AUTOSAR.Parameter` object.
   a. Open the Model Explorer (**Ctrl+H**).
   b. In the **Model Hierarchy** view, under **Simulink Root**, select **Base Workspace**.
   c. Select **Add** > **Add Custom**. The Model Explorer – Select Object dialog box opens.
   d. Specify a value in the **Object Name(s)** field, for example, `myPrm`.
   e. From the **Object class** drop-down list, select `AUTOSAR.Parameter`.
   f. Click **OK**. A new object `myPrm` appears in the base workspace.
2. In the **Contents** pane, select the object, for example, `myPrm`.
3. Using the **Dialog** pane, 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 a creating 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`. You must then 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`

   For information about the **Dialog** pane, see "Model Explorer: Property Dialog Pane" in the Simulink documentation.
4. In the Block Parameters dialog box, assign the data object to your model, for example:



Before you generate code, you must:

- Open the Configuration Parameters dialog box and select **Optimization** > **Signals and Parameters** > **Inline parameters**.
- Clear **Code Generation** > **Ignore custom storage classes**.

Specifying these parameters allows the software to export the calibration parameters. See “Generate Code with Custom Storage Classes”.

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

Configure AUTOSAR Calibration Component

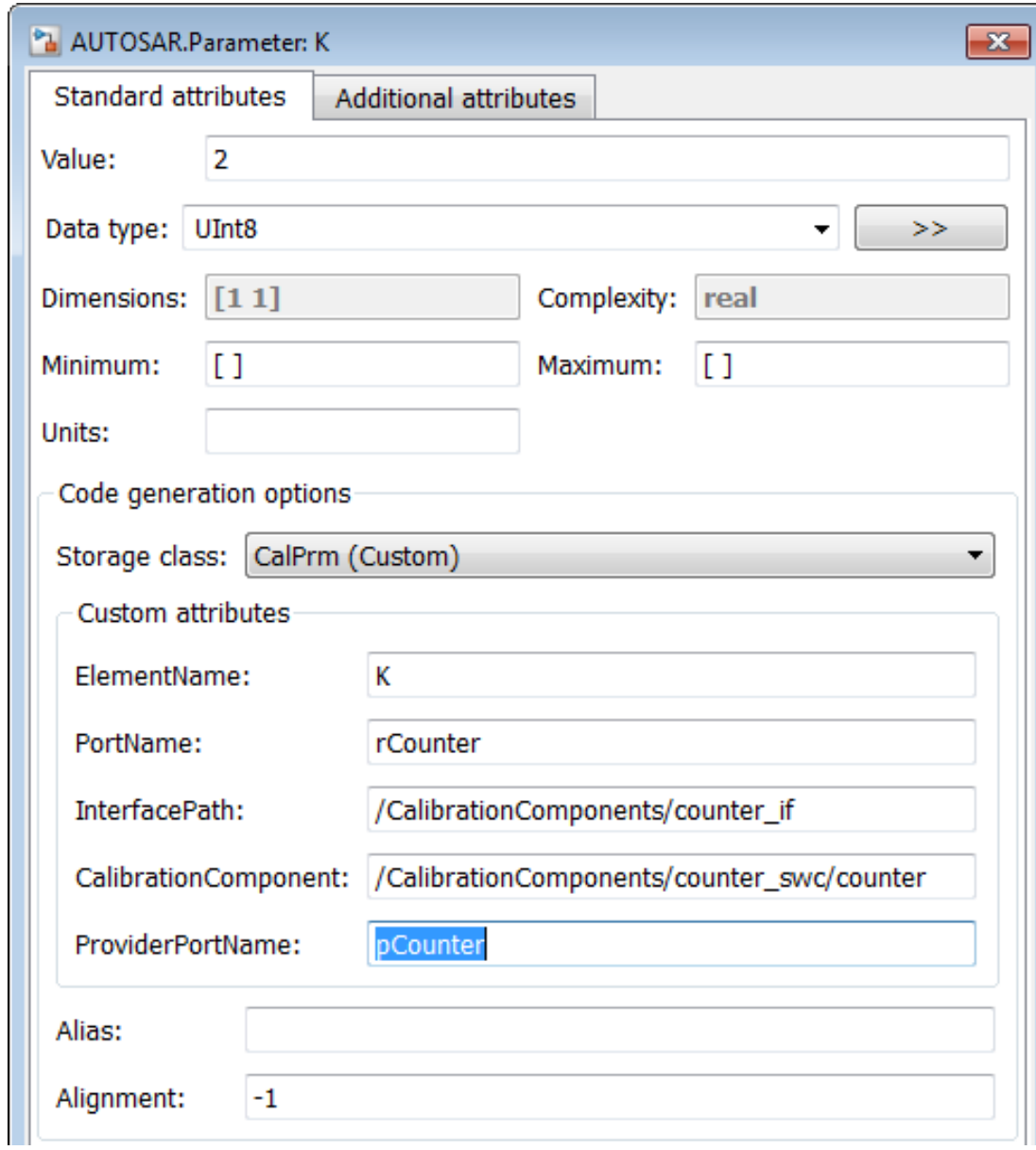
An AUTOSAR calibration component (`ParameterSwComponent`) contains calibration parameters that can be accessed by AUTOSAR software components (ASWCs) using an associated provider port. You can import a calibration component from `arxml` into Simulink or create a calibration component in Simulink.

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

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

```
<R-PORT-PROTOTYPE UUID="...">
  <SHORT-NAME>rCounter</SHORT-NAME>
  <REQUIRED-COM-SPECS>
...
    <PARAMETER-REQUIRE-COM-SPEC>
      <INIT-VALUE>
        <CONSTANT-REFERENCE>
```

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

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 AUTOSAR Data for Measurement and Calibration

In this section...

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`.

For more information, see "Configure SwCalibrationAccess" on page 4-80, "Configure swAddrMethod" on page 4-85, "Configure swAlignment" on page 4-87, and "swImplPolicy for Exported Data" on page 4-87

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 4-81
- "Specify SwCalibrationAccess for Signal and Parameter Data Objects" on page 4-83
- "Specify Default SwCalibrationAccess for Application Data Types" on page 4-84

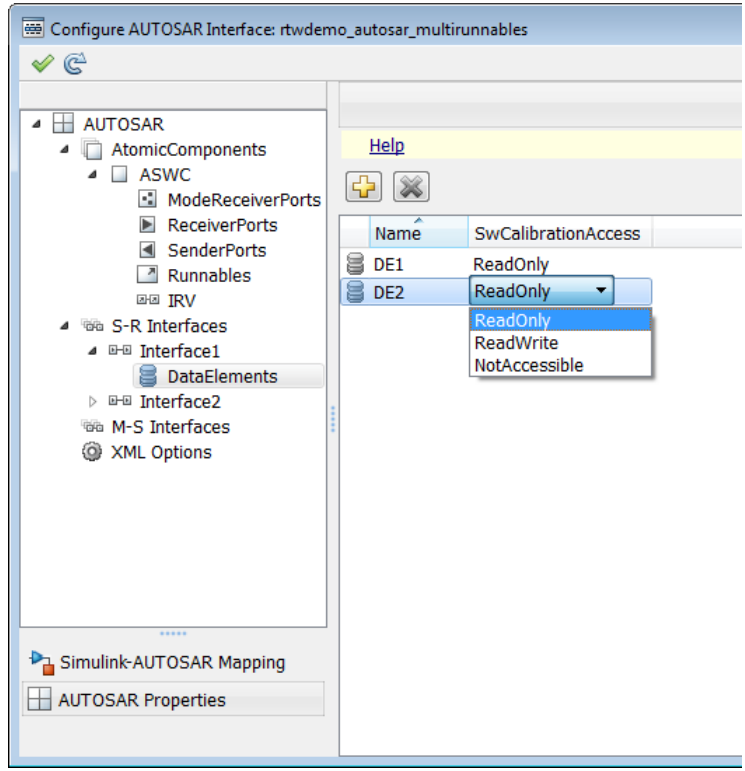
Specify `SwCalibrationAccess` for AUTOSAR Data Elements

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
- 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 **S-R Interfaces**, or to the **IRV** view under **Atomic Components**.
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 properties 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`

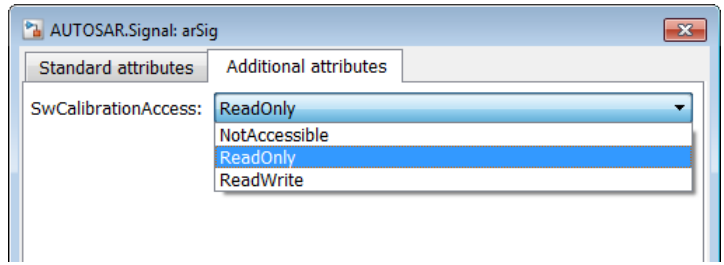
For example:

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
           SamplingMode: 'auto'
           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 properties 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 4-85
- "Add, Find, or Set swAddrMethod for Data" on page 4-86

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
- Cclient-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 "Memory Sections" 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 properties 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 = 'El1';
set(dataObj,[interfacePath dataElementName],'SwAddrMethod',sw1Path);
swPathGet = get(dataObj,[interfacePath dataElementName],'SwAddrMethod',...
  'PathType','FullyQualified');
```

Configure `swAlignment`

The `swAlignment` property describes the intended alignment of AUTOSAR data objects within a memory section. 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 = 'El1';
swAlignmentValue = '32';
set(dataObj,[interfacePath dataElementName],'SwAlignment',swAlignmentValue);
```

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

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

`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`. 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 Release 4.x Data Types

In this section...

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 sections.

For more information about modeling R4.x data types, see “Release 4.x Data Types”.

Control Application Data Type Generation

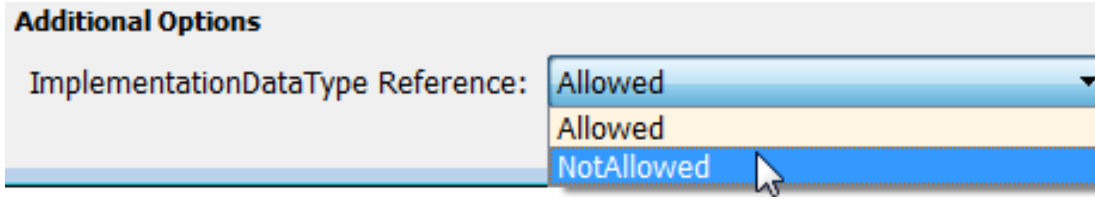
For AUTOSAR data types created in Simulink, by default, the software generates application base types only for fixed-point data types and enumerated date types with storage types. If you want to override the default behavior for generating application types, you can configure the `arxml` exporter to generate an application type, along with the implementation type and base type, for each exported AUTOSAR data type. Use the XML options parameter **ImplementationDataType Reference** (`XMLOptions` property `ImplementationDataTypeReference`), for which you can specify the following values:

- `Allowed` (default) — Allow direct reference of implementation types in the generated `arxml` code. If an application data type is not strictly required to describe an AUTOSAR data type, use an implementation data type reference.
- `NotAllowed` — Do not allow direct reference of implementation data types in the generated `arxml` code. Generate an application data type for each AUTOSAR data type.

To set the `ImplementationDataTypeReference` property in the MATLAB Command Window, use an AUTOSAR property `set` function call similar to the following:

```
open_system('rtwdemo_autosar_multirunnables');
arProps=autosar.api.getAUTOSARProperties('rtwdemo_autosar_multirunnables');
set(arProps,'XmlOptions','ImplementationTypeReference','NotAllowed');
get(arProps,'XmlOptions','ImplementationTypeReference')
```

To set the `ImplementationDataTypeReference` property in the Configure AUTOSAR Interface dialog box, select the AUTOSAR Properties Explorer and go to the **XML Options** view. 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` 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 properties function `addPackageableElement`. The following example specifies a custom data type mapping set package and name using MATLAB commands.

```
% Add a new data type mapping set
modelName = 'rtwdemo_autosar_multirunnables';
open_system(modelName);
propObj = autosar.api.getAUTOSARProperties(modelName);
newMappingSetPath = '/myPkg/mySubpkg/MyMappingSets';
newMappingSetName = 'MappingSetName';
newMappingSet = [newMappingSetPath '/' newMappingSetName];
addPackageableElement(propObj,'DataTypeMappingSet',newMappingSetPath,newMappingSetName);

% Configure the component behavior to use the new data type mapping set
swc = get(propObj,'XmlOptions','ComponentQualifiedName');
ib = get(propObj,swc,'Behavior','PathType','FullyQualified');
set(propObj,ib,'DataTypeMapping',newMappingSet);
```

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

% Build
rtwbuild(modelName);
```

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

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

Initialize Data with ApplicationValueSpecification

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

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

For AUTOSAR parameters typed by implementation data type, code generation exports `arxml` code that uses `NumericalValueSpecification`s and (for enumerated types) `TextValueSpecification`s to specify initial values. If initial values for parameters specify multiple values, generated code uses `ArrayValueSpecification`s.

Configure AUTOSAR CompuMethods

In this section...

CompuMethod Direction for Linear Functions

Embedded Coder software imports AUTOSAR computational methods (`CompuMethod`s) described in `arxml` code and preserves them across round-trips between an AUTOSAR authoring tool (AAT) and Simulink.

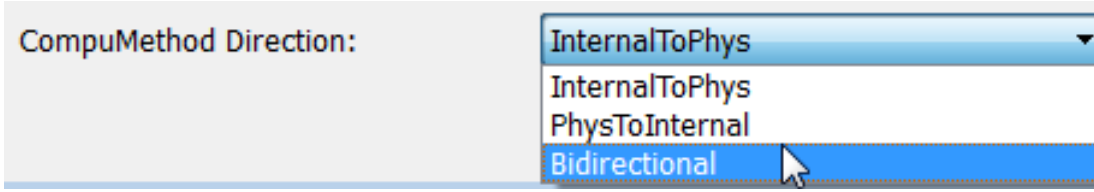
For designs originated in Simulink, you can control attributes for an exported `CompuMethod`, including the direction of `CompuMethod` conversion between internal and physical representations of a value. Using either the Configure AUTOSAR Interface dialog box or the AUTOSAR property `set` function, you 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 the AUTOSAR Properties Explorer and go to the **XML Options** view. Set the parameter **CompuMethod Direction** to the value that you want. Click **Apply**.



When you generate code for your model, the `CompuMethod`s in the exported `arxml` contain the requested directional sections. For example, here is a `CompuMethod` generated with the `CompuMethod` direction set to `Bidirectional`.

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

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

CompuMethod Unit References

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

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

```
<COMPU-METHOD UUID="...">
    <SHORT-NAME>COMPU_Boolean</SHORT-NAME>
    <CATEGORY>TEXTTABLE</CATEGORY>
    <UNIT-REF DEST="UNIT">/mymodel_pkg/mymodel_dt/NoUnit</UNIT-REF>
    ...
</COMPU-METHOD>
<UNIT UUID="...">
    <SHORT-NAME>NoUnit</SHORT-NAME>
    <FACTOR-SI-TO-UNIT>1</FACTOR-SI-TO-UNIT>
    <OFFSET-SI-TO-UNIT>0</OFFSET-SI-TO-UNIT>
</UNIT>
```

Providing a unit for each exported `CompuMethod` helps support measurement and calibration tool use of exported AUTOSAR data.

Rational Function CompuMethod for Dual-Scaled Parameter

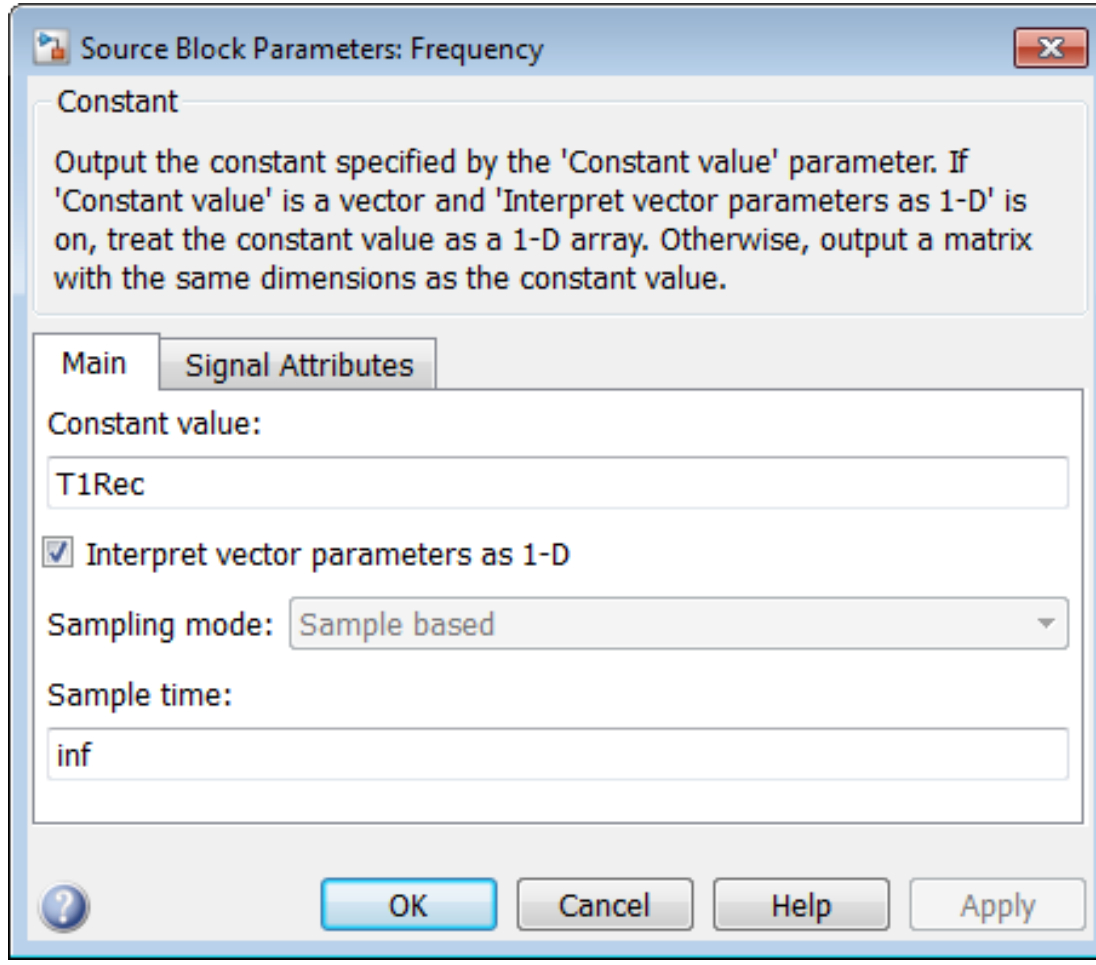
For an AUTOSAR dual-scaled parameter, which stores two scaled values of the same physical value, the software generates the `CompuMethod` category `RAT_FUNC`. The computation method can be a first-order rational function.

To configure and generate a dual-scaled parameter:

1. In the workspace, create an `AUTOSAR.DualScaledParameter` data object. For example:

   ```
   T1Rec = AUTOSAR.DualScaledParameter;
   ```

2. Open a model that is configured for AUTOSAR. Set up a Constant block to reference the dual-scaled parameter.



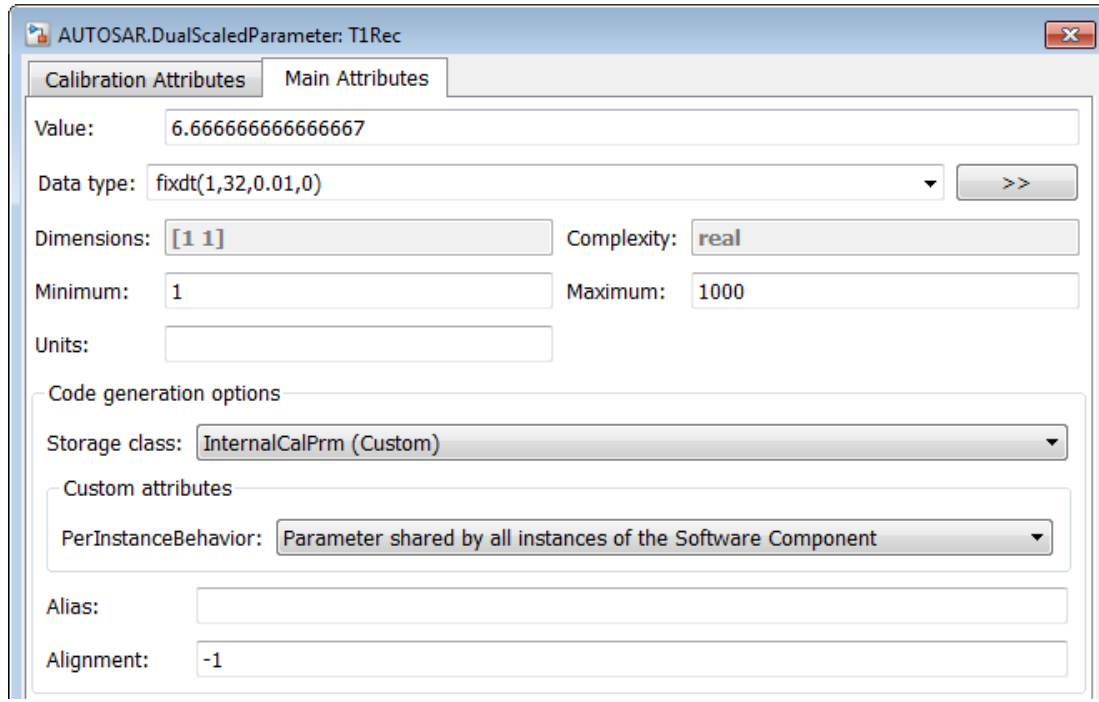
Connect the Constant block to a Simulink outport.

3 Configure the attributes of the dual-scaled parameter. To configure the parameter attributes, 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.

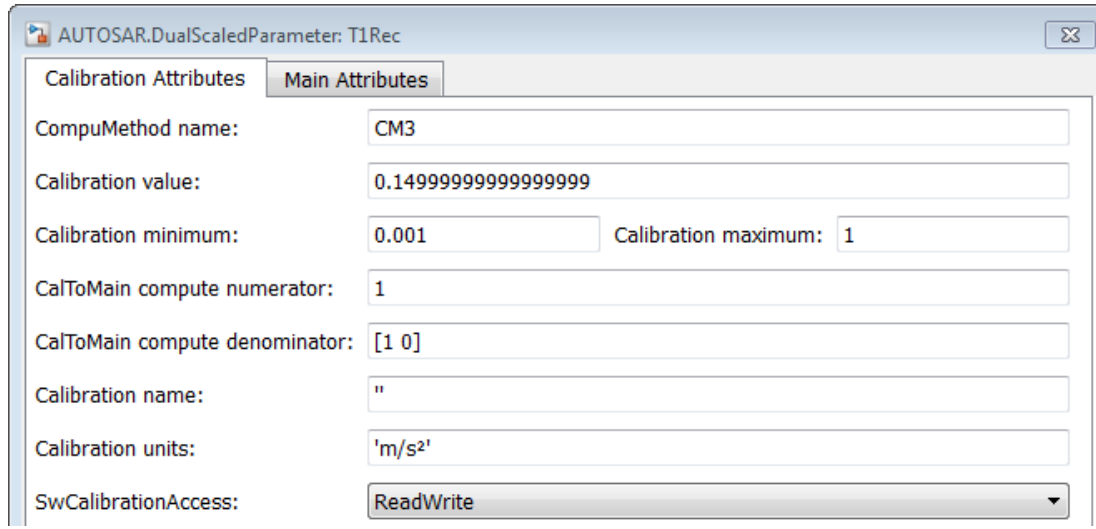
```
% Conversion from Time to Frequency
% F = 1/T
% In Other Words F = (0*T + 1)/(1*T+0);
```

```
% T1Rec = AUTOSAR.DualScaledParameter;
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/s²';
```

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



5 Here are the calibration attributes set by the MATLAB code. Beginning in R2014b, 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 Per-Instance Memory

You can model per-instance memory through the use of Data Store Memory blocks together with an `AUTOSAR.Signal` data object. For a detailed example, see `rtwdemo_autosar_PIM_script`. The following is an outline of the required steps:

1. In the base workspace, create an `AUTOSAR.Signal` object.
2. Open the data object and set the **Storage class** to `PerInstanceMemory (Custom)`. This selection enables custom attributes **needsNVRAMAccess** and **IsArTypedPerInstanceMemory**.
3. If you want to configure this per-instance memory to be a mirror block for an NVRAM block, select the **needsNVRAMAccess** option.
4. If you want to use AUTOSAR-typed per-instance memory, select the **IsArTypedPerInstanceMemory** option. Otherwise, the per-instance memory uses C types. AUTOSAR-typed per-instance memory (`arTypedPerInstanceMemory`) requires AUTOSAR schema version 4.0 or later.
5. Optionally, you can specify an **Initial value** for the global variable corresponding to per-instance memory.
6. Go to the **Additional attributes** tab, and use the **SwCalibrationAccess** parameter to configure software calibration access to the data as `NotAccessible`, `ReadOnly`, or `ReadWrite`
7. Create a Data Store Memory block that references the `AUTOSAR.Signal` object. See "Data Store Memory" in the Simulink Reference documentation.

Note: The software does *not* support per-instance memory modeling within a referenced model.

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 to indicate

that the per-instance memory is a RAM mirror block and must be serviced by the NvM manager module.

Create an AUTOSAR.Signal Object

To create an AUTOSAR.Signal object in the base workspace:

1. Open the Model Explorer (**Ctrl+H**).
2. In the **Model Hierarchy** view, under **Simulink Root**, select **Base Workspace**.
3. Select **Add** > **Add Custom**. The Model Explorer – Select Object dialog box opens.
4. Specify a value in the **Object Name(s)** field, for example, nvmImplicitRW.
5. From the **Object class** drop-down list, select AUTOSAR.Signal.
6. Click **OK**. A new object nvmImplicitRW appears in the base workspace.

Alternatively, you can use a MATLAB command such as

```
nvmImplicitRW = AUTOSAR.Signal
```

and double-click the object to open its dialog box.

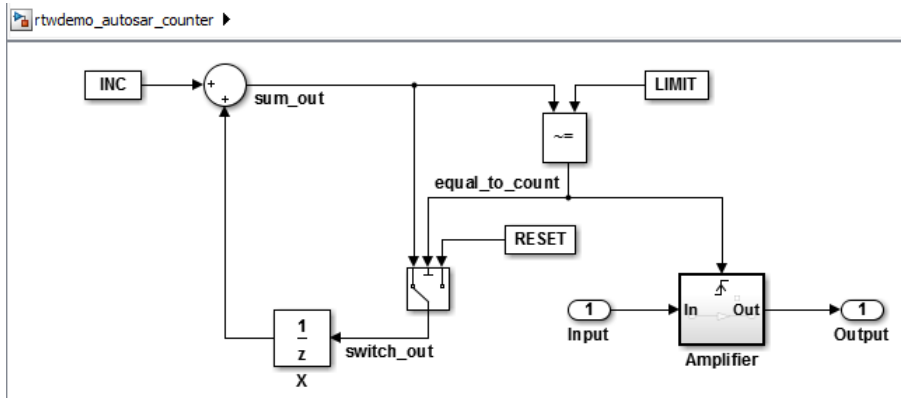
Configure AUTOSAR Static or Constant Memory

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:

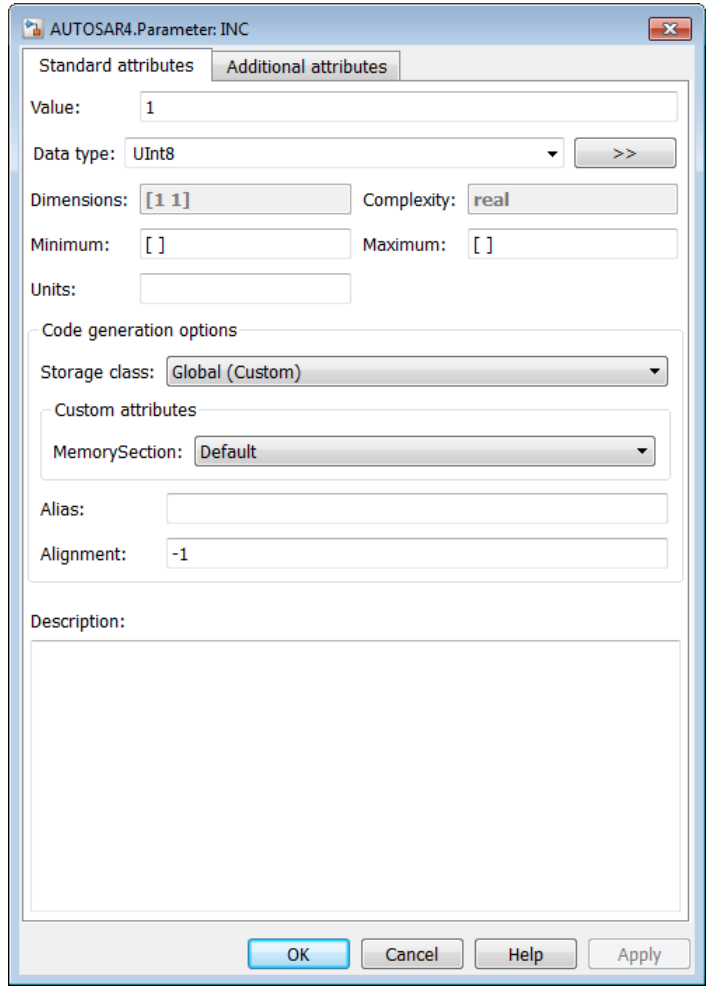
1. Open the example model rtwdemo_autosar_counter.



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. Open the `INC` data object, for example, by double-clicking it in the workspace. Examine the settings in the Properties dialog box.



Optionally, select the **Additional attribute tab** and use the **SwCalibrationAccess** parameter to configure software calibration access to the data. For more information, see "Configure AUTOSAR Data for Measurement and Calibration" on page 4-80. 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`:

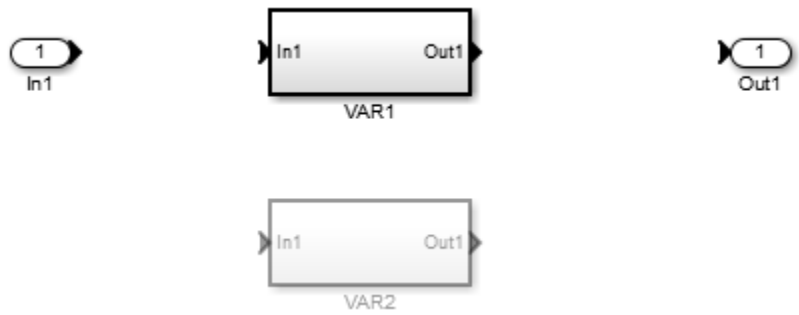
- `INC` now appears in a constant memory specification, rather than a calibration parameter.
- `sum_out` now appears in a static memory specification (previously was absent).

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.

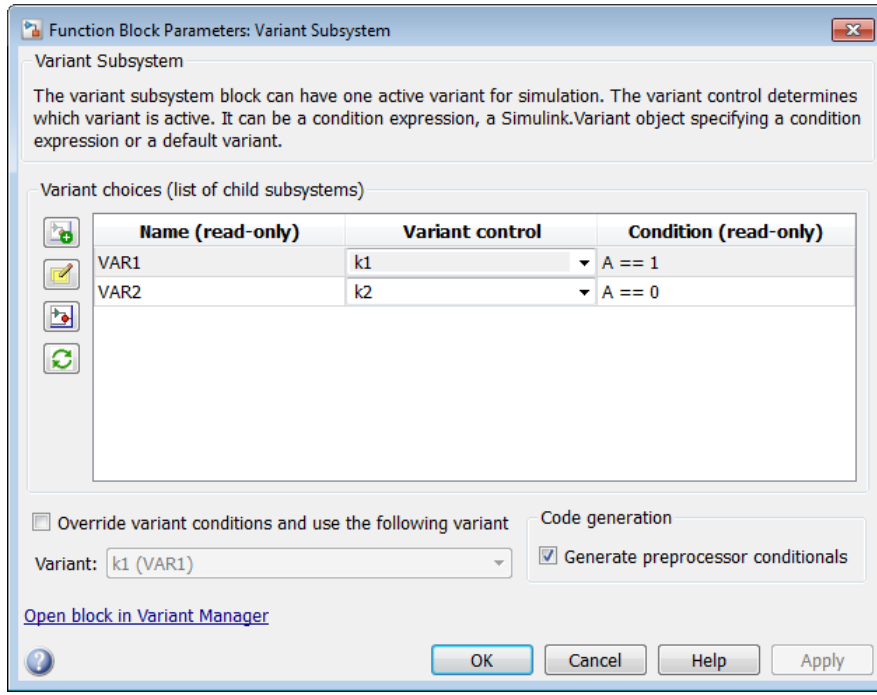
For example, the contents of a Variant Subsystem block are shown below. The variant choices are subsystems VAR1 and VAR2, The blocks are not connected at this level, because connectivity is automatically determined during simulation, based on the active variant.



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 (Custom)`, 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`.

Their relationship can be viewed 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 AUTOSAR Package Structure

In this section...

AUTOSAR Package Structure

In Simulink, you can configure the package structure into which AUTOSAR elements are exported. For example, you can configure an AUTOSAR package structure to:

- Conform to an externally-defined AUTOSAR package structure.
- Establish a namespace for elements in a package.
- Provide a basis for relative references to elements.

The package structure you define is preserved and exported in `arxml`. In general, AUTOSAR packages and their elements are preserved across round-trips between an AUTOSAR authoring tool (AAT) and Simulink

---

Note: The `arxml` exporter creates a configured package only if the model contains an element of the given category. For example, a software address method package is exported only if a software address method element is exported.

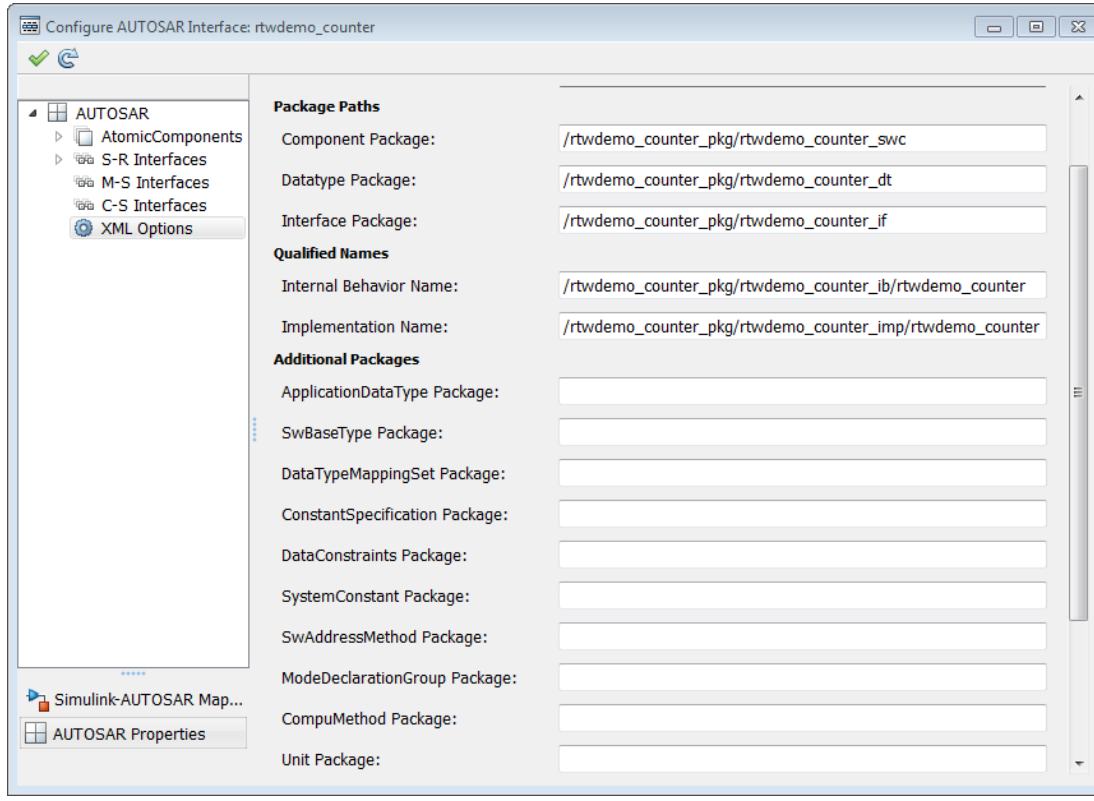
AUTOSAR Package Properties

You can use AUTOSAR XML options (`XmlOptions` properties) to define AUTOSAR packages for the following categories of AUTOSAR elements:

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

- 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)

If your AUTOSAR software component originated in Simulink, AUTOSAR configuration creation sets the initial values of only a subset of the AUTOSAR packaging properties. The following figure shows the **XML Options** view of the Configure AUTOSAR Interface dialog box after initial component creation of an AUTOSAR component for the example model `rtwdemo_counter`.



You can populate the package fields with paths, or leave them empty. If you leave them empty, the `arxml` exporter uses internal rules to calculate the package path. The application of internal rules is backward-compatible with earlier releases. The following table lists the XML option packaging properties with their rule-based default package paths.

Property Name	Package Paths Based on Internal Rules
`ComponentQualifiedName`	*`modelname`*`_pkg/`*`modelname`*`_swc/`*`modelname`* (dialog box displays the component path without the shortname)
`DataTypePackage`	*`modelname`*`_pkg/`*`modelname`*`_dt`
`InterfacePackage`	*`modelname`*`_pkg/`*`modelname`*`_if`

Property Name	Package Paths Based on Internal Rules
InternalBehaviorQualifiedName	*modelname*_pkg/*modelname*_ib/*modelname*
ImplementationQualifiedName	*modelname*_pkg/*modelname*_imp/*modelname*
ApplicationDataTypePackage	*DataTypePackage*/ApplDataTypes
SwBaseTypePackage	*DataTypePackage*/SwBaseTypes
DataTypeMappingPackage	*DataTypePackage*/DataTypeMappings
ConstantSpecificationPackage	*DataTypePackage*/Ground
DataConstraintPackage	*DataTypePackage*/ApplDataTypes/DataConstrs
SystemConstantPackage	*DataTypePackage*/SystemConstants
SwAddressMethodPackage	*DataTypePackage*/SwAddrMethods
ModeDeclarationGroupPackage	*DataTypePackage*
CompuMethodPackage	*DataTypePackage*
UnitPackage	*DataTypePackage*

To set a packaging property from the MATLAB Command Window or 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')
```

Note: You cannot configure the data constraints package for implementation data types created by the `arxml` exporter. The exporter determines the package path based on internal rules.

If your AUTOSAR software component originated in an AUTOSAR authoring tool (AAT), for most categories, the `arxml` importer preserves package-element relationships in the `arxml`. Also, the importer populates the XML option packaging properties. In cases where the imported `arxml` does not use a category-based mechanism for assigning AUTOSAR elements to AUTOSAR packages, the importer uses a heuristic to decide the best package for a given category.

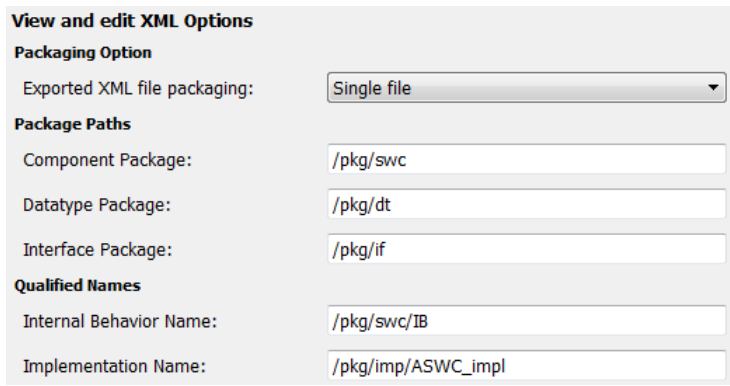
Note: The `arxml` importer does not preserve packages for software base types, and does not populate the software base type XML option package path. You must set the package path in the XML option properties.

For an example of configuring AUTOSAR package structure for export using the AUTOSAR Properties Explorer, see "Configure and Export AUTOSAR Packages" on page 4-112.

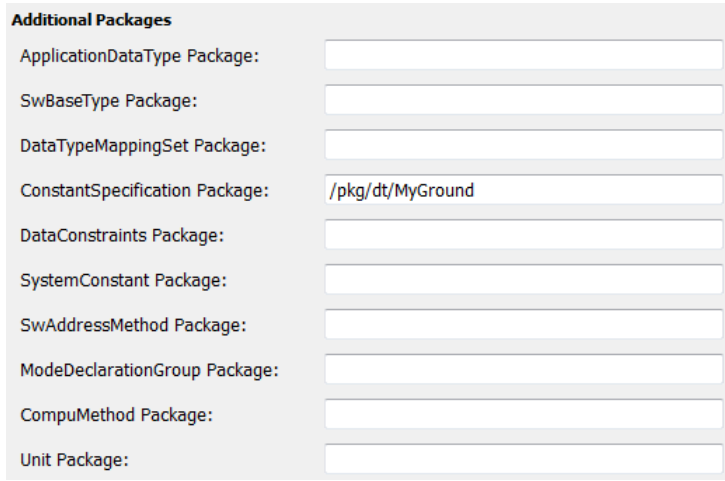
Configure and Export AUTOSAR Packages

To configure the AUTOSAR packages structure to export for your model:

1. Open a model that is configured for AUTOSAR. This example uses the example model rtwdemo_autosar_multirunnables.
2. Open the Configure AUTOSAR Interface dialog box, select the AUTOSAR Properties Explorer, and go to the **XML Options** view. Initially, only the first five AUTOSAR package parameters are configured:



3. Configure packages for one or more AUTOSAR elements that your model exports to `arxml`. The example model exports a large number of AUTOSAR constant specifications, so the example sets the **ConstantSpecification Package** parameter to `/pkg/dt/MyGround`. After entering package paths, click **Apply**.



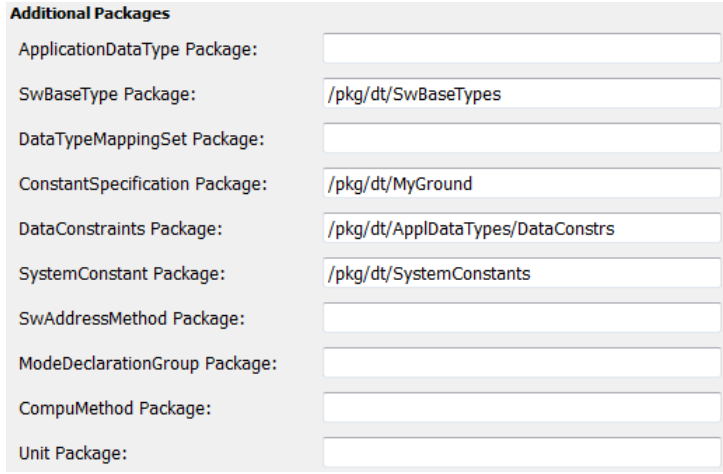
4 Validate the model configuration by clicking the Validate icon in the upper left corner of the dialog box.

5 Configure the model for code generation. The example model is configured to generate code only and to generate an HTML report. Generate code for the model.

6 Open the generated file *`modelname`*`.arxml` and search for the packages you configured. For the example model, searching `rtwdemo_autosar_multirunnables.arxml` for the string `'MyGround'` finds the package and many references to it, of which a sample appears below.

```
<AR-PACKAGES>
    <AR-PACKAGE UUID="392e8675-ce37-5b68-8065-e9b8461853a2">
        <SHORT-NAME>MyGround</SHORT-NAME>
        <ELEMENTS>
            <CONSTANT-SPECIFICATION UUID="355d4239-1e22-5b08-818d-9d11713ca564">
                <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="ef6ac993-eb04-539d-e95c-427959dcd83d">
                <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/dt/MyGround/DefaultInitValue_Double</CONSTANT-REF>
                            </CONSTANT-REFERENCE>
```

7 In the **XML Options** view of the AUTOSAR Properties Explorer, other package paths might have been filled in by the `arxml` exporter based on internal rules. For the example model, the exporter provided three additional package paths.



Note: The `arxml` exporter creates a configured package only if the model contains an element of the given category. For example, a software address method package is exported only if a software address method element is exported.

Configure and Map AUTOSAR Component Programmatically

In this section...

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 properties 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".

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
      - Runnables
      - IRV
  - S-R Interfaces
    - *`SRInterface1`*
      - DataElements
  - M-S Interfaces
    - *`MSInterface1`*
  - C-S Interfaces
    - *`CSInterface1`*
      - Operations
        - *`operation1`*
          - Arguments
  - XML Options

Properties of AUTOSAR Elements

The following table lists properties that are associated with AUTOSAR elements.

AUTOSAR Element Class	Properties
`AtomicComponent`	• `ReceiverPorts` (add/delete) • `SenderPorts` (add/delete) • `SenderReceiverPorts` (add/delete) • `ModeReceiverPorts` (add/delete) • `ClientPorts` (add/delete) • `ServerPorts` (add/delete) • `Behavior` (add/delete) • `Kind` • `Name`
`ApplicationComponentBehavior`	• `Runnables` (add/delete) • `Events` (add/delete) • `PIM` (add/delete) • `IRV` (add/delete) • `Parameters` (add/delete) • `DataTypeMapping` • `Name`
`DataSenderPort`	• `Interface` • `Name`
`DataReceiverPort`	• `Interface` • `Name`
`Runnable`	• `symbol` • `canBeInvokedConcurrently` • `SwAddrMethod` • `Name`
`TimingEvent`	• `Period` • `StartOnEvent` • `DisabledMode` • `Name`

AUTOSAR Element Class	Properties
`DataReceivedEvent`	• `Trigger` • `StartOnEvent` • `DisabledMode` • `Name`
`ModeSwitchEvent`	• `Trigger` • `Activation` • `StartOnEvent` • `DisabledMode` • `Name`
`OperationInvokedEvent`	• `Trigger` • `StartOnEvent` • `DisabledMode` • `Name`
`InitEvent`	• `StartOnEvent` • `Name`
`IrvData`	• `Type` • `SwAddrMethod` • `SwCalibrationAccess` • `SwAlignment` • `Name`
`SenderReceiverInterface`	• `DataElements` (add/delete) • `ModeGroup` (add/delete) • `IsService` • `Name`

AUTOSAR Element Class	Properties
`FlowData`	• `Type` • `SwAddrMethod` • `SwCalibrationAccess` • `SwAlignment` • `Name`
`ModeSwitchInterface`	• `ModeGroup` (add/delete) • `IsService` • `Name`
`ModeDeclarationGroupElement`	• `ModeGroup` • `SwCalibrationAccess` • `Name`
`ClientServerInterface`	• `Operations` (add/delete) • `IsService` • `Name`

Specify AUTOSAR Element Location

The AUTOSAR properties 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
```

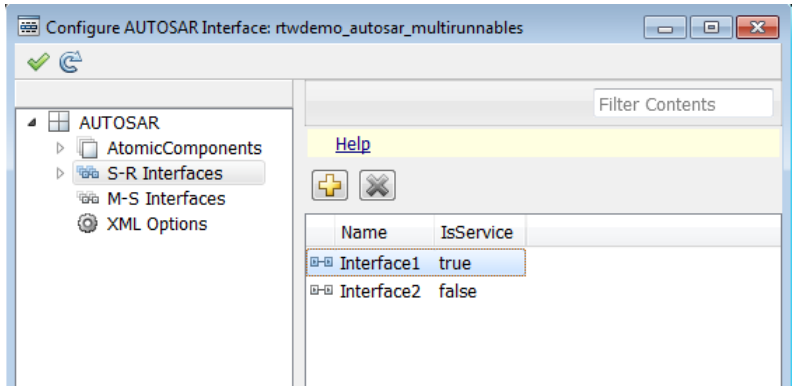
For AUTOSAR properties functions other than `addPackageableElement`, you can specify a partially-qualified path that does not include the package hierarchy, for example:

```
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.

```
>> propObj = autosar.api.getAUTOSARProperties('rtwdemo_autosar_multirunnables');
```

```
>> set(propObj,'Interface1','IsService',true);
```



If you added a Sender-Receiver Interface to a component package, you would specify a fully qualified path, for example:

```
>> propObj = autosar.api.getAUTOSARProperties('rtwdemo_autosar_multirunnables');
>> addPackageableElement(propObj,'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`.

Limitations and Tips

In this section...

Source of Initial Output Value for Function-Call Subsystem Outport

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

AUTOSAR Client Block in Referenced Model

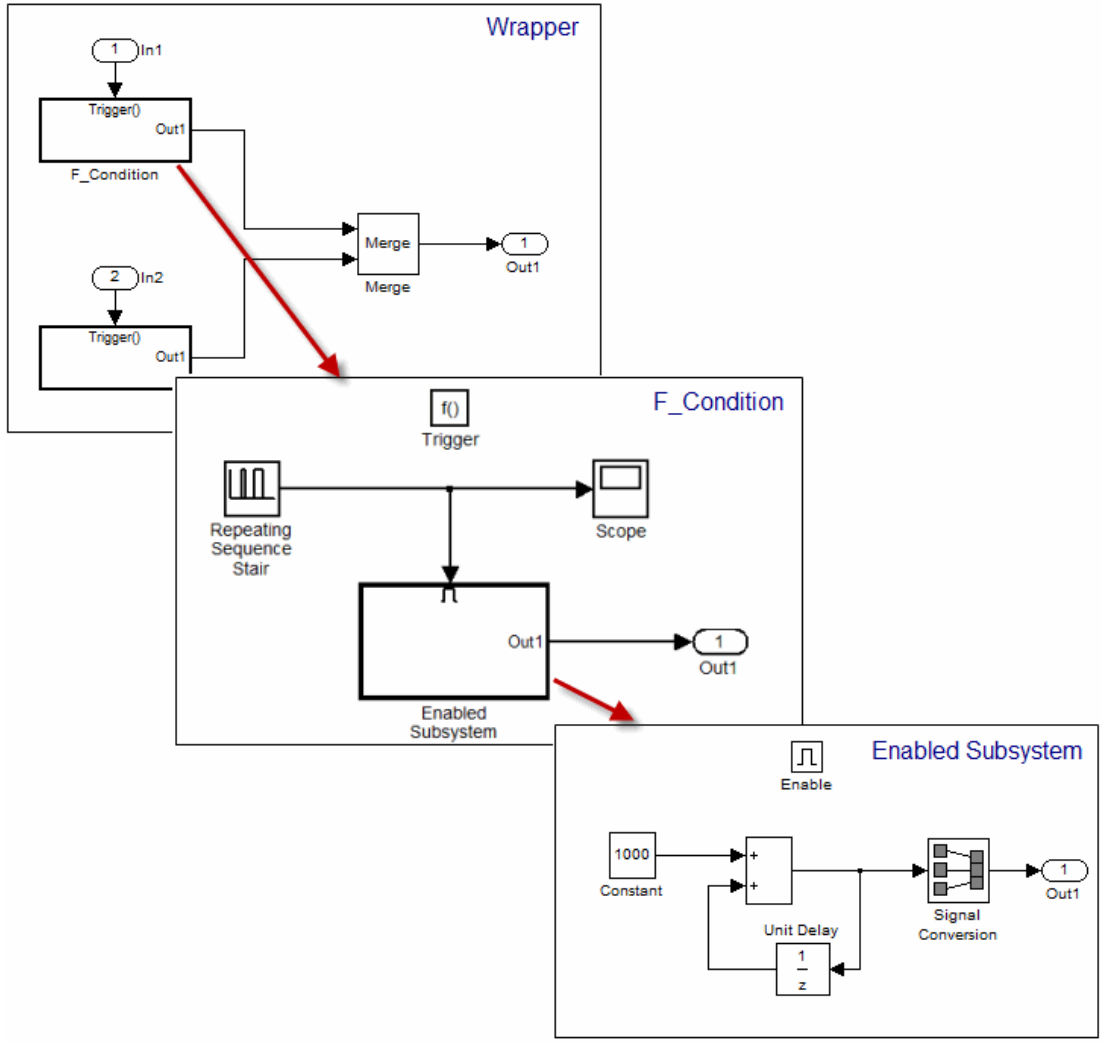
The software does not support the use of an AUTOSAR client block, such as Function Caller or Invoke AUTOSAR Server Operation, in a referenced model.

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 outport or one or more subsystems.
- If the output signal of the Merge block is connected to the inputs of one or more subsystems, assign the same signal name to the Merge block's output and inputs.

In addition, the signal from the function-call subsystem outport that enters a Merge block must not be conditionally computed. Consider the following example.



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

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

AUTOSAR Code Generation

Export AUTOSAR Component XML and C Code

In this section...

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.

Value	Schema Version
`4.1`	4.1.1
`4.0` (default)	4.0.3
`3.2`	3.2.1
`3.1`	3.1.4
`3.0`	3.0.2
`2.1`	2.1 (XSD rev 0017)

Note:

- 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.

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.

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`.

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 Packages for...
`Single file`	*`modelname`*`.arxml`	All elements.
`Modular`	*`modelname`*`_component.arxml`	Software components, including calibration components.

Exported XML File Packaging Value	Exported File Name	By Default Contains Packages for...
		This is the main `arxml` file exported for the Simulink model. In addition to component packages, the file includes elements for which packages have not been configured using the XML Options View of AUTOSAR Properties Explorer.
	modelname`_datatype.arxml`	Data types and related elements, including • 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) **Note:** Elements for which packages are not configured are placed in the main `arxml` file, *modelname*`_component.arxml`. For more information about configuring packages, see XML Options View of AUTOSAR Properties Explorer and "Configure AUTOSAR Package Structure".
	modelname`_implementation.arxml`	Software component implementation.
	modelname`_interface.arxml`	Interfaces, including S-R interfaces, C-S interfaces, and M-S interfaces.

Exported XML File Packaging Value	Exported File Name	By Default Contains Packages for...
	*modelname*_behavior.arxml	Software component internal behavior (generated only for schema 3.x or earlier).

You can merge the software component information back into an AUTOSAR authoring tool. This software component information is partitioned into separate files to facilitate merging. The partitioning attempts to minimize the number of merges that you must do. You do not need to merge the data type file into the authoring tool because data types are usually defined early in the design process. You must, however, merge the internal behavior file because this information is part of the model implementation.

To help support the round trip of AUTOSAR elements between an AAT and the Simulink model-based design environment, 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 3-9.

For examples of how to generate AUTOSAR-compliant code and export AUTOSAR Software Component description XML files from a Simulink model, see the following examples. (Go to Simulink Coder Examples in the help browser or enter the command `rtwdemos`.)

- AUTOSAR Code Generation
- AUTOSAR Code Generation for Multiple Runnable Entities

Code Replacement for AUTOSAR

In this section...

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, `Mfl.h` or `Mfx.h`)
- You have a build harness set up that can compile and link the AUTOSAR library with the generated code. For more information about building code for AUTOSAR, see "AUTOSAR Code Generation".

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 `RTW.viewTfl` 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.

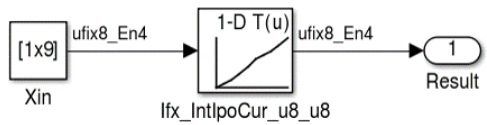
For more information on code replacement and code replacement libraries, see "What Is Code Replacement?" and "Code Replacement Libraries".

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()));

  /* Outport: '<Root>/Result' */
  look01_Y.Result[iU] = rtb_Ifx_IntIpoCur_u8_u8;
}
```

Required Algorithm Property Settings for IFL/IFX Function and Block Mappings

IFL/IFX Function and Block Mapping	Algorithm Property Parameters	Value
`Ifl_DPSearch` Prelookup	**Extrapolation method** `ExtrapMethod`	`Clip`
	Index search method `IndexSearchMethod`	`Linear search` or `Binary search`
	Use last breakpoint for input at or above upper limit `UseLastBreakPoint`	On
	Remove protection against out-of-range input `RemoveProtectionInput`	Off
	Integer rounding mode `RndMeth`	`Round` or `Zero`
`Ifl_IpoCur` Interpolation Using Prelookup	**Interpolation method** `InterpMethod`	`Linear`

IFL/IFX Function and Block Mapping	Algorithm Property Parameters	Value
	Extrapolation method `ExtrapMethod`	`Clip`
	Valid index input may reach last index `ValidIndexMayReachLast`	On
	Remove protection against out-of-range input `RemoveProtectionInput`	Off
	Integer rounding mode `RndMeth`	`Round` or `Zero`
`Ifl_IpoMap` Interpolation Using Prelookup	**Interpolation method** `InterpMethod`	`Linear`
	Extrapolation method `ExtrapMethod`	`Clip`
	Valid index input may reach last index `ValidIndexMayReachLast`	On
	Remove protection against out-of-range input `RemoveProtectionInput`	Off
	Integer rounding mode `RndMeth`	`Round` or `Zero`
`Ifl_IntIpoCur` n-D Lookup Table	**Interpolation method** `InterpMethod`	`Linear`

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

IFL/IFX Function and Block Mapping	Algorithm Property Parameters	Value
	Integer rounding mode `RndMeth`	`Round` or `Zero`
`Ifx_DPSearch` Prelookup	**Extrapolation method** `ExtrapMethod`	`Clip`
	Index search method `IndexSearchMethod`	`Linear search` or `Binary search`
	Use last breakpoint for input at or above upper limit `UseLastBreakPoint`	On
	Remove protection against out-of-range input `RemoveProtectionInput`	Off
	Integer rounding mode `RndMeth`	`Round` or `Zero`
`Ifx_IpoCur` Interpolation Using Prelookup	**Interpolation method** `InterpMethod`	`Linear`
	Extrapolation method `ExtrapMethod`	`Clip`
	Valid index input may reach last index `ValidIndexMayReachLast`	On
	Remove protection against out-of-range input `RemoveProtectionInput`	Off

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

IFL/IFX Function and Block Mapping	Algorithm Property Parameters	Value
	Integer rounding mode `RndMeth`	`Round` or `Zero`
`Ifx_LkUpMap` Interpolation Using Prelookup	**Interpolation method** `InterpMethod`	`Nearest`
	Extrapolation method `ExtrapMethod`	`Clip`
	Remove protection against out-of-range input `RemoveProtectionInput`	Off
	Integer rounding mode `RndMeth`	`Round` or `Zero`
	ValidIndexMayReachLast `ValidIndexMayReachLast`	On
`Ifx_LkUpBaseMap` Interpolation Using Prelookup	**Interpolation method** `InterpMethod`	`Flat`
	Extrapolation method `ExtrapMethod`	`Clip`
	Remove protection against out-of-range input `RemoveProtectionInput`	Off
	Integer rounding mode `RndMeth`	`Round` or `Zero`
	ValidIndexMayReachLast `ValidIndexMayReachLast`	On

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

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

IFL/IFX Function and Block Mapping	Algorithm Property Parameters	Value
	Model configuration parameter **Optimization > Signals and Parameters > Inline Parameters** `InlineParams`	On
	Breakpoint data should match power 2 spacing.	
`Ifx_IntLkUpFixCur` n-D Lookup Table	**Interpolation method** `InterpMethod`	`Flat`
	Extrapolation method `ExtrapMethod`	`Clip`
	Index search method `IndexSearchMethod`	`Evenly spaced points`
	Remove protection against out-of-range input `RemoveProtectionInput`	Off
	Integer rounding mode `RndMeth`	`Round` or `Zero`
	Model configuration parameter **Optimization > Signals and Parameters > Inline Parameters** `InlineParams`	On
	Breakpoint data must match power 2 spacing.	
`Ifx_IntIpoFixICur` n-D Lookup Table	**Interpolation method** `InterpMethod`	`Linear`

IFL/IFX Function and Block Mapping	Algorithm Property Parameters	Value
	Extrapolation method `ExtrapMethod`	`Clip`
	Index search method `IndexSearchMethod`	`Evenly spaced points`
	Use last table value for inputs at or above last `UseLastTableValue`	On
	Remove protection against out-of-range input `RemoveProtectionInput`	Off
	Integer rounding mode `RndMeth`	`Round` or `Zero`
	Breakpoint data must not match power 2 spacing.	
`Ifx_IntLkUpFixICur` n-D Lookup Table	**Interpolation method** `InterpMethod`	`Flat`
	Extrapolation method `ExtrapMethod`	`Clip`
	Index search method `IndexSearchMethod`	`Evenly spaced points`
	Remove protection against out-of-range input `RemoveProtectionInput`	Off
	Integer rounding mode `RndMeth`	`Round` or `Zero`

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

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

IFL/IFX Function and Block Mapping	Algorithm Property Parameters	Value
	Extrapolation method `ExtrapMethod`	`Clip`
	Index search method `IndexSearchMethod`	`Evenly spaced points`
	Use last table value for inputs at or above last `UseLastTableValue`	On
	Remove protection against out-of-range input `RemoveProtectionInput`	Off
	Integer rounding mode `RndMeth`	`Round` or `Zero`
	Model configuration parameter **Optimization > Signals and Parameters > Inline Parameters** `InlineParams`	On
	Breakpoint data must match power 2 spacing.	
`Ifx_IntLkUpFixMap` n-D Lookup Table	**Interpolation method** `InterpMethod`	`Nearest`
	Extrapolation method `ExtrapMethod`	`Clip`
	Index search method `IndexSearchMethod`	`Evenly spaced points`

IFL/IFX Function and Block Mapping	Algorithm Property Parameters	Value
	Remove protection against out-of-range input `RemoveProtectionInput`	Off
	Integer rounding mode `RndMeth`	`Round` or `Zero`
	Use last table value for inputs at or above last `UseLastTableValue`	On
	Model configuration parameter **Optimization > Signals and Parameters > Inline Parameters** `InlineParams`	On
	Breakpoint data must match power 2 spacing.	
`Ifx_IntLkUpFixBaseMap` n-D Lookup Table	**Interpolation method** `InterpMethod`	`Flat`
	Extrapolation method `ExtrapMethod`	`Clip`
	Index search method `IndexSearchMethod`	`Evenly spaced points`
	Remove protection against out-of-range input `RemoveProtectionInput`	Off
	Integer rounding mode `RndMeth`	`Round` or `Zero`

IFL/IFX Function and Block Mapping	Algorithm Property Parameters	Value
	Use last table value for inputs at or above last `UseLastTableValue`	On
	Model configuration parameter **Optimization > Signals and Parameters > Inline Parameters** `InlineParams`	On
	Breakpoint data must match power 2 spacing.	
`Ifx_IntIpoFixIMap` n-D Lookup Table	**Interpolation method** `InterpMethod`	`Linear`
	Extrapolation method `ExtrapMethod`	`Linear`
	Index search method `IndexSearchMethod`	`Evenly spaced points`
	Use last table value for inputs at or above last `UseLastTableValue`	On
	Remove protection against out-of-range input `RemoveProtectionInput`	Off
	Integer rounding mode `RndMeth`	`Round` or `Zero`
	Breakpoint data must not match power 2 spacing.	

IFL/IFX Function and Block Mapping	Algorithm Property Parameters	Value
`Ifx_IntLkUpFixIMap` n-D Lookup Table	**Interpolation method** `InterpMethod`	`Nearest`
	Extrapolation method `ExtrapMethod`	`Clip`
	Index search method `IndexSearchMethod`	`Evenly spaced points`
	Remove protection against out-of-range input `RemoveProtectionInput`	Off
	Integer rounding mode `RndMeth`	`Round` or `Zero`
	Use last table value for inputs at or above last `UseLastTableValue`	On
	Breakpoint data must not match power 2 spacing.	
`Ifx_IntLkUpFixIBaseMap` n-D Lookup Table	**Interpolation method** `InterpMethod`	`Flat`
	Extrapolation method `ExtrapMethod`	`Clip`
	Index search method `IndexSearchMethod`	`Evenly spaced points`
	Remove protection against out-of-range input `RemoveProtectionInput`	Off

IFL/IFX Function and Block Mapping	Algorithm Property Parameters	Value
	Integer rounding mode `RndMeth`	`Round` or `Zero`
	Use last table value for inputs at or above last `UseLastTableValue`	On
	Breakpoint data must not match power 2 spacing.	

Code Replacement Checks for AUTOSAR Lookup Table Functions

The following checks occur during the code replacement match process for AUTOSAR lookup table functions:

Function Type	Match Process Checks Whether
n-D lookup	• Input and corresponding breakpoint arguments have the same data type. • Output and the table data argument have the same data type.
Interpolation with 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

In this section...

Overview

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.

Use the SIL and PIL Simulation Modes

You can run a top model or Model block that is configured for the AUTOSAR target (`autosar.tlc`) using the `Software-in-the-Loop (SIL)` and `Processor-in-the-Loop (PIL)` simulation modes.

For more information, see "Top-Model SIL or PIL Simulation" and "Model Block SIL or PIL Simulation".

For limitations, see "Limitations and Tips" on page 5-29.

Use a SIL or PIL Block for AUTOSAR Verification

To verify source code, you create a SIL block, which wraps the generated code in an S-function. The AUTOSAR target automatically configures the generated S-function to route simulation data using AUTOSAR run-time environment (RTE) API calls.

To verify the behavior of production-intent object code, you create a PIL block. You must provide an implementation of the target connectivity API for this block.

To carry out a verification using a SIL or PIL block:

1. In the Configuration Parameters dialog box, select the **Code Generation** pane and clear **Generate code only**. If you select **Generate code only**, the software does not create a SIL or PIL block.

2. Select the **Code Generation > Verification** pane.
3. From the **Create block** drop-down list, select either `SIL` or `PIL`. Click **OK**.
4. To create your SIL or PIL block, generate code in the usual way. See "Export AUTOSAR Software Component" on page 5-5 and "Configure AUTOSAR Multiple Runnables" on page 4-42.
5. Once the SIL or PIL block is built, replace the existing component in your model with the new block.
6. Simulate the model and check the output to verify that the code produces the same data as the original subsystem.

Note: The software does not propagate non-zero outport initialization inside an AUTOSAR model to the outports (via the RTE) until the step function executes. When you run the generated code in a SIL simulation, you do not see the outport initialization until the SIL wrapper executes the step function for the first time.

For more information about configuring and running simulations with SIL or PIL blocks, see "Use a SIL or PIL Block".

For limitations, see "Limitations and Tips" on page 5-29.

Limitations and Tips

In this section...

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 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.

AUTOSAR Top Model SIL and PIL

Through signal logging, you can configure your top model to log invariant output signals. However, the software will log these invariant signals as periodically sampled data.

AUTOSAR Model Block SIL and PIL

The software supports testing of AUTOSAR components that are modeled as model reference components. These model reference components are implemented as standard model reference Simulink Coder targets and do not contain special AUTOSAR behavior.

AUTOSAR SIL and PIL Block

- The software does not support SIL or PIL block verification for code generated from models that have Simulink Function and Function Caller blocks, for example, in an AUTOSAR client-server configuration. Use the Model block SIL/PIL approach, with the **Code under test** block parameter set to `Top model`. For more information, see "SIL and PIL Simulation Support and Limitations".

- Consider a runnable (function-call subsystem) in a model, which contains a Stateflow chart using absolute-time temporal logic. Replace the runnable with a SIL block and run a simulation with the model. If the SIL block is executed conditionally in the model, then the results of the SIL simulation differ from the results of the Normal mode simulation.
- If your model has function-call subsystems and you export a runnable that has context-dependent inputs (for example, feedback signals), then the results of a SIL/PIL simulation with the generated code may not match the results of the Normal mode simulation of your model. See “Exported Functions in Feedback Loops”.

Functions — Alphabetical List

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.

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"
- "Configure the AUTOSAR Interface"

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” 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.

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”
- “Configure the AUTOSAR Interface”

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” 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.

Construction

arxml.importer	Construct `arxml.importer` object

Methods

createCalibrationComponentObjects	Create Simulink calibration objects from AUTOSAR calibration component
createComponentAsModel	Create AUTOSAR atomic software component as Simulink model
getApplicationComponentNames	Get list of application software component names
getCalibrationComponentNames	Get calibration component names
getClientServerInterfaceNames	Get list of client-server interfaces

getComponentNames	Get application and sensor/actuator software component names
getDependencies	Get list of XML dependency files
getFile	Return software component XML file name
getSensorActuatorComponentNames	Get list of sensor/actuator software component names
setDependencies	Set XML file dependencies
setFile	Set software component XML file name
updateModel	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.

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”

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.

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”
- “Configure the AUTOSAR Interface”

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.

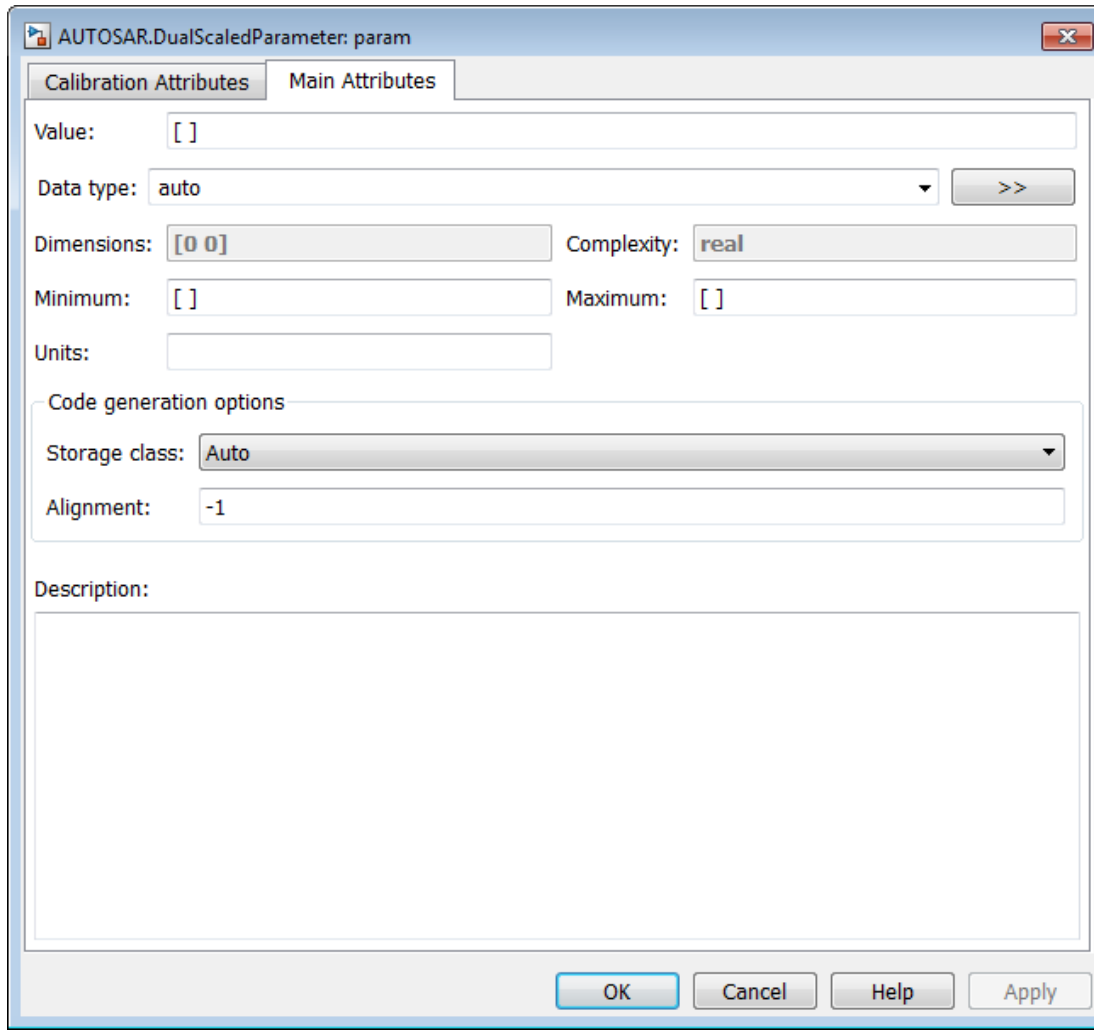
If you provide the calibration minimum and maximum values, the parameter computes minimum and maximum values of the main value. Simulink performs range checking of parameter values. The software alerts you when the parameter object value lies outside a range that corresponds to its specified minimum 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`.

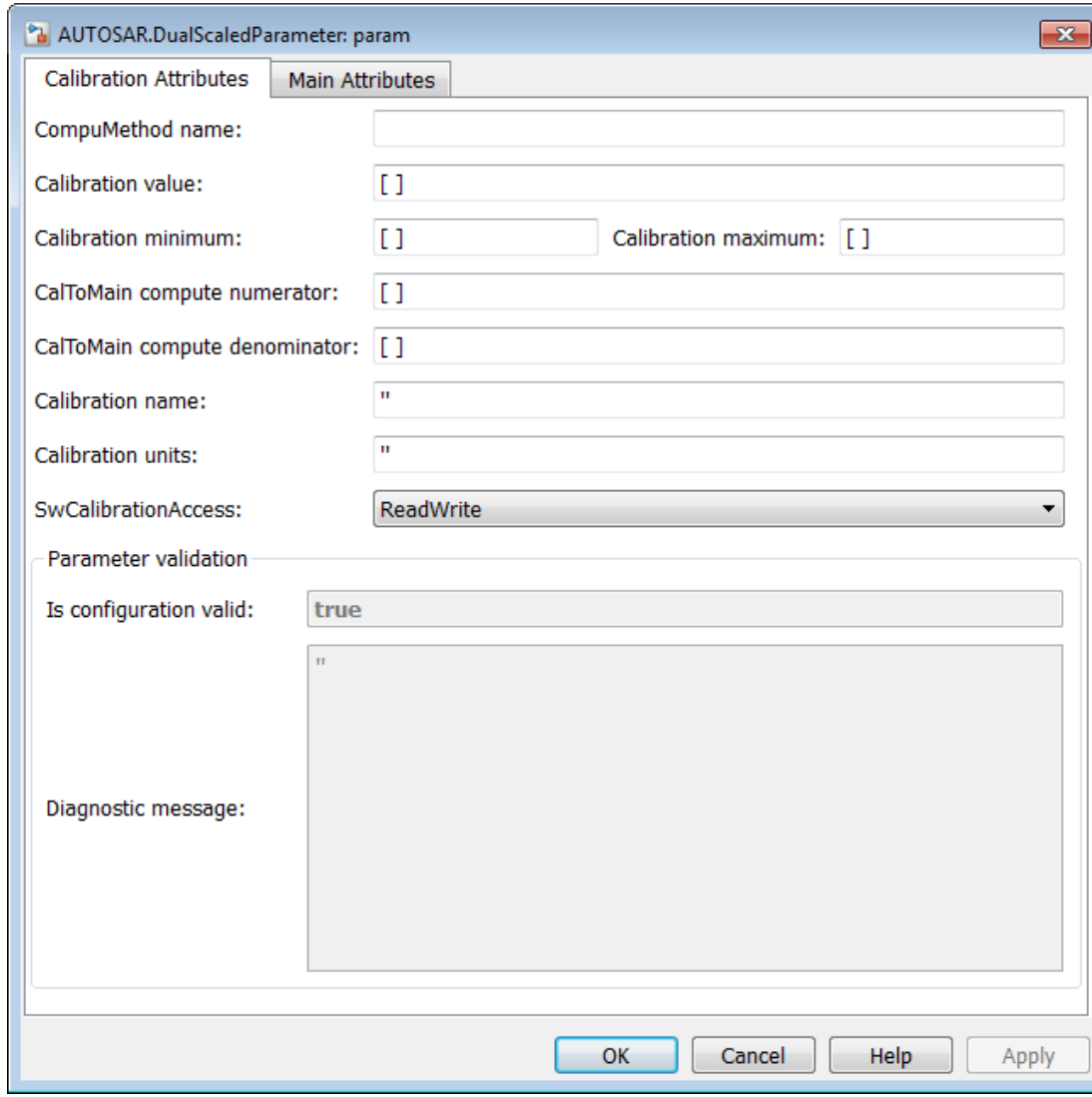
Property Dialog Box

Main Attributes Tab



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

Calibration Attributes Tab



CompuMethod name

Name of the AUTOSAR computation method (`CompuMethod`) to generate for the parameter in `arxml` code. For an AUTOSAR dual-scaled parameter, the software

generates the `CompuMethod` category RAT_FUNC. For an example, see "Rational function CompuMethod for dual-scaled parameter".

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.

Cal2Main 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 **Cal2Main compute numerator**, you cannot change it.

Cal2Main compute denominator

Specify the denominator coefficients `c` and `c` of the first-order linear equation:

$$y = \frac{ax + b}{cx + d}$$

The default value is `[ ]` (unspecified). Specify finite, real, double scalar values for `c` and `d`. For example, `[1 1]`.

Once you have applied **Cal2Main 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`.

Is configuration valid

Simulink indicates whether the configuration is valid. The default value is `true`. If Simulink detects an issue with the configuration, it sets this field to `false` and provides information in the **Diagnostic message** field. You cannot set this field.

Diagnostic message

If you specify invalid parameter settings, Simulink displays a message in this field. Use the diagnostic information to help you fix an invalid configuration issue. You cannot set this field.

Properties

Name	Access	Description
`CompuMethodName`	`RW`	Name of AUTOSAR `CompuMethod` to generate for this parameter in `arxml` code. (CompuMethod name)
`CalibrationValue`	`RW`	Calibration value of this parameter. (Calibration value)
`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 `CalToMainCompuNumerator`, you cannot change it.
`CalToMainCompuDenominator`	`RW`	Denominator coefficients of the computational method. (CalToMain compute denominator)

Name	Access	Description
		Once you have applied `CalToMainCompuDenominator`, you cannot change it.
`CalibrationName`	`RW`	Name of the calibration parameter. (Calibration name)
`CalibrationDocUnits`	`RW`	Measurement units for this calibration parameter's value. (Calibration units)
`SwCalibrationAccess`	`RW`	Measurement and calibration tool access to the calibration parameter — `ReadOnly`, `ReadWrite`, or `NotAccessible`. (SwCalibrationAccess)
`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'
                       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;
```

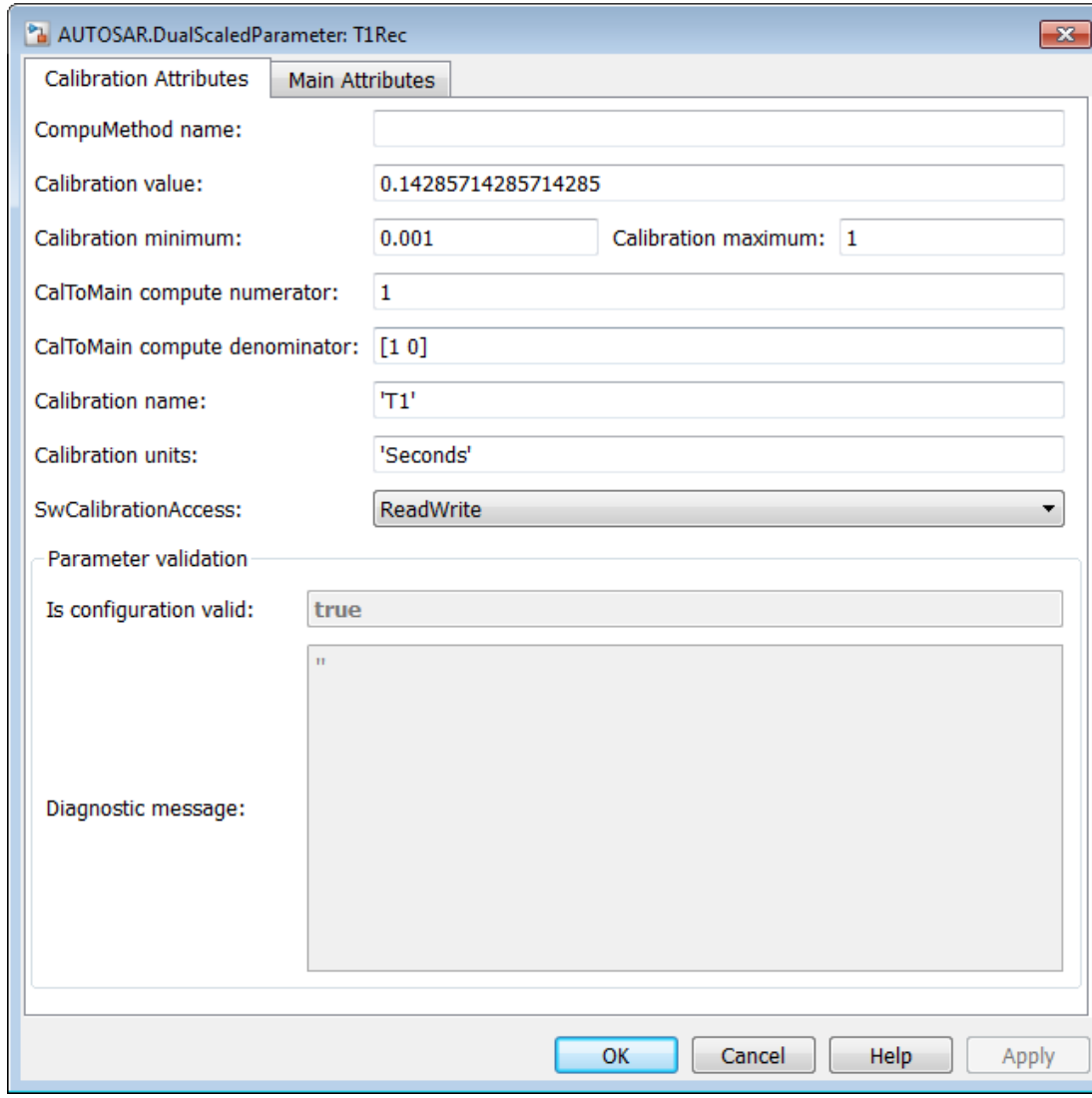
If you specify a value outside this allowable range, Simulink generates a warning.

Specify the units that Simulink uses.

```
T1Rec.DocUnits = 'Hertz';
```

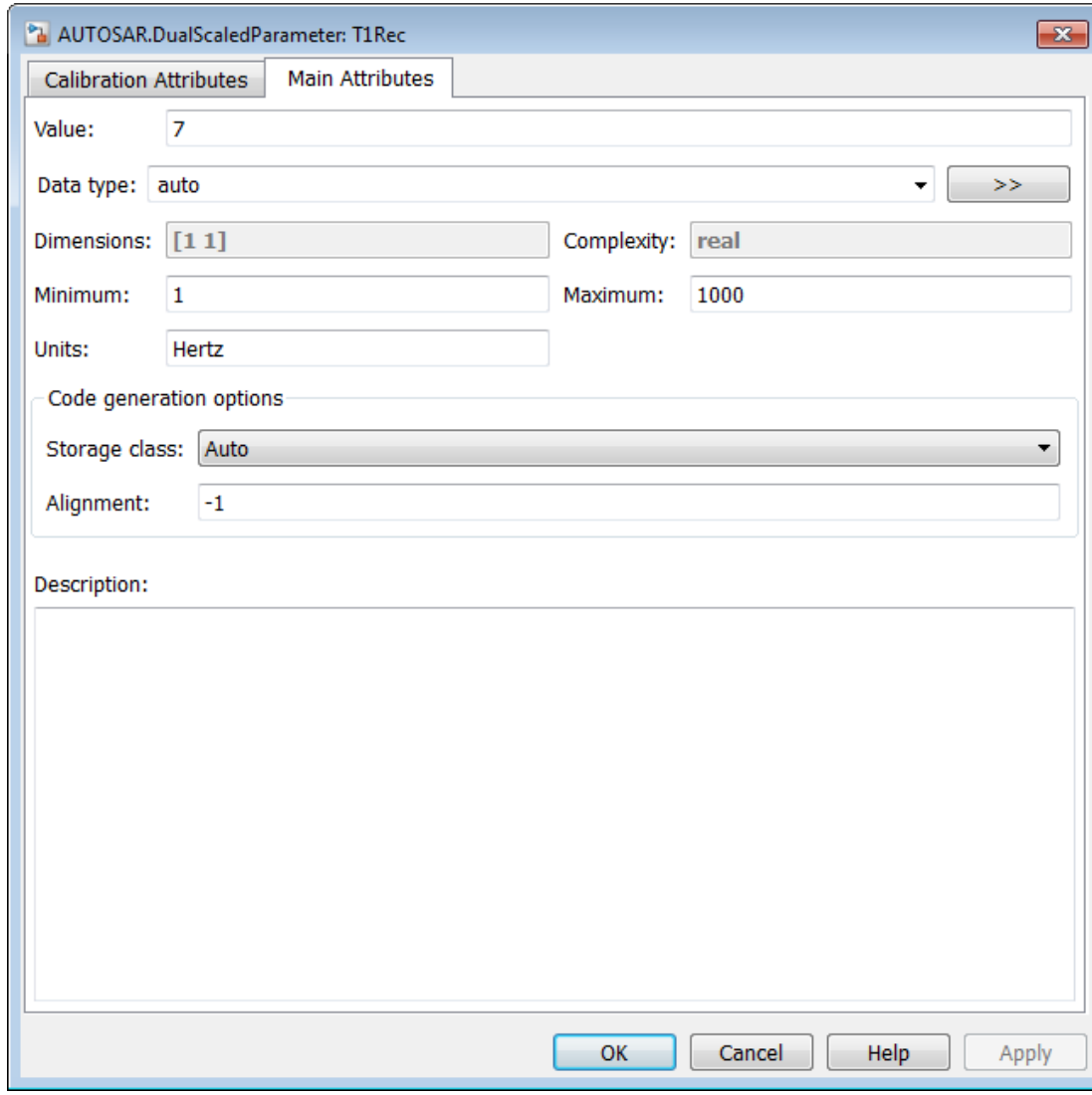
Open the AUTOSAR.DualScaledParameter dialog box.

```
open T1Rec
```



The **Calibration Attributes** tab displays the calibration value and the computational method that you specified.

In the dialog box, click the **Main Attributes** tab.


AUTOSAR.DualScaledParameter: T1Rec
Calibration Attributes
Main Attributes
Value:
7
Data type:
auto
>>
Dimensions:
[1 1]
Complexity:
real
Minimum:
1
Maximum:
1000
Units:
Hertz
Code generation options
Storage class:
Auto
Alignment:
-1
Description:
OK
Cancel
Help
Apply

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 generated ARXML file 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'
                       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 file 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 and enable the code generation report, when you generate code for the model, Embedded Coder generates an ARXML file 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.

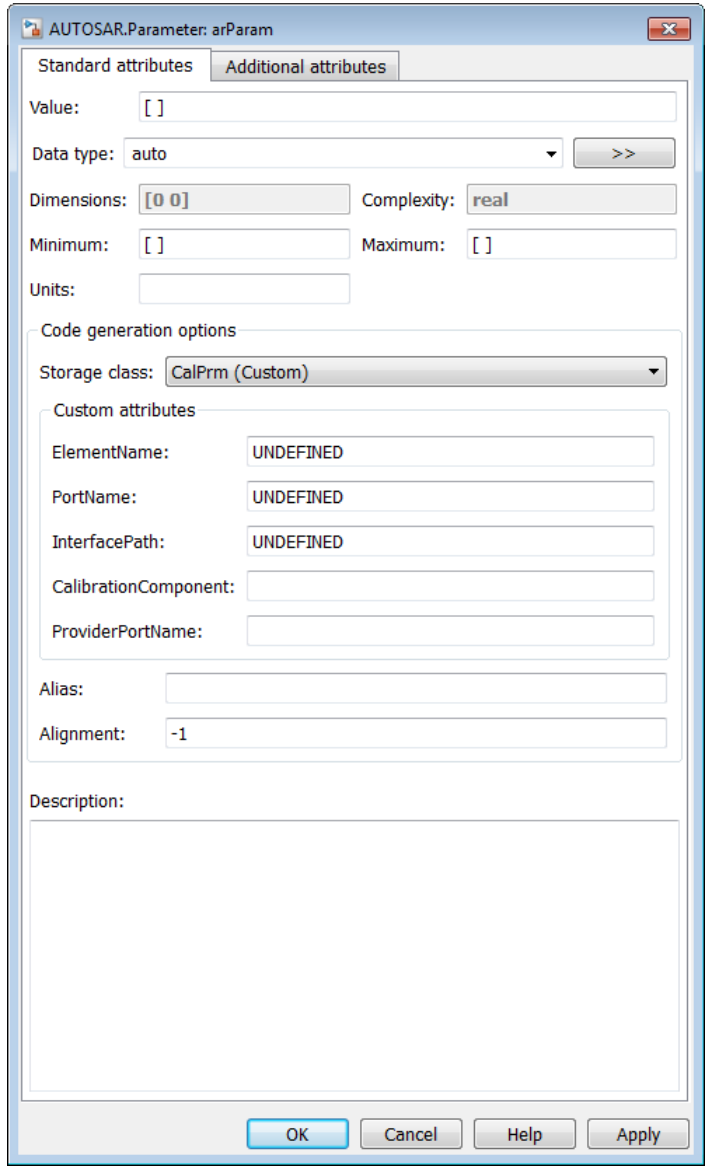
This class extends the `Simulink.Parameter` class. With parameter objects, you can specify the value of a parameter and other information about the parameter, such as its purpose, its dimensions, or its minimum and maximum values. Some Simulink products use this information, for example, to determine whether the parameter is tunable (see "Tunable 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



The `Simulink.Parameter` reference page describes the dialog box parameters in detail. The `AUTOSAR.Parameter` class extends the `Simulink.Parameter` class with the following additional selections for the **Storage class** parameter:

- `CalPrm (Custom)` — Calibration parameters belong to a calibration component which can be accessed by multiple AUTOSAR Software Components. Selecting this storage class enables the parameters **ElementName**, **PortName**, **InterfacePath**, **CalibrationComponent**, **ProviderPortName**, **Alias**, and **SwCalibrationAccess**.
  - Optionally, you can use **ElementName**, **PortName**, and **InterfacePath** to associate the calibration parameter with a specific AUTOSAR element, AUTOSAR port, or AUTOSAR interface.
  - Optionally, you can use **CalibrationComponent** and **ProviderPortName** to configure the calibration parameter to be exported in an AUTOSAR calibration component (`ParameterSwComponent`), which AUTOSAR software components (ASWCs) can access using an associated provider port. **CalibrationComponent** specifies the qualified name of the calibration component to be exported, and **ProviderPortName** specifies the short name of the associated provider port.
  - Optionally, you can use **Alias** to specify an identifier to represent the parameter in generated code.
  - **SwCalibrationAccess** allows you to specify measurement and calibration software access to the data as `NotAccessible`, `ReadOnly`, or `ReadWrite`.
- `InternalCalPrm (Custom)` — Internal calibration parameters are defined and accessed by only one AUTOSAR software component. Selecting this storage class enables the parameters **PerInstanceBehavior** and **Alias**.
  - **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`.
  - Optionally, you can use **Alias** to specify an identifier to represent the parameter in generated code.
  - **SwCalibrationAccess** allows you to specify measurement and calibration software access to the data as `NotAccessible`, `ReadOnly`, or `ReadWrite`.

Note: These **Storage class** selections require inline parameters to be enabled for code generation.

- `SystemConstant (Custom)` — Allows you to control the storage of a systemwide constant in generated code. Selecting this storage class enables the parameters **Alias** and **SwCalibrationAccess**.
  - Optionally, you can use **Alias** to specify an identifier to represent the parameter in generated code.
  - **SwCalibrationAccess** allows you to specify measurement and calibration software access to the data as `NotAccessible`, `ReadOnly`, or `ReadWrite`.

For more information, see:

- "Tunable Parameter Storage"
- "Model AUTOSAR Calibration Parameters"
- "Configure AUTOSAR Calibration Parameters"
- "Configure AUTOSAR Calibration Component"
- "Variation Point Proxies"
- "Configure AUTOSAR Variation Point Proxies"
- "Configure AUTOSAR Data for Measurement and Calibration"

See Also

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

AUTOSAR.Signal

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

Description

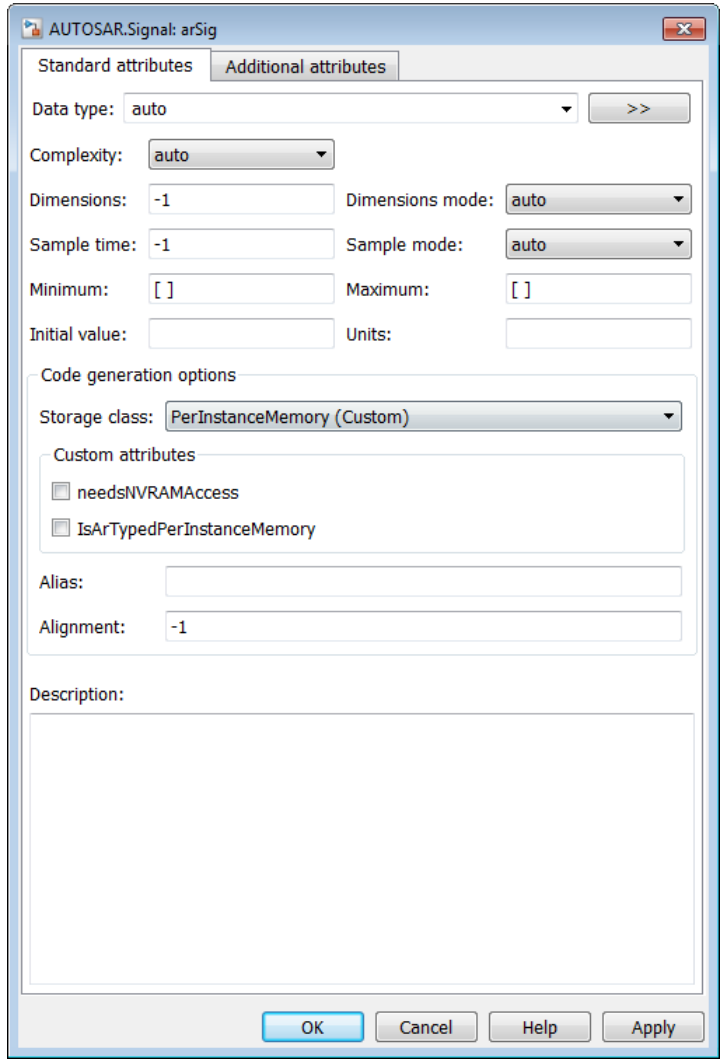
With this class, you can create workspace objects for modeling per-instance memory for AUTOSAR runnables. You can create an `AUTOSAR.Signal` object in the base MATLAB workspace.

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 dialog box parameters in detail. The `AUTOSAR.Signal` class extends the `Simulink.Signal` class with the following additional selection for the **Storage class** parameter:

- `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 parameters **needsNVRAMAccess**, **IsArTypedPerInstanceMemory**, **Alias**, and **SwCalibrationAccess**.

  - **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.
  - Optionally, you can use **Alias** to specify an identifier to represent the signal in generated code.
  - **SwCalibrationAccess** allows you to specify measurement and calibration software access to the data as `NotAccessible`, `ReadOnly`, or `ReadWrite`.

After you create an `AUTOSAR.Signal` object, you can reference it in a Data Store Memory block. For more information, see

- "Model AUTOSAR Per-Instance Memory"
- "Configure AUTOSAR Per-Instance Memory"
- "Configure AUTOSAR Data for Measurement and Calibration"

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”

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”

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.

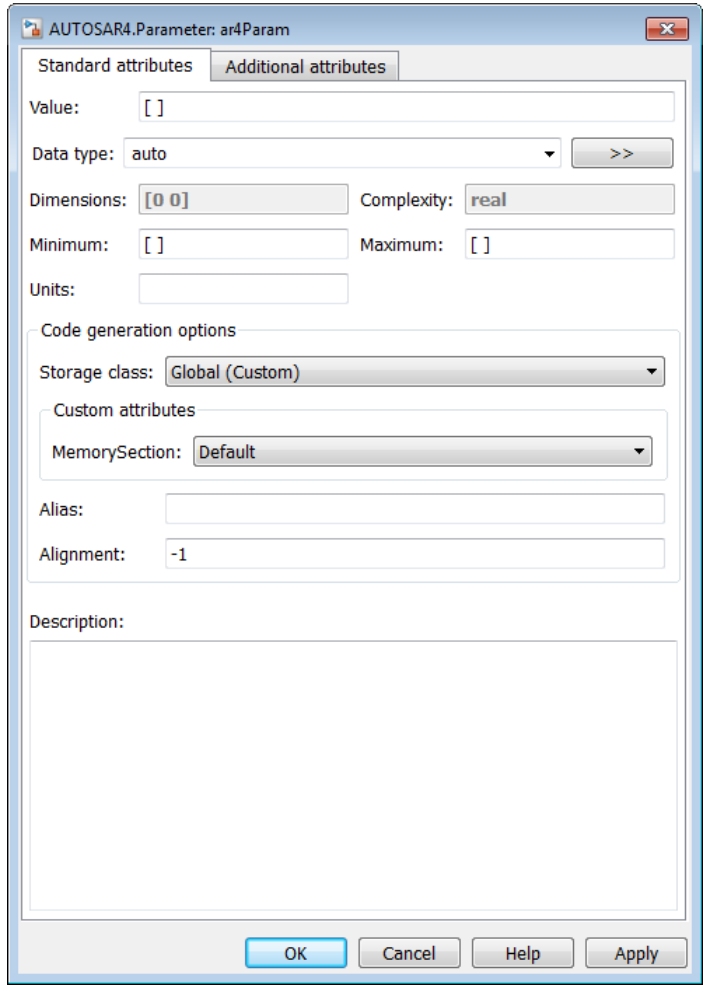
This class extends the `Simulink.Parameter` class. With parameter objects, you can specify the value of a parameter and other information about the parameter, such as its purpose, its dimensions, or its minimum and maximum values. Some Simulink products use this information, for example, to determine whether the parameter is tunable (see "Tunable 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 `Simulink.Parameter` reference page describes the dialog box parameters in detail. The `AUTOSAR4.Parameter` class extends the `Simulink.Parameter` class with the following additional selections for the **Storage class** parameter:

- `Global (Custom)` — Allows you to map internal global parameters to AUTOSAR Memory Sections. Selecting this storage class enables the parameters **MemorySection**, **Alias**, and **SwCalibrationAccess**.
  - **MemorySection** allows you to explicitly select AUTOSAR Memory Section `VAR` or `CAL`, or accept the `Default`.
  - Optionally, you can use **Alias** to specify an identifier to represent the signal in generated code.
  - **SwCalibrationAccess** allows you to specify measurement and calibration software access to the data as `NotAccessible`, `ReadOnly`, or `ReadWrite`.

For more information, see

- "Model AUTOSAR Static and Constant Memory"
- "Configure AUTOSAR Static or Constant Memory"
- "Configure AUTOSAR Data for Measurement and Calibration"

See Also

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

AUTOSAR4.Signal

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

Description

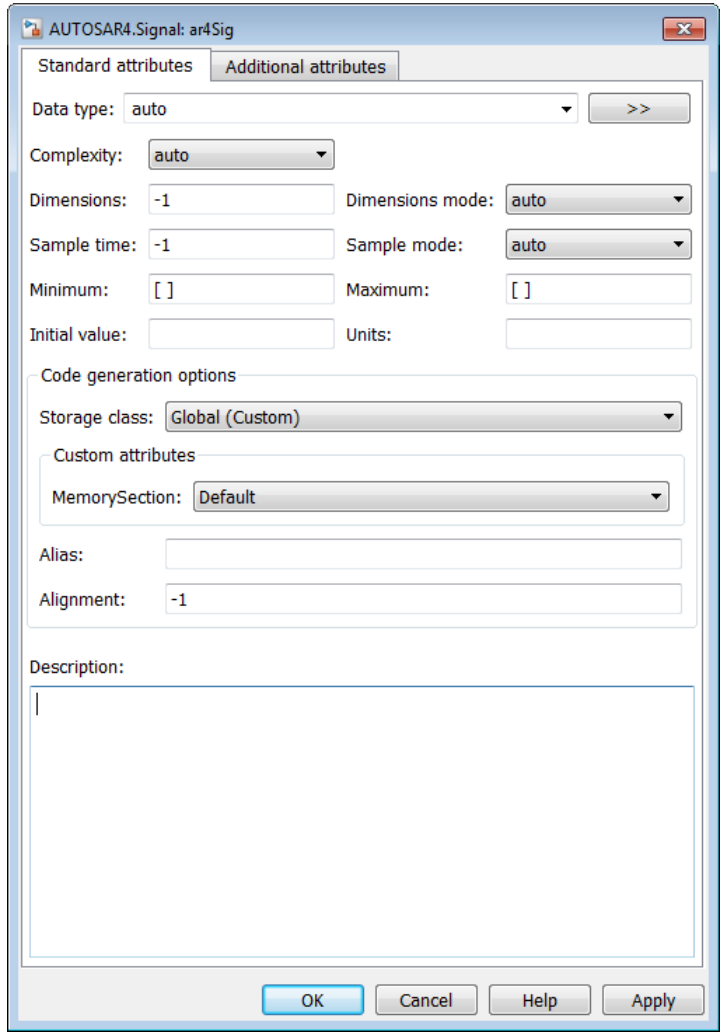
With this class, you can create workspace objects for mapping internal global signals to AUTOSAR Memory Sections. You can create an `AUTOSAR4.Signal` object in the base MATLAB workspace.

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 dialog box parameters in detail. The `AUTOSAR4.Signal` class extends the `Simulink.Signal` class with the following additional selection for the **Storage class** parameter:

- `Global (Custom)` — Allows you to map internal global signals to AUTOSAR Memory Sections. Selecting this storage class enables the parameters **MemorySection**, **Alias**, and **SwCalibrationAccess**.
  - **MemorySection** allows you to explicitly select AUTOSAR Memory Section `VAR` or `CAL`, or accept the `Default`.
  - Optionally, you can use **Alias** to specify an identifier to represent the signal in generated code.
  - **SwCalibrationAccess** allows you to specify measurement and calibration software access to the data as `NotAccessible`, `ReadOnly`, or `ReadWrite`.

For more information, see

- "Model AUTOSAR Static and Constant Memory"
- "Configure AUTOSAR Static or Constant Memory"
- "Configure AUTOSAR Data for Measurement and Calibration"

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,P
```

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,P` specifies additional options for Simulink calibration object creation.

Input Arguments

importerObj	Handle to imported AUTOSAR information created in a previous call to `arxml.importer`.
componentName	Absolute short name path of calibration parameter component.
PropertyN,ValueN	Optional property/value pairs representing creation options. You can specify values for the following properties: `'CreateSimulinkObject'` `true` (default) or `false`. If `true`, the function creates `Simulink.AliasType` and `Simulink.NumericType` data

objects corresponding to the AUTOSAR data types in the XML file.

`'DataDictonary'`

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"

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,Va
```

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,Va`
specifies additional options for Simulink model creation.

Input Arguments

`importerObj`	Handle to imported AUTOSAR information created in a previous call to `arxml.importer`.
`ComponentName`	Absolute short name path of the atomic software component.
`PropertyN,ValueN`	Optional property/value pairs representing creation options. You can specify values for the following properties: `'AutoSave'` `true` or `false` (default). If `true`, the function automatically saves the generated Simulink model.

`'CreateInternalBehavior'`

`true` or `false` (default). If `true`, the function automatically imports the internal behavior of a multi-runnable AUTOSAR software component into the Simulink model. The importer:

- Adds subsystem blocks in the model and maps them to corresponding runnables imported from the AUTOSAR software component.
- Adds signal lines in the model and maps them to corresponding interrunnable variables (IRVs) imported from the AUTOSAR software component.

`'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.

`'DataDictonary'`

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')
```

Automatically import the internal behavior of a multi-runnable AUTOSAR software component into a Simulink 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"

delete

Delete AUTOSAR element

Syntax

```
delete(arProps,elementPath)
```

Description

`delete(arProps,elementPath)` deletes the AUTOSAR element at elementPath.

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”
- “Configure the AUTOSAR Interface”

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.

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"
- "Import AUTOSAR Software Component"
- "Configure the AUTOSAR Interface"

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.

Examples

Find Sender-Receiver Interfaces That Are Not Services

For a model, find sender-receiver interfaces for which the property `IsService` is `false` and return fully qualified paths.

```
rtwdemo_autosar_multirunnables
arProps=autosar.api.getAUTOSARProperties('rtwdemo_autosar_multirunnables');
ifPaths=find(arProps,[],'SenderReceiverInterface',...
  'IsService',false,'PathType','FullyQualified')

ifPaths =
```

```
    '/pkg/if/Interface1'    '/pkg/if/Interface2'
```

Find Mode-Switch Interface Paths

For a model, add a mode-switch interface and then use `find` to list paths for mode-switch interfaces in the model.

```
rtwdemo_autosar_multirunnables
arProps=autosar.api.getAUTOSARProperties('rtwdemo_autosar_multirunnables');
addPackageableElement(arProps,'ModeSwitchInterface','/pkg/if','Interface3',...
  'IsService',true);
ifPaths=find(arProps,[],'ModeSwitchInterface','PathType','FullyQualified')
```

```
ifPaths =
    '/pkg/if/Interface3'
```

Input Arguments

`arProps` — AUTOSAR properties information for a model

handle

AUTOSAR properties information for a model, previously returned by *`arProps`* `= autosar.api.getAUTOSARProperties(`*`model`*`).` *`model`* is a handle or 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” 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`

Related Examples

- “Configure and Map AUTOSAR Component Programmatically”
- “Configure the AUTOSAR Interface”

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.

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"
- "Configure the AUTOSAR Interface"

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.

Input Arguments

`importerObj`	Handle to imported AUTOSAR information created in a previous call to `arxml.importer`.

Output Arguments

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

See Also

`arxml.importer.getSensorActuatorComponentNames` | `arxml.importer.getComponentNames`

How To

- "Import AUTOSAR Software Component"

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*.

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"

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.

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"
- "Import AUTOSAR Software Component"

- “Configure AUTOSAR Client-Server Communication”

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`*.

Note: `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.

Input Arguments

`importerObj`	Handle to imported AUTOSAR information created in a previous call to `arxml.importer`.

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"

getDataTransfer

Get AUTOSAR mapping information for Simulink data transfer line

Syntax

```
[arIrvName,arDataAccessMode]=getDataTransfer(slMap,
slDataTransferName)
```

Description

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

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
slMap=autosar.api.getSimulinkMapping('rtwdemo_autosar_multirunnables');
[arIrvName,arDataAccessMode]=getDataTransfer(slMap,'irv4')

arIrvName =
IRV4

arDataAccessMode =
Implicit
```

- “Configure and Map AUTOSAR Component Programmatically”
- “Configure the AUTOSAR Interface”

Input Arguments

`slMap` — Simulink to AUTOSAR mapping information for a model

handle

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

Example: `slMap`

`slDataTransferName` — 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: `arDataAcessMode`

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`*.

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"

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`*.

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"

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.

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"
- "Configure the AUTOSAR Interface"

Input Arguments

`slMap` — Simulink to AUTOSAR mapping information for a model

handle

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

Example: `slMap`

`slFcnName` — Name of Simulink entry point function

string

Name of the Simulink entry point function for which to return AUTOSAR mapping information.

Example: `'InitializeFunction'`

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(slMap,slFcnName)
```

Description

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

Examples

Get AUTOSAR Mapping Information for Function Caller Block

Get AUTOSAR mapping information for a function-caller block in a model in which AUTOSAR client function invocation is being modeled. The model has a function-caller block for Simulink function `readData`.

```
open_system('mControllerWithInterface_client')
slMapC = autosar.api.getSimulinkMapping('mControllerWithInterface_client');
mapFunctionCaller(slMapC,'readData','cPort','readData');
[arPort,arOp] = getFunctionCaller(slMapC,'readData')

arPort =
cPort

arOp =
readData
```

- “Configure and Map AUTOSAR Component Programmatically”
- “Configure the AUTOSAR Interface”

Input Arguments

`slMap` — Simulink to AUTOSAR mapping information for a model

handle

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

Example: `slMap`

`slFcnName` — 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(slMap,
slPortName)
```

Description

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

Examples

Get AUTOSAR Mapping Information for Model Inport

Get AUTOSAR mapping information for a model inport in the example model `rtwdemo_autosar_multirunnable`. The model has an inport named `RPort_DE1`.

```
rtwdemo_autosar_multirunnables
slMap=autosar.api.getSimulinkMapping('rtwdemo_autosar_multirunnables');
[arPortName,arDataElementName,arDataAccessMode]=getInport(slMap,'RPort_DE1')

arPortName =
RPort

arDataElementName =
DE1

arDataAccessMode =
ImplicitReceive
```

- “Configure and Map AUTOSAR Component Programmatically”
- “Configure the AUTOSAR Interface”

Input Arguments

`slMap` — Simulink to AUTOSAR mapping information for a model

handle

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

Example: `slMap`

`slPortName` — 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`, or `ModeReceive`.

Example: `arDataAcessMode`

See Also

`mapInport`

getOutport

Get AUTOSAR mapping information for Simulink outport

Syntax

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

Description

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

Examples

Get AUTOSAR Mapping Information for Model Outport

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

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

arPortName =
PPort

arDataElementName =
DE1

arDataAccessMode =
ImplicitSend
```

- “Configure and Map AUTOSAR Component Programmatically”
- “Configure the AUTOSAR Interface”

Input Arguments

`slMap` — Simulink to AUTOSAR mapping information for a model

handle

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

Example: `slMap`

`slPortName` — Name of model outport

string

Name of the model outport 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 outport.

Example: `arPortName`

`arDataElementName` — Name of AUTOSAR data element

string

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

Example: `arDataElementName`

`arDataAccessMode` — Value of AUTOSAR data access mode

string

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

Example: `arDataAcessMode`

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.

Input Arguments

`importerObj`	Handle to imported AUTOSAR information created in a previous call to `arxml.importer`.

Output Arguments

`sensoractuator Software ComponentNames`	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”

mapDataTransfer

Map Simulink data transfer line to AUTOSAR inter-runnable variable

Syntax

```
mapDataTransfer(slMap,slDataTransferName,arIrvName,arDataAccessMode)
```

Description

`mapDataTransfer(slMap,slDataTransferName,arIrvName,arDataAccessMode)` maps the Simulink data transfer line slDataTransferName to AUTOSAR inter-runnable variable arIrvName and AUTOSAR data access mode arDataAccessMode.

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
slMap=autosar.api.getSimulinkMapping('rtwdemo_autosar_multirunnables');
mapDataTransfer(slMap,'irv4','IRV4','Explicit');
[arIrvName,arDataAccessMode]=getDataTransfer(slMap,'irv4')

arIrvName =
IRV4

arDataAccessMode =
Explicit
```

- “Configure and Map AUTOSAR Component Programmatically”
- “Configure the AUTOSAR Interface”

Input Arguments

`slMap` — Simulink to AUTOSAR mapping information for a model

handle

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

Example: `slMap`

`slDataTransferName` — 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(slMap,slFcnName,arRunnableName)
```

Description

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

Examples

Set AUTOSAR Mapping Information for Simulink Entry-Point Function

Set AUTOSAR mapping information for a Simulink entry point function in the example model `rtwdemo_autosar_multirunnables`. The model has an initialization entry-point function named `InitializeFunction` and three exported entry-point functions named `Runnable1`, `Runnable2`, and `Runnable3`.

```
open_system('rtwdemo_autosar_multirunnables')
slMap=autosar.api.getSimulinkMapping('rtwdemo_autosar_multirunnables');
mapFunction(slMap,'InitializeFunction','Runnable_Init');
arRunnableName=getFunction(slMap,'InitializeFunction')
```

```
arRunnableName =
Runnable_Init
```

- “Configure and Map AUTOSAR Component Programmatically”
- “Configure the AUTOSAR Interface”

Input Arguments

`slMap` — Simulink to AUTOSAR mapping information for a model

handle

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

Example: `slMap`

`slFcnName` — Name of Simulink entry point function

string

Name of the Simulink entry point function for which to set AUTOSAR mapping information.

Example: `'InitializeFunction'`

`arRunnableName` — Name of AUTOSAR runnable

string

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

Example: `'Runnable_Init'`

See Also

`getFunction`

mapFunctionCaller

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

Syntax

```
mapFunctionCaller(slMap,slFcnName,arPortName,arOperationName)
```

Description

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

If your model has multiple callers of Simulink function slFcnName, this function maps all of them to the AUTOSAR client port and operation.

Examples

Set AUTOSAR Mapping Information for Function Caller Block

Set AUTOSAR mapping information for a function-caller block in a model in which AUTOSAR client function invocation is being modeled. The model has a function-caller block for Simulink function `readData`.

```
open_system('mControllerWithInterface_client')
slMapC = autosar.api.getSimulinkMapping('mControllerWithInterface_client');
mapFunctionCaller(slMapC,'readData','cPort','readData');
[arPort,arOp] = getFunctionCaller(slMapC,'readData')
```

```
arPort =
cPort

arOp =
readData
```

- “Configure and Map AUTOSAR Component Programmatically”
- “Configure the AUTOSAR Interface”

Input Arguments

`slMap` — Simulink to AUTOSAR mapping information for a model

handle

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

Example: `slMap`

`slFcnName` — 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(slMap,slPortName,arPortName,arDataElementName,
arDataAccessMode)
```

Description

`mapInport(slMap,slPortName,arPortName,arDataElementName, arDataAccessMode)` maps the Simulink inport slPortName to the AUTOSAR data element arDataElementName at AUTOSAR receiver port arPortName. The AUTOSAR data access mode for the receiver port is set to arDataAccessMode.

Examples

Set AUTOSAR Mapping Information for Model Inport

Set AUTOSAR mapping information for a model inport in the example model `rtwdemo_autosar_multirunnables`. The model has an inport named `RPort_DE1`. This example changes the AUTOSAR data access mode for `RPort_DE1` from `ImplicitReceive` to `ExplicitReceive`.

```
rtwdemo_autosar_multirunnables
slMap=autosar.api.getSimulinkMapping('rtwdemo_autosar_multirunnables');
mapInport(slMap,'RPort_DE1','RPort','DE1','ExplicitReceive');
[arPortName,arDataElementName,arDataAccessMode]=getInport(slMap,'RPort_DE1')
```

```
arPortName =
RPort

arDataElementName =
DE1

arDataAccessMode =
ExplicitReceive
```

- “Configure and Map AUTOSAR Component Programmatically”

- “Configure the AUTOSAR Interface”

Input Arguments

`slMap` — Simulink to AUTOSAR mapping information for a model

handle

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

Example: `slMap`

`slPortName` — 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`, or `ModeReceive`.

Example: `'ExplicitReceive'`

See Also

`getInport`

mapOutport

Map Simulink outport to AUTOSAR port

Syntax

```
mapOutport(slMap,slPortName,arPortName,arDataElementName,
arDataAccessMode)
```

Description

`mapOutport(slMap,slPortName,arPortName,arDataElementName, arDataAccessMode)` maps the Simulink outport slPortName to the AUTOSAR data element arDataElementName at AUTOSAR provider port arPortName. The AUTOSAR data access mode for the provider port is set to arDataAccessMode.

Examples

Set AUTOSAR Mapping Information for Model Outport

Set AUTOSAR mapping information for a model outport in the example model `rtwdemo_autosar_multirunnables`. The model has an outport named `PPort_DE1`. This example changes the AUTOSAR data access mode for `PPort_DE1` from `ImplicitSend` to `ExplicitSend`.

```
rtwdemo_autosar_multirunnables
slMap=autosar.api.getSimulinkMapping('rtwdemo_autosar_multirunnables');
mapOutport(slMap,'PPort_DE1','PPort','DE1','ExplicitSend');
[arPortName,arDataElementName,arDataAccessMode]=getOutport(slMap,'PPort_DE1')
```

```
arPortName =
PPort

arDataElementName =
DE1

arDataAccessMode =
ExplicitSend
```

- “Configure and Map AUTOSAR Component Programmatically”
- “Configure the AUTOSAR Interface”

Input Arguments

`slMap` — Simulink to AUTOSAR mapping information for a model

handle

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

Example: `slMap`

`slPortName` — Name of model outport

string

Name of the model outport 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 outport.

Example: `'Output'`

`arDataElementName` — Name of AUTOSAR data element

string

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

Example: `'Output'`

`arDataAccessMode` — Value of AUTOSAR data access mode

string

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

Example: `'ExplicitSend'`

See Also

`getOutport`

set

Set property of AUTOSAR element

Syntax

```
set(arProps,elementPath,property,value)
```

Description

`set(arProps,elementPath,property,value)` sets the specified property of the AUTOSAR element at elementPath to value. For properties that reference other elements, value is s path. To set XML packaging options, specify elementPath as `XmlOptions`.

Examples

Set IsService Property for Sender-Receiver Interface

For a model, set the `IsService` property for sender-receiver interface `Interface1` to `true` (1).

```
rtwdemo_autosar_multirunnables
arProps=autosar.api.getAUTOSARProperties('rtwdemo_autosar_multirunnables');
set(arProps,'Interface1','IsService',true);
isService=get(arProps,'Interface1','IsService')
```

```
isService =
     1
```

Set Runnable Symbol Name

For a model, set the `symbol` property for runnable `Runnable1` to `test_symbol`.

```
rtwdemo_autosar_multirunnables
arProps=autosar.api.getAUTOSARProperties('rtwdemo_autosar_multirunnables');
compQName=get(arProps,'XmlOptions','ComponentQualifiedName');
runnables=find(arProps,compQName,'Runnable','PathType','FullyQualified');
runnables(2)
```

```
ans =
    '/pkg/swc/ASWC/Behavior/Runnable1'
```

```
get(arProps,runnables{2},'symbol')

ans =
Runnable1

set(arProps,runnables{2},'symbol','test_symbol2')
get(arProps,runnables{2},'symbol')

ans =
test_symbol2
```

- "Configure and Map AUTOSAR Component Programmatically"
- "Configure the AUTOSAR Interface"

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`*.

Input Arguments

`importerObj`	Handle to imported AUTOSAR information created in a previous call to `arxml.importer`.
`dependencies`	Can be: • a cell array of strings (for a list of dependencies) • a char array (for a single dependency) • or the empty array [] (for removing a dependency)

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

How To

- "Import AUTOSAR Software Component"

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`*.

Input Arguments

`importerObj`	Handle to imported AUTOSAR information created in a previous call to `arxml.importer`.
`filename`	XML file name. Only atomic software components described in this file can be imported.

Examples

Set the name of the main software component file associated with an `arxml.importer` object.

```
>> obj = arxml.importer({'control_system_component.arxml',...
                         'control_system_interface.arxml', ...
                         'control_system_datatype.arxml'});
>> setFile(obj,'control_system_component2.arxml')
>> obj

obj =

The file "H:\wrk\control_system_component2.arxml" contains:
```

```
  1 Application-Software-Component-Type:
    '/ComponentType/controlSystem'

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

See Also

`arxml.importer.getFile`

How To

- "Import AUTOSAR Software Component"

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 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.

Input Arguments

`importerObj`	Handle to imported AUTOSAR information created in a previous call to `arxml.importer`.
`modelname`	Name of the model to be updated with changes in the imported AUTOSAR description represented by *`importerObj`*. The model must be open.

Examples

Update model `mySWC` with AUTOSAR `arxml` changes described in `updatedSWC.arxml`.

```
>> 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"