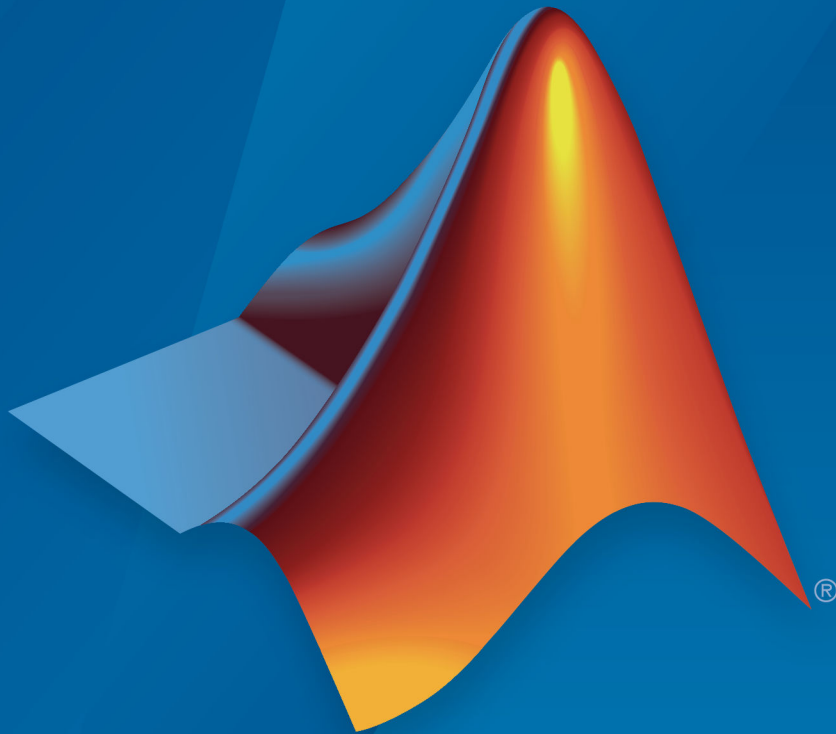


# AUTOSAR Blockset

Reference



# MATLAB® & SIMULINK®

R2019a



# How to Contact MathWorks



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



Phone: 508-647-7000



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

## *AUTOSAR Blockset Reference*

© COPYRIGHT 2019 by The MathWorks, Inc.

The software described in this document is furnished under a license agreement. The software may be used or copied only under the terms of the license agreement. No part of this manual may be photocopied or reproduced in any form without prior written consent from The MathWorks, Inc.

FEDERAL ACQUISITION: This provision applies to all acquisitions of the Program and Documentation by, for, or through the federal government of the United States. By accepting delivery of the Program or Documentation, the government hereby agrees that this software or documentation qualifies as commercial computer software or commercial computer software documentation as such terms are used or defined in FAR 12.212, DFARS Part 227.72, and DFARS 252.227-7014. Accordingly, the terms and conditions of this Agreement and only those rights specified in this Agreement, shall pertain to and govern the use, modification, reproduction, release, performance, display, and disclosure of the Program and Documentation by the federal government (or other entity acquiring for or through the federal government) and shall supersede any conflicting contractual terms or conditions. If this License fails to meet the government's needs or is inconsistent in any respect with federal procurement law, the government agrees to return the Program and Documentation, unused, to The MathWorks, Inc.

### **Trademarks**

MATLAB and Simulink are registered trademarks of The MathWorks, Inc. See [www.mathworks.com/trademarks](http://www.mathworks.com/trademarks) for a list of additional trademarks. Other product or brand names may be trademarks or registered trademarks of their respective holders.

### **Patents**

MathWorks products are protected by one or more U.S. patents. Please see [www.mathworks.com/patents](http://www.mathworks.com/patents) for more information.

### **Revision History**

March 2019      Online only      New for Version 2.0 (Release 2019a)

## Functions – Alphabetical List

1

## Blocks – Alphabetical List

2

## Tools – Alphabetical List

3

## Model Advisor Checks

4

<b>MathWorks AUTOSAR Blockset Checks</b> .....	<b>4-2</b>
MathWorks Automotive Advisory Board Checks .....	<b>4-2</b>
Check model configuration parameters for AUTOSAR compliance .....	<b>4-2</b>
Check compatibility of AUTOSAR Interpolation Routines .....	<b>4-4</b>



# Functions — Alphabetical List

---

## add

**Package:** autosar.api

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.

```
hModel = 'autosar_swc_expfncs';
open_system(hModel);
arProps=autosar.api.getAUTOSARProperties(hModel);
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`.

```
addpath(fullfile(matlabroot, '/help/toolbox/autosar/examples'));
hModel = 'mAutosarMsConfigAfter';
open_system(hModel);
arProps=autosar.api.getAUTOSARProperties(hModel);
addPackageableElement(arProps, 'ModeSwitchInterface', '/pkg/if', 'Interface3', ...
    'IsService', true);
ifPaths=find(arProps, [], 'ModeSwitchInterface', 'PathType', 'FullyQualified')

ifPaths =
    {'/pkg/if/myMsIf'}    {'/pkg/if/MsIf2'}    {'/pkg/if/Interface3'}

add(arProps, '/pkg/if/Interface3', 'ModeGroup', 'mgModes');
```

## 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, character vector, or string scalar representing the model name.

Example: `arProps`

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

character vector | string scalar

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

Example: `'Input'`

### **property** — Type of property

character vector | string scalar

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

Example: `'DataElements'`

### **name** — Name of child property element

character vector | string scalar

Name of the child property element to add.

Example: 'DE1'

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

name (character vector or string scalar), 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

## Topics

“Configure and Map AUTOSAR Component Programmatically”

“AUTOSAR Component Configuration”

**Introduced in R2013b**



# addPackageableElement

**Package:** autosar.api

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`.

```
hModel = 'autosar_swc_expfncns';
open_system(hModel);
arProps=autosar.api.getAUTOSARProperties(hModel);
addPackageableElement(arProps, 'SenderReceiverInterface', '/pkg/if', 'Interface3', ...
    'IsService', true);
ifPaths=find(arProps, [], 'SenderReceiverInterface', ...
    'IsService', true, '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, character vector, or string scalar representing the model name.

Example: `arProps`

### **category — Element category**

character vector | string scalar

Category of element to add. Valid category values are 'ClientServerInterface', 'DataTypeMappingSet', 'ModeDeclarationGroup', 'ModeSwitchInterface', 'Package', 'ParameterComponent', 'ParameterInterface', 'SenderReceiverInterface', 'SwAddrMethod', and 'SystemConst'.

Example: 'SenderReceiverInterface'

### **package — Package path**

character vector | string scalar

Fully-qualified path to the element package.

Example: '/pkg/if'

### **name — Element name**

character vector | string scalar

Name of the element to add.

Example: 'Interface3'

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

name (character vector or string scalar), 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

## **Topics**

“Configure and Map AUTOSAR Component Programmatically”

“AUTOSAR Component Configuration”

**Introduced in R2014b**

## arxml.importer

Import AUTOSAR component XML

### Description

Use `arxml.importer` functions to import AUTOSAR components into Simulink® in a controlled manner. For example, you can parse an AUTOSAR software component description XML file exported by an AUTOSAR authoring tool, and then import the component into a Simulink model. After importing the component, use the Simulink representation of the component for further configuration, algorithm development, C/C++ code generation, and `arxml` export.

### Creation

### Syntax

```
ar = arxml.importer(filename)
ar = arxml.importer({filename1,filename2,...,filenameN})
```

### Description

`ar = arxml.importer(filename)` creates object `ar`, which represents the AUTOSAR information in XML file `filename`.

`ar = arxml.importer({filename1,filename2,...,filenameN})` creates object `ar`, which represents the AUTOSAR information in the specified XML files.

---

**Tip** If you enter the `arxml.importer` function call without a terminating semicolon (;), the importer lists the AUTOSAR content of the specified XML file or files. The information includes paths to software components in the AUTOSAR package structure, which you can specify in calls to `createComponentAsModel` and `createCompositionAsModel`.

---

## Input Arguments

### **filename — AUTOSAR XML filename**

character vector | string scalar

Name of XML file containing AUTOSAR information.

Example: 'mySWC.arxml'

### **filename1,filename2,...,filenameN — AUTOSAR XML filenames**

cell array of character vectors | string array

Cell array of names of XML files containing AUTOSAR information.

Example: {'mySWC.arxml','DataTypes.arxml','MiscDefs.arxml'}

## Object Functions

createComponentAsModel	Create Simulink representation of AUTOSAR arxml atomic software component
createCompositionAsModel	Create Simulink representation of AUTOSAR arxml software composition
getComponentNames	Get AUTOSAR software component names from arxml files
updateAUTOSARProperties	Update model with arxml definitions of AUTOSAR elements
updateModel	Update AUTOSAR model with arxml changes

## Examples

### **Create arxml.importer Object from AUTOSAR XML File**

Call the `arxml.importer` function to create object `ar`, which represents the AUTOSAR information in XML file `mySWC.arxml`. Use the returned object to import AUTOSAR software component `/pkg/swc` and create an initial Simulink representation of the component.

```
ar = arxml.importer('mySWC.arxml')
createComponentAsModel(ar, '/pkg/swc', 'ModelPeriodicRunnablesAs', 'AtomicSubsystem')
```

## Create `arxml.importer` Object from Multiple AUTOSAR XML Files

Call the `arxml.importer` function to create object `ar`, which represents the AUTOSAR information in XML files `mySWC.arxml`, `DataTypes.arxml`, and `MiscDefs.arxml`. Use the returned object to import AUTOSAR software component `/pkg/swc` and create an initial Simulink representation of the component.

```
ar = arxml.importer({'mySWC.arxml', 'DataTypes.arxml', 'MiscDefs.arxml'})
createComponentAsModel(ar, '/pkg/swc', 'ModelPeriodicRunnablesAs', 'AtomicSubsystem')
```

## See Also

### Topics

“Import AUTOSAR Software Component”

“AUTOSAR `arxml` Importer”

“Round-Trip Preservation of AUTOSAR XML File Structure and Element Information”

“Import AUTOSAR Software Component Updates”

“Reuse AUTOSAR Element Descriptions”

**Introduced in R2008a**

# autosar.api.create

Create or update mapped AUTOSAR component model

## Syntax

```
autosar.api.create(model)
autosar.api.create(model, mode)
```

## Description

`autosar.api.create(model)` creates or updates mapped AUTOSAR software component model `model`. The default function behavior depends on the mapping state of the model.

- If the model is not mapped to an AUTOSAR software component, the function creates a Simulink to AUTOSAR mapping in `default` mode. In this mapping, Simulink inports and outports are mapped to AUTOSAR ports with default AUTOSAR properties.
- If the model is already mapped to an AUTOSAR software component, the function updates the existing mapping in `incremental` mode. The function finds and maps unmapped model elements, and updates the AUTOSAR Dictionary for deleted model elements.

`autosar.api.create(model, mode)` additionally specifies a mapping mode — `default`, `init`, or `incremental`.

## Examples

### Create Default AUTOSAR Properties and Mapping

Create AUTOSAR properties and Simulink to AUTOSAR mapping for an Embedded Coder® model in which the model configuration parameter **System target file** has been changed from `ert.tlc` to `autosar.tlc` or `autosar_adaptive.tlc`. Map model inports and outports to AUTOSAR ports with default AUTOSAR properties.

```
open_system('rtwdemo_counter');  
set_param('rtwdemo_counter','SystemTargetFile','autosar.tlc');  
autosar.api.create('rtwdemo_counter');
```

## Incrementally Update Mapped AUTOSAR Component for Model Changes

For a mapped AUTOSAR software component model, update the mapping to account for incremental model changes. Find and map unmapped model elements and update the AUTOSAR Dictionary for deleted model elements.

```
open_system('autosar_swc');  
autosar.api.create('autosar_swc','incremental');
```

## Input Arguments

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

handle | character vector | string scalar

Model for which to create or update AUTOSAR properties and Simulink to AUTOSAR mapping, specified as a handle, character vector, or string scalar representing the model name.

Example: 'my\_model'

### **mode** — Mode in which to map model elements

default | init | incremental

The default mode value depends on the mapping state of the model — `default` for an unmapped model or `incremental` for a mapped model.

Specify `default` to create AUTOSAR properties and Simulink to AUTOSAR mapping for a model. As part of the mapping, the function maps model inports and outputs to AUTOSAR ports with default AUTOSAR properties. If the model is already mapped, the function overwrites the existing mapping.

Specify `init` to create AUTOSAR properties and Simulink to AUTOSAR mapping for a model. As part of the mapping, the function does *not* map model inports and outputs. If the model is already mapped, the function overwrites the existing mapping.



Specify `incremental` to update the existing mapping in a mapped AUTOSAR software component model. The function finds and maps unmapped model elements and updates the AUTOSAR Dictionary for deleted model elements.

Example: `'default'`

## See Also

`autosar.api.delete` | `autosar_ui_launch` | `updateAUTOSARProperties`

## Topics

*"Configure and Map AUTOSAR Component Programmatically"*

*"AUTOSAR Component Configuration"*

**Introduced in R2013b**

## createEnumeration

**Package:** autosar.api

Create Simulink enumeration data type definition to work with imported AUTOSAR element

### Syntax

```
createEnumeration(arProps,name,applicationDataTypePath)
createEnumeration(arProps,name,compuMethodPath,
implementationDataTypePath)
createEnumeration(arProps,compuMethodPath)
```

### Description

`createEnumeration(arProps,name,applicationDataTypePath)` creates Simulink Enumeration from its application data type that can be used to work with imported AUTOSAR elements.

`createEnumeration(arProps,name,compuMethodPath,implementationDataTypePath)` creates Simulink Enumeration from its implementation data type and CompuMethod.

`createEnumeration(arProps,compuMethodPath)` creates family of Simulink enumerations from its CompuMethod.

### Examples

#### Create an Enumeration Data Type from its Application Data Type

Create a Simulink enumeration data type definition with the name `myEnum` from the application data type at path `'/AUTOSAR_PlatformTypes/ApplicationDataTypes/MyAppType'`.

```
dataObj = ...
autosar.api.getAUTOSARProperties mdlName);
createEnumeration(dataObj, 'myEnum', ...
    '/AUTOSAR_PlatformTypes/ApplicationDataTypes/MyAppType');
```

## Create an Enumeration Data Type from its Implementation Data Type and CompuMethod

Create a Simulink enumeration data type definition with the name `myEnum` from the implementation data type at path `'/AUTOSAR_PlatformTypes/ImplementationDataTypes/uint16'` by using the computation method from path `'/a/b/myCM'`.

```
dataObj = ...
autosar.api.getAUTOSARProperties mdlName);
createEnumeration(dataObj, 'myEnum', '/a/b/myCM', ...
    '/AUTOSAR_PlatformTypes/ImplementationDataTypes/uint16');
```

## Input Arguments

### **arProps** — AUTOSAR properties information for a model

handle (default)

AUTOSAR properties information for a model, previously returned by `arProps = autosar.api.getAUTOSARProperties(model)`. The parameter `model` is a handle, character vector, or string scalar representing the model name.

Example: `arProps`

Data Types: `function_handle`

### **name** — Name of Simulink enumeration data type

character vector (default) | string scalar

Name of enumeration data type created for Simulink representation of an AUTOSAR element.

In the Simulink environment, this enumeration data type is mapped to both an application data type and an implementation data type. The application data type for the enumeration provides application-level physical attributes such as real-world range of values, data

structure, and physical semantics. The implementation data type provides implementation-level attributes, such as stored-integer minimum and maximum specifications and primitive type (for example, integer).

Example: 'myEnum'

Data Types: char | string

### **applicationDataTypePath — Path to enumeration application data type**

character vector (default) | string scalar

Path to application data type for created Simulink enumeration data type. The application data type for the enumeration provides application-level physical attributes such as real-world range of values, data structure, and physical semantics. The application data type is used in simulation.

Example: '/AUTOSAR\_PlatformTypes/ApplicationDataTypes/MyAppType'

Data Types: char | string

### **compuMethodPath — Path to CompuMethod used to convert enumeration data types**

character vector (default) | string scalar

Path to the CompuMethod. This method converts the enumeration implementation data type to the enumeration application data type.

Example: '/a/b/myCM'

Data Types: char | string

### **implementationDataTypePath — Path to enumeration implementation data type**

character vector (default) | string scalar

Path to Simulink enumeration implementation data type. The implementation data type provides implementation-level attributes, such as stored-integer minimum and maximum specifications and primitive type (for example, integer). Implementation data types are used in code generation.

Example: '/AUTOSAR\_PlatformTypes/ImplementationDataTypes/uint16'

Data Types: char | string

## **See Also**

createNumericType

## **Topics**

“AUTOSAR Component Configuration”

“Model AUTOSAR Data Types”

## createNumericType

**Package:** autosar.api

Create Simulink numeric data type definition to work with imported AUTOSAR element

### Syntax

```
createNumericType(arProps, name, applicationDataTypePath)
createNumericType(arProps, name, compuMethodPath,
implementationDataTypePath)
```

### Description

`createNumericType(arProps, name, applicationDataTypePath)` creates a `Simulink.NumericType` object from its application data type that can be used to work with imported AUTOSAR elements.

`createNumericType(arProps, name, compuMethodPath, implementationDataTypePath)` creates a `Simulink.NumericType` object from its implementation data type and `CompuMethod`.

### Examples

#### Create a Numeric Data Type from its Application Data Type

Create a Simulink numeric data type with the name `myDataType` from its application data type at path `'/AUTOSAR_PlatformTypes/ApplicationDataTypes/MyAppType'`.

```
dataObj = ...
    autosar.api.getAUTOSARProperties mdlName);
createNumericType(dataObj, 'myDataType', ...
    '/AUTOSAR_PlatformTypes/ApplicationDataTypes/MyAppType');
```

## Create a Numeric Data Type from its Implementation Data Type and CompuMethod

Create a Simulink numeric data type with the name `myDataType` from the implementation data type at path `'/AUTOSAR_PlatformTypes/ImplementationDataTypes/uint32'` by using the computation method from path `'/a/b/myCM'`.

```
dataObj = ...
    autosar.api.getAUTOSARProperties mdlName);
    createNumericType(dataObj, 'myDataType', '/a/b/myCM', ...
        '/AUTOSAR_PlatformTypes/ImplementationDataTypes/uint32');
```

## Input Arguments

### **arProps** — AUTOSAR properties information for a model

handle (default)

AUTOSAR properties information for a model, previously returned by `arProps = autosar.api.getAUTOSARProperties(model)`. The parameter `model` is a handle, character vector, or string scalar representing the model name.

Example: `arProps`

Data Types: `function_handle`

### **name** — Name of Simulink numeric data type

character vector (default) | string scalar

Name of numeric data type created for Simulink representation of an AUTOSAR element.

In the Simulink environment, this numeric data type is mapped to both an application data type and an implementation data type. The application data type provides application-level physical attributes such as real-world range of values, data structure, and physical semantics. The implementation data type provides implementation-level attributes, such as stored-integer minimum and maximum specifications and primitive type (for example, integer).

Example: `'myDataType'`

Data Types: `char` | `string`

## **applicationDataTypePath — Path to numeric application data type**

character vector (default) | string scalar

Path to application data type for created Simulink numeric data type. The application data type provides application-level physical attributes such as real-world range of values, data structure, and physical semantics. The application data type is used in simulation.

Example: '/AUTOSAR\_PlatformTypes/ApplicationDataTypes/MyAppType'

Data Types: char | string

## **compuMethodPath — Path to CompuMethod used to convert numeric data types**

character vector (default) | string scalar

Path to the CompuMethod. This method converts the numeric implementation data type to the numeric application data type.

Example: '/a/b/myCM'

Data Types: char | string

## **implementationDataTypePath — Path to numeric implementation data type**

character vector (default) | string scalar

Path to Simulink numeric implementation data type. The numeric data type provides implementation-level attributes, such as stored-integer minimum and maximum specifications and primitive type (for example, integer). Implementation data types are used in code generation.

Example: '/AUTOSAR\_PlatformTypes/ImplementationDataTypes/uint32'

Data Types: char | string

## **See Also**

createEnumeration

## **Topics**

“Configure and Map AUTOSAR Component Programmatically”

“AUTOSAR Component Configuration”



# autosar.api.delete

Delete AUTOSAR properties and mapping for Simulink model

## Syntax

```
autosar.api.delete(model)
```

## Description

`autosar.api.delete(model)` deletes AUTOSAR properties and Simulink to AUTOSAR mapping for `model`. The resulting model does not represent and map an AUTOSAR software component.

## Examples

### Remove AUTOSAR Component Representation from Model

Delete AUTOSAR properties and Simulink to AUTOSAR mapping for a model.

```
hModel = 'autosar_sw_counter';  
open_system(hModel);  
autosar.api.delete(hModel);
```

## Input Arguments

**model** — Model for which to delete AUTOSAR properties and mapping

handle | character vector | string scalar

Model for which to delete AUTOSAR properties and Simulink to AUTOSAR mapping, specified as a handle, character vector, or string scalar representing the model name.

Example: 'my\_model'

## **See Also**

`autosar.api.create`

## **Topics**

“AUTOSAR Component Configuration”

**Introduced in R2017b**

# autosar.api.getAUTOSARProperties

Configure AUTOSAR software component elements and properties

## Description

In an AUTOSAR software component model, use AUTOSAR property functions to configure AUTOSAR elements from an AUTOSAR component perspective. You can add AUTOSAR elements, find elements, get and set properties of elements, delete elements, and define arxml packaging of elements.

## Creation

## Syntax

```
arProps = autosar.api.getAUTOSARProperties(model)
```

## Description

`arProps = autosar.api.getAUTOSARProperties(model)` creates object `arProps`, which represents AUTOSAR properties information for `model`. The specified model must be open.

## Input Arguments

**model** — AUTOSAR model

handle | character vector | string scalar

Model for which to create AUTOSAR properties object, specified as a handle, character vector, or string scalar representing the model name.

Example: 'my\_model'

## Object Functions

add	Add property to AUTOSAR element
addPackageableElement	Add element to AUTOSAR package in model
createEnumeration	Create Simulink enumeration data type definition to work with imported AUTOSAR element
createNumericType	Create Simulink numeric data type definition to work with imported AUTOSAR element
delete	Delete AUTOSAR element
deleteUnmappedComponents	Delete unmapped AUTOSAR components from model
find	Find AUTOSAR elements
get	Get property of AUTOSAR element
set	Set property of AUTOSAR element

## Examples

### Create AUTOSAR Properties Object and Set IsService Property

Call the `autosar.api.getAUTOSARProperties` function to create object `arProps`, which represents AUTOSAR properties information for model `autosar_swc_slfcns`. Use the returned object to set the `IsService` property for client-server interface `CSIf` to `true` (1), indicating that the port interface is used for AUTOSAR services.

```
hModel = 'autosar_swc_slfcns';
open_system(hModel);
arProps = autosar.api.getAUTOSARProperties(hModel);
set(arProps, 'CSIf', 'IsService', true);
isService = get(arProps, 'CSIf', 'IsService')

isService =
    logical
     1
```

## See Also

### Topics

“Configure and Map AUTOSAR Component Programmatically”  
“AUTOSAR Property and Map Function Examples”  
“AUTOSAR Component Configuration”

**Introduced in R2013b**

## **autosar.api.getSimulinkMapping**

Map Simulink elements to AUTOSAR elements

### **Description**

In an AUTOSAR software component model, use AUTOSAR map functions to map model elements to AUTOSAR component elements from a Simulink model perspective. For example, you can:

- Map a Simulink inport or outport to an AUTOSAR receiver or sender port and a sender-receiver data element.
- Map a Simulink entry-point function to an AUTOSAR runnable and optional software address methods.
- Map a Simulink data transfer line to an AUTOSAR inter-runnable variable (IRV).
- Map a Simulink function caller to an AUTOSAR client port and a client-server operation.
- Map a Simulink model workspace parameter to an AUTOSAR component internal parameter.
- Map a Simulink block signal or state to an AUTOSAR variable.
- Map a Simulink data store to an AUTOSAR variable.

### **Creation**

### **Syntax**

```
sLMap = autosar.api.getSimulinkMapping(model)
```

### **Description**

`sLMap = autosar.api.getSimulinkMapping(model)` creates object `sLMap`, which represents AUTOSAR mapping information for `model`. The specified model must be open.

## Input Arguments

### **model** — AUTOSAR model

handle | character vector | string scalar

Model for which to create AUTOSAR mapping object, specified as a handle, character vector, or string scalar representing the model name.

Example: 'my\_model'

## Object Functions

getDataStore	Get AUTOSAR mapping information for Simulink data store
getDataTransfer	Get AUTOSAR mapping information for Simulink data transfer
getFunction	Get AUTOSAR mapping information for Simulink entry-point function
getFunctionCaller	Get AUTOSAR mapping information for Simulink function-caller block
getInport	Get AUTOSAR mapping information for Simulink inport
getOutport	Get AUTOSAR mapping information for Simulink outport
getParameter	Get AUTOSAR mapping information for Simulink model workspace parameter
getSignal	Get AUTOSAR mapping information for Simulink block signal
getState	Get AUTOSAR mapping information for Simulink block state
mapDataStore	Map Simulink data store to AUTOSAR variable
mapDataTransfer	Map Simulink data transfer to AUTOSAR inter-runnable variable
mapFunction	Map Simulink entry-point function to AUTOSAR runnable and software address methods
mapFunctionCaller	Map Simulink function-caller block to AUTOSAR client port and operation
mapInport	Map Simulink inport to AUTOSAR port
mapOutport	Map Simulink outport to AUTOSAR port
mapParameter	Map Simulink model workspace parameter to AUTOSAR component internal parameter
mapSignal	Map Simulink block signal to AUTOSAR variable
mapState	Map Simulink block state to AUTOSAR variable

## Examples

## Create AUTOSAR Mapping Object and Map Entry-Point Function to AUTOSAR Runnable

Call the `autosar.api.getSimulinkMapping` function to create object `slMap`, which represents AUTOSAR mapping information for model `autosar_sw.c`. Use the returned object to map the Simulink initialize entry-point function to AUTOSAR runnable `Runnable_Init`.

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

```
arRunnableName =  
    'Runnable_Init'
```

## See Also

### Topics

“Configure and Map AUTOSAR Component Programmatically”  
“AUTOSAR Property and Map Function Examples”  
“AUTOSAR Component Configuration”

**Introduced in R2013b**



# autosar.api.syncModel

Update Simulink to AUTOSAR mapping of model with Simulink modifications

## Syntax

```
autosar.api.syncModel(model)
```

## Description

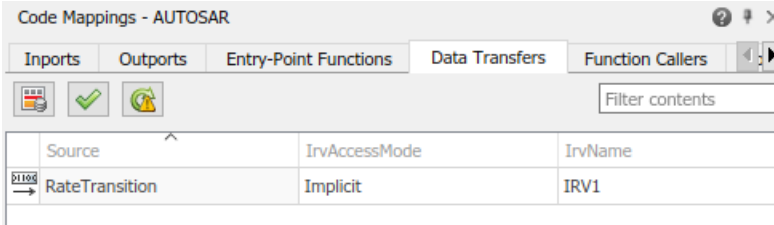
`autosar.api.syncModel(model)` updates the Simulink to AUTOSAR mapping of `model` with modifications made to Simulink elements, such as data transfers, entry-point functions, and function callers.

This function is equivalent to using the **Update** button  in the Code Mappings editor view of an AUTOSAR component model.

## Examples

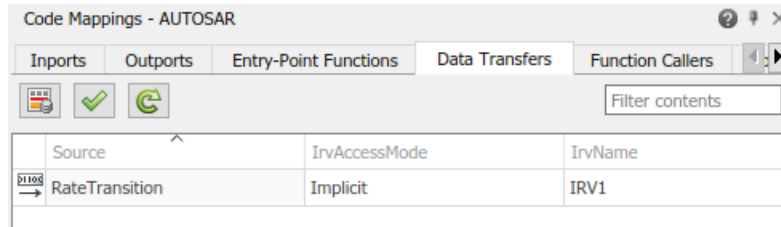
### Update Simulink to AUTOSAR Mapping of Model

When you create or modify an AUTOSAR model, Simulink to AUTOSAR mapping potentially is not current with the model content. For example, the **Update** button in this display indicates that Simulink elements need loading or updating.



Source	IrvAccessMode	IrvName
RateTransition	Implicit	IRV1

This example opens and updates a model. After calling `autosar.api.syncModel`, the Simulink to AUTOSAR mapping reflects the current model content.



```
hModel = 'autosar_sw';  
open_system(hModel);  
autosar.api.syncModel(hModel)
```

## Input Arguments

### **model** — Model to update

handle | character vector | string scalar

Loaded or open model for which to update Simulink to AUTOSAR mapping with model changes, specified as a handle, character vector, or string scalar representing the model name.

Example: 'my\_model'

## See Also

`autosar.api.validateModel`

## Topics

“AUTOSAR Property and Map Function Examples”

“AUTOSAR Component Configuration”

**Introduced in R2016a**

# autosar.api.validateModel

Validate AUTOSAR properties and mapping of Simulink model

## Syntax

```
autosar.api.validateModel(model)
```

## Description

`autosar.api.validateModel(model)` validates the AUTOSAR properties and Simulink to AUTOSAR mapping of `model`.

This function is equivalent to using the **Validate** button  in the Code Mappings editor view of an AUTOSAR component model.

## Examples

### Validate AUTOSAR Properties and Mapping of Model

This example opens a model in which a Simulink inport is not mapped to an AUTOSAR port and data element. Initial validation reports the error and fails. After the inport is mapped, validation succeeds.

```
hModel = 'autosar_model_with_unmapped_port';  
load_system(hModel);  
  
% Initial validation fails  
try  
    autosar.api.validateModel(hModel)  
catch validationErr  
    throw(validationErr)  
end
```

Block 'autosar\_model\_with\_unmapped\_port/Input' is not mapped to an AUTOSAR port element.

```
% Map the unmapped port  
slMap=autosar.api.getSimulinkMapping(hModel);  
mapInport(slMap,'Input','Input','Input','ImplicitReceive');
```

```
% Second validation succeeds  
autosar.api.validateModel(hModel)
```

## Input Arguments

### **model** — Model to validate

handle | character vector | string scalar

Loaded or open model for which to validate AUTOSAR properties and Simulink to AUTOSAR mapping, specified as a handle, character vector, or string scalar representing the model name.

Example: 'my\_model'

## See Also

`autosar.api.syncModel`

## Topics

“AUTOSAR Property and Map Function Examples”

“AUTOSAR Component Configuration”

**Introduced in R2016a**

# autosar\_ui\_close

Close AUTOSAR Dictionary dialog box

## Syntax

```
autosar_ui_close(model)
```

## Description

`autosar_ui_close(model)` closes the AUTOSAR Dictionary dialog box for the specified open model.

## Examples

### Close AUTOSAR Dictionary Dialog Box for Example Model

Open the AUTOSAR Dictionary dialog box with settings for an AUTOSAR example model, and then close the dialog box.

```
hModel = 'autosar_swc';  
open_system(hModel)  
autosar_ui_launch(hModel)  
autosar_ui_close(hModel)
```

## Input Arguments

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

handle | character vector | string scalar

Model for which to close the AUTOSAR Dictionary dialog box, specified as a handle, character vector, or string scalar representing the model name.

Example: 'autosar\_swc'

## **See Also**

`autosar.api.create` | `autosar_ui_launch`

## **Topics**

“AUTOSAR Component Configuration”

**Introduced in R2014b**

# autosar\_ui\_launch

Open AUTOSAR Dictionary dialog box

## Syntax

```
autosar_ui_launch(model)
```

## Description

`autosar_ui_launch(model)` opens the AUTOSAR Dictionary dialog box with settings for the specified open model.

## Examples

### Open AUTOSAR Dictionary Dialog Box for Example Model

Open the AUTOSAR Dictionary dialog box with settings for an AUTOSAR example model.

```
hModel = 'autosar_swc';  
open_system(hModel)  
autosar_ui_launch(hModel)
```

## Input Arguments

**model** — Model for which to open the AUTOSAR Dictionary dialog box

handle | character vector | string scalar

Model for which to open the AUTOSAR Dictionary dialog box, specified as a handle, character vector, or string scalar representing the model name.

Example: 'autosar\_swc'

## **See Also**

`autosar.api.create` | `autosar_ui_close`

## **Topics**

“AUTOSAR Component Configuration”

**Introduced in R2013b**



# createComponentAsModel

**Package:** arxml

Create Simulink representation of AUTOSAR arxml atomic software component

## Syntax

```
createComponentAsModel(ar, ComponentName)  
[mdl, sts] = createComponentAsModel(ar, ComponentName, Name, Value)
```

## Description

`createComponentAsModel(ar, ComponentName)` creates a Simulink model corresponding to AUTOSAR atomic software component `ComponentName`. The component description is part of AUTOSAR information previously imported from AUTOSAR XML files, which is represented by `ar.xml.importer` object `ar`. The importer creates an initial Simulink representation of the imported AUTOSAR component, with an initial, default mapping of Simulink model elements to AUTOSAR component elements. The initial representation provides a starting point for further AUTOSAR configuration and Model-Based Design. For more information, see “AUTOSAR arxml Importer”.

The initial representation of AUTOSAR component behavior in the created model depends on the XML description:

- If the XML description of the component does not describe component behavior, the importer creates a model with a default representation of AUTOSAR runnables and ports.
- If the XML description of the component describes component behavior, the importer creates a model based on AUTOSAR elements that are accessed in the component.

For example, AUTOSAR ports must be accessed by runnables in order to generate the corresponding Simulink elements. If a sender-receiver or client-server port in XML code is not accessed by a runnable, the importer does not create the corresponding inports, outports, or Simulink functions.

`[mdl, sts] = createComponentAsModel(ar, ComponentName, Name, Value)` specifies additional options for Simulink model creation with one or more `Name, Value` pair arguments.

## Examples

### Import AUTOSAR Component and Model Periodic Runnables as Atomic Subsystems

Import AUTOSAR software component `/pkg/swc` from XML file `mySWC.arxml` and create an initial Simulink representation of the component. Model AUTOSAR periodic runnables as atomic subsystems with periodic rates.

```
ar = arxml.importer('mySWC.arxml')
createComponentAsModel(ar, '/pkg/swc', 'ModelPeriodicRunnablesAs', 'AtomicSubsystem')
```

### Import AUTOSAR Component and Model Periodic Runnables as Function-Call Subsystems

Import AUTOSAR software component `/pkg/swc` from XML file `mySWC.arxml` and create an initial Simulink representation of the component. Model AUTOSAR periodic runnables as function-call subsystems with periodic rates.

```
ar = arxml.importer('mySWC.arxml')
createComponentAsModel(ar, '/pkg/swc', 'ModelPeriodicRunnablesAs', 'FunctionCallSubsystem')
```

### Import AUTOSAR Component and Use Data Dictionary

Import AUTOSAR software component `/pkg/swc` from XML file `mySWC.arxml` and create an initial Simulink representation of the component. Place Simulink data objects corresponding to AUTOSAR data types into data dictionary `ardata.sldd`.

```
ar = arxml.importer('mySWC.arxml')
createComponentAsModel(ar, '/pkg/swc', 'ModelPeriodicRunnablesAs', 'AtomicSubsystem', ...
    'DataDictionary', 'ardata.sldd')
```

## Import AUTOSAR Component and Designate Initialization Runnable

Import AUTOSAR software component /pkg/swc from XML file mySWC.arxml and create an initial Simulink representation of the component. Configure AUTOSAR runnable Runnable\_Init as the initialization runnable for the component.

```
ar = arxml.importer('mySWC.arxml')
createComponentAsModel(ar, '/pkg/swc', 'ModelPeriodicRunnablesAs', 'AtomicSubsystem', ...
    'InitializationRunnable', 'Runnable_Init')
```

## Import AUTOSAR Component and Use PredefinedVariant to Resolve Variation Points

Import AUTOSAR software component /pkg/swc from XML file mySWC.arxml and create an initial Simulink representation of the component. Use PredefinedVariant Senior to resolve variation points in the component at model creation time.

```
ar = arxml.importer('mySWC.arxml')
createComponentAsModel(ar, '/pkg/swc', 'ModelPeriodicRunnablesAs', 'AtomicSubsystem', ...
    'PredefinedVariant', '/pkg/body/Variants/Senior');
```

## Import AUTOSAR Component and Use SwSystemconstantValueSets to Resolve Variation Points

Import AUTOSAR software component /pkg/swc from XML file mySWC.arxml and create an initial Simulink representation of the component. Use SwSystemconstantValueSets A and B to resolve variation points in the component at model creation time.

```
ar = arxml.importer('mySWC.arxml')
createComponentAsModel(ar, '/pkg/swc', 'ModelPeriodicRunnablesAs', 'AtomicSubsystem', ...
    'SystemConstValueSets', {'/pkg/body/SystemConstantValues/A', '/pkg/body/SystemConstantValues/B'});
```

## Input Arguments

**ar** — `arxml.importer` object  
object

AUTOSAR information previously imported from XML files, specified as an `arxml.importer` object.

## **ComponentName — Component path**

character vector | string scalar

Absolute short-name path of the atomic software component.

Example:  `'/Company/Powertrain/Components/ASWC '`

## **Name-Value Pair Arguments**

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Example:  `'ModelPeriodicRunnablesAs', 'AtomicSubsystem'`  directs the importer to model AUTOSAR periodic runnables as atomic subsystems with periodic rates.

## **DataDictionary — Simulink data dictionary**

character vector | string scalar

Simulink data dictionary into which to import data objects corresponding to AUTOSAR data types in the XML file. If the specified dictionary does not already exist, the importer creates it. The model is then associated with that data dictionary.

Example:  `'DataDictionary', 'ardata.slidd'`

## **InitializationRunnable — Initialization runnable**

character vector | string scalar

Name of an existing AUTOSAR runnable to select as the initialization runnable for the component.

Example:  `'InitializationRunnable', 'Runnable_Init'`

## **ModelPeriodicRunnablesAs — Subsystem type for periodic runnables**

`'AtomicSubsystem'`  (default) |  `'FunctionCallSubsystem'`  |  `'Auto'`

By default,  `createComponentAsModel`  imports AUTOSAR periodic runnables found in  `arxml`  files and models them as atomic subsystems with periodic rates. If conditions prevent use of atomic subsystems, the importer throws an error.

To model periodic runnables as function-call subsystems with periodic rates, specify  `FunctionCallSubsystem` .

If you specify `Auto`, the importer attempts to model periodic runnables as atomic subsystems. If conditions prevent use of atomic subsystems, the importer models periodic runnables as function-call subsystems.

For more information, see “Import AUTOSAR Software Component with Multiple Runnables”.

Example: `'ModelPeriodicRunnablesAs', 'AtomicSubsystem'`

Data Types: `char`

### **PredefinedVariant — Path to AUTOSAR predefined variant**

`character vector` | `string scalar`

Path to a `PredefinedVariant` defined in the AUTOSAR XML file. A `PredefinedVariant` describes a combination of system constant values among potentially multiple valid combinations to apply to an AUTOSAR software component. Use this property to resolve variation points in the AUTOSAR software component at model creation time. If specified, the importer uses the `PredefinedVariant` to initialize `SwSystemconst` data that serves as input to control variation points.

For more information, see “Control AUTOSAR Variants with Predefined Value Combinations”.

Example: `'PredefinedVariant', '/pkg/body/Variants/Senior'`

### **SystemConstValueSets — Paths to one or more AUTOSAR system constant value sets**

`cell array of character vectors` | `string array`

Paths to one or more `SystemConstValueSets` defined in the AUTOSAR XML file. A `SystemConstValueSet` specifies a set of system constant values to apply to an AUTOSAR software component. Use this property to resolve variation points in the AUTOSAR software component at model creation time. If specified, the importer uses the `SystemConstValueSets` to initialize `SwSystemconst` data that serves as input to control variation points.

For more information, see “Control AUTOSAR Variants with Predefined Value Combinations”.

Example: `'SystemConstValueSets', '{ '/pkg/body/SystemConstantValues/A', '/pkg/body/SystemConstantValues/B' }'`

## Output Arguments

### **mdl** — Model handle

handle

Variable that returns a handle to created model.

### **sts** — Success or failure

true or false

Variable that returns true if the import is successful. Otherwise, returns false.

## Tips

- If you enter the `arxml.importer` object function call without a terminating semicolon (;), the importer lists the AUTOSAR content of the specified XML file or files. The information includes paths to software components in the AUTOSAR package structure, which you can specify in calls to `createComponentAsModel` and `createCompositionAsModel`.
- When importing an AUTOSAR software component into a model, it is recommended that you explicitly specify the `ModelPeriodicRunnablesAs` property. This property determines how the importer models AUTOSAR periodic runnables in the created model. See the property description under “Name-Value Pair Arguments” on page 1-40.

## See Also

`arxml.importer`

## Topics

“Import AUTOSAR Software Component”

“Import AUTOSAR Component to Simulink”

“AUTOSAR arxml Importer”

“Control AUTOSAR Variants with Predefined Value Combinations”

**Introduced in R2008a**

# createCompositionAsModel

**Package:** arxml

Create Simulink representation of AUTOSAR arxml software composition

## Syntax

```
createCompositionAsModel(ar,CompositionName)
[mdl, sts] = createCompositionAsModel(ar,CompositionName,Name,Value)
```

## Description

`createCompositionAsModel(ar,CompositionName)` creates a Simulink model corresponding to AUTOSAR software composition `CompositionName`. The composition description is part of AUTOSAR information previously imported from AUTOSAR XML files, which is represented by `arxml.importer` object `ar`. The importer creates an initial Simulink representation of the imported AUTOSAR composition. The initial representation provides a starting point for further AUTOSAR configuration and Model-Based Design. For more information, see “AUTOSAR arxml Importer”.

`[mdl, sts] = createCompositionAsModel(ar,CompositionName,Name,Value)` specifies additional options for Simulink model creation with one or more `Name, Value` pair arguments.

## Examples

### Import AUTOSAR Composition

Import AUTOSAR software composition `/pkg/rootComposition` from XML file `mySWCs.arxml` and create an initial Simulink representation of the composition.

```
ar = arxml.importer('mySWCs.arxml')
createCompositionAsModel(ar, '/pkg/rootComposition')
```

## Import AUTOSAR Composition and Include Existing Component Models

Import AUTOSAR software composition `/pkg/rootComposition` from XML file `mySWCs.arxml` and create an initial Simulink representation of the composition. For components `mySwc1` and `mySwc2` contained within the composition, use existing Simulink component models rather than creating new ones.

```
ar = arxml.importer('mySWCs.arxml')
createCompositionAsModel(ar, '/pkg/rootComposition', 'ComponentModels', {'mySwc1', 'mySwc2'})
```

## Import AUTOSAR Composition and Use Data Dictionary

Import AUTOSAR software composition `/pkg/rootComposition` from XML file `mySWCs.arxml` and create an initial Simulink representation of the composition. Place Simulink data objects corresponding to AUTOSAR data types into data dictionary `ardata.sldd`.

```
ar = arxml.importer('mySWCs.arxml')
createCompositionAsModel(ar, '/pkg/rootComposition', 'DataDictionary', 'ardata.sldd')
```

## Import AUTOSAR Composition and Model Periodic Runnables as Function-Call Subsystems

Import AUTOSAR software composition `/pkg/rootComposition` from XML file `mySWCs.arxml` and create an initial Simulink representation of the composition. Model AUTOSAR periodic runnables as function-call subsystems with periodic rates.

```
ar = arxml.importer('mySWCs.arxml')
createCompositionAsModel(ar, '/pkg/rootComposition', ...
    'ModelPeriodicRunnablesAs', 'FunctionCallSubsystem')
```

## Import AUTOSAR Composition and Use PredefinedVariant to Resolve Variation Points

Import AUTOSAR software composition `/pkg/rootComposition` from XML file `mySWCs.arxml` and create an initial Simulink representation of the composition. Use `PredefinedVariant Senior` to resolve variation points in components at model creation time.

```
ar = arxml.importer('mySWCs.arxml')
createCompositionAsModel(ar, '/pkg/rootComposition', ...
    'PredefinedVariant', '/pkg/body/Variants/Senior');
```



## Import AUTOSAR Composition and Use SwSystemconstantValueSets to Resolve Variation Points

Import AUTOSAR software composition `/pkg/rootComposition` from XML file `mySWCs.arxml` and create an initial Simulink representation of the composition. Use `SwSystemconstantValueSets` A and B to resolve variation points in components at model creation time.

```
ar = arxml.importer('mySWCs.arxml')
createCompositionAsModel(ar, '/pkg/rootComposition', ...
    'SystemConstValueSets', {'/pkg/body/SystemConstantValues/A', '/pkg/body/SystemConstantValues/B'});
```

## Input Arguments

### **ar** — `arxml.importer` object

object

AUTOSAR information previously imported from XML files, specified as an `arxml.importer` object.

### **CompositionName** — Composition path

character vector | string scalar

Absolute short-name path of the software composition.

Example: `'/Company/Powertrain/Components/RootComposition'`

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Example: `'ModelPeriodicRunnablesAs', 'AtomicSubsystem'` directs the importer to model AUTOSAR periodic runnables as atomic subsystems with periodic rates.

### **ComponentModels** — Simulink component models

cell array of character vectors | string array

Names of existing atomic software component models to use when creating a Simulink representation of the composition. The function incorporates the specified existing component models in the composition model instead of creating new ones.

Example: `'ComponentModels', {'mySwc1', 'mySwc2'}`

### **DataDictionary** — Simulink data dictionary

character vector | string scalar

Simulink data dictionary into which to import data objects corresponding to AUTOSAR data types in the XML file. If the specified dictionary does not already exist, the importer creates it. The model is then associated with that data dictionary.

Example: `'DataDictionary', 'ardata.sldd'`

### **ModelPeriodicRunnablesAs** — Subsystem type for periodic runnables

'Auto' (default) | 'AtomicSubsystem' | 'FunctionCallSubsystem'

By default, `createCompositionAsModel` imports AUTOSAR periodic runnables found in `arxml` files and attempts to model them as atomic subsystems with periodic rates. If conditions prevent use of atomic subsystems, the function models the periodic runnables as function-call subsystems with periodic rates.

To model periodic runnables only as atomic subsystems, specify `AtomicSubsystem`. If conditions prevent use of atomic subsystems, the function throws an error.

To model periodic runnables only as function-call subsystems, specify `FunctionCallSubsystem`.

For more information, see “Import AUTOSAR Software Component with Multiple Runnables”.

Example: `'ModelPeriodicRunnablesAs', 'AtomicSubsystem'`

Data Types: char

### **PredefinedVariant** — Path to AUTOSAR predefined variant

character vector | string scalar

Path to a `PredefinedVariant` defined in the AUTOSAR XML file. A `PredefinedVariant` describes a combination of system constant values among potentially multiple valid combinations to apply to AUTOSAR software components. Use this property to resolve variation points in AUTOSAR software components at model

creation time. If specified, the importer uses the `PredefinedVariant` to initialize `SwSystemconst` data that serves as input to control variation points.

For more information, see “Control AUTOSAR Variants with Predefined Value Combinations”.

Example: `'PredefinedVariant', '/pkg/body/Variants/Senior'`

### **SystemConstValueSets — Paths to one or more AUTOSAR system constant value sets**

cell array of character vectors | string array

Paths to one or more `SystemConstValueSets` defined in the AUTOSAR XML file. A `SystemConstValueSet` specifies a set of system constant values to apply to AUTOSAR software components. Use this property to resolve variation points in AUTOSAR software components at model creation time. If specified, the importer uses the `SystemConstValueSets` to initialize `SwSystemconst` data that serves as input to control variation points.

For more information, see “Control AUTOSAR Variants with Predefined Value Combinations”.

Example: `'SystemConstValueSets', '{'/pkg/body/SystemConstantValues/A', '/pkg/body/SystemConstantValues/B'}'`

## **Output Arguments**

### **mdl — Model handle**

handle

Variable that returns a handle to created model.

### **sts — Success or failure**

true or false

Variable that returns true if the import is successful. Otherwise, returns false.

## **Tip**

If you enter the `arxml.importer` object function call without a terminating semicolon (;), the importer lists the AUTOSAR content of the specified XML file or files. The

information includes paths to software components in the AUTOSAR package structure, which you can specify in calls to `createCompositionAsModel` and `createComponentAsModel`.

### See Also

`arxml.importer`

### Topics

“Import AUTOSAR Software Component”

“Import AUTOSAR Composition to Simulink”

“AUTOSAR arxml Importer”

“Control AUTOSAR Variants with Predefined Value Combinations”

### Introduced in R2017b

# delete

**Package:** autosar.api

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.

```
hModel = 'autosar_swc_expfncs';
open_system(hModel);
arProps = autosar.api.getAUTOSARProperties(hModel);

% Add Interface3
addPackageableElement(arProps,'SenderReceiverInterface','/pkg/if','Interface3');
ifPaths=find(arProps,[],'SenderReceiverInterface','PathType','FullyQualified')

ifPaths =
    {'/pkg/if/Interface1'    {'/pkg/if/Interface2'}    {'/pkg/if/Interface3'}

% Find AUTOSAR DataReceiverPort and change its interface from Interface1 to Interface3
arPortType = 'DataReceiverPort';
aswcPath = find(arProps,[],'AtomicComponent','PathType','FullyQualified');
rPorts=find(arProps,aswcPath{1},arPortType,'PathType','FullyQualified');
rPort=rPorts{1};
set(arProps,rPort,'Interface','Interface3')

% Delete Interface1
```

```
delete(arProps, 'Interface1');
ifPaths=find(arProps,[],'SenderReceiverInterface','PathType','FullyQualified')
ifPaths =
    {'/pkg/if/Interface2'}    {'/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, character vector, or string scalar representing the model name.

Example: `arProps`

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

character vector | string scalar

Path to the AUTOSAR element to delete.

Example: `'Input'`

## See Also

`add`

## Topics

“Configure and Map AUTOSAR Component Programmatically”

“AUTOSAR Component Configuration”

**Introduced in R2013b**

# deleteUnmappedComponents

**Package:** autosar.api

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 a `rxml` code. This function does not remove calibration components.

## Examples

### Remove Unmapped Atomic Software Components From AUTOSAR Model

After importing AUTOSAR information from a `rxml` 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 a `rxml`.

```
arProps=autosar.api.getAUTOSARProperties('my_autosar_model');  
deleteUnmappedComponents(arProps);
```

## 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, character vector, or string scalar representing the model name.

Example: `arProps`

## See Also

`arxml.importer`

## Topics

“Configure and Map AUTOSAR Component Programmatically”

“Import AUTOSAR Software Component”

“AUTOSAR Component Configuration”

**Introduced in R2014b**



# find

**Package:** autosar.api

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.

```
hModel = 'autosar_swc_expcns';
open_system(hModel);
arProps=autosar.api.getAUTOSARProperties(hModel);
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.

```
addpath(fullfile(matlabroot,'/help/toolbox/autosar/examples'));  
hModel = 'mAutosarMsConfigAfter';  
open_system(hModel);  
arProps=autosar.api.getAUTOSARProperties(hModel);  
addPackageableElement(arProps,'ModeSwitchInterface','/pkg/if','Interface3',...  
    'IsService',true);  
ifPaths=find(arProps,[],'ModeSwitchInterface','PathType','FullyQualified')  
  
ifPaths =  
    {'/pkg/if/myMsIf'}    {'/pkg/if/MsIf2'}    {'/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, character vector, or string scalar representing the model name.

Example: `arProps`

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

character vector | string scalar | []

Path specifying the starting point at which to look for the specified type of AUTOSAR elements. [] indicates the root of the component.

Example: []

### **category** — Type of AUTOSAR element

character vector | string scalar

Type of AUTOSAR element for which to return paths.

Example: 'SenderReceiverInterface'

**'PathType', value — Whether the returned paths are fully qualified or partially qualified**

'PartiallyQualified' (default) | 'FullyQualified'

Specify FullyQualified to return fully qualified paths.

Example: 'PathType', 'FullyQualified'

**property, value — Property and value**

name (character vector or string scalar), 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 — Paths to AUTOSAR elements**

cell array of character vectors

Variable that returns paths to AUTOSAR elements.

Example: ifPaths

## See Also

add | delete | get | set

## Topics

“Configure and Map AUTOSAR Component Programmatically”

“AUTOSAR Component Configuration”

**Introduced in R2013b**

## get

**Package:** autosar.api

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.

```
hModel = 'autosar_swc_expfncs';  
open_system(hModel);  
arProps=autosar.api.getAUTOSARProperties(hModel);  
isService=get(arProps,'Interface1','IsService')
```

```
isService =  
    logical  
     0
```

## 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, character vector, or string scalar representing the model name.

Example: `arProps`

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

character vector | string scalar

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

Example: `'Input'`

### **property — Type of property**

character vector | string scalar

Type of property to add for which to return a value, among valid properties for the AUTOSAR element.

Example: `'IsService'`

## **Output Arguments**

### **pValue — Property value or path**

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`

## **Topics**

“Configure and Map AUTOSAR Component Programmatically”

“AUTOSAR Component Configuration”

**Introduced in R2013b**

# getComponentNames

**Package:** arxml

Get AUTOSAR software component names from arxml files

## Syntax

```
names = getComponentNames(ar)
names = getComponentNames(ar, compKind)
```

## Description

`names = getComponentNames(ar)` returns the names of AUTOSAR software components found in the XML files associated with `arxml.importer` object `ar`. By default, the function returns the names of atomic software components, including application, sensor/actuator, complex device driver, ECU abstraction, and service proxy software components.

`names = getComponentNames(ar, compKind)` uses the `compKind` argument to specify the type of software component to return. You can narrow the search to a specific type of atomic software component, such as 'Application' or 'SensorActuator', or specify a nonatomic component, such as 'Composition' or 'Parameter'.

## Examples

### Get AUTOSAR Atomic Software Component Names from arxml Files

Get the names of AUTOSAR atomic software components present in arxml files.

```
addpath(fullfile(matlabroot, 'examples', 'autosarblockset'));
ar = arxml.importer('ThrottlePositionControlComposition.arxml');
names = getComponentNames(ar)

names =
    {'/Company/Components/Controller' }
```

```
{'/Company/Components/ThrottlePositionMonitor' }  
{'/Company/Components/AccelerationPedalPositionSensor' }  
{'/Company/Components/ThrottlePositionActuator' }  
{'/Company/Components/ThrottlePositionSensor' }
```

## Get AUTOSAR Sensor-Actuator Software Component Names from arxml Files

Get the names of AUTOSAR sensor-actuator software components present in arxml files.

```
addpath(fullfile(matlabroot, 'examples', 'autosarblockset'));  
ar = arxml.importer('ThrottlePositionControlComposition.arxml');  
names = getComponentNames(ar, 'SensorActuator')  
  
names =  
    {'/Company/Components/AccelerationPedalPositionSensor'}  
    {'/Company/Components/ThrottlePositionActuator' }  
    {'/Company/Components/ThrottlePositionSensor' }
```

## Input Arguments

### ar — arxml.importer object

object

AUTOSAR information previously imported from XML files, specified as an arxml.importer object.

### compKind — Component type

'Atomic' (default) | 'Application' | 'ComplexDeviceDriver' | 'Composition' | 'EcuAbstraction' | 'Parameter' | 'SensorActuator' | 'ServiceProxy'

Type of software component to return.

## Output Argument

### names — Names array

cell array of character vectors

Variable that returns an array of component names. Each array element is the absolute short-name path of an AUTOSAR software component.

Example: {'/pkg/swc/tpSensor', '/pkg/swc/tpActuator'}



## **See Also**

arxml.importer

## **Topics**

“Import AUTOSAR Software Component”

“AUTOSAR arxml Importer”

**Introduced in R2008a**

## getDataStore

**Package:** autosar.api

Get AUTOSAR mapping information for Simulink data store

### Syntax

```
arValue=getDataStore(slMap,slBlockHandle)  
arValue=getDataStore(slMap,slBlockHandle,arProperty)
```

### Description

`arValue=getDataStore(slMap,slBlockHandle)` returns the type of AUTOSAR variable mapped to Simulink data store memory block `slBlockHandle`. AUTOSAR variable types include `ArTypedPerInstanceMemory` and `StaticMemory`.

`arValue=getDataStore(slMap,slBlockHandle,arProperty)` returns the value of property `arProperty` for the AUTOSAR variable to which the Simulink data store is mapped.

### Examples

#### Get AUTOSAR Mapping Information for Simulink Data Stores

Get AUTOSAR mapping and property information for Simulink data store memory block `Data Store Memory` in example model `autosar_bsw_sensor1`.

```
hModel = 'autosar_bsw_sensor1';  
hBlock = 'autosar_bsw_sensor1/Data Store Memory';  
  
open_system(hModel);  
slMap = autosar.api.getSimulinkMapping(hModel);  
mapDataStore(slMap,hBlock,'ArTypedPerInstanceMemory','NeedsNVRAMAccess','true');  
arMappedTo = getDataStore(slMap,hBlock)  
arNvram = getDataStore(slMap,hBlock,'NeedsNVRAMAccess')
```

```

arMappedTo =
    'ArTypedPerInstanceMemory'

arNvram =
    'true'

```

## 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, character vector, or string scalar representing the model name.

Example: `sLMap`

### **sLBlockHandle** — Simulink data store memory block handle

handle

Name or handle of Simulink data store memory block for which to return AUTOSAR mapping information.

Example: `'autosar_bsw_sensor1/Data Store Memory'`

### **arProperty** — AUTOSAR property

character vector | string scalar

Name of AUTOSAR variable property. Valid property names include `ShortName`, `SwAddrMethod`, `SwCalibrationAccess`, and `DisplayFormat`. For `ArTypedPerInstanceMemory`, you can also specify `NeedsNVRAMAccess`. For `StaticMemory`, you can also specify C type qualifier properties `IsVolatile` or `Qualifier` (AUTOSAR additional native type qualifier). For property descriptions, see `mapDataStore`.

Example: `'SwCalibrationAccess'`

## Output Arguments

### **arValue** — Value of AUTOSAR variable type or property

character vector

Variable that returns either the type of the mapped AUTOSAR variable or the value of a variable property.

Example: `arValue`

## **See Also**

Data Store Memory | `mapDataStore`

## **Topics**

“Map Data Stores to AUTOSAR Variables”

“Configure AUTOSAR Per-Instance Memory”

“Configure AUTOSAR Static Memory”

“Configure and Map AUTOSAR Component Programmatically”

“AUTOSAR Component Configuration”

**Introduced in R2019a**

# getDataTransfer

**Package:** autosar.api

Get AUTOSAR mapping information for Simulink data transfer

## Syntax

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

## Description

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

## Examples

### Get AUTOSAR Mapping Information for Simulink Data Transfer Line

Get AUTOSAR mapping information for a data transfer line in the example model autosar\_swc\_expfncns. The model has data transfer lines named irv1, irv2, irv3, and irv4.

```
hModel = 'autosar_swc_expfncns';  
open_system(hModel);  
slMap=autosar.api.getSimulinkMapping(hModel);  
[arIrvName, arDataAccessMode]=getDataTransfer(slMap, 'irv4')  
  
arIrvName =  
    'IRV4'  
arDataAccessMode =  
    'Implicit'
```

## Get AUTOSAR Mapping Information for Rate Transition Block

Get AUTOSAR mapping information for a Rate Transition block in the example model `mMultitasking_4rates`. The model has Rate Transition blocks named `RateTransition`, `RateTransition1`, and `RateTransition2`, which are located at the top level of the model.

```
addpath(fullfile(matlabroot, '/help/toolbox/autosar/examples'));
hModel = 'mMultitasking_4rates';
open_system(hModel);
slMap=autosar.api.getSimulinkMapping(hModel);
[arIrvName,arDataAccessMode]=getDataTransfer(slMap,'mMultitasking_4rates/RateTransition')

arIrvName =
    'IRV1'
arDataAccessMode =
    'Implicit'
```

## 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, character vector, or string scalar representing the model name.

Example: `slMap`

### **slDataTransfer** — Simulink data transfer line name or Rate Transition full block path

character vector | string scalar

Name of the Simulink data transfer line or full block path to the Rate Transition block for which to return AUTOSAR mapping information.

Example: `'irv4'`

Example: `'myModel/RateTransition2'`

## Output Arguments

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

character vector

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

Example: `arIrvName`

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

character vector

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

Example: `arDataAccessMode`

## **See Also**

`mapDataTransfer`

## **Topics**

“Configure and Map AUTOSAR Component Programmatically”

“AUTOSAR Component Configuration”

**Introduced in R2013b**

# getFunction

**Package:** autosar.api

Get AUTOSAR mapping information for Simulink entry-point function

## Syntax

```
arRunnableName = getFunction(slMap,slFcnName)
[arRunnableName,arRunnableSwAddrMethod,arInternalDataSwAddrMethod] =
getFunction(slMap,slFcnName)
```

## Description

`arRunnableName = getFunction(slMap,slFcnName)` returns the name of the AUTOSAR runnable `arRunnableName` mapped to Simulink entry-point function `slFcnName`.

`[arRunnableName,arRunnableSwAddrMethod,arInternalDataSwAddrMethod] = getFunction(slMap,slFcnName)` returns the names of function and internal data software address methods (`SwAddrMethods`) defined for the mapped AUTOSAR runnable. If a `SwAddrMethod` is not defined, the function returns '`<None>`'.

## Examples

### Get AUTOSAR Runnable Name for Simulink Entry-Point Function

Get the name of the AUTOSAR runnable mapped to a Simulink entry point function in the example model `autosar_sw.c`. The model has an initialize entry-point function named `Runnable_Init` and rate-based entry-point functions named `Runnable_1s` and `Runnable_2s`.

```
hModel = 'autosar_sw.c';
open_system(hModel);
slMap=autosar.api.getSimulinkMapping(hModel);
arRunnableName=getFunction(slMap,'InitializeFunction')
```



```
arRunnableName =
    'Runnable_Init'
```

## Get AUTOSAR SwAddrMethod Names for Simulink Entry-Point Function

Get AUTOSAR SwAddrMethod names for a Simulink entry point function in the example model `autosar_swc_counter`. The model has a rate-based entry-point step function.

```
hModel = 'autosar_swc_counter';
open_system(hModel);

% Add SwAddrMethods myCODE and myVAR to the AUTOSAR component
arProps = autosar.api.getAUTOSARProperties(hModel);
addPackageableElement(arProps, 'SwAddrMethod', ...
    '/Company/Powertrain/DataTypes/SwAddrMethods', 'myCODE', ...
    'SectionType', 'Code')
swAddrPaths = find(arProps, [], 'SwAddrMethod', 'PathType', 'FullyQualified', ...
    'SectionType', 'Code')
addPackageableElement(arProps, 'SwAddrMethod', ...
    '/Company/Powertrain/DataTypes/SwAddrMethods', 'myVAR', ...
    'SectionType', 'Var')
swAddrPaths = find(arProps, [], 'SwAddrMethod', 'PathType', 'FullyQualified', ...
    'SectionType', 'Var')

% Set code generation parameter for runnable internal data SwAddrMethods
set_param(hModel, 'GroupInternalDataByFunction', 'on')

% Map step runnable function and internal data to myCODE and myVAR SwAddrMethods
slMap = autosar.api.getSimulinkMapping(hModel);
mapFunction(slMap, 'StepFunction', 'Runnable_Step', ...
    'SwAddrMethod', 'myCODE', 'SwAddrMethodForInternalData', 'myVAR')

% Return AUTOSAR mapping information for step function
[arRunnableName, arRunnableSwAddrMethod, arInternalDataSwAddrMethod] = ...
    getFunction(slMap, 'StepFunction')

swAddrPaths =
    {'/Company/Powertrain/DataTypes/SwAddrMethods/CODE'}
    {'/Company/Powertrain/DataTypes/SwAddrMethods/myCODE'}

swAddrPaths =
    {'/Company/Powertrain/DataTypes/SwAddrMethods/VAR'}
    {'/Company/Powertrain/DataTypes/SwAddrMethods/myVAR'}

arRunnableName =
    'Runnable_Step'

arRunnableSwAddrMethod =
    'myCODE'
```

```
arInternalDataSwAddrMethod =  
    'myVAR'
```

## 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, character vector, or string scalar representing the model name.

Example: `sLMap`

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

character vector | string scalar

Name of the Simulink entry point function for which to return AUTOSAR mapping information, specified as follows:

- For an initialize function, 'InitializeFunction'.
- For a reset function, a model-wide reset event name. For example, 'reset'.
- For a terminate function, 'TerminateFunction'.
- For a rate-based function, 'StepFunction' for the base-rate task or 'StepFunctionN' for a substrate task, where *N* is the task identifier.
- For an exported function, 'FunctionCallName', where *FunctionCallName* is the name of the Inport block that drives the control port of the function-call subsystem. For example, 'Trigger\_1s' in example model `autosar_swc_slfcns` or 'FunctionTrigger' in example model `autosar_swc_fcncalls`.
- For a global Simulink function in a client-server configuration, 'SLFunctionName', where *SLFunctionName* is the name of a Simulink function for which Trigger block parameter **Function visibility** is set to `global`. For example, 'readData' in the example model used in “Configure AUTOSAR Server”.

Example: 'StepFunction2'

## Output Arguments

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

character vector

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

Example: `arRunnableName`

### **arRunnableSwAddrMethod — Name of function SwAddrMethod**

character vector

Variable that returns the name of the `SwAddrMethod` defined for the AUTOSAR runnable function.

Example: `arRunnableSwAddrMethod`

### **arInternalDataSwAddrMethod — Name of internal data SwAddrMethod**

character vector

Variable that returns the name of the `SwAddrMethod` defined for the AUTOSAR runnable internal data.

Example: `arInternalDataSwAddrMethod`

## See Also

`mapFunction`

## Topics

“Configure and Map AUTOSAR Component Programmatically”

“AUTOSAR Component Configuration”

**Introduced in R2013b**

# getFunctionCaller

**Package:** autosar.api

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`.

```
addpath(fullfile(matlabroot,'/help/toolbox/autosar/examples'));
hModel = 'mControllerWithInterface_client';
open_system(hModel);
slMapC = autosar.api.getSimulinkMapping(hModel);
mapFunctionCaller(slMapC,'readData','cPort','readData');
[arPort,arOp] = getFunctionCaller(slMapC,'readData')

arPort =
cPort
```

```
arOp =  
readData
```

## 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, character vector, or string scalar representing the model name.

Example: `sLMap`

### **sLFcnName** — Name of Simulink function

character vector | string scalar

Name of the Simulink function for the function-caller block for which to return AUTOSAR mapping information.

Example: `'readData'`

## Output Arguments

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

character vector

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

Example: `arPort`

### **arOperationName** — Name of AUTOSAR operation

character vector

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

Example: `arOp`

## **See Also**

mapFunctionCaller

## **Topics**

“Configure and Map AUTOSAR Component Programmatically”  
“AUTOSAR Component Configuration”

**Introduced in R2014b**

# getInport

**Package:** autosar.api

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 autosar\_swc\_expfncns. The model has an inport named RPort\_DE1.

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

```
arPortName =  
RPort
```

```
arDataElementName =  
DE1
```

`arDataAccessMode =  
ImplicitReceive`

## 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, character vector, or string scalar representing the model name.

Example: `slMap`

### **slPortName** — Name of model inport

character vector | string scalar

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

Example: `'Input'`

## Output Arguments

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

character vector

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

Example: `arPortName`

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

character vector

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

Example: `arDataElementName`

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

character vector



Variable that returns the value of the AUTOSAR data access mode mapped to the specified Simulink inport. The value can be `ImplicitReceive`, `ExplicitReceive`, `QueuedExplicitReceive`, `ErrorStatus`, `ModeReceive`, `IsUpdated`, `EndToEndRead`, or `ExplicitReceiveByVal`

Example: `arDataAccessMode`

## See Also

`mapInport`

## Topics

[“Configure and Map AUTOSAR Component Programmatically”](#)  
[“AUTOSAR Component Configuration”](#)

**Introduced in R2013b**

## getOutputport

**Package:** autosar.api

Get AUTOSAR mapping information for Simulink outputport

## Syntax

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

## Description

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

## Examples

### Get AUTOSAR Mapping Information for Model Outputport

Get AUTOSAR mapping information for a model outputport in the example model autosar\_swc\_expfcns. The model has an outputport named PPort\_DE1.

```
hModel = 'autosar_swc_expfcns';  
open_system(hModel);  
slMap=autosar.api.getSimulinkMapping(hModel);  
[arPortName,arDataElementName,arDataAccessMode]=getOutputport(slMap,'PPort_DE1')  
  
arPortName =  
PPort  
  
arDataElementName =  
DE1
```

```
arDataAccessMode =  
ImplicitSend
```

## 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, character vector, or string scalar representing the model name.

Example: `sLMap`

### **sLPortName** — Name of model output

character vector | string scalar

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

Example: `'Output'`

## Output Arguments

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

character vector

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

Example: `arPortName`

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

character vector

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

Example: `arDataElementName`

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

character vector

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

Example: `arDataAccessMode`

## **See Also**

`mapOutput`

## **Topics**

*“Configure and Map AUTOSAR Component Programmatically”*

*“AUTOSAR Component Configuration”*

**Introduced in R2013b**

# getParameter

**Package:** autosar.api

Get AUTOSAR mapping information for Simulink model workspace parameter

## Syntax

```
arValue=getParameter(sLMap,sLParameter)
arValue=getParameter(sLMap,sLParameter,arProperty)
```

## Description

`arValue=getParameter(sLMap,sLParameter)` returns the type of AUTOSAR parameter mapped to Simulink model workspace parameter `sLParameter`. AUTOSAR parameter types include `SharedParameter`, `PerInstanceParameter`, and `ConstantMemory`.

`arValue=getParameter(sLMap,sLParameter,arProperty)` returns the value of property `arProperty` for the AUTOSAR parameter to which model workspace parameter `sLParameter` is mapped.

## Examples

### Get AUTOSAR Mapping Information for Simulink Model Workspace Parameters

Get AUTOSAR mapping and property information for Simulink model workspace parameters `K` and `INC` in example model `autosar_swc_counter`.

```
hModel = 'autosar_swc_counter';
open_system(hModel);
sLMap = autosar.api.getSimulinkMapping(hModel);

mapParameter(sLMap,'K','SharedParameter')
arMappedTo = getParameter(sLMap,'K')
arValue = getParameter(sLMap,'K','SwCalibrationAccess')
```

```
mapParameter(sLMMap, 'INC', 'ConstantMemory', 'SwCalibrationAccess', 'ReadOnly')
arMappedTo = getParameter(sLMMap, 'INC')
arValue = getParameter(sLMMap, 'INC', 'SwCalibrationAccess')

arMappedTo =
    'SharedParameter'

arValue =
    'ReadWrite'

arMappedTo =
    'ConstantMemory'

arValue =
    'ReadOnly'
```

## Input Arguments

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

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

Example: `sLMMap`

**sLParameter** — Simulink model workspace parameter character vector | string scalar

Name of Simulink model workspace parameter for which to return AUTOSAR mapping information.

Example: `'INC'`

**arProperty** — AUTOSAR property character vector | string scalar

Name of AUTOSAR parameter property. Valid property names include `SwAddrMethod`, `SwCalibrationAccess`, and `DisplayFormat`. For `ConstantMemory`, you can also specify C type qualifier properties `IsConst`, `IsVolatile`, or `Qualifier` (AUTOSAR additional native type qualifier). For property descriptions, see `mapParameter`.

Example: `'SwCalibrationAccess'`

## Output Arguments

### **arValue** — Value of AUTOSAR parameter type or property

character vector

Variable that returns either the type of the mapped AUTOSAR component internal parameter or the value of a parameter property.

Example: arValue

## See Also

mapParameter

## Topics

“Configure and Map AUTOSAR Component Programmatically”

“AUTOSAR Component Configuration”

**Introduced in R2018b**

## getSignal

**Package:** autosar.api

Get AUTOSAR mapping information for Simulink block signal

### Syntax

```
arValue=getSignal(slMap,slPortHandle)
arValue=getSignal(slMap,slPortHandle,arProperty)
```

### Description

`arValue=getSignal(slMap,slPortHandle)` returns the type of AUTOSAR variable mapped to the named or test-pointed Simulink block signal associated with output port handle `slPortHandle`. AUTOSAR variable types include `ArTypedPerInstanceMemory` and `StaticMemory`.

`arValue=getSignal(slMap,slPortHandle,arProperty)` returns the value of property `arProperty` for the AUTOSAR variable to which the Simulink block signal is mapped.

### Examples

#### Get AUTOSAR Mapping Information for Simulink Block Signals

Get AUTOSAR mapping and property information for the Simulink block signals for blocks `RelOpt` and `Sum` in example model `autosar_sw_counter`.

```
hModel = 'autosar_sw_counter';
open_system(hModel);
slMap = autosar.api.getSimulinkMapping(hModel);

portHandles = get_param('autosar_sw_counter/RelOpt','portHandles');
outportHandle = portHandles.Outport;
mapSignal(slMap,outportHandle,'StaticMemory')
arMappedTo = getSignal(slMap,outportHandle)
```



```

arValue = getSignal(slMap,outportHandle,'SwCalibrationAccess')

portHandles = get_param('autosar_sw_counter/Sum','portHandles');
outportHandle = portHandles.Outport;
mapSignal(slMap,outportHandle,'ArTypedPerInstanceMemory',...
'SwCalibrationAccess','ReadWrite')
arMappedTo = getSignal(slMap,outportHandle)
arValue = getSignal(slMap,outportHandle,'SwCalibrationAccess')

arMappedTo =
    'StaticMemory'

arValue =
    'ReadOnly'

arMappedTo =
    'ArTypedPerInstanceMemory'

arValue =
    'ReadWrite'

```

## 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, character vector, or string scalar representing the model name.

Example: `slMap`

**slPortHandle** — Simulink outpost port handle for a block signal  
handle

Outpost port handle for a named or test-pointed Simulink block signal for which to return AUTOSAR mapping information. Use MATLAB® commands to construct the outpost port handle. For example, for a Relational Operator block named `RelOpt`:

```

portHandles = get_param('autosar_sw_counter/RelOpt','portHandles');
outportHandle = portHandles.Outport;

```

Example: `outportHandle`

**arProperty** — AUTOSAR property  
character vector | string scalar

Name of AUTOSAR variable property. Valid property names include `ShortName`, `SwAddrMethod`, `SwCalibrationAccess`, and `DisplayFormat`. For `StaticMemory`, you can also specify C type qualifier properties `IsVolatile` or `Qualifier` (AUTOSAR additional native type qualifier). For property descriptions, see `mapSignal`.

Example: `'SwCalibrationAccess'`

## Output Arguments

**arValue** — Value of AUTOSAR variable type or property

character vector

Variable that returns either the type of the mapped AUTOSAR variable or the value of a variable property.

Example: `arValue`

## See Also

`mapSignal`

## Topics

“Configure and Map AUTOSAR Component Programmatically”

“AUTOSAR Component Configuration”

**Introduced in R2018b**

# getState

**Package:** autosar.api

Get AUTOSAR mapping information for Simulink block state

## Syntax

```
arValue=getState(slMap,slStateOwnerBlock)
arValue=getState(slMap,slStateOwnerBlock,slState)
arValue=getState(slMap,slStateOwnerBlock,slState,arProperty)
```

## Description

`arValue=getState(slMap,slStateOwnerBlock)` returns the type of AUTOSAR variable mapped to the Simulink block state associated with state owner block `slStateOwnerBlock`. AUTOSAR variable types include `ArTypedPerInstanceMemory` and `StaticMemory`.

`arValue=getState(slMap,slStateOwnerBlock,slState)` returns the type of AUTOSAR variable mapped to Simulink state `slState` associated with state owner block `slStateOwnerBlock`. Specify a nonempty `slState` argument only for blocks with multiple states.

`arValue=getState(slMap,slStateOwnerBlock,slState,arProperty)` returns the value of property `arProperty` for the AUTOSAR variable to which the Simulink block state is mapped.

## Examples

### Get AUTOSAR Mapping Information for Simulink Block State

Get AUTOSAR mapping and property information for the Simulink block state for Unit Delay block X in example model `autosar_sw_counter`. The state owner block has one state.

```
hModel = 'autosar_sw_counter';
open_system(hModel);
slMap = autosar.api.getSimulinkMapping(hModel);

mapState(slMap, 'autosar_sw_counter/X', '', 'ArTypedPerInstanceMemory', ...
    'SwCalibrationAccess', 'ReadWrite')
arMappedTo = getState(slMap, 'autosar_sw_counter/X')
arValue = getState(slMap, 'autosar_sw_counter/X', '', 'SwCalibrationAccess')

arMappedTo =
    'ArTypedPerInstanceMemory'

arValue =
    'ReadWrite'
```

## 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, character vector, or string scalar representing the model name.

Example: `slMap`

### **slStateOwnerBlock** — Simulink state owner block handle or path

handle | character vector | string scalar

Handle or path to Simulink state owner block for which to return AUTOSAR mapping information.

Example: `'autosar_sw_counter/X'`

### **slState** — Simulink state

character vector | string scalar

Name of Simulink state associated with state owner block `slStateOwnerBlock`. Specify a nonempty state name only for blocks with multiple states. If `slState` is empty, the function returns mapping information for the first state in the block.

Example: `''`

### **arProperty** — AUTOSAR property

character vector | string scalar

Name of AUTOSAR variable property. Valid property names include `ShortName`, `SwAddrMethod`, `SwCalibrationAccess`, and `DisplayFormat`. For `StaticMemory`, you can also specify C type qualifier properties `IsVolatile` or `Qualifier` (AUTOSAR additional native type qualifier). For property descriptions, see `mapState`.

Example: `'SwCalibrationAccess'`

## Output Arguments

**arValue** — Value of AUTOSAR variable type or property

character vector

Variable that returns either the type of the mapped AUTOSAR variable or the value of a variable property.

Example: `arValue`

## See Also

`mapState`

## Topics

“Configure and Map AUTOSAR Component Programmatically”

“AUTOSAR Component Configuration”

**Introduced in R2018b**

## mapDataStore

**Package:** autosar.api

Map Simulink data store to AUTOSAR variable

### Syntax

```
mapDataStore(slMap, slBlockHandle, arVarType)
mapDataStore(slMap, slBlockHandle, arVarType, Name, Value)
```

### Description

`mapDataStore(slMap, slBlockHandle, arVarType)` maps Simulink data store memory block `slBlockHandle` to an AUTOSAR variable of type `arVarType` for AUTOSAR run-time calibration. AUTOSAR variable types include `ArTypedPerInstanceMemory` and `StaticMemory`.

`mapDataStore(slMap, slBlockHandle, arVarType, Name, Value)` specifies additional properties for an AUTOSAR `ArTypedPerInstanceMemory` or `StaticMemory` variable by using one or more `Name, Value` pair arguments.

### Examples

#### Set AUTOSAR Mapping Information for Simulink Data Stores

Set AUTOSAR mapping and property information for Simulink data store memory block `Data Store Memory` in example model `autosar_bsw_sensor1`.

```
hModel = 'autosar_bsw_sensor1';
hBlock = 'autosar_bsw_sensor1/Data Store Memory';

open_system(hModel);
slMap = autosar.api.getSimulinkMapping(hModel);
mapDataStore(slMap, hBlock, 'ArTypedPerInstanceMemory', 'NeedsNVRAMAccess', 'true');
arMappedTo = getDataStore(slMap, hBlock)
arNvram = getDataStore(slMap, hBlock, 'NeedsNVRAMAccess')
```

```

arMappedTo =
    'ArTypedPerInstanceMemory'

arNvram =
    'true'

```

## 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, character vector, or string scalar representing the model name.

Example: `sLMap`

### **sLBlockHandle** — Simulink data store memory block handle

handle

Name or handle of Simulink data store memory block for which to set AUTOSAR mapping information.

Example: `'autosar_bsw_sensor1/Data Store Memory'`

### **arVarType** — Type of AUTOSAR variable

character vector | string scalar

Type of AUTOSAR variable to which to map the specified Simulink data store. Valid AUTOSAR variable types include `ArTypedPerInstanceMemory`, `StaticMemory`, and `Auto`. To accept software mapping defaults, specify `Auto`.

Example: `'StaticMemory'`

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Example: `'SwCalibrationAccess', 'ReadWrite'` specifies read-write access to the variable for run-time calibration.

## **DisplayFormat — Calibration display format**

character vector | string scalar

Specify display format for the AUTOSAR variable. AUTOSAR display format specifications control the width and precision display for measurement and calibration data. For more information, see “Configure DisplayFormat”.

Example: 'DisplayFormat', '%2.6f'

## **IsVolatile — C volatile type qualifier flag (StaticMemory only)**

character vector | string scalar

Specify whether to include C type qualifier `volatile` in generated code for the AUTOSAR variable.

Example: 'IsVolatile', 'true'

## **NeedsNVRAMAccess — NeedsNVRAMAccess flag (ArTypedPerInstanceMemory only)**

character vector | string scalar

Specify whether the AUTOSAR variable needs access to nonvolatile RAM on a processor. Specify `true` to configure the per-instance memory to be a mirror block for a specific NVRAM block.

Example: 'NeedsNVRAMAccess', 'true'

## **Qualifier — C AdditionalNativeTypeQualifier flag (StaticMemory only)**

character vector | string scalar

Optionally specify an AUTOSAR additional native type qualifier to include in generated code for the AUTOSAR variable.

Example: 'Qualifier', 'test\_qualifier'

## **ShortName — Variable short name**

character vector | string scalar

Specify short name for the AUTOSAR variable. If unspecified, `arxml` export automatically generates a short name, which can differ from the data store name.

Example: 'ShortName', 'LowSetPoint'

## **SwAddrMethod — Name of variable SwAddrMethod**

character vector | string scalar



Specify a `SwAddrMethod` name that is valid for the AUTOSAR variable. Code generation uses the `SwAddrMethod` name to group AUTOSAR variables in a memory section for access by measurement and calibration tools. For a list of valid `SwAddrMethod` values for the variable, see the Code Mappings editor, **Data Stores** tab. For more information, see “Configure `SwAddrMethod`”.

Example: `'SwAddrMethod', 'VAR'`

### **SwCalibrationAccess – Calibration access mode**

character vector | string scalar

Specify how measurement and calibration tools can access the AUTOSAR variable. Valid access values include `ReadOnly`, `ReadWrite`, and `NotAccessible`. For more information, see “Configure `SwCalibrationAccess`”.

Example: `'SwCalibrationAccess', 'ReadWrite'`

## **See Also**

Data Store Memory | `getDataStore`

## **Topics**

“Map Data Stores to AUTOSAR Variables”

“Configure AUTOSAR Per-Instance Memory”

“Configure AUTOSAR Static Memory”

“Configure and Map AUTOSAR Component Programmatically”

“AUTOSAR Component Configuration”

## **Introduced in R2019a**

# mapDataTransfer

**Package:** autosar.api

Map Simulink data transfer to AUTOSAR inter-runnable variable

## Syntax

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

## Description

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

## Examples

### Set AUTOSAR Mapping Information for Simulink Data Transfer Line

Set AUTOSAR mapping information for a data transfer line in the example model `autosar_swc_expfncns`. 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`.

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

arIrvName =
IRV4

arDataAccessMode =
Explicit
```

## Set AUTOSAR Mapping Information for Rate Transition Block

Set AUTOSAR mapping information for a Rate Transition block in the example model `mMultitasking_4rates`. The model has Rate Transition blocks named `RateTransition`, `RateTransition1`, and `RateTransition2`, which are located at the top level of the model. This example changes the AUTOSAR data access mode for `RateTransition` from `Implicit` to `Explicit`.

```
addpath(fullfile(matlabroot, '/help/toolbox/autosar/examples'));
hModel = 'mMultitasking_4rates';
open_system(hModel);
slMap=autosar.api.getSimulinkMapping(hModel);
mapDataTransfer(slMap, 'mMultitasking_4rates/RateTransition', 'IRV1', 'Explicit');
[arIrvName, arDataAccessMode]=getDataTransfer(slMap, 'mMultitasking_4rates/RateTransition')

arIrvName =
IRV1

arDataAccessMode =
Explicit
```

## 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, character vector, or string scalar representing the model name.

Example: `slMap`

### **slDataTransfer** — Simulink data transfer line name or Rate Transition full block path

character vector | string scalar

Name of the Simulink data transfer line or full block path to the Rate Transition block for which to set AUTOSAR mapping information.

Example: `'irv4'`

Example: `'myModel/RateTransition2'`

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

character vector | string scalar

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

Example: 'IRV4'

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

character vector | string scalar

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

Example: 'Explicit'

## See Also

`getDataTransfer`

## Topics

“Configure and Map AUTOSAR Component Programmatically”

“AUTOSAR Component Configuration”

**Introduced in R2013b**

# mapFunction

**Package:** autosar.api

Map Simulink entry-point function to AUTOSAR runnable and software address methods

## Syntax

```
mapFunction(slMap, slFcnName, arRunnableName)
mapFunction(slMap, slFcnName, arRunnableName, Name, Value)
```

## Description

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

`mapFunction(slMap, slFcnName, arRunnableName, Name, Value)` specifies additional properties for the AUTOSAR runnable by using one or more `Name, Value` pair arguments. You can specify software address methods (`SwAddrMethods`) for runnable function code and internal data.

## Examples

### Set AUTOSAR Mapping Information for Simulink Entry-Point Function

Set AUTOSAR mapping information for a Simulink entry point function in the example model `autosar_sw_c`. The model has an initialize entry-point function named `Runnable_Init` and rate-based entry-point functions named `Runnable_1s` and `Runnable_2s`.

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

```
arRunnableName =  
    'Runnable_Init'
```

## Set AUTOSAR SwAddrMethods for Simulink Entry-Point Function

Set AUTOSAR SwAddrMethods for a Simulink entry point function in the example model `autosar_swc_counter`. The model has a rate-based entry-point step function.

```
hModel = 'autosar_swc_counter';  
open_system(hModel);  
  
% Add SwAddrMethods myCODE and myVAR to the AUTOSAR component  
arProps = autosar.api.getAUTOSARProperties(hModel);  
addPackageableElement(arProps, 'SwAddrMethod', ...  
    '/Company/Powertrain/DataTypes/SwAddrMethods', 'myCODE', ...  
    'SectionType', 'Code')  
swAddrPaths = find(arProps, [], 'SwAddrMethod', 'PathType', 'FullyQualified', ...  
    'SectionType', 'Code')  
addPackageableElement(arProps, 'SwAddrMethod', ...  
    '/Company/Powertrain/DataTypes/SwAddrMethods', 'myVAR', ...  
    'SectionType', 'Var')  
swAddrPaths = find(arProps, [], 'SwAddrMethod', 'PathType', 'FullyQualified', ...  
    'SectionType', 'Var')  
  
% Set code generation parameter for runnable internal data SwAddrMethods  
set_param(hModel, 'GroupInternalDataByFunction', 'on')  
  
% Map step runnable function and internal data to myCODE and myVAR SwAddrMethods  
slMap = autosar.api.getSimulinkMapping(hModel);  
mapFunction(slMap, 'StepFunction', 'Runnable_Step', ...  
    'SwAddrMethod', 'myCODE', 'SwAddrMethodForInternalData', 'myVAR')  
  
% Return AUTOSAR mapping information for step function  
[arRunnableName, arRunnableSwAddrMethod, arInternalDataSwAddrMethod] = ...  
    getFunction(slMap, 'StepFunction')  
  
swAddrPaths =  
    {'/Company/Powertrain/DataTypes/SwAddrMethods/CODE'}  
    {'/Company/Powertrain/DataTypes/SwAddrMethods/myCODE'}  
  
swAddrPaths =  
    {'/Company/Powertrain/DataTypes/SwAddrMethods/VAR'}  
    {'/Company/Powertrain/DataTypes/SwAddrMethods/myVAR'}  
  
arRunnableName =  
    'Runnable_Step'  
  
arRunnableSwAddrMethod =  
    'myCODE'
```

```
arInternalDataSwAddrMethod =
    'myVAR'
```

## 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, character vector, or string scalar representing the model name.

Example: `slMap`

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

character vector | string scalar

Name of the Simulink entry point function for which to set AUTOSAR mapping information, specified as follows.

- For an initialize function, 'InitializeFunction'.
- For a reset function, a model-wide reset event name. For example, 'reset'.
- For a terminate function, 'TerminateFunction'.
- For a rate-based function, 'StepFunction' for the base-rate task or 'StepFunctionN' for a substrate task, where *N* is the task identifier.
- For an exported function, 'FunctionCallName', where *FunctionCallName* is the name of the Inport block that drives the control port of the function-call subsystem. For example, 'Trigger\_1s' in example model `autosar_swc_slfcns` or 'FunctionTrigger' in example model `autosar_swc_fcncalls`.
- For a global Simulink function in a client-server configuration, 'SLFunctionName', where *SLFunctionName* is the name of a Simulink function for which Trigger block parameter **Function visibility** is set to `global`. For example, 'readData' in the example model used in “Configure AUTOSAR Server”.

Example: 'StepFunction2'

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

character vector | string scalar

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

Example: 'Runnable2'

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1, Value1, . . . , NameN, ValueN`.

Example: 'SwAddrMethod', 'CODE' specifies SwAddrMethod CODE for an AUTOSAR runnable function.

### SwAddrMethod — Name of function SwAddrMethod

character vector | string scalar

Specify a `SwAddrMethod` name that is valid for the AUTOSAR function. Code generation uses the `SwAddrMethod` name to group AUTOSAR runnable functions in a memory section. For a list of valid `SwAddrMethod` values for the function, see the Code Mappings editor, **Entry-Point Functions** tab. For more information, see “Configure SwAddrMethod”.

Example: 'SwAddrMethod', 'CODE'

### SwAddrMethodForInternalData — Name of internal data SwAddrMethod

character vector | string scalar

Specify a `SwAddrMethod` name that is valid for the AUTOSAR internal data. Code generation uses the `SwAddrMethod` name to group AUTOSAR runnable internal data in a memory section. For a list of valid `SwAddrMethod` values for the internal data, see the Code Mappings editor, **Entry-Point Functions** tab. For more information, see “Configure SwAddrMethod”.

Code generation for runnable internal data SwAddrMethods requires setting the model configuration option **Code Generation > Interface > Generate separate internal data per entry-point function** (GroupInternalDataByFunction) to on.

Example: 'SwAddrMethodForInternalData', 'VAR'

## See Also

getFunction



## **Topics**

“Configure and Map AUTOSAR Component Programmatically”

“AUTOSAR Component Configuration”

**Introduced in R2013b**

# mapFunctionCaller

**Package:** autosar.api

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`.

```
addpath(fullfile(matlabroot,'/help/toolbox/autosar/examples'));
hModel = 'mControllerWithInterface_client';
open_system(hModel);
slMapC = autosar.api.getSimulinkMapping(hModel);
mapFunctionCaller(slMapC,'readData','cPort','readData');
[arPort,arOp] = getFunctionCaller(slMapC,'readData')

arPort =
cPort
```

```
arOp =  
readData
```

## 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, character vector, or string scalar representing the model name.

Example: `sLMap`

### **sLFcnName** — Name of Simulink function

character vector | string scalar

Name of the Simulink function for the function-caller block for which to set AUTOSAR mapping information.

Example: `'readData'`

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

character vector | string scalar

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

Example: `'cPort'`

### **arOperationName** — Name of AUTOSAR operation

character vector | string scalar

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

Example: `'readData'`

## See Also

`getFunctionCaller`

## Topics

“Configure and Map AUTOSAR Component Programmatically”

“AUTOSAR Component Configuration”

**Introduced in R2014b**

# mapInport

**Package:** autosar.api

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 `autosar_sw_c_expfcns`. The model has an inport named `RPort_DE1`. This example changes the AUTOSAR data access mode for `RPort_DE1` from `ImplicitReceive` to `ExplicitReceive`.

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

DE1

```
arDataAccessMode =  
ExplicitReceive
```

## 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, character vector, or string scalar representing the model name.

Example: `sLMap`

### **sLPortName** — Name of model inport

character vector | string scalar

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

Example: `'Input'`

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

character vector | string scalar

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

Example: `'Input'`

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

character vector | string scalar

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

Example: `'Input'`

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

character vector | string scalar

Value of the AUTOSAR data access mode to which to map the specified Simulink inport. The value can be `ImplicitReceive`, `ExplicitReceive`, `QueuedExplicitReceive`, `ErrorStatus`, `ModeReceive`, `IsUpdated`, `EndToEndRead`, or `ExplicitReceiveByVal`.

Example: 'ExplicitReceive'

## **See Also**

getInport

## **Topics**

“Configure and Map AUTOSAR Component Programmatically”

“AUTOSAR Component Configuration”

**Introduced in R2013b**

## mapOutputport

**Package:** autosar.api

Map Simulink outputport to AUTOSAR port

### Syntax

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

### Description

`mapOutputport(slMap,slPortName,arPortName,arDataElementName,arDataAccessMode)` maps the Simulink outputport `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 Outputport

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

```
hModel = 'autosar_sw_c_expfcns';  
open_system(hModel);  
slMap=autosar.api.getSimulinkMapping(hModel);  
mapOutputport(slMap,'PPort_DE1','PPort','DE1','ExplicitSend');  
[arPortName,arDataElementName,arDataAccessMode]=getOutputport(slMap,'PPort_DE1')  
  
arPortName =  
PPort  
  
arDataElementName =
```



```
DE1
```

```
arDataAccessMode =  
ExplicitSend
```

## 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, character vector, or string scalar representing the model name.

Example: `sLMap`

### **sLPortName** — Name of model output

character vector | string scalar

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

Example: `'Output'`

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

character vector | string scalar

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

Example: `'Output'`

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

character vector | string scalar

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

Example: `'Output'`

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

character vector | string scalar

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

Example: 'ExplicitSend'

## **See Also**

getOutport

## **Topics**

“Configure and Map AUTOSAR Component Programmatically”

“AUTOSAR Component Configuration”

**Introduced in R2013b**

# mapParameter

**Package:** autosar.api

Map Simulink model workspace parameter to AUTOSAR component internal parameter

## Syntax

```
mapParameter(sLMap, sLParameter, arParamType)
mapParameter(sLMap, sLParameter, arParamType, Name, Value)
```

## Description

`mapParameter(sLMap, sLParameter, arParamType)` maps the Simulink model workspace parameter `sLParameter` to an AUTOSAR parameter of type `arParamType` for AUTOSAR run-time calibration. AUTOSAR parameter types include `SharedParameter`, `PerInstanceParameter`, and `ConstantMemory`.

`mapParameter(sLMap, sLParameter, arParamType, Name, Value)` specifies additional properties for an AUTOSAR `SharedParameter`, `PerInstanceParameter`, or `ConstantMemory` parameter by using one or more `Name, Value` pair arguments.

## Examples

### Set AUTOSAR Mapping Information for Simulink Model Workspace Parameters

Set AUTOSAR mapping and property information for Simulink model workspace parameters `K` and `INC` in example model `autosar_swc_counter`.

```
hModel = 'autosar_swc_counter';
open_system(hModel);
sLMap = autosar.api.getSimulinkMapping(hModel);

mapParameter(sLMap, 'K', 'SharedParameter')
arMappedTo = getParameter(sLMap, 'K')
arValue = getParameter(sLMap, 'K', 'SwCalibrationAccess')
```

```
mapParameter(sLMap, 'INC', 'ConstantMemory', 'SwCalibrationAccess', 'ReadOnly')
arMappedTo = getParameter(sLMap, 'INC')
arValue = getParameter(sLMap, 'INC', 'SwCalibrationAccess')

arMappedTo =
    'SharedParameter'

arValue =
    'ReadWrite'

arMappedTo =
    'ConstantMemory'

arValue =
    'ReadOnly'
```

## 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, character vector, or string scalar representing the model name.

Example: `sLMap`

**sLParameter** — Name of Simulink model workspace parameter  
character vector | string scalar

Name of the Simulink model workspace parameter for which to set AUTOSAR mapping information.

Example: `'INC'`

**arParamType** — Type of AUTOSAR parameter  
character vector | string scalar

Type of AUTOSAR component internal parameter to which to map the specified Simulink model workspace parameter. Valid AUTOSAR parameter types include `SharedParameter`, `PerInstanceParameter`, `ConstantMemory`, and `Auto`. To accept software mapping defaults, specify `Auto`.

Example: `'SharedParameter'`

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1, Value1, . . . , NameN, ValueN`.

Example: `'SwCalibrationAccess', 'ReadOnly'` specifies read-only access to the parameter for run-time calibration.

### **DisplayFormat** — Calibration display format

character vector | string scalar

Specify display format for the AUTOSAR parameter. AUTOSAR display format specifications control the width and precision display for measurement and calibration data. For more information, see “Configure DisplayFormat”.

Example: `'DisplayFormat', '%2.6f'`

### **IsConst** — C const type qualifier flag (ConstantMemory only)

character vector | string scalar

Specify whether to include C type qualifier `const` in generated code for the AUTOSAR parameter.

Example: `'IsConst', 'true'`

### **IsVolatile** — C volatile type qualifier flag (ConstantMemory only)

character vector | string scalar

Specify whether to include C type qualifier `volatile` in generated code for the AUTOSAR parameter.

Example: `'IsVolatile', 'true'`

### **Qualifier** — C AdditionalNativeTypeQualifier flag (ConstantMemory only)

character vector | string scalar

Optionally specify an AUTOSAR additional native type qualifier to include in generated code for the AUTOSAR parameter.

Example: `'Qualifier', 'test_qualifier'`

### **SwAddrMethod** — Name of parameter SwAddrMethod

character vector | string scalar

Specify a `SwAddrMethod` name that is valid for the AUTOSAR parameter. Code generation uses the `SwAddrMethod` name to group AUTOSAR parameters in a memory section for access by measurement and calibration tools. For a list of valid `SwAddrMethod` values for the parameter, see the Code Mappings editor, **Parameters** tab. For more information, see “Configure `SwAddrMethod`”.

Example: `'SwAddrMethod', 'CONST'`

### **SwCalibrationAccess — Calibration access mode**

character vector | string scalar

Specify how measurement and calibration tools can access the AUTOSAR parameter. Valid access values include `ReadOnly`, `ReadWrite`, and `NotAccessible`. For more information, see “Configure `SwCalibrationAccess`”.

Example: `'SwCalibrationAccess', 'ReadOnly'`

## **See Also**

`getParameter`

## **Topics**

“Configure and Map AUTOSAR Component Programmatically”

“AUTOSAR Component Configuration”

**Introduced in R2018b**

# mapSignal

**Package:** autosar.api

Map Simulink block signal to AUTOSAR variable

## Syntax

```
mapSignal(slMap,slPortHandle,arVarType)
mapSignal(slMap,slPortHandle,arVarType,Name,Value)
```

## Description

`mapSignal(slMap,slPortHandle,arVarType)` maps the named or test-pointed Simulink block signal associated with output port handle `slPortHandle` to an AUTOSAR variable of type `arVarType` for AUTOSAR run-time calibration. AUTOSAR variable types include `ArTypedPerInstanceMemory` and `StaticMemory`.

`mapSignal(slMap,slPortHandle,arVarType,Name,Value)` specifies additional properties for an AUTOSAR `ArTypedPerInstanceMemory` or `StaticMemory` variable by using one or more `Name,Value` pair arguments.

## Examples

### Set AUTOSAR Mapping Information for Simulink Block Signals

Set AUTOSAR mapping and property information for the Simulink block signals for blocks `RelOpt` and `Sum` in example model `autosar_sw_counter`.

```
hModel = 'autosar_sw_counter';
open_system(hModel);
slMap = autosar.api.getSimulinkMapping(hModel);

portHandles = get_param('autosar_sw_counter/RelOpt','portHandles');
outputHandle = portHandles.Outport;
mapSignal(slMap,outputHandle,'StaticMemory')
arMappedTo = getSignal(slMap,outputHandle)
```

```
arValue = getSignal(sLMMap,outputHandle,'SwCalibrationAccess')

portHandles = get_param('autosar_sw_counter/Sum','portHandles');
outputHandle = portHandles.Outport;
mapSignal(sLMMap,outputHandle,'ArTypedPerInstanceMemory',...
'SwCalibrationAccess','ReadWrite')
arMappedTo = getSignal(sLMMap,outputHandle)
arValue = getSignal(sLMMap,outputHandle,'SwCalibrationAccess')

arMappedTo =
    'StaticMemory'

arValue =
    'ReadOnly'

arMappedTo =
    'ArTypedPerInstanceMemory'

arValue =
    'ReadWrite'
```

## Input Arguments

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

handle

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

Example: `sLMMap`

### **sLPortHandle** — Simulink output port handle for a block signal

handle

Output port handle for a named or test-pointed Simulink block signal for which to set AUTOSAR mapping information. Use MATLAB commands to construct the output port handle. For example, for a Relational Operator block named `RelOpt`:

```
portHandles = get_param('autosar_sw_counter/RelOpt','portHandles');
outputHandle = portHandles.Outport;
```

Example: `outputHandle`

### **arVarType** — Type of AUTOSAR variable

character vector | string scalar



Type of AUTOSAR variable to which to map the specified Simulink block signal. Valid AUTOSAR variable types include `ArTypedPerInstanceMemory`, `StaticMemory`, and `Auto`. To accept software mapping defaults, specify `Auto`.

Example: `'StaticMemory'`

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1, Value1, . . . , NameN, ValueN`.

Example: `'SwCalibrationAccess', 'ReadWrite'` specifies read-write access to the variable for run-time calibration.

### DisplayFormat — Calibration display format

character vector | string scalar

Specify display format for the AUTOSAR variable. AUTOSAR display format specifications control the width and precision display for measurement and calibration data. For more information, see “Configure DisplayFormat”.

Example: `'DisplayFormat', '%2.6f'`

### IsVolatile — C volatile type qualifier flag (StaticMemory only)

character vector | string scalar

Specify whether to include C type qualifier `volatile` in generated code for the AUTOSAR variable.

Example: `'IsVolatile', 'true'`

### Qualifier — C AdditionalNativeTypeQualifier flag (StaticMemory only)

character vector | string scalar

Optionally specify an AUTOSAR additional native type qualifier to include in generated code for the AUTOSAR variable.

Example: `'Qualifier', 'test_qualifier'`

### ShortName — Variable short name

character vector | string scalar

Specify short name for the AUTOSAR variable. If unspecified, arxml export automatically generates a short name, which can differ from the signal name.

Example: 'ShortName', 'SM\_equal\_to\_count'

### **SwAddrMethod — Name of variable SwAddrMethod**

character vector | string scalar

Specify a SwAddrMethod name that is valid for the AUTOSAR variable. Code generation uses the SwAddrMethod name to group AUTOSAR variables in a memory section for access by measurement and calibration tools. For a list of valid SwAddrMethod values for the variable, see the Code Mappings editor, **Signals/States** tab. For more information, see “Configure SwAddrMethod”.

Example: 'SwAddrMethod', 'VAR'

### **SwCalibrationAccess — Calibration access mode**

character vector | string scalar

Specify how measurement and calibration tools can access the AUTOSAR variable. Valid access values include `ReadOnly`, `ReadWrite`, and `NotAccessible`. For more information, see “Configure SwCalibrationAccess”.

Example: 'SwCalibrationAccess', 'ReadWrite'

## **See Also**

`getSignal`

## **Topics**

“Configure and Map AUTOSAR Component Programmatically”

“AUTOSAR Component Configuration”

**Introduced in R2018b**

# mapState

**Package:** autosar.api

Map Simulink block state to AUTOSAR variable

## Syntax

```
mapState(slMap, slStateOwnerBlock, '', arVarType)
mapState(slMap, slStateOwnerBlock, slState, arVarType)
mapState(slMap, slStateOwnerBlock, slState, arVarType, Name, Value)
```

## Description

`mapState(slMap, slStateOwnerBlock, '', arVarType)` maps the Simulink block state associated with state owner block `slStateOwnerBlock` to an AUTOSAR variable of type `arVarType` for AUTOSAR run-time calibration. AUTOSAR variable types include `ArTypedPerInstanceMemory` and `StaticMemory`.

`mapState(slMap, slStateOwnerBlock, slState, arVarType)` maps Simulink block state `slState` associated with state owner block `slStateOwnerBlock` to an AUTOSAR variable of type `arVarType`. Specify a nonempty `slState` argument only for blocks with multiple states.

`mapState(slMap, slStateOwnerBlock, slState, arVarType, Name, Value)` specifies additional properties for an AUTOSAR `ArTypedPerInstanceMemory` or `StaticMemory` variable by using one or more `Name, Value` pair arguments.

## Examples

### Set AUTOSAR Mapping Information for Simulink Block State

Set AUTOSAR mapping and property information for the Simulink block state for Unit Delay block X in example model `autosar_swc_counter`. The state owner block has one state.

```
hModel = 'autosar_sw_counter';
open_system(hModel);
slMap = autosar.api.getSimulinkMapping(hModel);

mapState(slMap, 'autosar_sw_counter/X', '', 'ArTypedPerInstanceMemory', ...
    'SwCalibrationAccess', 'ReadWrite')
arMappedTo = getState(slMap, 'autosar_sw_counter/X')
arValue = getState(slMap, 'autosar_sw_counter/X', '', 'SwCalibrationAccess')

arMappedTo =
    'ArTypedPerInstanceMemory'

arValue =
    'ReadWrite'
```

## 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, character vector, or string scalar representing the model name.

Example: `slMap`

### **slStateOwnerBlock** — Simulink state owner block handle or path

handle | character vector | string scalar

Handle or path to Simulink state owner block for which to set AUTOSAR mapping information.

Example: `'autosar_sw_counter/X'`

### **slState** — Simulink state

character vector | string scalar

Name of Simulink state associated with state owner block `slStateOwnerBlock`. Specify a nonempty state name only for blocks with multiple states. If `slState` is empty, the function sets mapping information for the first state in the block.

Example: `''`

### **arVarType** — Type of AUTOSAR variable

character vector | string scalar

Type of AUTOSAR variable to which to map the specified Simulink block state. Valid AUTOSAR variable types include `ArTypedPerInstanceMemory`, `StaticMemory`, and `Auto`. To accept software mapping defaults, specify `Auto`.

Example: `'ArTypedPerInstanceMemory'`

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1, Value1, . . . , NameN, ValueN`.

Example: `'SwCalibrationAccess', 'ReadWrite'` specifies read-write access to the variable for run-time calibration.

### DisplayFormat — Calibration display format

character vector | string scalar

Specify display format for the AUTOSAR variable. AUTOSAR display format specifications control the width and precision display for measurement and calibration data. For more information, see “Configure `DisplayFormat`”.

Example: `'DisplayFormat', '%2.6f'`

### IsVolatile — C volatile type qualifier flag (StaticMemory only)

character vector | string scalar

Specify whether to include C type qualifier `volatile` in generated code for the AUTOSAR variable.

Example: `'IsVolatile', 'true'`

### Qualifier — C AdditionalNativeTypeQualifier flag (StaticMemory only)

character vector | string scalar

Optionally specify an AUTOSAR additional native type qualifier to include in generated code for the AUTOSAR variable.

Example: `'Qualifier', 'test_qualifier'`

### ShortName — Variable short name

character vector | string scalar

Specify short name for the AUTOSAR variable. If unspecified, arxml export automatically generates a short name, which is based on the state name if one exists. If the state is unnamed, the generated short name can differ from the block name.

Example: 'ShortName', 'PIM\_X'

### **SwAddrMethod — Name of variable SwAddrMethod**

character vector | string scalar

Specify a `SwAddrMethod` name that is valid for the AUTOSAR variable. Code generation uses the `SwAddrMethod` name to group AUTOSAR variables in a memory section for access by measurement and calibration tools. For a list of valid `SwAddrMethod` values for the variable, see the Code Mappings editor, **Signals/States** tab. For more information, see “Configure `SwAddrMethod`”.

Example: 'SwAddrMethod', 'VAR'

### **SwCalibrationAccess — Calibration access mode**

character vector | string scalar

Specify how measurement and calibration tools can access the AUTOSAR variable. Valid access values include `ReadOnly`, `ReadWrite`, and `NotAccessible`. For more information, see “Configure `SwCalibrationAccess`”.

Example: 'SwCalibrationAccess', 'ReadWrite'

## **See Also**

`getState`

## **Topics**

“Configure and Map AUTOSAR Component Programmatically”

“AUTOSAR Component Configuration”

**Introduced in R2018b**

# set

**Package:** autosar.api

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 an AUTOSAR model, set the IsService property for sender-receiver interface Interfacel1 to true (1), indicating that the port interface is used for AUTOSAR services.

```
hModel = 'autosar_swc_expfncns';
open_system(hModel);
arProps = autosar.api.getAUTOSARProperties(hModel);
set(arProps,'Interfacel1','IsService',true);
isService = get(arProps,'Interfacel1','IsService')

isService =
    logical
     1
```

## Set Runnable Symbol Name

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

```
hModel = 'autosar_swc_expfncs';
open_system(hModel);
arProps = autosar.api.getAUTOSARProperties(hModel);
compQName = get(arProps, 'XmlOptions', 'ComponentQualifiedNames');
runnables = find(arProps, compQName, 'Runnable', 'PathType', 'FullyQualified');
runnables(2)

ans =
    {'/pkg/swc/ASWC/IB/Runnable1'}

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

ans =
    Runnable1

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

ans =
    test_symbol
```

## 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, character vector, or string scalar representing the model name.

Example: `arProps`

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

character vector | string scalar

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

Example: `'Input'`

### **property** — Type of property

character vector | string scalar



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

## Topics

“Configure and Map AUTOSAR Component Programmatically”

“AUTOSAR Component Configuration”

**Introduced in R2013b**

## updateAUTOSARProperties

**Package:** arxml

Update model with arxml definitions of AUTOSAR elements

### Syntax

```
updateAUTOSARProperties(ar,modelname)  
updateAUTOSARProperties(ar,modelname,Name,Value)
```

### Description

`updateAUTOSARProperties(ar,modelname)` updates the specified open model with AUTOSAR elements in the XML files associated with `arxml.importer` object `ar`. The update operation generates a report that details the AUTOSAR elements added to the model. For `updateAUTOSARProperties`, unlike `updateModel`, the associated XML definition files are not required to contain the AUTOSAR software component mapped by the model.

For each imported AUTOSAR element, the function also imports the element dependencies. For example, importing `CompuMethod` elements also imports `Unit` and `PhysicalDimension` elements.

By default, the function imports AUTOSAR elements as read-only definitions, which prevents changes. To allow imported elements to be modified, set the `ReadOnly` property to `false`.

`updateAUTOSARProperties(ar,modelname,Name,Value)` updates the specified open model with AUTOSAR elements, using a `Name,Value` argument pair to specify a specific element category, package, or path.

## Examples

### Reuse AUTOSAR SwAddrMethod Elements in Component Model

Suppose that you are developing an AUTOSAR software component model, into which you want to import predefined SwAddrMethod elements that are shared by multiple product lines and teams. This example imports definitions from shared descriptions file SwAddrMethods.arxml into example model autosar\_swc and generates an update report.

```
addpath(fullfile(matlabroot, '/examples/autosarblockset'));
modelName = 'autosar_swc';
open_system(modelName);
ar = arxml.importer('SwAddrMethods.arxml');
updateAUTOSARProperties(ar, modelName);

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

## AUTOSAR Update Report for autosar\_swc

Software component: `/Company/Powertrain/Components/ASWC`  
Original model saved as: `autosar_swc_backup`

This report details the updates applied to Simulink model `autosar_swc` 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 `autosar_swc_backup` ([compare models](#)). The report also recommends manual model changes.

### Simulink

---

### AUTOSAR

---

#### Automatic AUTOSAR Element Changes

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

## Update Model with Specific AUTOSAR Elements

Update model `mySWC` with two AUTOSAR elements, specified by root paths `/ExternalElements/CompuMethods/RpmCm` and `/AUTOSAR_PlatformTypes/ImplementationDataTypes/uint16`.

```
open_system('mySWC')
ar = arxml.importer('ExternalElements.arxml');
updateAUTOSARProperties(ar, 'mySWC', 'RootPath', {'/ExternalElements/CompuMethods/RpmCm', ...
'/AUTOSAR_PlatformTypes/ImplementationDataTypes/uint16'});
```

## Allow Modification of Imported AUTOSAR Elements

Import XML definitions of AUTOSAR software address methods as read-write elements. By default, the function imports AUTOSAR elements as read-only definitions, which prevents changes.

```
open_system('mySWC')
ar = arxml.importer('SwAddressMethods.arxml');
updateAUTOSARProperties(ar,'mySWC','ReadOnly',false);
```

## Update Model with AUTOSAR Elements From a Specific Package

Update model mySWC with AUTOSAR elements from package / AUTOSAR\_PlatformTypes/CompuMethods.

```
open_system('mySWC')
ar = arxml.importer('ExternalElements.arxml');
updateAUTOSARProperties(ar,'mySWC','Package',{'/AUTOSAR_PlatformTypes/CompuMethods'});
```

## Update Model with AUTOSAR Elements From a Specific Category

Update model mySWC with AUTOSAR elements of category ImplementationDataType. Importing ImplementationDataType elements also imports dependent elements, such as SwBaseType elements.

```
open_system('mySWC')
ar = arxml.importer('ExternalElements.arxml');
updateAUTOSARProperties(ar,'mySWC','Category',{'ImplementationDataType'});
```

## Input Arguments

### **ar** — arxml.importer object

object

AUTOSAR information previously imported from XML files, specified as an arxml.importer object.

### **modelName** — Model name

character vector | string scalar

Name of the open model to be updated with definitions of AUTOSAR elements in the XML files associated with an arxml.importer object.

Example: 'mySWC'

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1, Value1, . . . , NameN, ValueN`.

Example: `'Category', {'ImplementationDataType'}` directs the importer to update a model with AUTOSAR elements of category `ImplementationDataType`.

### Category — AUTOSAR element categories

cell array of character vectors | string array

One or more AUTOSAR element categories from which to import elements.

Example: `'Category', {'ImplementationDataType'}`

### Package — AUTOSAR element packages

cell array of character vectors | string array

Paths to one or more AUTOSAR element packages from which to import elements.

Example: `'Package', {'/AUTOSAR_PlatformTypes/CompuMethods'}`

To refine a category or package import, you can specify both a category and a package from which to import elements. For example:

```
'Category', {'ImplementationDataType'}, ...  
'Package', {'/AUTOSAR_PlatformTypes/ImplementationDataTypes'}
```

### ReadOnly — Designate elements as read-only or read-write

true (default) | false

Specify whether to treat imported elements as read-only (the default), preventing definition changes, or read-write.

Example: `'ReadOnly', false`

### RootPath — AUTOSAR elements

cell array of character vectors | string array

Root paths to one or more specific AUTOSAR elements to import.

Example: `'RootPath', {'/ExternalElements/CMs/RpmCm', '/AUTOSAR_PlatformTypes/IDTs/uint16'}`

**DataTypeMappingSet — AUTOSAR data type mapping sets**

cell array of character vectors | string array

Paths to one or more AUTOSAR data type mapping sets associated with application data type elements.

Example: { '/AUTOSAR\_PlatformTypes/DataTypeMappingSets/MapSet1' }

**See Also**

arxml.importer | updateModel

**Topics**

“Reuse AUTOSAR Element Descriptions”  
“Reuse AUTOSAR Elements in Component Model”  
“Import AUTOSAR Software Component”  
“AUTOSAR arxml Importer”

**Introduced in R2019a**

## updateModel

**Package:** arxml

Update AUTOSAR model with arxml changes

### Syntax

```
updateModel(ar, modelName)
```

### Description

`updateModel(ar, modelName)` updates the specified open model with changes detected in the XML files associated with `arxml.importer` object `ar`. The update generates and opens a report that details the updates applied to the model, and required changes that were not made automatically. The XML files must contain the AUTOSAR software component mapped by the model.

### Example

#### Update Model with AUTOSAR arxml Changes

Update model `mySWC` with the AUTOSAR `arxml` changes described in `updatedSWC.arxml` and open an update report.

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

### Updating model mySWC
### Saving original model as mySWC_backup.slx
### Creating HTML report mySWC_update_report.html
```

### Input Arguments

**ar** — `arxml.importer` object  
object



AUTOSAR information previously imported from XML files, specified as an `arxml.importer` object.

**modelName — Model name**

character vector | string scalar

Name of an open model to be updated with changes in the XML files associated with an `arxml.importer` object.

Example: 'mySWC'

## See Also

`arxml.importer`

## Topics

"Import AUTOSAR Component to Simulink"

"Import AUTOSAR Composition to Simulink"

"Import AUTOSAR Software Component"

"AUTOSAR arxml Importer"

"Import AUTOSAR Software Component Updates"

## Introduced in R2014a



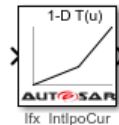
# Blocks — Alphabetical List

---

# Curve

Approximate one-dimensional function

**Library:** AUTOSAR Blockset / Classic Platform / Library  
Routines / Interpolation



## Description

The Curve block performs one-dimensional interpolated table lookup, including index searches. The table is a sampled representation of a function. Breakpoint sets relate the input values to positions in the table. You can also use the Prelookup and Prelookup Using Curve blocks together to perform the same operations as this block.

If you select the AUTOSAR 4.0 code replacement library (CRL) for your AUTOSAR model, code generated from this block is replaced with the AUTOSAR library routine that you configure in the block parameters dialog box.

## Ports

### Input

#### **u1 — First-dimension inputs**

scalar | vector | matrix

Real-valued inputs to the **u1** port, mapped to an output value by looking up or interpolating the table of values that you define.

Example: 0:10

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | fixed point

## Output

### **Port\_1 – Output computed by looking up or estimating table values**

scalar | vector | matrix

Output generated by looking up or estimating table values based on the input values. If the inputs match the index values of breakpoint sets the curve block provides a table value as output. If the block inputs do not match index values in breakpoint sets, but are within range, the block performs the interpolation method you selected and provides an estimated value from the table values as output.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | fixed point

## Parameters

### **Targeted Routine Library – Indicates the AUTOSAR routine library used for block code replacement**

IFX(fixed-point) (default) | IFL(floating-point)

If you select the AUTOSAR 4.0 code replacement library (CRL) for your model, code generated from this block is replaced from the selected AUTOSAR routine library. This parameter enables you to choose either fixed-point (IFX) or floating-point (IFL) code replacement and validation checks.

### **Targeted Routine – AUTOSAR library routine used for code replacement**

Ifx\_IntIpoMap (default)

This parameter reflects the name of the AUTOSAR code replacement library (CRL) routine used to replace the code generated by this block. The naming convention includes the targeted routine library, interpolation method, and block type. This parameter is reference-only and must not be edited.

## Table Specification

### **Data specification – Method of table and breakpoint specification**

Table and breakpoints (default) | Lookup table object

From the list, select:

- **Table and breakpoints** — Specify the table data and breakpoints. Selecting this option enables these parameters:
  - **Table data**
  - **Breakpoints specification**
  - **Breakpoints**
  - **Edit table and breakpoints**
- **Lookup table object** — Use an existing lookup table (`Simulink.LookupTable`) object. Selecting this option enables the **Name** field and the **Edit table and breakpoints** button.

### **Programmatic Use**

**Block Parameter:** `DataSpecification`

**Type:** character vector

**Values:** 'Table and breakpoints' | 'Lookup table object'

**Default:** 'Table and breakpoints'

### **Name — Name of the lookup table object**

[ ] (default) | `Simulink.LookupTable` object

Enter the name of the lookup table (`Simulink.LookupTable`) object.

### **Dependencies**

To enable this parameter, set **Data specification** to `Lookup table object`.

### **Programmatic Use**

**Block Parameter:** `LookupTableObject`

**Type:** character vector

**Values:** name of a `Simulink.LookupTable` object

**Default:** ''

### **Table data — Define the table of output values**

[1 2 4] (default) | character vector

Enter the table of output values.

During simulation, the matrix must be one-dimensional. However, during block diagram editing, you can enter an empty matrix (specified as [ ]) or an undefined workspace variable. This technique lets you postpone specifying a correctly dimensioned matrix for the table data and continue editing the block diagram.

**Dependencies**

To enable this parameter, set **Data specification** to Table and breakpoints.

**Programmatic Use**

**Block Parameter:** Table

**Type:** character vector

**Values:** matrix of table values

**Default:** '[1 2 4]'

**Breakpoints specification – Method of breakpoint specification**

Explicit values (default) | Even spacing

Specify whether to enter data as explicit breakpoints or as parameters that generate evenly spaced breakpoints.

- To explicitly specify breakpoint data, set this parameter to `Explicit values` and enter breakpoint data in the text box next to the **Breakpoints** parameters.
- To specify parameters that generate evenly spaced breakpoints, set this parameter to `Even spacing` and enter values for the **First point** and **Spacing** parameters for each dimension of breakpoint data. The block calculates the number of points to generate from the table data.

**Dependencies**

To enable this parameter, set **Data specification** to Table and breakpoints.

**Programmatic Use**

**Block Parameter:** BreakpointsSpecification

**Type:** character vector

**Values:** 'Explicit values' | 'Even spacing'

**Default:** 'Explicit values'

**Breakpoints – Explicit breakpoint values, or first point and spacing of breakpoints**

[10, 22, 31] (default) | 1-by-n or n-by-1 vector of monotonically increasing values

Specify the breakpoint data explicitly or as evenly-spaced breakpoints, based on the value of the **Breakpoints specification** parameter.

- If you set **Breakpoints specification** to `Explicit values`, enter the breakpoint set that corresponds to each dimension of table data in each **Breakpoints** row. For each

dimension, specify breakpoints as a 1-by-n or n-by-1 vector whose values are strictly monotonically increasing.

- If you set **Breakpoints specification** to `Even spacing`, enter the parameters **First point** and **Spacing** in each **Breakpoints** row to generate evenly-spaced breakpoints in the respective dimension. Your table data determines the number of evenly spaced points.

### Dependencies

To enable this parameter, set **Data specification** to `Table` and `breakpoints`.

### Programmatic Use

**Block Parameter:** `BreakpointsForDimension1`

**Type:** character vector

**Values:** 1-by-n or n-by-1 vector of monotonically increasing values

**Default:** `'[10, 22, 31]'`

### First point — First point in evenly spaced breakpoint data

1 (default) | scalar

Specify the first point in your evenly spaced breakpoint data as a real-valued, finite, or scalar. This parameter is available when you set the **Breakpoints specification** to `Even spacing`.

### Dependencies

To enable this parameter, set **Data specification** to `Table` and `breakpoints` and **Breakpoints specification** to `Even spacing`.

### Programmatic Use

**Block Parameter:** `BreakpointsForDimension1FirstPoint`

**Type:** character vector

**Values:** real-valued, finite, scalar

**Default:** `'1'`

### Spacing — Spacing between evenly spaced breakpoints

1 (default) | scalar

Specify the spacing between points in your evenly-spaced breakpoint data.

### Dependencies

To enable this parameter, set **Data specification** to `Table` and `breakpoints` and **Breakpoints specification** to `Even spacing`.



**Programmatic Use****Block Parameter:** BreakpointsForDimension1Spacing**Type:** character vector**Values:** positive, real-valued, finite, scalar**Default:** '1'**Edit table and breakpoints – Launch Lookup Table Editor dialog box**  
button

Click this button to open the Lookup Table Editor. You can then edit the object and save the new values for the object. For more information, see “Edit Lookup Tables” (Simulink) in the Simulink documentation.

**Algorithm****Index search method – Method of calculating table indices**

Linear search (default) | Evenly spaced points | Binary search

Select Evenly spaced points, Linear search, or Binary search. Each search method has speed advantages in different circumstances:

- For evenly spaced breakpoint sets (for example, 10, 20, 30, and so on), you achieve optimal speed by selecting Evenly spaced points to calculate table indices. This algorithm uses only the first two breakpoints of a set to determine the offset and spacing of the remaining points.

---

**Note** When using the Simulink.LookupTable object to specify table data and the **Breakpoints Specification** parameter of the referenced Simulink.LookupTable object is set to Even spacing, set the **Index search method** to Evenly spaced points.

---

- For unevenly spaced breakpoint sets, follow these guidelines:
  - If input signals do not vary significantly between time steps, selecting Linear search with **Begin index search using previous index result** produces the best performance.
  - If input signals jump more than one or two table intervals per time step, selecting Binary search produces the best performance.

A suboptimal choice of an index search method can lead to slow performance of models that rely heavily on lookup tables.

The generated code stores only the first breakpoint, the spacing, and the number of breakpoints when:

- The breakpoint data is not tunable.
- The index search method is `Evenly spaced points`.

### **Programmatic Use**

**Block Parameter:** `IndexSearchMethod`

**Type:** character vector

**Values:** `'Binary search'` | `'Evenly spaced points'` | `'Linear search'`

**Default:** `'Linear search'`

### **Begin index search using previous index result — Start using the index from the previous time step**

`off (default)` | `on`

Select this check box when you want the block to start its search using the index found at the previous time step. For inputs that change slowly with respect to the interval size, enabling this option can improve performance. Otherwise, the linear search and binary search methods can take longer, especially for large breakpoint sets.

### **Dependencies**

To enable this parameter, set **Index search method** to `Linear search` or `Binary search`.

### **Programmatic Use**

**Block Parameter:** `BeginIndexSearchUsing PreviousIndexResult`

**Type:** character vector

**Values:** `'off'` | `'on'`

**Default:** `'off'`

### **Interpolation method — Method of interpolation between breakpoint values**

`Linear point-slope (default)` | `Flat`

When an input falls between breakpoint values, the block interpolates the output value by using neighboring breakpoints. For more information, see “Interpolation Methods” (Simulink).

### **Programmatic Use**

**Block Parameter:** `InterpMethod`

**Type:** character vector

**Values:** 'Linear point-slope' | 'Flat'

**Default:** 'Linear point-slope'

### **Integer rounding mode — Rounding mode for fixed-point operations**

Round (default) | Zero

Specify the rounding mode for fixed-point or floating-point lookup table calculations that occur during simulation or execution of code generated from the model.

This option does not affect rounding of block parameter values. Simulink rounds such values to the nearest representable integer value. To control the rounding of a block parameter, enter an expression using a MATLAB rounding function into the edit field on the block dialog box.

#### **Programmatic Use**

**Block Parameter:** RndMeth

**Type:** character vector

**Values:** 'Round' | 'Zero'

**Default:** 'Round'


## **Data Types**

### **Table data — Data type of table data**

Inherit: Same as output (default) | double | single | int8 | uint8 | int16 | uint16 | int32 | uint32 | fixdt(1,16) | fixdt(1,16,0) | fixdt(1,16,2^0,0) | <data type expression>

Specify the table data type. The block validates that the selected types are compatible with the specification of the targeted routine. You can set the table data type to:

- A rule that inherits a data type, for example, Inherit: Same as output
- The name of a built-in data type, for example, single
- The name of a data type object, for example, a Simulink.NumericType object
- An expression that evaluates to a data type, for example, fixdt(1,16,0)

Click the **Show data type assistant** button  to display the **Data Type Assistant**, which helps you set the data type attributes. For more information, see “Specify Data Types Using Data Type Assistant” (Simulink).

---

**Tip** Specify a table data type different from the output data type in these cases:

- Lower memory requirement for storing table data that uses a smaller type than the output signal.
  - Sharing of prescaled table data between two Map blocks that have different output data types.
  - Sharing of custom storage table data in the generated code for blocks that have different output data types.
- 

### **Programmatic Use**

**Block Parameter:** TableDataTypeStr

**Type:** character vector

**Values:** 'Inherit: Inherit from 'Table data'' | 'Inherit: Same as output' | 'double' | 'single' | 'int8' | 'uint8' | 'int16' | 'uint16' | 'int32' | 'uint32' | 'fixdt(1,16)' | 'fixdt(1,16,0)' | 'fixdt(1,16,2^0,0)' | '<data type expression>'

**Default:** 'Inherit: Same as output'


### **Breakpoints — Breakpoint data type**

Inherit: Same as corresponding input (default) | double | single | int8 | uint8 | int16 | uint16 | int32 | uint32 | fixdt(1,16) | fixdt(1,16,0) | fixdt(1,16,2^0,0) | Enum: <class name> | <data type expression>

Specify the data type for a set of breakpoint data. You can set the breakpoint data type to:

- A rule that inherits a data type, for example, `Inherit: Same as corresponding input`
- The name of a built-in data type, for example, `single`
- The name of a data type class, for example, an enumerated data type class
- The name of a data type object, for example, a `Simulink.NumericType` object
- An expression that evaluates to a data type, for example, `fixdt(1,16,0)`

A limitation for using enumerated data with this block is that it does not support out-of-range input for enumerated data. When specifying enumerated data, include the entire enumeration set in the breakpoint data set.

Click the **Show data type assistant** button  to display the **Data Type Assistant**, which helps you set the data type attributes. For more information, see “Specify Data Types Using Data Type Assistant” (Simulink).

#### **Programmatic Use**

**Block Parameter:** BreakpointsForDimension1DataTypeStr | BreakpointsForDimension2DataTypeStr

**Type:** character vector

**Values:** 'Inherit: Same as corresponding input' | 'Inherit: Inherit from 'Breakpoint data'' | 'double' | 'single' | 'int8' | 'uint8' | 'int16' | 'uint16' | 'int32' | 'uint32' | 'fixdt(1,16)' | 'fixdt(1,16,0)' | 'fixdt(1,16,2^0,0)' | '<data type expression>'

**Default:** 'Inherit: Same as corresponding input'

## **Extended Capabilities**

### **C/C++ Code Generation**

Generate C and C++ code using Simulink® Coder™.

### **See Also**

Curve Using Prelookup | Map | Map Using Prelookup | Prelookup

### **Topics**

“Configure Lookup Tables for AUTOSAR Measurement and Calibration”

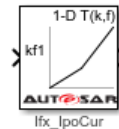
“Code Generation with AUTOSAR Code Replacement Library”

**Introduced in R2019a**

## Curve Using Prelookup

Use previously calculated index and fraction values to accelerate approximation of one-dimensional function

**Library:** AUTOSAR Blockset / Classic Platform / Library  
Routines / Interpolation



### Description

The Curve Using Prelookup block is intended for use with a Prelookup block. This block enables a prelookup result to drive multiple interpolation results. The Prelookup block computes the index and interval fraction that specify how its input value  $u$  relates to the breakpoint data set and feeds the resulting index and fraction values into the Curve Using Prelookup block to interpolate a one-dimensional table. The Prelookup and Curve Using Prelookup blocks have distributed algorithms that when used together perform the same algorithm operation as the Curve block but offer greater flexibility and more efficient simulation and code generation.

If you select the AUTOSAR 4.0 code replacement library (CRL) for your AUTOSAR model, code generated from this block is replaced with the AUTOSAR library routine that you configure in the block parameters dialog box.

### Ports

#### Input

**kf1** — Input containing index  $k$  and fraction  $f$

bus object

Inputs to the **kf1** port contain index  $k$  and fraction  $f$  specified as a bus object.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | fixed point | bus

**T — Table data**

scalar | vector | matrix | 1-d array

Table data values provided as input to port **T**. These table values correspond to the breakpoint data sets specified in Prelookup blocks. The Interpolation Using Prelookup block generates output by looking up or estimating table values based on index (k) and interval fraction (f) values fed from Prelookup blocks.

**Dependencies**

To enable this port, set **Source** to Input port.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | fixed point

**Output****Port 1 — Approximation of one-dimensional function**

scalar | vector | matrix

Approximation of the one-dimensional function computed by interpolating table data that uses values from the input index, k, and the fraction, f.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | fixed point

**Parameters****Targeted Routine Library — Indicates the AUTOSAR routine library used for block code replacement**

IFX(fixed-point) (default) | IFL(floating-point)

If you select the AUTOSAR 4.0 code replacement library (CRL) for your model, code generated from this block is replaced from the selected AUTOSAR routine library. This parameter enables you to choose either fixed-point (IFX) or floating-point (IFL) code replacement and validation checks.

**Targeted Routine — AUTOSAR library routine used for code replacement**

Ifx\_IpoCur (default)

This parameter reflects the name of the AUTOSAR code replacement library (CRL) routine used to replace the code generated by this block. The naming convention includes

the targeted routine library, interpolation method, and block type. This parameter is reference-only and must not be edited.

### Table Specification

#### Data Specification — Choose how to enter table data

Explicit values (default) | Lookup table object

Specify whether to enter table data directly or use a lookup table object. If you set this parameter to:

- Explicit values, the **Table Data** parameter is visible in the dialog box.
- Lookup table object, the **Name** parameter is visible in the dialog box.

#### Programmatic Use

**Block Parameter:** TableSpecification

**Type:** character vector


**Values:** 'Explicit values' | 'Lookup table object'

**Default:** 'Explicit values'

#### Name — Name of a Simulink.LookupTable object

Simulink.LookupTable object

Specify the name of a Simulink.LookupTable object. A lookup table object references Simulink breakpoint objects. If a Simulink.LookupTable object does not exist, click the

action button  and select **Create**. The corresponding parameters of the new lookup table object are populated with the block information.

#### Dependencies

To enable this parameter, set **Data Specification** to Lookup table object.

#### Programmatic Use

**Block Parameter:** LookupTableObject

**Type:** character vector

**Value:** Simulink.LookupTable object

**Default:** ''

#### Table data — Define the table of output values

[1 2 4] (default) | character vector

Enter the table of output values.



During simulation, the matrix size must be one-dimensional. However, during block diagram editing, you can enter an empty matrix (specified as `[]`) or an undefined workspace variable. This technique lets you postpone specifying a correctly dimensioned matrix for the table data and continue editing the block diagram.

### Dependencies

To enable this parameter, set **Data specification** to `Table` and `breakpoints`.

### Programmatic Use

**Block Parameter:** `Table`

**Type:** character vector

**Values:** matrix of table values

**Default:** `'[1 2 4]'`

### Edit table and breakpoints — Launch Lookup Table Editor dialog box button

Click this button to open the Lookup Table Editor. For more information, see “Edit Lookup Tables” (Simulink) in the Simulink documentation.

Clicking this button for a lookup table object lets you edit the object and save the new values for the object.

## Algorithm

### Interpolation method — Select Linear point-slope or Flat interpolation methods

`Linear point-slope (default) | Flat`

Specify the method that the block uses to interpolate table data. You can select `Linear point-slope` or `Flat`. For more information, see “Interpolation Methods” (Simulink).

### Programmatic Use

**Block Parameter:** `InterpMethod`

**Type:** character vector

**Values:** `'Flat' | 'Linear point-slope'`

**Default:** `'Linear point-slope'`

### Integer rounding mode — Rounding mode for fixed-point operations

`Round (default) | Zero`

Specify the rounding mode for fixed-point or floating-point lookup table calculations that occur during simulation or execution of code generated from the model.

This option does not affect rounding of values of block parameters. Simulink rounds such values to the nearest representable integer value. To control the rounding of a block parameter, enter an expression using a MATLAB rounding function into the edit field on the block dialog box.

### **Programmatic Use**

**Block Parameter:** RndMeth

**Type:** character vector

**Values:** 'Round' | 'Zero'

**Default:** 'Round'


## **Data Types**

### **Table data — Data type of table data**

Inherit: Same as output (default) | double | single | int8 | uint8 | int16 | uint16 | int32 | uint32 | fixdt(1,16) | fixdt(1,16,0) | fixdt(1,16,2^0,0) | <data type expression>

Specify the table data type. The block validates that the selected types are compatible with the specification of the targeted routine. You can set it to:

- A rule that inherits a data type, for example, `Inherit: Same as output`
- The name of a built-in data type, for example, `single`
- The name of a data type object, for example, a `Simulink.NumericType` object
- An expression that evaluates to a data type, for example, `fixdt(1,16,0)`

Click the **Show data type assistant** button  to display the **Data Type Assistant**, which helps you set the data type attributes. For more information, see “Specify Data Types Using Data Type Assistant” (Simulink).

---

**Tip** Specify a table data type different from the output data type for these cases:

- Lower memory requirement for storing table data that uses a smaller type than the output signal
- Sharing of prescaled table data between two Curve blocks with different output data types

- Sharing of custom storage table data in the generated code for blocks with different output data types
- 

### Programmatic Use

**Block Parameter:** TableDataTypeStr

**Type:** character vector

**Values:** 'Inherit: Inherit from 'Table data'' | 'Inherit: Same as output' | 'double' | 'single' | 'int8' | 'uint8' | 'int16' | 'uint16' | 'int32' | 'uint32' | 'fixdt(1,16)' | 'fixdt(1,16,0)' | 'fixdt(1,16,2^0,0)' | '<data type expression>'

**Default:** 'Inherit: Same as output'

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

### See Also

Curve | Map | Map Using Prelookup | Prelookup

### Topics

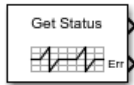
“Configure Lookup Tables for AUTOSAR Measurement and Calibration”

“Code Generation with AUTOSAR Code Replacement Library”

**Introduced in R2019a**

# DiagnosticInfoCaller

Call AUTOSAR Diagnostic Event Manager (Dem) service interface `DiagnosticInfo`



## Library

AUTOSAR Blockset / Classic Platform / Basic Software / Diagnostic Event Manager (Dem)

## Description

Call AUTOSAR Dem service interface `DiagnosticInfo` using a specified operation.

## Parameters

### Client port name

Name of the client port used by the AUTOSAR software component for the function call, set to `DiagnosticInfo` by default.

### Operation

Name of an operation defined by the AUTOSAR standard for the Dem service interface `DiagnosticInfo`:

- `GetEventStatus` (default)
- `GetEventFailed`
- `GetEventTested`
- `GetDTCOfEvent`
- `GetFaultDetectionCounter`
- `GetEventExtendedDataRecord`
- `GetEventFreezeFrameData`

**Data type for DTCFormat** (GetDTCOfEvent only)

Name of a data type that defines Dem format type values for the function input DTCFormat. By default, the data type is set to Enum: Dem\_DTCFormatType. For more information about format type values, see the AUTOSAR standard *Specification of Diagnostic Event Manager*.

**Sample time**

Block sample time, set to -1 (inherited) by default.

**See Also**

DiagnosticMonitorCaller

Diagnostic Service Component

**Related Examples**

“Configure Calls to AUTOSAR Diagnostic Event Manager Service”

“Configure AUTOSAR Basic Software Service Implementations for Simulation”

“Simulate AUTOSAR Basic Software Services and Run-Time Environment”

**Introduced in R2016b**

## DiagnosticMonitorCaller

Call AUTOSAR Diagnostic Event Manager (Dem) service interface `DiagnosticMonitor`



## Library

AUTOSAR Blockset / Classic Platform / Basic Software / Diagnostic Event Manager (Dem)

## Description

Call AUTOSAR Dem service interface `DiagnosticMonitor` using a specified operation.

## Parameters

### Client port name

Name of the client port used by the AUTOSAR software component for the function call, set to `DiagnosticMonitor` by default.

### Operation

Name of an operation defined by the AUTOSAR standard for the Dem service interface `DiagnosticMonitor`:

- `SetEventStatus` (default)
- `ResetEventStatus`
- `PrestoreFreezeFrame`
- `ClearPrestoredFreezeFrame`
- `SetEventDisabled`

### Data type for `EventStatus` (`SetEventStatus` only)

Name of a data type that defines Dem event status values for the function input `EventStatus`. By default, the data type is set to Enum: `Dem_EventStatusType`.

For more information about event status values, see the AUTOSAR standard *Specification of Diagnostic Event Manager*.

**Sample time**

Block sample time, set to -1 (inherited) by default.

**See Also**

DiagnosticInfoCaller

Diagnostic Service Component

**Related Examples**

“Configure Calls to AUTOSAR Diagnostic Event Manager Service”

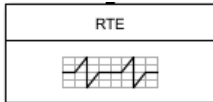
“Configure AUTOSAR Basic Software Service Implementations for Simulation”

“Simulate AUTOSAR Basic Software Services and Run-Time Environment”

**Introduced in R2016b**

# Diagnostic Service Component

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



## Library

AUTOSAR Blockset / Classic Platform / Basic Software / Diagnostic Event Manager (Dem)

## Description

The Diagnostic Service Component block provides reference implementations of Diagnostic Event Manager (Dem) services supported by AUTOSAR Basic Software (BSW) caller blocks. When coupled with Dem caller blocks, the reference implementations allow you to configure and run system- or composition-level simulations of AUTOSAR Dem service calls.

The block has prepopulated parameters, including **Counter-Based Debouncing** parameters and RTE parameters. Examine the parameter settings and consider if modifications are required, based on how you are using the Dem service operations.

The **Counter-Based Debouncing** parameters control the counter-based debounce algorithm provided by the Dem service reference implementations. During multiple simulation runs, you can tune event step size and threshold parameters and observe the effects.

Debouncing provides a means to determine when a monitored event is regarded as passed or failed. The software maintains a counter for each event ID. When PREFAIL events arrive, the software increments the event ID counter by a fixed step value. When PREPASS events arrive, the software decrements the event ID counter by a fixed step value. When a counter reaches a lower or upper threshold, the event is considered passed or failed.

In the Dem reference implementations, the step size and threshold parameters apply globally to event IDs, not to individual IDs.



## Parameters

### Increment step size

Specify a fixed-step value, 1 to 32767, by which to increment a Dem event ID counter when PREFAIL events arrive.

### Decrement step size

Specify a fixed-step value, 1 to 32767, by which to decrement a Dem event ID counter when PREPASS events arrive.

### Failed threshold

Specify a Dem event ID counter threshold value, 1 to 32767, to represent failed status.

### Passed threshold

Specify a Dem event ID counter threshold value, -32768 to -1, to represent passed status.

### Event ID

The RTE tab table lists component client ports and their mapping to Dem service event IDs. Each row in the table represents a call into Dem services from a Basic Software caller block. Calls that act on the same event should use the same event ID. Check the event ID mappings. For an example of mapping Dem client ports to shared event IDs, see “Simulate AUTOSAR Basic Software Services and Run-Time Environment”.

## See Also

DiagnosticInfoCaller

DiagnosticMonitorCaller

## Related Examples

“Configure Calls to AUTOSAR Diagnostic Event Manager Service”

“Configure AUTOSAR Basic Software Service Implementations for Simulation”

“Simulate AUTOSAR Basic Software Services and Run-Time Environment”

**Introduced in R2017b**

# Event Receive

Convert input event to signal



## Library

AUTOSAR Blockset / Adaptive Platform / Signal Routing

## Description

At the top level of an AUTOSAR adaptive model, use the Event Receive and Event Send blocks to set up event-based communication.

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

## Parameters

### Data type

Data type of the outgoing data.

## See Also

Event Send

## **Related Examples**

“Configure AUTOSAR Adaptive Software Components”

**Introduced in R2019a**

# Event Send

Convert input signal to event



## Library

AUTOSAR Blockset / Adaptive Platform / Signal Routing

## Description

At the top level of an AUTOSAR adaptive model, use the Event Receive and Event Send blocks to set up event-based communication.

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

## Parameters

### Data type

Data type of the outgoing data.

## See Also

Event Receive

## **Related Examples**

“Configure AUTOSAR Adaptive Software Components”

**Introduced in R2019a**

# Map

Approximate two-dimensional function

**Library:** AUTOSAR Blockset / Classic Platform / Library  
Routines / Interpolation



## Description

The Map block performs two-dimensional, interpolated table lookup, including index searches. The table is a sampled representation of a function in two variables. Breakpoint sets relate the input values to positions in the table. You can also use the Prelookup and Prelookup Using Map blocks together to perform the same operations as this block.

When you set the **Math and Data Types > Use algorithms optimized for row-major array layout** configuration parameter, the block behavior changes from column-major to row-major. For these blocks, the column-major and row-major algorithms might differ in the order of the output calculations, possibly resulting in slightly different numeric values. This capability requires Simulink Coder or Embedded Coder software. For more information on row-major support, see “Code Generation of Matrices and Arrays” (Simulink Coder).

If you select the AUTOSAR 4.0 code replacement library (CRL) for your AUTOSAR model, code generated from this block is replaced with the AUTOSAR library routine that you configure in the block parameters dialog box.

## Ports

### Input

#### u1 — First dimension input values

scalar | vector | matrix

Real-valued inputs to the first port, mapped to an output value by looking up or interpolating the table of values that you define.

Example: 0:10

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | fixed point

### **u2 — Second dimension input values**

scalar | vector | matrix

Real-valued inputs to the second port, mapped to an output value by looking up or interpolating the table of values that you define.

Example: 0:10

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | fixed point

## **Output**

### **y — Output computed by looking up or estimating table values**

scalar | vector | matrix

Output generated by looking up or estimating table values based on input values. If the inputs match the index values of breakpoint sets, the map block provides a table value as output. If the block inputs do not match index values in breakpoint sets, but are within range, the block performs the configured interpolation method and provides an estimated value from the table as output.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | fixed point

## **Parameters**

### **Targeted Routine Library — Indicates the AUTOSAR routine library used for block code replacement**

IFX(fixed-point) (default) | IFL(floating-point)

If you select the AUTOSAR 4.0 code replacement library (CRL) for your model, code generated from this block is replaced from the selected AUTOSAR routine library. This parameter enables you to choose either fixed-point (IFX) or floating-point (IFL) code replacement and validation checks.



### Targeted Routine — AUTOSAR library routine used for code replacement

Ifx\_IntIpoMap (default)

This parameter reflects the name of the AUTOSAR code replacement library (CRL) routine used to replace the code generated by this block. The naming convention includes the targeted routine library, interpolation method, and block type. This parameter is reference-only and must not be edited.

## Table Specification

### Data specification — Method of table and breakpoint specification

Table and breakpoints (default) | Lookup table object

From the list, select:

- **Table and breakpoints** — Specify the table data and breakpoints. Selecting this option enables these parameters:
  - **Table data**
  - **Breakpoints specification**
  - **Breakpoints 1**
  - **Breakpoints 2**
  - **Edit table and breakpoints**
- **Lookup table object** — Use an existing lookup table (`Simulink.LookupTable`) object. Selecting this option enables the **Name** field and the **Edit table and breakpoints** button.

#### Programmatic Use

**Block Parameter:** DataSpecification

**Type:** character vector

**Values:** 'Table and breakpoints' | 'Lookup table object'

**Default:** 'Table and breakpoints'

#### Name — Name of the lookup table object

[] (default) | `Simulink.LookupTable` object

Enter the name of the lookup table (`Simulink.LookupTable`) object.

#### Dependencies

To enable this parameter, set **Data specification** to `Lookup table object`.

### Programmatic Use

**Block Parameter:** LookupTableObject

**Type:** character vector

**Values:** name of a Simulink.LookupTable object

**Default:** ''

### Table data — Define the table of output values

[4 5 6;16 19 20;10 18 23] (default) | character vector

Enter the table of output values.

During simulation, the matrix size must be two dimensional. However, during block diagram editing, you can enter an empty matrix (specified as []) or an undefined workspace variable. This technique lets you postpone specifying a correctly dimensioned matrix for the table data and continue editing the block diagram.

### Dependencies

To enable this parameter, set **Data specification** to Table and breakpoints.

### Programmatic Use

**Block Parameter:** Table

**Type:** character vector

**Values:** matrix of table values

**Default:** '[4 5 6;16 19 20;10 18 23]'

### Breakpoints specification — Method of breakpoint specification

Explicit values (default) | Even spacing

Specify whether to enter data as explicit breakpoints or as parameters that generate evenly spaced breakpoints.

- To explicitly specify breakpoint data, set this parameter to **Explicit values** and enter breakpoint data in the text box next to the **Breakpoints** parameters.
- To specify parameters that generate evenly spaced breakpoints, set this parameter to **Even spacing** and enter values for the **First point** and **Spacing** parameters for each dimension of breakpoint data. The block calculates the number of points to generate from the table data.

### Dependencies

To enable this parameter, set **Data specification** to Table and breakpoints.

**Programmatic Use****Block Parameter:** BreakpointsSpecification**Type:** character vector**Values:** 'Explicit values' | 'Even spacing'**Default:** 'Explicit values'**Breakpoints — Explicit breakpoint values, or first point and spacing of breakpoints**

[10, 22, 31] (default) | 1-by-n or n-by-1 vector of monotonically increasing values

Specify the breakpoint data explicitly or as evenly-spaced breakpoints, based on the value of the **Breakpoints specification** parameter.

- If you set **Breakpoints specification** to `Explicit values`, enter the breakpoint set that corresponds to each dimension of table data in each **Breakpoints** row. For each dimension, specify breakpoints as a 1-by-n or n-by-1 vector whose values are strictly monotonically increasing.
- If you set **Breakpoints specification** to `Even spacing`, enter the parameters **First point** and **Spacing** in each **Breakpoints** row to generate evenly-spaced breakpoints in the respective dimension. Your table data determines the number of evenly spaced points.

**Dependencies**

To enable this parameter, set **Data specification** to `Table` and `breakpoints`.

**Programmatic Use****Block Parameter:** BreakpointsForDimension1 | BreakpointsForDimension2**Type:** character vector**Values:** 1-by-n or n-by-1 vector of monotonically increasing values**Default:** '[10, 22, 31]'**First point — First point in evenly spaced breakpoint data**

1 (default) | scalar

Specify the first point in your evenly spaced breakpoint data as a real-valued, finite, or scalar. This parameter is available when you set the **Breakpoints specification** to `Even spacing`.

**Dependencies**

To enable this parameter, set **Data specification** to `Table` and `breakpoints` and **Breakpoints specification** to `Even spacing`.

### **Programmatic Use**

**Block Parameter:** BreakpointsForDimension1FirstPoint | BreakpointsForDimensionSecondPoint

**Type:** character vector

**Values:** real-valued, finite, scalar

**Default:** '1'

### **Spacing — Spacing between evenly spaced breakpoints**

1 (default) | scalar

Specify the spacing between points in your evenly-spaced breakpoint data.

### **Dependencies**

To enable this parameter, set **Data specification** to Table and breakpoints and **Breakpoints specification** to Even spacing.

### **Programmatic Use**

**Block Parameter:** BreakpointsForDimension1Spacing | BreakpointsForDimension2Spacing

**Type:** character vector

**Values:** positive, real-valued, finite, scalar

**Default:** '1'

### **Edit table and breakpoints — Launch Lookup Table Editor dialog box**

button

Click this button to open the Lookup Table Editor. You can then edit the object and save the new values for the object. For more information, see “Edit Lookup Tables” (Simulink) in the Simulink documentation.

## **Algorithm**

### **Index search method — Method of calculating table indices**

Linear search (default) | Evenly spaced points | Binary search

Select Evenly spaced points, Linear search, or Binary search. Each search method has speed advantages in different circumstances:

- For evenly spaced breakpoint sets (for example, 10, 20, 30, and so on), you achieve optimal speed by selecting Evenly spaced points to calculate table indices. This algorithm uses only the first two breakpoints of a set to determine the offset and spacing of the remaining points.

---

**Note** When using the `Simulink.LookupTable` object to specify table data and the **Breakpoints Specification** parameter of the referenced `Simulink.LookupTable` object is set to `Even spacing`, set the **Index search method** to `Evenly spaced points`.

---

- For unevenly spaced breakpoint sets, follow these guidelines:
  - If input signals do not vary significantly between time steps, selecting `Linear search` with **Begin index search using previous index result** produces the best performance.
  - If input signals jump more than one or two table intervals per time step, selecting `Binary search` produces the best performance.

A suboptimal choice of an index search method can lead to slow performance of models that rely heavily on lookup tables.

The generated code stores only the first breakpoint, the spacing, and the number of breakpoints when:

- The breakpoint data is not tunable.
- The index search method is `Evenly spaced points`.

#### **Programmatic Use**

**Block Parameter:** `IndexSearchMethod`

**Type:** character vector

**Values:** `'Binary search'` | `'Evenly spaced points'` | `'Linear search'`

**Default:** `'Linear search'`

#### **Begin index search using previous index result — Start using the index from the previous time step**

`off (default)` | `on`

Select this check box when you want the block to start its search using the index found at the previous time step. For inputs that change slowly with respect to the interval size, enabling this option can improve performance. Otherwise, the linear search and binary search methods can take longer, especially for large breakpoint sets.

#### **Dependencies**

To enable this parameter, set **Index search method** to `Linear search` or `Binary search`.

**Programmatic Use****Block Parameter:** BeginIndexSearchUsing PreviousIndexResult**Type:** character vector**Values:** 'off' | 'on'**Default:** 'off'**Interpolation method — Method of interpolation between breakpoint values**

Linear point-slope (default) | Flat

When an input falls between breakpoint values, the block interpolates the output value by using neighboring breakpoints. For more information, see “Interpolation Methods” (Simulink).

**Programmatic Use****Block Parameter:** InterpMethod**Type:** character vector**Values:** 'Linear point-slope' | 'Flat'**Default:** 'Linear point-slope'**Integer rounding mode — Rounding mode for fixed-point operations**

Round (default) | Zero

Specify the rounding mode for fixed-point or floating-point lookup table calculations that occur during simulation or execution of code generated from the model.


This option does not affect rounding of block parameter values. Simulink rounds such values to the nearest representable integer value. To control the rounding of a block parameter, enter an expression using a MATLAB rounding function into the edit field on the block dialog box.

**Programmatic Use****Block Parameter:** RndMeth**Type:** character vector**Values:** 'Round' | 'Zero'**Default:** 'Round'**Data Types****Table data — Data type of table data**

Inherit: Same as output (default) | double | single | int8 | uint8 | int16 | uint16 | int32 | uint32 | fixdt(1,16) | fixdt(1,16,0) | fixdt(1,16,2^0,0) | <data type expression>

Specify the table data type. The block validates that the selected types are compatible with the specification of the targeted routine. You can set the table data type to:

- A rule that inherits a data type, for example, `Inherit: Same as output`
- The name of a built-in data type, for example, `single`
- The name of a data type object, for example, a `Simulink.NumericType` object
- An expression that evaluates to a data type, for example, `fixdt(1,16,0)`

Click the **Show data type assistant** button  to display the **Data Type Assistant**, which helps you set the data type attributes. For more information, see “Specify Data Types Using Data Type Assistant” (Simulink).

---

**Tip** Specify a table data type different from the output data type in these cases:

- Lower memory requirement for storing table data that uses a smaller type than the output signal.
  - Sharing of prescaled table data between two Map blocks that have different output data types.
  - Sharing of custom storage table data in the generated code for blocks that have different output data types.
- 

### Programmatic Use

**Block Parameter:** `TableDataTypeStr`

**Type:** character vector

**Values:** `'Inherit: Inherit from 'Table data'' | 'Inherit: Same as output' | 'double' | 'single' | 'int8' | 'uint8' | 'int16' | 'uint16' | 'int32' | 'uint32' | 'fixdt(1,16)' | 'fixdt(1,16,0)' | 'fixdt(1,16,2^0,0)' | '<data type expression>'`

**Default:** `'Inherit: Same as output'`


### Breakpoints — Breakpoint data type

`Inherit: Same as corresponding input (default) | double | single | int8 | uint8 | int16 | uint16 | int32 | uint32 | fixdt(1,16) | fixdt(1,16,0) | fixdt(1,16,2^0,0) | Enum: <class name> | <data type expression>`

Specify the data type for a set of breakpoint data. You can set the breakpoint data type to:

- A rule that inherits a data type, for example, `Inherit: Same as corresponding input`
- The name of a built-in data type, for example, `single`
- The name of a data type class, for example, an enumerated data type class
- The name of a data type object, for example, a `Simulink.NumericType` object
- An expression that evaluates to a data type, for example, `fixdt(1,16,0)`

A limitation for using enumerated data with this block is that it does not support out-of-range input for enumerated data. When specifying enumerated data, include the entire enumeration set in the breakpoint data set.

Click the **Show data type assistant** button  to display the **Data Type Assistant**, which helps you set the data type attributes. For more information, see “Specify Data Types Using Data Type Assistant” (Simulink).

### Programmatic Use

**Block Parameter:** `BreakpointsForDimension1DataTypeStr` | `BreakpointsForDimension2DataTypeStr`

**Type:** character vector

**Values:** `'Inherit: Same as corresponding input'` | `'Inherit: Inherit from 'Breakpoint data''` | `'double'` | `'single'` | `'int8'` | `'uint8'` | `'int16'` | `'uint16'` | `'int32'` | `'uint32'` | `'fixdt(1,16)'` | `'fixdt(1,16,0)'` | `'fixdt(1,16,2^0,0)'` | `'<data type expression>'`

**Default:** `'Inherit: Same as corresponding input'`

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

### See Also

[Curve](#) | [Curve Using Prelookup](#) | [Map Using Prelookup](#) | [Prelookup](#)



## **Topics**

“Configure Lookup Tables for AUTOSAR Measurement and Calibration”

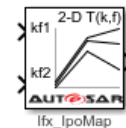
“Code Generation with AUTOSAR Code Replacement Library”

**Introduced in R2019a**

## Map Using Prelookup

Use previously calculated index and fraction values to accelerate approximation of two-dimensional function

**Library:** AUTOSAR Blockset / Classic Platform / Library  
Routines / Interpolation



## Description

The Map Using Prelookup block is intended for use with a Prelookup block. This block enables a prelookup result to drive multiple interpolation results. The Prelookup block calculates the index and interval fraction that specify how its input value  $u$  relates to the breakpoint data set and feeds the resulting index and fraction values into a Map Using Prelookup block to interpolate a two-dimensional table. The Prelookup and Map Using Prelookup blocks have distributed algorithms that when used together perform the same algorithm operation as the Map block but offer greater flexibility and more efficient simulation and code generation.

If you select the AUTOSAR 4.0 code replacement library (CRL) for your AUTOSAR model, code generated from this block is replaced with the AUTOSAR library routine that you configure in the block parameters dialog box.

## Ports

### Input

**kf1** — Input containing index  $k$  and fraction  $f$

bus object

Inputs to the **kf1** port contain index  $k$  and fraction  $f$  specified as a bus object.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | fixed point | bus

**T — Table data**

scalar | vector | matrix | 2-d array

Table data values provided as input to port **T**. These table values correspond to the breakpoint data sets specified in Prelookup blocks. The Interpolation Using Prelookup block generates output by looking up or estimating table values based on index (*k*) and interval fraction (*f*) values fed from a Prelookup block.

**Dependencies**

To enable this port, set **Source** to Input port.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | fixed point

**Output****Port 1 — Approximation of two-dimensional function**

scalar | vector | matrix

Approximation of the two-dimensional function computed by interpolating table data that uses values from the input index, *k*, and the fraction, *f*.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | fixed point

**Parameters****Targeted Routine Library — Indicates the AUTOSAR routine library used for block code replacement**

IFX(fixed-point) (default) | IFL(floating-point)

If you select the AUTOSAR 4.0 code replacement library (CRL) for your model, code generated from this block is replaced from the selected AUTOSAR routine library. This parameter enables you to choose either fixed-point (IFX) or floating-point (IFL) code replacement and validation checks.

**Targeted Routine — AUTOSAR library routine used for code replacement**

Ifx\_IpoMap (default)

This parameter reflects the name of the AUTOSAR code replacement library (CRL) routine used to replace the code generated by this block. The naming convention includes

the targeted routine library, interpolation method, and block type. This parameter is reference-only and must not be edited.

### Table Specification

#### Data Specification — Choose how to enter table data

Explicit values (default) | Lookup table object

Specify whether to enter table data directly or use a lookup table object. If you set this parameter to:

- Explicit values, the **Table Data** parameter is visible in the dialog box.
- Lookup table object, the **Name** parameter is visible in the dialog box.

#### Programmatic Use

**Block Parameter:** TableSpecification

**Type:** character vector


**Values:** 'Explicit values' | 'Lookup table object'

**Default:** 'Explicit values'

#### Name — Name of a Simulink.LookupTable object

Simulink.LookupTable object

Specify the name of a Simulink.LookupTable object. A lookup table object references Simulink breakpoint objects. If a Simulink.LookupTable object does not exist, click the

action button  and select **Create**. The corresponding parameters of the new lookup table object are populated with the block information.

#### Dependencies

To enable this parameter, set **Data Specification** to Lookup table object.

#### Programmatic Use

**Block Parameter:** LookupTableObject

**Type:** character vector

**Value:** Simulink.LookupTable object

**Default:** ''

#### Table data — Define the table of output values

[4 5 6;16 19 20;10 18 23] (default) | character vector

Enter the table of output values.

During simulation, the matrix size must be two-dimensional. However, during block diagram editing, you can enter an empty matrix (specified as `[]`) or an undefined workspace variable. This technique lets you postpone specifying a correctly dimensioned matrix for the table data and continue editing the block diagram.

### Dependencies

To enable this parameter, set **Data specification** to `Table` and `breakpoints`.

### Programmatic Use

**Block Parameter:** `Table`

**Type:** character vector

**Values:** matrix of table values

**Default:** `[4 5 6;16 19 20;10 18 23]'`

### Edit table and breakpoints — Launch Lookup Table Editor dialog box button

Click this button to open the Lookup Table Editor. For more information, see “Edit Lookup Tables” (Simulink) in the Simulink documentation.

Clicking this button for a lookup table object lets you edit the object and save the new values for the object.

## Algorithm

### Interpolation method — Select Linear point-slope or Flat interpolation methods

`Linear point-slope (default) | Flat`

Specify the method that the block uses to interpolate table data. You can select `Linear point-slope` or `Flat`. For more information, see “Interpolation Methods” (Simulink).

### Programmatic Use

**Block Parameter:** `InterpMethod`

**Type:** character vector

**Values:** `'Flat' | 'Linear point-slope'`

**Default:** `'Linear point-slope'`

### Integer rounding mode — Rounding mode for fixed-point operations

`Round (default) | Zero`

Specify the rounding mode for fixed-point or floating-point lookup table calculations that occur during simulation or execution of code generated from the model.

This option does not affect rounding of values of block parameters. Simulink rounds such values to the nearest representable integer value. To control the rounding of a block parameter, enter an expression using a MATLAB rounding function into the edit field on the block dialog box.

### **Programmatic Use**

**Block Parameter:** RndMeth

**Type:** character vector

**Values:** 'Round' | 'Zero'

**Default:** 'Round'


## **Data Types**

### **Table data — Data type of table data**

Inherit: Same as output (default) | double | single | int8 | uint8 | int16 | uint16 | int32 | uint32 | fixdt(1,16) | fixdt(1,16,0) | fixdt(1,16,2^0,0) | <data type expression>

Specify the table data type. The block will validate that the selected types are compatible with the specification of the targeted routine. You can set it to:

- A rule that inherits a data type, for example, `Inherit: Same as output`
- The name of a built-in data type, for example, `single`
- The name of a data type object, for example, a `Simulink.NumericType` object
- An expression that evaluates to a data type, for example, `fixdt(1,16,0)`

Click the **Show data type assistant** button  to display the **Data Type Assistant**, which helps you set the data type attributes. For more information, see “Specify Data Types Using Data Type Assistant” (Simulink).

---

**Tip** Specify a table data type different from the output data type for these cases:

- Lower memory requirement for storing table data that uses a smaller type than the output signal
- Sharing of prescaled table data between two Curve blocks with different output data types

- Sharing of custom storage table data in the generated code for blocks with different output data types
- 

**Programmatic Use****Block Parameter:** TableDataTypeStr**Type:** character vector**Values:** 'Inherit: Inherit from 'Table data'' | 'Inherit: Same as output' | 'double' | 'single' | 'int8' | 'uint8' | 'int16' | 'uint16' | 'int32' | 'uint32' | 'fixdt(1,16)' | 'fixdt(1,16,0)' | 'fixdt(1,16,2^0,0)' | '<data type expression>'**Default:** 'Inherit: Same as output'

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

### See Also

Curve | Curve Using Prelookup | Map | Prelookup

### Topics

"Configure Lookup Tables for AUTOSAR Measurement and Calibration"

"Code Generation with AUTOSAR Code Replacement Library"

**Introduced in R2019a**

# NvMAdminCaller

Call AUTOSAR NVRAM Manager (NvM) service interface `NvMAdmin`



## Library

AUTOSAR Blockset / Classic Platform / Basic Software / NVRAM Manager (NvM)

## Description

Call AUTOSAR NvM service interface `NvMAdmin` using a specified operation.

## Parameters

### Client port name

Name of the client port used by the AUTOSAR software component for the function call, set to `NvMAdmin` by default.

### Operation

Name of an operation defined by the AUTOSAR standard for the NvM service interface `NvMAdmin`. One operation is supported: `SetBlockProtection`.

### Sample time

Block sample time, set to -1 (inherited) by default.

## See Also

`NvMServiceCaller`

NVRAM Service Component



## **Related Examples**

“Configure Calls to AUTOSAR NVRAM Manager Service”

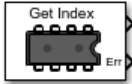
“Configure AUTOSAR Basic Software Service Implementations for Simulation”

“Simulate AUTOSAR Basic Software Services and Run-Time Environment”

**Introduced in R2016b**

## NvMServiceCaller

Call AUTOSAR NVRAM Manager (NvM) service interface NvMService



## Library

AUTOSAR Blockset / Classic Platform / Basic Software / NVRAM Manager (NvM)

## Description

Call AUTOSAR NvM service interface NvMService using a specified operation.

## Parameters

### Client port name

Name of the client port used by the AUTOSAR software component for the function call, set to NvMService by default.

### Operation

Name of an operation defined by the AUTOSAR standard for the NvM service interface NvMService:

- GetDataIndex (default)
- GetErrorStatus
- EraseNvBlock
- InvalidateNvBlock
- ReadBlock
- RestoreBlockDefaults
- SetDataIndex

- SetRamBlockStatus
- WriteBlock

**Argument specification** (ReadBlock, RestoreBlockDefaults, and WriteBlock only)

MATLAB expression that specifies data type and dimensions for data to be read or written by the operation. The default expression is `uint8(1)`. For examples, see “Argument Specification for Simulink Function Blocks” (Simulink).

**Sample time**

Block sample time, set to -1 (inherited) by default.

## See Also

NvMAdminCaller

NVRAM Service Component

## Related Examples

“Configure Calls to AUTOSAR NVRAM Manager Service”

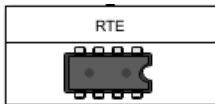
“Configure AUTOSAR Basic Software Service Implementations for Simulation”

“Simulate AUTOSAR Basic Software Services and Run-Time Environment”

**Introduced in R2016b**

# NVRAM Service Component

Configure AUTOSAR NVRAM Services and Runtime Environment (RTE) for emulation



## Library

AUTOSAR Blockset / Classic Platform / Basic Software / NVRAM Manager (NvM)

## Description

The NVRAM Service Component block provides reference implementations of NVRAM Manager (NvM) services supported by AUTOSAR Basic Software (BSW) caller blocks. When coupled with NvM caller blocks, the reference implementations allow you to configure and run system- or composition-level simulations of AUTOSAR NvM service calls.

The block has prepopulated parameters, including **NVRAM Properties** parameters and RTE parameters. Examine the parameter settings and consider if modifications are required, based on how you are using the NvM service operations.

## Parameters

### Maximum number of memory blocks

Specify the maximum number of memory blocks to use in NvM service operations.

### Block ID

The RTE tab table lists component client ports and their mapping to NvM service block IDs. Each row in the table represents a call into NvM services from a Basic Software caller block. Calls that act on the same NvM block should use the same block ID. Check the block ID mappings.

## **See Also**

NvMAdminCaller

NvMServiceCaller

## **Related Examples**

*“Configure Calls to AUTOSAR NVRAM Manager Service”*

*“Configure AUTOSAR Basic Software Service Implementations for Simulation”*

*“Simulate AUTOSAR Basic Software Services and Run-Time Environment”*

**Introduced in R2017b**

## Prelookup

Compute index and fraction for a Curve Using Prelookup or Map Using Prelookup block

**Library:** AUTOSAR Blockset / Classic Platform / Library  
Routines / Interpolation



## Description

The Prelookup block computes the index and fraction that specify how its input value  $u$  relates to the breakpoint dataset. The Prelookup block feeds the resulting output index and fraction values as a bus into a Curve Using Prelookup block to interpolate a one-dimensional table or a Map Using Prelookup block to interpolate a two-dimensional table. When a Prelookup block is used with either a Curve Using Prelookup or Map Using Prelookup block, they perform the same algorithm operation as the Curve or Map blocks. The use of the two blocks together offers greater flexibility and more efficient simulation and code generation.

If you select the AUTOSAR 4.0 code replacement library (CRL) for your AUTOSAR model, code generated from this block is replaced with the AUTOSAR library routine that you configure in the block parameters dialog box.

## Ports

### Input

#### Port\_1 — Input signal, $u$

scalar | vector | matrix

The Prelookup block accepts real-valued signals of any numeric data type that Simulink supports, except Boolean. The Prelookup block supports fixed-point data types for signals and breakpoint data.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 |  
fixed point | bus

## Output

### Port\_2 — Bus containing output index and fraction

bus

Outputs the index and fraction as a bus, which specifies the interval containing the input and the normalized position of the input on the interval. The bus type is defined automatically based on if the **Targeted Routine Library** is set to fixed-point (IFX) or floating-point (IFL) code replacement.

Data Types: bus

## Parameters

### Targeted Routine Library — Indicates the AUTOSAR routine library used for block code replacement

IFX(fixed-point) (default) | IFL(floating-point)

If you select the AUTOSAR 4.0 code replacement library (CRL) for your model, code generated from this block is replaced from the selected AUTOSAR routine library. This parameter enables you to choose either fixed-point (IFX) or floating-point (IFL) code replacement and validation checks.

### Targeted Routine — AUTOSAR library routine used for code replacement

Ifx\_DPSearch (default)

This parameter reflects the name of the AUTOSAR code replacement library (CRL) routine used to replace the code generated by this block. The naming convention includes the targeted routine library, interpolation method, and block type. This parameter is reference-only and must not be edited.

## Table Specification

### Breakpoints Specification — Choose how to enter breakpoint data

Explicit values (default) | Breakpoint object

If you set this parameter to:

- `Explicit values`, the **Breakpoints** and parameter becomes visible in the dialog box.

- Breakpoint object, the **Name** parameter is visible in the dialog box.

### **Programmatic Use**

**Block Parameter:** BreakpointsSpecification

**Type:** character vector

**Values:** 'Explicit values' | 'Breakpoint object'

**Default:** 'Explicit values'

### **Breakpoints — Breakpoint data values**

[1 2 3] (default)

Explicitly specify the breakpoint data. Each breakpoint data set must be a strictly monotonically increasing vector that contains two or more elements.

### **Dependencies**

To enable this parameter, set **Breakpoints Specification** to Explicit values.

### **Programmatic Use**

**Block Parameter:** BreakpointsData

**Type:** character vector


**Values:** '[1 2 3]'

**Default:** '[1 2 3]'

### **Name — Name of a Simulink.Breakpoint object**

no default | Simulink.Breakpoint

Specify the name of a Simulink.Breakpoint object. If a Simulink.Breakpoint

object does not exist, click the action button  and select **Create**. The corresponding parameters of the new breakpoint object are populated with the block information.

### **Dependencies**

To enable this parameter, set **Breakpoints Specification** to Breakpoint object.

### **Programmatic Use**

**Block Parameter:** BreakpointObject

**Type:** character vector

**Values:** Simulink.Breakpoint object

**Default:** ''



## Algorithms

### Index search method — Method for searching breakpoint data

Linear search (default) | Binary search

Each search method has speed advantages in different situations:

- If input values for  $u$  do not vary significantly between time steps, selecting **Linear search** with **Begin index search using previous index result** produces the best performance.
- If input values for  $u$  jump more than one or two table intervals per time step, selecting **Binary search** produces the best performance.

A suboptimal choice of index search method can lead to slow performance of models that rely heavily on lookup tables.

### Begin index search using previous index result — Start search using the index found at the previous time step

off (default) | on

For input values of  $u$  that change slowly with respect to the interval size, enabling this option can improve performance. Otherwise, the linear search and binary search methods can take longer, especially for large breakpoint sets.

#### Programmatic Use

**Block Parameter:** IndexSearchMethod

**Values:** 'Binary search' | 'Linear search'

**Type:** character vector

**Default:** 'Binary search'

### Integer rounding mode — Rounding mode for fixed-point operations

Round (default) | Zero

Specify the rounding mode for fixed-point or floating-point lookup table calculations that occur during simulation or execution of code generated from the model.

This option does not affect rounding of block parameter values. Simulink rounds such values to the nearest representable integer value. To control the rounding of a block parameter, enter an expression using a MATLAB rounding function into the edit field on the block dialog box.

**Programmatic Use**

**Block Parameter:** RndMeth

**Type:** character vector

**Values:** 'Round' | 'Zero'

**Default:** 'Round'

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

### See Also

[Curve](#) | [Curve Using Prelookup](#) | [Map](#) | [Map Using Prelookup](#)

### Topics

[“Configure Lookup Tables for AUTOSAR Measurement and Calibration”](#)

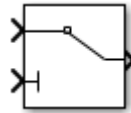
[“Code Generation with AUTOSAR Code Replacement Library”](#)

**Introduced in R2019a**

# Signal Invalidation

Control AUTOSAR root output data element invalidation

**Library:** AUTOSAR Blockset / Classic Platform / Signal Routing



## Description

Relay the first input, a data value, to the output, based on the value of the second input, an invalidation control flag.

If the input data value is valid (invalidation control flag is `false`), the software relays the input data value to the output.

If the input data value is invalid (invalidation control flag is `true`), the resulting action is determined by the value of the block parameter **Signal invalidation policy**:

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

This block must be connected directly to a root output block. It cannot be used within a reusable subsystem.

## Ports

### Input

#### Port\_1 — Input data value

numeric value

Input data value to be relayed if valid.

Example: 4

Data Types: single | double | base integer | Boolean | fixed point | enumerated | bus

### **Port\_2 — Invalidation control flag**

true | false

The invalidation control flag determines whether the input data value is valid and can be relayed (`false`), or is invalid and must be handled based on an invalidation policy (`true`).

Example: false

Data Types: Boolean

## **Output**

### **Port\_1 — Output data value**

numeric value

Output data value produced by the combination of the input data value and the invalidation control flag.

Data Types: single | double | base integer | Boolean | fixed point | enumerated | bus

## **Parameters**

### **Signal invalidation policy — Invalidation policy**

Keep (default) | Replace | DontInvalidate

Specify an AUTOSAR data element invalidation policy, which determines how an invalid data element is handled.

### **Initial value — Data element initial value**

0 (default) | numeric value

Specify a data element initial value. If the input data value is flagged as invalid, and if the **Signal invalidation policy** is `Replace`, the software replaces the input data value with the specified initial value.

## **Extended Capabilities**

### **C/C++ Code Generation**

Generate C and C++ code using Simulink® Coder™.

### **See Also**

#### **Topics**

“Configure AUTOSAR Sender-Receiver Data Invalidation”

**Introduced in R2015b**



# Tools — Alphabetical List

---

## Code Mappings Editor

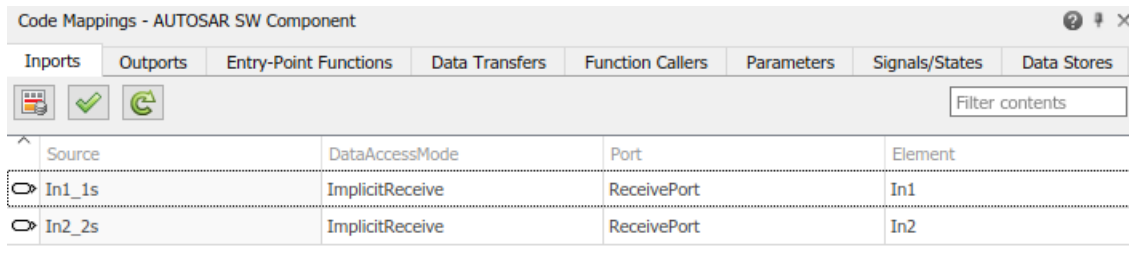
Map AUTOSAR elements for code generation

### Description

Code Mappings editor is a graphical interface for mapping AUTOSAR elements for code generation. Map Simulink model elements such as inports, outports, and entry-point functions to AUTOSAR component elements such as receiver ports, sender ports, and runnables.

Using a tabbed table format, Code Mappings editor displays model inports, outports, and other model elements relevant to your AUTOSAR platform. Use this view to map model elements to AUTOSAR component elements from a Simulink model perspective. The mappings that you configure are reflected in generated AUTOSAR-compliant C code and exported arxml descriptions.

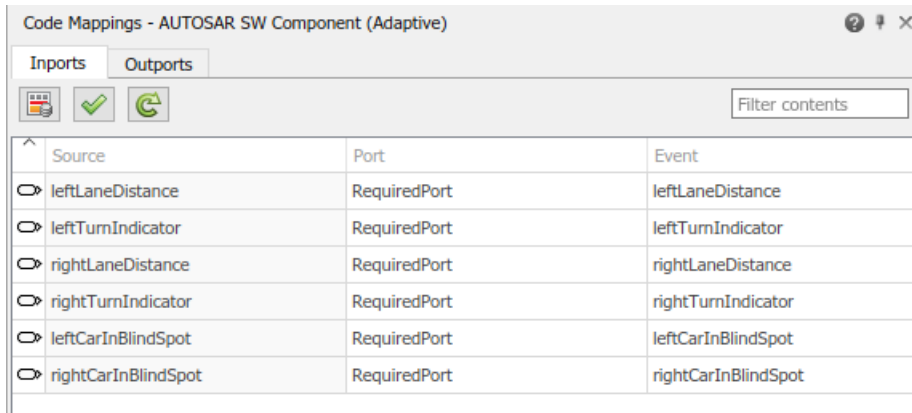
For more information, see “Map AUTOSAR Elements for Code Generation” or “Map AUTOSAR Adaptive Elements for Code Generation”.






The screenshot shows the 'Code Mappings - AUTOSAR SW Component' window. It has a tabbed interface with 'Inports' selected. Below the tabs are icons for a grid, a checkmark, and a refresh button, along with a 'Filter contents' search box. The main area contains a table with the following data:

Source	DataAccessMode	Port	Element
In1_1s	ImplicitReceive	ReceivePort	In1
In2_2s	ImplicitReceive	ReceivePort	In2





Code Mappings editor provides in-canvas access to AUTOSAR mapping information, with batch editing, element filtering, easy navigation to model elements and AUTOSAR properties, and model element traceability. As you progressively configure the model representation of the AUTOSAR component, you can apply these Code Mappings editor controls:

- **Filter contents** field - Selectively display some elements, while omitting others, in the current view.
- **AUTOSAR Dictionary** button  - Switch from the Code Mappings editor view of a Simulink element to the AUTOSAR Dictionary view of the corresponding AUTOSAR element.
- **Validate** button  - Validate the AUTOSAR component configuration.
- **Update** button  - Update the Simulink to AUTOSAR mapping of the model to reflect model changes, such as adding, changing, or removing Simulink data transfers, entry-point functions, and function callers.

## Open the Code Mappings Editor

- If your model already has a mapped AUTOSAR software component, in the model window, do one of the following:
  - Select **Code > C/C++ Code > Configure Model in Code Perspective**.

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

The model opens in the AUTOSAR code perspective, which includes Code Mappings editor.

- If your model does not have a mapped AUTOSAR component, in the model window, do one of the following:
  - Use AUTOSAR Component Quick Start.
    - 1 Open the Configuration Parameters dialog box, **Code Generation** pane, and set the system target file to either `autosar.tlc` or `autosar_adaptive.tlc`. Click **OK**.
    - 2 To open AUTOSAR Component Quick Start, select **Code > C/C++ Code > Configure Model in Code Perspective**.
    - 3 Work through the component quick-start procedure and click **Finish**.
  - Use Embedded Coder Quick Start.
    - 1 Select **Code > C/C++ Code > Embedded Coder Quick Start**.
    - 2 As you work through the quick-start procedure, in the Output window, select output option **C code compliant with AUTOSAR** or **C++ code compliant with AUTOSAR Adaptive Platform**.
    - 3 Click **Finish**.

The model opens in the AUTOSAR code perspective, which includes Code Mappings editor.

## Examples

### Map Model Elements to AUTOSAR Component Elements

If you are modeling for the AUTOSAR Classic Platform, navigate Code Mappings editor tabs to perform these actions:

- “Map Inports and Outports to AUTOSAR Sender-Receiver Ports”
- “Map Entry-Point Functions to AUTOSAR Runnables”
- “Map Data Transfers to AUTOSAR Inter-Runnable Variables”

- “Map Function Callers to AUTOSAR Client-Server Ports and Operations”
- “Map Model Workspace Parameters to AUTOSAR Component Internal Parameters”
- “Map Block Signals and States to AUTOSAR Variables”
- “Map Data Stores to AUTOSAR Variables”

If you are modeling for the AUTOSAR Adaptive Platform, navigate Code Mappings editor tabs to:

“Map Inports and Outports to AUTOSAR Required and Provided Service Ports”

## See Also

### Topics

“Map AUTOSAR Elements for Code Generation”

“Configure AUTOSAR Elements and Properties”

“Map AUTOSAR Adaptive Elements for Code Generation”

“Configure AUTOSAR Adaptive Elements and Properties”

“Configure and Map AUTOSAR Component Programmatically”

“AUTOSAR Component Configuration”

### Introduced in R2018a



# Model Advisor Checks

---

## MathWorks AUTOSAR Blockset Checks

### MathWorks Automotive Advisory Board Checks

MathWorks® AUTOSAR Blockset and checks facilitate designing and troubleshooting models from which code is generated for automotive applications.

The Model Advisor performs a checkout of the Simulink Check™ license when you run the AUTOSAR Blockset checks.

### Check model configuration parameters for AUTOSAR compliance

**Check ID:** `mathworks.autosar.autosar_configset`

#### Description

Check configuration settings in the model configuration that apply to AUTOSAR compatibility.

Available with AUTOSAR Blockset.

#### Results and Recommended Actions

Condition	Recommended Action
One or more of the model configuration parameters are not compatible with AUTOSAR.	Set the listed configuration parameters to the recommended values. alternatively, you can automatically fix the parameters to the recommended actions by using Auto-Fix option.

Following are the model parameters the check will look into provided that the the **AUTOSAR Compliance** is set to **on** by using a proper license (TLC file).

Parameter	Recommended Values	Auto Fix	Condition Dependencies
AutoInsertRateTransitionBlock	off	off	STC = STIndependent && SolverMode = SingleTasking
AutosarCompliant	On	On	
AutosarMaxShortNameLength	range(32,128)	128	~isAdaptiveAutosar
CombineOutputUpdateFcns	on	on	
ERTFilePackagingFormat	Modular	Modular	CodeInterfacePackaging = reusable function
InlineParams	On	On	CodeInterfacePackaging = reusable function
RateTransitionBlockCode	inline	inline	
SFInvalidInputDataAccessInChartInitDiag	warning error	warning	
SimulationMode	normal external SIL PIL	normal	
SupportComplex	off	off	~isAdaptiveAutosar
SupportContinuousTime	off	off	
SupportNonFinite	off	off	
SupportNonInlinedSFCns	off	off	

### Capabilities and Limitations

- Runs on library models.
- Allows exclusions of blocks and charts.

## Check compatibility of AUTOSAR Interpolation Routines

**Check ID:** `mathworks.autosar.lut_replacement_check`

### Description

Identifies the Simulink Lookup Table blocks that are incompatible with AUTOSAR Interpolation Routines.

Available with AUTOSAR Blockset.

### Results and Recommended Actions

Condition	Recommended Action
The Model Parameter Code Replacement Library is set to <b>None</b> .	The Model Parameter Code Replacement Library should not be set to <b>None</b> .

BlockType	Condition	Recommended Action
Prelookup	ParameterExtrapMethod is set to Clip.	Consider using AUTOSAR Prelookup block for better compatibility.
n-D Lookup Table	Parameter Dimensions set to <b>1</b> and the parameterExtrapMethod is set to Clip	Consider using AUTOSAR Curve block for better compatibility.
	Parameter Dimensions set to <b>2</b> and the parameterExtrapMethod is set to Clip	Consider using AUTOSAR Map block for better compatibility.
Interpolation Using Prelookup	Parameter Dimensions set to <b>1</b> and the parameterExtrapMethod is set to Clip	Consider using AUTOSAR Curve Using Prelookup block



BlockType	Condition	Recommended Action
	Parameter Dimensions is set to <b>2</b> and the parameter ExtrapMethod is set to Clip	Consider using AUTOSAR Map Using Prelookup block

### Capabilities and Limitations

- Runs on library models.
- Analyzes content of library-linked blocks. By default, the input parameter **Follow links** is set to on.
- Analyzes content in masked subsystems. By default, the input parameter **Look under masks** is set to graphical.
- Allows exclusions of blocks and charts.

