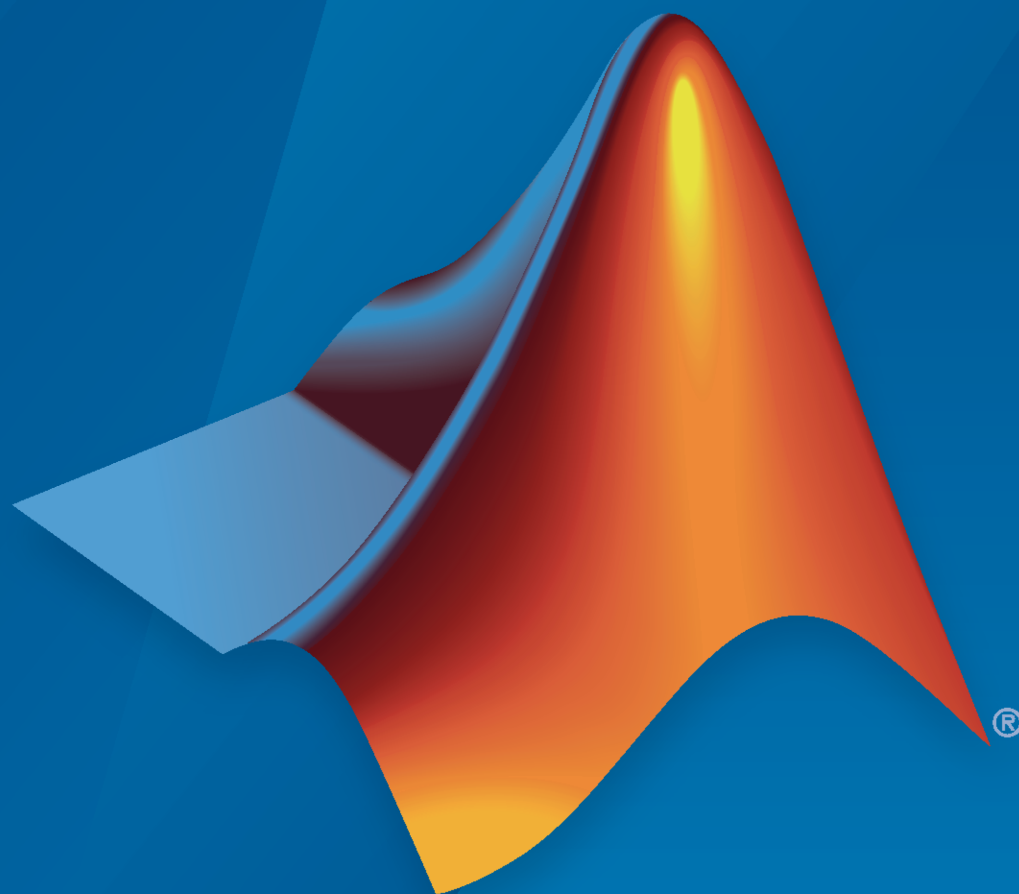
AUTOSAR Blockset

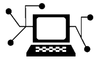
Reference



MATLAB® & SIMULINK®

R2022a

How to Contact MathWorks



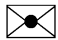
Latest news: www.mathworks.com

Sales and services: www.mathworks.com/sales_and_services

User community: www.mathworks.com/matlabcentral

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

Phone: 508-647-7000



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

AUTOSAR Blockset Reference

© COPYRIGHT 2019–2022 by The MathWorks, Inc.

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

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

Trademarks

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

Patents

MathWorks products are protected by one or more U.S. patents. Please see www.mathworks.com/patents for more information.

Revision History

March 2019	Online only	New for Version 2.0 (Release 2019a)
September 2019	Online only	Revised for Version 2.1 (Release 2019b)
March 2020	Online only	Revised for Version 2.2 (Release 2020a)
September 2020	Online only	Revised for Version 2.3 (Release 2020b)
March 2021	Online only	Revised for Version 2.4 (Release 2021a)
September 2021	Online only	Revised for Version 2.5 (Release 2021b)
March 2022	Online only	Revised for Version 2.6 (Release 2022a)

Contents

1 Functions

2 Blocks

3 Apps

4 Tools

5 Model Advisor Checks

AUTOSAR Blockset Checks **5-2**
MathWorks Automotive Advisory Board Checks **5-2**
Check model configuration parameters for AUTOSAR compliance **5-2**
Check compatibility of AUTOSAR Interpolation Routines **5-3**

Functions

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.

```
addpath(fullfile(matlabroot,'/examples/autosarblockset/main'));
hModel = 'autosar_swc_expfcns';
open_system(hModel);
arProps = autosar.api.getAUTOSARProperties(hModel);
add(arProps,'Interface1','DataElements','DE3');
get(arProps,'Interface1','DataElements')

ans =
  1×3 cell array
    {'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 =
  1×3 cell array
    {'/pkg/if/myMsIf'}    {'/pkg/if/MsIf2'}    {'/pkg/if/Interface3'}

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

```
ans =
    'Interface3/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
`autosar.api.getAUTOSARProperties` | `delete`

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

Introduced in R2013b

addBSWService

Package: `autosar.arch`

Add Basic Software component to AUTOSAR architecture model

Syntax

`bswBlock = addBSWService(archCM,bswKind)`

Description

`bswBlock = addBSWService(archCM,bswKind)` adds a Basic Software (BSW) service component block of type `bswKind` to composition or architecture model `archCM`. Valid values for `bswKind` are `'dem'` for Diagnostic Event Manager and `'nvm'` for NVRAM Manager (not case-sensitive). The `archCM` argument is a composition or architecture model handle returned by a previous call to `addComposition`, `autosar.arch.createModel`, or `autosar.arch.loadModel`. The `bswBlock` output argument returns a block handle.

Examples

Add BSW Service Component Blocks to AUTOSAR Architecture Model Top Level

Add NVRAM Service Component and Diagnostic Service Component blocks to the top level of an AUTOSAR architecture model.

```
% Create AUTOSAR architecture model
modelName = 'myArchModel';
archModel = autosar.arch.createModel(modelName);

% Add components inside the architecture model
addComponent(archModel,'Controller1');
actuator = addComponent(archModel,'Actuator');
set(actuator,'Kind','SensorActuator');

% Add Basic Software service component blocks
addBSWService(archModel,'nvm');
addBSWService(archModel,'dem');
layout(archModel);  % Auto-arrange layout
```

Input Arguments

`archCM` — Composition or architecture model

handle

AUTOSAR composition or architecture model to which to add a BSW component. The argument is a composition or architecture model handle returned by a previous call to `addComposition`, `autosar.arch.createModel`, or `autosar.arch.loadModel`.

Example: `archModel`

`bswKind` — BSW service component block type

`'dem'` | `'nvm'`

Type of AUTOSAR BSW service component block to add to the specified composition or architecture model. Valid values are `'dem'` for Diagnostic Event Manager and `'nvm'` for NVRAM Manager (not case-sensitive).

Example: `'dem'`

Output Arguments

`bswBlock` — BSW block
handle

Returns an AUTOSAR BSW block handle.

See Also

Diagnostic Service Component | NVRAM Service Component | `layout`

Topics

"Configure AUTOSAR Architecture Model Programmatically"
"Configure AUTOSAR Scheduling and Simulation"
"Author AUTOSAR Compositions and Components in Architecture Model"

Introduced in R2020a

addComponent

Package: `autosar.arch`

Add component to AUTOSAR architecture model

Syntax

```
components = addComponent(archCM,compNames)
components = addComponent(archCM,compNames,'Kind',value)
```

Description

`components = addComponent(archCM,compNames)` adds one or more components specified in the `compNames` argument to composition or architecture model `archCM`. The `archCM` argument is a composition or architecture model handle returned by a previous call to `addComposition`, `autosar.arch.createModel`, or `autosar.arch.loadModel`. The `components` output argument returns one or more component handles, which are `autosar.arch.Component` objects.

`components = addComponent(archCM,compNames,'Kind',value)` allows you to specify the component type for all added components. Valid component types are `Application` (the default), `SensorAccuator`, `ComplexDeviceDriver`, `EcuAbstraction`, and `ServiceProxy`.

Examples

Add Components to AUTOSAR Architecture Model

In an AUTOSAR architecture model:

1. Add a composition named `Sensors` and, inside the composition, add AUTOSAR sensor-actuator components named `PedalSnsr` and `ThrottleSnsr`.
2. At the top level of the model, add an application component named `Controller1` and a sensor-actuator component named `Actuator`.

```
% Create AUTOSAR architecture model
modelName = 'myArchModel';
archModel = autosar.arch.createModel(modelName);

% Add a composition
composition = addComposition(archModel,'Sensors');

% Add 2 components inside Sensors
names = {'PedalSnsr','ThrottleSnsr'};
sensorSWCs = addComponent(composition,names,'Kind','SensorActuator');
layout(composition); % auto-arrange layout

% Add components at architecture model top level
addComponent(archModel,'Controller1');
actuator = addComponent(archModel,'Actuator');
```

```
set(actuator,'Kind','SensorActuator');
layout(archModel);  % Auto-arrange layout
```

Input Arguments

`archCM` — Composition or architecture model
handle

AUTOSAR composition or architecture model to which to add one or more components. The argument is a composition or architecture model handle returned by a previous call to `addComposition`, `autosar.arch.createModel`, or `autosar.arch.loadModel`.

Example: `archModel`

`compNames` — Component names
character vector | string scalar | cell array of character vectors | string array

Names of the components to add to the specified composition or architecture model.

Example: `{'PedalSnsr','ThrottleSnsr'}`

`'Kind',value` — Specify component type
`'Application'` (default) | `'SensorAccuator'` | `'ComplexDeviceDriver'` | `'EcuAbstraction'` | `'ServiceProxy'`

Type of AUTOSAR components to add to the specified composition or architecture model. The specified type applies to all added components.

Example: `'Kind','SensorActuator'`

Output Arguments

`components` — Added components
handle | array of handles

Returns one or more AUTOSAR component handles, which are `autosar.arch.Component` objects, with component properties.

See Also

Software Component | `addComposition` | `addPort` | `connect` | `destroy` | `importFromARXML` | `layout`

Topics

"Configure AUTOSAR Architecture Model Programmatically"
"Add and Connect AUTOSAR Compositions and Components"
"Author AUTOSAR Compositions and Components in Architecture Model"

Introduced in R2020a

addComposition

Package: `autosar.arch`

Add composition to AUTOSAR architecture model

Syntax

`compositions = addComposition(archCM,compNames)`

Description

`compositions = addComposition(archCM,compNames)` adds one or more compositions specified in the `compNames` argument to composition or architecture model `archCM`. The `archCM` argument is a composition or architecture model handle returned by a previous call to `addComposition`, `autosar.arch.createModel`, or `autosar.arch.loadModel`. The `compositions` output argument returns one or more composition handles, which are `autosar.arch.Composition` objects.

Examples

Add Composition with Components to AUTOSAR Architecture Model

In an AUTOSAR architecture model, add a composition named `Sensors`. Inside the composition, add AUTOSAR sensor-actuator components named `PedalSnsr` and `ThrottleSnsr`.

```
% Create AUTOSAR architecture model
modelName = 'myArchModel';
archModel = autosar.arch.createModel(modelName);

% Add a composition
composition = addComposition(archModel,'Sensors');

% Add 2 components inside Sensors
names = {'PedalSnsr','ThrottleSnsr'};
sensorSWCs = addComponent(composition,names,'Kind','SensorActuator');
layout(composition); % Auto-arrange layout
```

Input Arguments

`archCM` — Composition or architecture model

handle

AUTOSAR composition or architecture model to which to add one or more compositions. The argument is a composition or architecture model handle returned by a previous call to `addComposition`, `autosar.arch.createModel`, or `autosar.arch.loadModel`.

Example: `archModel`

`compNames` — Composition names

character vector | string scalar | cell array of character vectors | string array

Names of the compositions to add to the specified composition or architecture model.

Example: `{'Sensors','Actuators'}`

Output Arguments

`compositions` — Added compositions
handle | array of handles

Returns one or more AUTOSAR composition handles, which are `autosar.arch.Composition` objects, with composition properties.

See Also

Software Composition | `addComponent` | `addPort` | `connect` | `destroy` | `importFromARXML` | `layout`

Topics

"Configure AUTOSAR Architecture Model Programmatically"
"Add and Connect AUTOSAR Compositions and Components"
"Author AUTOSAR Compositions and Components in Architecture Model"

Introduced in R2020a

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`.

```
addpath(fullfile(matlabroot,'/examples/autosarblockset/main'));
hModel = 'autosar_swc_expfcns';
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 =
  1×1 cell array
    {'/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

`autosar.api.getAUTOSARProperties` | `delete`

Topics

"AUTOSAR Property and Map Function Examples"
"Configure and Map AUTOSAR Component Programmatically"
"AUTOSAR Component Configuration"

Introduced in R2014b

addPort

Package: autosar.arch

Add port to AUTOSAR component, composition, or architecture model

Syntax

ports = addPort(archCCM,portKind,portNames)

Description

ports = addPort(archCCM,portKind,portNames) adds one or more ports of type portKind to component, composition, or architecture model archCCM. Valid values for portKind are 'Receiver' and 'Sender'. The portNames argument specifies the names of one or more ports to add. The archCCM argument is a component, composition, or architecture model handle returned by a previous call to addComponent, addComposition, autosar.arch.createModel, or autosar.arch.loadModel. The ports output argument returns one or more port handles, which are autosar.arch.CompPort or autosar.arch.ArchPort objects.

Examples

Add Ports to AUTOSAR Architecture Model, Composition, and Components

In an AUTOSAR architecture model:

1. Add a composition named Sensors.
2. At the top level of the model, add an application component named Controller1 and a sensor-actuator component named Actuator.
3. For the architecture model, add two receiver (input) ports and a sender (output) port. The ports appear at the architecture model boundary.
4. For the composition block, add two receiver ports and two sender ports. The composition receiver port names match the names of the architecture model receiver ports to which they connect.
5. For the component blocks, add receiver and sender ports. The component receiver and sender port names match the names of the component, composition, or architecture model ports to which they connect.

```
% Create AUTOSAR architecture model
modelName = 'myArchModel';
archModel = autosar.arch.createModel(modelName);

% Add a composition
composition = addComposition(archModel,'Sensors');

% Add components at architecture model top level
addComponent(archModel,'Controller1');
actuator = addComponent(archModel,'Actuator');
set(actuator,'Kind','SensorActuator');

% Add architecture ports
```

```
addPort(archModel,'Receiver',{'TPS_Hw','APP_Hw'});
addPort(archModel,'Sender','ThrCmd_Hw');

% Add composition ports
addPort(composition,'Receiver',{'TPS_Hw','APP_Hw'});
addPort(composition,'Sender',{'TPS_Perc','APP_Perc'});

% Add component ports
controller = find(archModel,'Component','Name','Controller1');
addPort(controller,'Receiver',{'TPS_Perc','APP_Perc'});
addPort(controller,'Sender','ThrCmd_Perc');
addPort(actuator,'Receiver','ThrCmd_Perc');
addPort(actuator,'Sender','ThrCmd_Hw');

layout(archModel);  % Auto-arrange layout
```

Input Arguments

archCCM — Component, composition, or architecture model

handle

AUTOSAR component, composition, or architecture model to which to add one or more ports. The argument is a component, composition, or architecture model handle returned by a previous call to `addComponent`, `addComposition`, `autosar.arch.createModel`, or `autosar.arch.loadModel`.

Example: `archModel`

portKind — Port type

`'Receiver'` | `'Sender'`

Type of AUTOSAR ports to add to the specified component, composition, or architecture model. The specified type applies to all added ports.

Example: `'Receiver'`

portNames — Port names

character vector | string scalar | cell array of character vectors | string array

Names of the ports to add to the specified component, composition, or architecture model.

Example: `{'TPS_Hw','APP_Hw'}`

Output Arguments

ports — Added ports

handle | array of handles

Returns one or more AUTOSAR port handles, which are `autosar.arch.CompPort` or `autosar.arch.ArchPort` objects, with port properties.

See Also

`addComponent` | `addComposition` | `connect` | `destroy` | `importFromARXML` | `layout`

Topics

“Configure AUTOSAR Architecture Model Programmatically”
“Add and Connect AUTOSAR Compositions and Components”
“Author AUTOSAR Compositions and Components in Architecture Model”

Introduced in R2020a

addSignal

Package: `autosar.api`

Add Simulink block signal to AUTOSAR mapping

Syntax

`addSignal(slMap,slPortHandle)`

Description

`addSignal(slMap,slPortHandle)` adds the Simulink® block signal associated with outport port handle `slPortHandle` to AUTOSAR mapping. The signal then can be mapped to an AUTOSAR variable, for example, by using the `mapSignal` function.

Examples

Add and Map Simulink Block Signal

In example model `autosar_swc_counter`:

1. Create a new default AUTOSAR mapping.
2. Add Simulink signal `equal_to_count`, which originates in the `RelOpt` block, to the AUTOSAR component signal mapping.
3. Map the signal to AUTOSAR static memory and set `ReadWrite` calibration access.

```
hModel = 'autosar_swc_counter';
addpath(fullfile(matlabroot,'/examples/autosarblockset/main'));
open_system(hModel);
autosar.api.create(hModel,'default'); % Create default AUTOSAR mapping
slMap = autosar.api.getSimulinkMapping(hModel);

portHandles = get_param('autosar_swc_counter/RelOpt','portHandles');
outportHandle = portHandles.Outport;
addSignal(slMap,outportHandle)

mapSignal(slMap,outportHandle,'StaticMemory',...
  'SwCalibrationAccess','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 outport port handle for a block signal
handle

Outport port handle for a Simulink block signal to add to AUTOSAR mapping. Use MATLAB® commands to construct the outport port handle. For example, for a Relational Operator block named `RelOpt`:

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

Example: `outportHandle`

See Also

`autosar.api.getSimulinkMapping` | `getSignal` | `mapSignal` | `removeSignal`

Topics

"Map Block Signals and States to AUTOSAR Variables"
"Map Submodel Signals and States to AUTOSAR Variables"
"Configure AUTOSAR Per-Instance Memory"
"Configure AUTOSAR Static Memory"
"AUTOSAR Property and Map Function Examples"
"AUTOSAR Component Configuration"

Introduced in R2020b

arxml.importer

Import AUTOSAR XML descriptions of software components, compositions, or packages

Description

Use `arxml.importer` functions to import AUTOSAR software components, compositions, or packages of shared elements from ARXML files into Simulink. 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, you can use the Simulink representation of the component for further configuration, algorithm development, C/C++ code generation, and ARXML export.

For a list of schema versions supported for ARXML import and export, see “Select AUTOSAR Classic Schema” or “Select AUTOSAR Adaptive Schema”.

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 from AUTOSAR element packages
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 XML Descriptions Into Simulink"
"Import AUTOSAR Component to Simulink"
"Import AUTOSAR Composition to Simulink"
"Import AUTOSAR Software Component Updates"
"Import and Reference Shared AUTOSAR Element Definitions"
"Import AUTOSAR Package into Component Model"
"Import AUTOSAR Adaptive Software Descriptions"
"Import AUTOSAR Adaptive Components to Simulink"
"Import AUTOSAR Package into Adaptive Component Model"
"AUTOSAR ARXML Importer"
"Round-Trip Preservation of AUTOSAR XML File Structure and Element Information"

Introduced in R2008a

autosar.api.create

Create or update mapped AUTOSAR component model

Syntax

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

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`.

`autosar.api.create(model,mode,Name,Value)` specifies additional options for mapping with one or more `Name,Value` pair arguments.

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

Map Submodel Referenced From AUTOSAR Component Model

Create AUTOSAR properties and Simulink to AUTOSAR mapping for a submodel referenced from an AUTOSAR component model.

```
addpath(fullfile(matlabroot,'/examples/autosarblockset/main'));
open_system('autosar_subcomponent');
autosar.api.create('autosar_subcomponent','default','ReferencedFromComponentModel',true);
```

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 outports 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 outports. 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'`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `'ReferencedFromComponentModel',true` maps a model as a referenced submodel.

`ReferencedFromComponentModel` — Designate whether the model is referenced from a component model

`false` (default) | `true`

Specify whether the model is a submodel referenced from an AUTOSAR software component model. In a mapped submodel, you can use the Code Mappings editor to configure the submodel internal data for calibration.

Example: `'ReferencedFromComponentModel',true`

See Also

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

Topics

"Incrementally Update AUTOSAR Mapping After Model Changes"
"Map Calibration Data for Submodels Referenced from AUTOSAR Component Models"
"Configure and Map AUTOSAR Component Programmatically"
"AUTOSAR Component Configuration"

Introduced in R2013b

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_swc_counter';
addpath(fullfile(matlabroot,'/examples/autosarblockset/main'));
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 from imported AUTOSAR data elements
createManifest	Create manifest file for AUTOSAR adaptive model
createNumericType	Create Simulink numeric data type definition from imported AUTOSAR data elements
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.

```
addpath(fullfile(matlabroot,'/examples/autosarblockset/main'));
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. In an AUTOSAR adaptive model, use AUTOSAR map functions to configure generated C++ class names and namespaces for your adaptive application. For example, you can:

- Map a Simulink entry-point function to an AUTOSAR runnable and optional software address methods.
- Map a Simulink inport or outport to an AUTOSAR receiver or sender port and a sender-receiver data element.
- Map a Simulink model workspace parameter to an AUTOSAR component parameter.
- Map a Simulink data store to an AUTOSAR variable.
- Add or remove Simulink block signals from AUTOSAR component mapping.
- Map a Simulink block signal or state to an AUTOSAR variable.
- Set the default data packaging for Simulink internal data stores, signals, and states in AUTOSAR generated code.
- 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.
- Control generated C++ class name or namespace for adaptive applications.

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

addSignal	Add Simulink block signal to AUTOSAR mapping
getClassName	Get class name of model
getClassNamespace	Get class namespace for a model
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
getInternalDataPackaging	Get default internal data packaging for AUTOSAR component model
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 parameter
mapSignal	Map Simulink block signal to AUTOSAR variable
mapState	Map Simulink block state to AUTOSAR variable
removeSignal	Remove Simulink block signal from AUTOSAR mapping
setClassName	Set class name of model
setClassNamespace	Set class namespace of model
setInternalDataPackaging	Set default internal data packaging for AUTOSAR component model

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_swc`. Use the returned object to map the Simulink initialize entry-point function to AUTOSAR runnable `Runnable_Init`.

```
hModel = 'autosar_swc';
addpath(fullfile(matlabroot,'/examples/autosarblockset/main'));
open_system(hModel);
slMap = autosar.api.getSimulinkMapping(hModel);
mapFunction(slMap,'Initialize','Runnable_Init');
arRunnableName = getFunction(slMap,'Initialize')
```

```
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

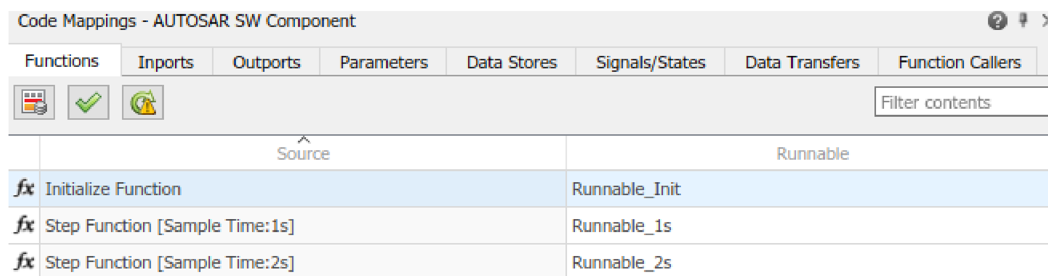
`autosar.api.syncModel(model)` updates the Simulink to AUTOSAR mapping of `model` with modifications made to Simulink elements, such as data transfers, entry-point functions, and function callers.

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

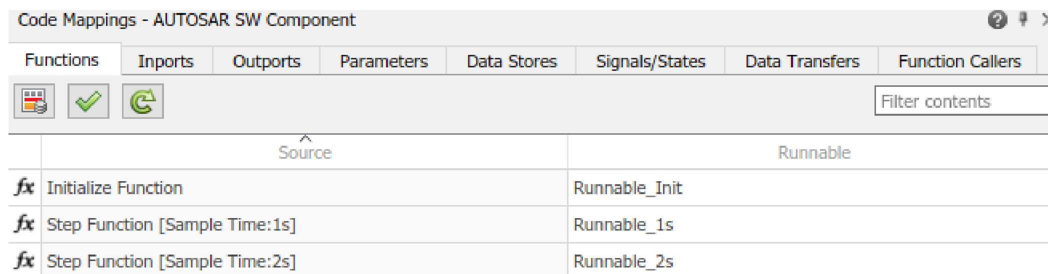
Examples

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 Code Mappings editor display indicates that Simulink elements need loading or updating.



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



```
hModel = 'autosar_swc';
addpath(fullfile(matlabroot,'/examples/autosarblockset/main'));
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`.

If Simulink Coder™ and Embedded Coder are not licensed on your system, the function validates only the 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.arch.createModel

Create AUTOSAR architecture model

Syntax

```
archModel = autosar.arch.createModel(modelName)
archModel = autosar.arch.createModel(modelName,openFlag)
```

Description

`archModel = autosar.arch.createModel(modelName)` creates and opens AUTOSAR architecture model `modelName`. The output argument `archModel` returns a model handle, which is an `autosar.arch.Model` object.

`archModel = autosar.arch.createModel(modelName,openFlag)` allows you to control whether the created AUTOSAR architecture model opens in the editor. Specify `openFlag` as `false` to create an architecture model without opening it in the editor.

Examples

Create and Open AUTOSAR Architecture Model

Create an AUTOSAR architecture model named `myArchModel`, open the model in the editor, and return model properties.

```
modelName = 'myArchModel';
archModel = autosar.arch.createModel(modelName)
```

```
archModel = 

  Model with properties:

            Name: 'myArchModel'
      Components: [0×0 autosar.arch.Component]
    Compositions: [0×0 autosar.arch.Composition]
           Ports: [0×0 autosar.arch.PortBase]
      Connectors: [0×0 autosar.arch.Connector]
  SimulinkHandle: 1.2207e-04
```

Create AUTOSAR Architecture Model without Opening

Create an AUTOSAR architecture model without opening it in the editor, and return model properties.

```
modelName = 'myArchModel';
archModel = autosar.arch.createModel(modelName,false)
```

```
archModel = 

  Model with properties:

            Name: 'myArchModel'
      Components: [0×0 autosar.arch.Component]
    Compositions: [0×0 autosar.arch.Composition]
           Ports: [0×0 autosar.arch.PortBase]
```

```
        Connectors: [0×0 autosar.arch.Connector]
    SimulinkHandle: 2.4414e-04
```

Input Arguments

`modelName` — Architecture model name
character vector | string scalar

Name of the AUTOSAR architecture model to create.

Example: `'myArchModel'`

`openFlag` — Open flag
true (default) | false

Specify `false` to create an architecture model without opening it in the editor.

Example: `false`

Output Arguments

`archModel` — Architecture model
handle

Returns an AUTOSAR architecture model handle, which is an `autosar.arch.Model` object.

See Also

`autosar.arch.loadModel` | `close` | `open` | `save`

Topics
"Configure AUTOSAR Architecture Model Programmatically"
"Create AUTOSAR Architecture Models"
"Author AUTOSAR Compositions and Components in Architecture Model"

Introduced in R2020a

autosar.arch.loadModel

Load AUTOSAR architecture model

Syntax

`archModel = autosar.arch.loadModel(modelName)`

Description

`archModel = autosar.arch.loadModel(modelName)` loads AUTOSAR architecture model `modelName` into memory without opening the model in the editor. The output argument `archModel` returns a model handle, which is an `autosar.arch.Model` object. After you load a model into memory, you can work with it by using architecture functions or open the model in the editor by using the `open` function. Save changes by using the `save` function.

Examples

Load AUTOSAR Architecture Model

Load an AUTOSAR architecture model into memory without opening the model in the editor, and return model properties.

```
modelName = 'autosar_tpc_composition';
addpath(fullfile(matlabroot,'/examples/autosarblockset/main'));
archModel = autosar.arch.loadModel(modelName)
```

```
archModel = 

  Model with properties:

              Name: 'autosar_tpc_composition'
        Components: [2×1 autosar.arch.Component]
      Compositions: [1×1 autosar.arch.Composition]
             Ports: [4×1 autosar.arch.ArchPort]
        Connectors: [7×1 autosar.arch.Connector]
    SimulinkHandle: 0.0031
```

Input Arguments

`modelName` — Architecture model name
character vector | string scalar

Name of the AUTOSAR architecture model to load into memory.

Example: `'myArchModel'`

Output Arguments

`archModel` — Architecture model
handle

Returns an AUTOSAR architecture model handle, which is an `autosar.arch.Model` object.

See Also

`autosar.arch.createModel` | `close` | `open` | `save`

Topics

"Configure AUTOSAR Architecture Model Programmatically"
"Create AUTOSAR Architecture Models"
"Author AUTOSAR Compositions and Components in Architecture Model"

Introduced in R2020a

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';
addpath(fullfile(matlabroot,'/examples/autosarblockset/main'));
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_ui_launch` | `autosar.api.create`

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

close

Package: `autosar.arch`

Close AUTOSAR architecture model

Syntax

```
close(archModel)
close(archModel,'Force')
```

Description

`close(archModel)` closes architecture model `archModel`. The `archModel` argument is a model handle returned by a previous call to `autosar.arch.createModel` or `autosar.arch.loadModel`. The model must be open or loaded with no unsaved changes.

`close(archModel,'Force')` closes the model without saving any unsaved changes.

Examples

Close AUTOSAR Architecture Model After Saving Change

Create an AUTOSAR architecture model, add a composition, save the change, and close the model.

```
% Create AUTOSAR architecture model
modelName = 'myArchModel';
archModel = autosar.arch.createModel(modelName);

% Add a composition
composition = addComposition(archModel,'Sensors2');

% Save the model
save(archModel);

% Close the model
close(archModel);
```

Input Arguments

`archModel` — Architecture model
handle

AUTOSAR architecture model to close. The argument is a model handle returned by a previous call to `autosar.arch.createModel` or `autosar.arch.loadModel`. The model must be open or loaded with no unsaved changes.

Example: `archModel`

See Also
`autosar.arch.createModel` | `autosar.arch.loadModel` | `open` | `save`

Topics
"Configure AUTOSAR Architecture Model Programmatically"
"Create AUTOSAR Architecture Models"
"Author AUTOSAR Compositions and Components in Architecture Model"

Introduced in R2020a

connect

Package: `autosar.arch`

Connect AUTOSAR architecture components and compositions

Syntax

```
connectors = connect(archModel,comp1,comp2)
connectors = connect(archCM,[],comp2)
connectors = connect(archCM,comp1,[])
connectors = connect(archModel,port1,port2)
```

Description

`connectors = connect(archModel,comp1,comp2)` connects the output ports of component or composition `comp1` to the input ports of component or composition `comp2`, based on matching port names. The `archModel` argument is a model handle returned by a previous call to `autosar.arch.createModel` or `autosar.arch.loadModel`. The `comp1` and `comp2` arguments are component or composition handles returned by previous calls to `addComponent`, `addComposition`, or `find`. The `connectors` output argument returns one or more connector handles, which are `autosar.arch.Connector` objects.

`connectors = connect(archCM,[],comp2)` connects the root input ports of parent composition or architecture model `archCM` to the input ports of child component or composition `comp2`, based on matching port names.

`connectors = connect(archCM,comp1,[])` connects the output ports of child component or composition `comp1` to the root output ports of parent composition or architecture model `archCM`, based on matching port names.

`connectors = connect(archModel,port1,port2)` connects component, composition, or root architecture port `port1` to component, composition or root architecture port `port2`. The `port1` and `port2` arguments are port handles returned by previous calls to `addPort` or `find`.

Examples

Connect AUTOSAR Architecture Components and Compositions Based On Matching or Specified Ports

In an AUTOSAR architecture model:

1. At the top level of the model, add a composition, an application component, and a sensor-actuator component.
2. For the architecture model, add two receiver (input) ports and a sender (output) port. The ports appear at the architecture model boundary.
3. For the composition block, add two receiver ports and two sender ports. The composition receiver port names match the names of the architecture model receiver ports to which they connect.

4. For the component blocks, add receiver and sender ports. The component receiver and sender port names match the names of the component, composition, or architecture model ports to which they connect.
5. At the top level of the model, connect the composition and the components based on matching port names.
6. Connect the architecture root ports to composition and component ports. Rather than relying on matching port names to make connections, use port handles to identify specific architecture, composition, and component ports.
7. Inside the `Sensors` composition, add sensor-actuator components named `PedalSnsr` and `ThrottleSnsr`.
8. For the component blocks, add receiver and sender ports. The component receiver and sender port names match the names of the composition root ports to which they connect.
9. Connect the `Sensors` composition root ports to component ports based on matching port names.

```
% Create AUTOSAR architecture model
modelName = 'myArchModel';
archModel = autosar.arch.createModel(modelName);

% Add a composition
composition = addComposition(archModel,'Sensors');

% Add components at architecture model top level
addComponent(archModel,'Controller1');
actuator = addComponent(archModel,'Actuator');
set(actuator,'Kind','SensorActuator');

% Add architecture ports
addPort(archModel,'Receiver',{'TPS_Hw','APP_Hw'});
addPort(archModel,'Sender','ThrCmd_Hw');

% Add composition ports
addPort(composition,'Receiver',{'TPS_Hw','APP_Hw'});
addPort(composition,'Sender',{'TPS_Perc','APP_Perc'});

% Add component ports
controller = find(archModel,'Component','Name','Controller1');
addPort(controller,'Receiver',{'TPS_Perc','APP_Perc'});
addPort(controller,'Sender','ThrCmd_Perc');
addPort(actuator,'Receiver','ThrCmd_Perc');
addPort(actuator,'Sender','ThrCmd_Hw');

% At top level, connect composition and components based on matching port names
connect(archModel,composition,controller);
connect(archModel,controller,actuator);

% Connect specified arch root ports to specified composition and component ports
connect(archModel,archModel.Ports(1),composition.Ports(1));
% Use find to construct port specifications
connect(archModel,...
    find(archModel,'Port','Name','APP_Hw'),...
    find(composition,'Port','Name','APP_Hw'));
connect(archModel,actuator.Ports(2),archModel.Ports(3));
% ALTERNATIVELY, connect architecture root ports based on matching port names
% connect(archModel,[],composition);
% connect(archModel,actuator,[]);

layout(archModel);  % Auto-arrange architecture model layout

% Add 2 components inside Sensors composition
names = {'PedalSnsr','ThrottleSnsr'};
sensorSWCs = addComponent(composition,names,'Kind','SensorActuator');

% Add component ports inside Sensors
pSnsr = find(composition,'Component','Name','PedalSnsr');
tSnsr = find(composition,'Component','Name','ThrottleSnsr');
```

```
addPort(pSnsr,'Receiver','APP_Hw');
addPort(pSnsr,'Sender','APP_Perc');
addPort(tSnsr,'Receiver','TPS_Hw');
addPort(tSnsr,'Sender','TPS_Perc');

% Connect composition root ports to component ports based on matching port names
connect(composition,[],pSnsr);
connect(composition,pSnsr,[]);
connect(composition,[],tSnsr);
connect(composition,tSnsr,[]);

layout(composition);  % Auto-arrange composition layout
```

Input Arguments

`archModel` — Architecture model

handle

AUTOSAR architecture model in which to connect ports. The argument is a model handle returned by a previous call to `autosar.arch.createModel` or `autosar.arch.loadModel`.

Example: `archModel`

`archCM` — Composition or architecture model

handle

AUTOSAR composition or architecture model in which to connect parent and child ports based on matching port names. The argument is a composition or architecture model handle returned by a previous call to `addComposition`, `autosar.arch.createModel`, or `autosar.arch.loadModel`.

Example: `archModel`

`comp1` — Component or composition

handle

Component or composition for which to connect output ports based on matching port names. The argument is a component or composition handle returned by a previous call to `addComponent`, `addComposition`, or `find`.

Example: `composition`

`comp2` — Component or composition

handle

Component or composition for which to connect input ports based on matching port names. The argument is a component or composition handle returned by a previous call to `addComponent`, `addComposition`, or `find`.

Example: `controller`

`port1` — Component, composition, or root architecture port

handle

Component, composition, or root architecture port to connect to another specified port. The argument is a port handle returned by a previous call to `addPort` or `find`.

Example: `archModel.Ports(1)`

`port2` — Component, composition, or root architecture port

handle

Component, composition, or root architecture port to connect to another specified port. The argument is a port handle returned by a previous call to `addPort` or `find`.

Example: `composition.Ports(1)`

Output Arguments

`connectors` — Added connectors
handle | array of handles

Returns one or more AUTOSAR connector handles, which are `autosar.arch.Connector` objects, with connector properties.

See Also

`addComponent` | `addComposition` | `addPort` | `destroy` | `find` | `importFromARXML` | `layout`

Topics
"Configure AUTOSAR Architecture Model Programmatically"
"Add and Connect AUTOSAR Compositions and Components"
"Author AUTOSAR Compositions and Components in Architecture Model"

Introduced in R2020a

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 `arxml.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

handle

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

`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 pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

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.sldd'`

`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'`

`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 argument 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 argument 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'` argument. This argument determines how the importer models AUTOSAR periodic runnables in the created model. See the argument description under "Name-Value Pair Arguments" on page 1-47.

See Also

`arxml.importer` | `getComponentNames`

Topics

"Import AUTOSAR XML Descriptions Into Simulink"
"Import AUTOSAR Component to Simulink"
"Import AUTOSAR Adaptive Software Descriptions"
"Import AUTOSAR Adaptive Components 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 `/Company/Components/ThrottlePositionControlComposition` from the file `ThrottlePositionControlComposition.arxml`. The ARXML file is located at `*matlabroot*/examples/autosarblockset/data`, which is on the default MATLAB path. Create an initial Simulink representation of the composition.

```
ar = arxml.importer('ThrottlePositionControlComposition.arxml');
names = getComponentNames(ar,'Composition')

names =
  1×1 cell array
    {'/Company/Components/ThrottlePositionControlComposition'}

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

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 Share AUTOSAR Dictionary

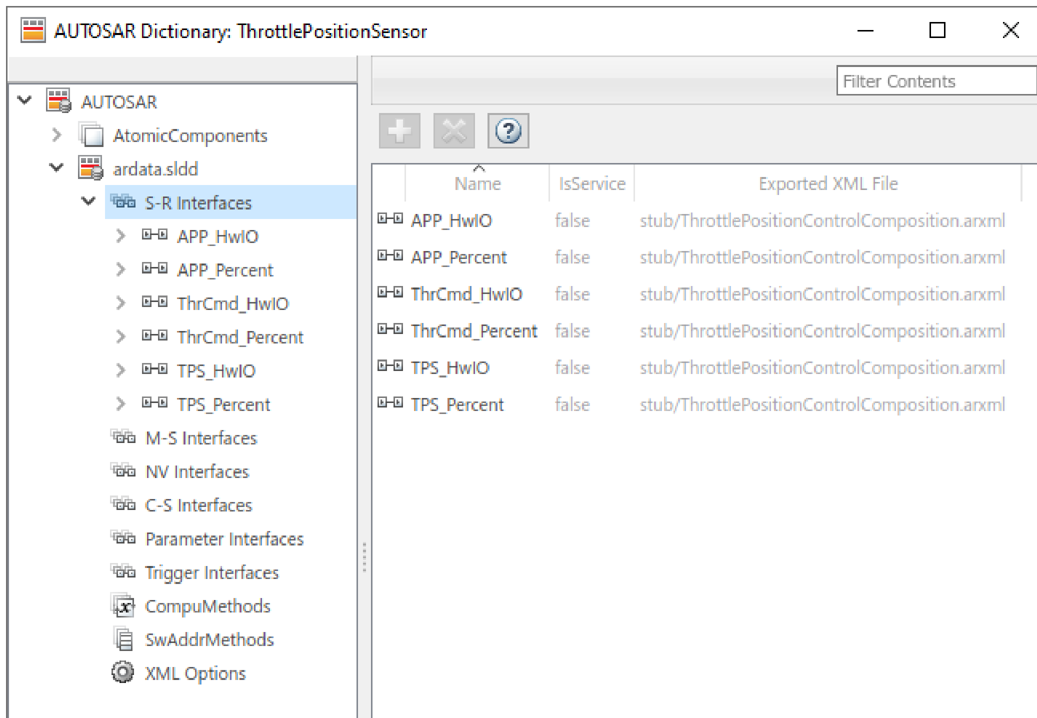
Import AUTOSAR software composition `/Company/Components/ThrottlePositionControlComposition` from the file `ThrottlePositionControlComposition.arxml`. The ARXML file is located at *`matlabroot`*`/examples/autosarblockset/data`, which is on the default MATLAB path. Create an initial Simulink representation of the composition.

For each imported component, the importer stores sharable AUTOSAR properties, such as interfaces and data types, in data dictionary `ardata.sldd`. Components within the composition can then share the stored properties.

```
ar = arxml.importer('ThrottlePositionControlComposition.arxml')
createCompositionAsModel(ar,'/Company/Components/ThrottlePositionControlComposition',...
    'ModelPeriodicRunnablesAs','Auto',...
    'DataDictionary','ardata.sldd','ShareAUTOSARProperties',true);
```

To view the shared properties, open the AUTOSAR dictionary for a component model. This example opens `ThrottlePositionSensor`. Expand the AUTOSAR dictionary node **ardata.sldd**. You can view read-only properties, such as shared component interfaces, and modify XML options for composition and component export.

```
autosar_ui_launch('ThrottlePositionSensor')
```



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
handle

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

`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 pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

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.

If you specify `true` for the `'ShareAUTOSARProperties'` argument, the specified data dictionary also stores sharable AUTOSAR properties, such as interfaces and data types, for sharing among components in the composition.

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'`

`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 argument 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'`

`ShareAUTOSARProperties` — Add AUTOSAR component properties to shared dictionary
`false` (default) | `true`

To improve the performance of common tasks in AUTOSAR composition modeling, composition import can store sharable component properties, such as interfaces and data types, into a Simulink data dictionary. Components within the composition can then share the stored properties.

For compositions containing more than 20 software components, sharing AUTOSAR properties among components can significantly improve performance for composition workflows, including import, dictionary navigation, AUTOSAR validation, and code generation. Limiting property replication among components can reduce component model file sizes.

The shared AUTOSAR dictionary provides a central location for viewing and configuring AUTOSAR composition and component properties. You can view read-only properties, such as shared component interfaces, and modify XML options for composition and component export.

To share AUTOSAR properties, specify `true`. For each imported component, the function stores sharable AUTOSAR properties, such as interfaces and data types, in the Simulink data dictionary specified by the `'DataDictionary'` argument. The `'DataDictionary'` argument must be specified.

For more information, see "Import AUTOSAR Composition and Share AUTOSAR Dictionary" on page 1-51.

Example: `'ShareAUTOSARProperties',true`

`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 argument 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 | getComponentNames

Topics
“Import AUTOSAR XML Descriptions Into Simulink”
“Import AUTOSAR Composition to Simulink”
“AUTOSAR ARXML Importer”
“Control AUTOSAR Variants with Predefined Value Combinations”

Introduced in R2017b

createEnumeration

Package: `autosar.api`

Create Simulink enumeration data type definition from imported AUTOSAR data elements

Syntax

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

Description

`createEnumeration(arProps,name,applicationDataTypePath)` creates a Simulink enumeration type from an AUTOSAR application data type. The function can be used to work with AUTOSAR elements that you imported by using `updateAUTOSARProperties`.

`createEnumeration(arProps,name,compuMethodPath,implementationDataTypePath)` creates a Simulink enumeration type from an AUTOSAR implementation data type and CompuMethod.

`createEnumeration(arProps,compuMethodPath)` creates a family of Simulink enumeration types from an AUTOSAR CompuMethod.

Examples

Create Enumeration Data Type from AUTOSAR Application Data Type

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

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

Create Enumeration Data Type from AUTOSAR Implementation Data Type and CompuMethod

Create a Simulink enumeration data type definition with the name `myEnum` from the AUTOSAR 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 AUTOSAR application data type for created Simulink enumeration 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 enumeration data types
character vector (default) | string scalar

Path to the AUTOSAR CompuMethod, which is used to translate between the enumeration implementation data type and 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 AUTOSAR implementation data type for created Simulink enumeration 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

`autosar.api.getAUTOSARProperties` | `updateAUTOSARProperties` | `createNumericType`

Topics

"AUTOSAR Property and Map Function Examples"
"Configure and Map AUTOSAR Component Programmatically"
"AUTOSAR Component Configuration"
"Model AUTOSAR Data Types"

Introduced in R2019a

createManifest

Package: `autosar.api`

Create manifest file for AUTOSAR adaptive model

Syntax

`createManifest(arProps)`

Description

`createManifest(arProps)` creates an execution manifest `JSON` file for the adaptive application. The manifest file modifies the default logging behavior of the adaptive application Linux® executable, providing properties such as the logging mode and verbosity level.

Examples

Create AUTOSAR Properties Object and Create Manifest File

Call the `autosar.api.getAUTOSARProperties` function to create object `arProps`, which represents AUTOSAR properties information for the model `autosar_LaneGuidance`. Use the returned object to create an execution manifest `JSON` file for the specified adaptive model.

```
addpath(fullfile(matlabroot,'/examples/autosarblockset/main'));
hModel = 'autosar_LaneGuidance';
open_system(hModel);
arProps = autosar.api.getAUTOSARProperties(hModel);
createManifest(arProps);
```

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`

See Also

`autosar.api.getAUTOSARProperties`

Topics
"Configure Run-Time Logging for AUTOSAR Adaptive Executables"

Introduced in R2021a

createModel

Package: `autosar.arch`

Create Simulink implementation model for AUTOSAR architecture component

Syntax

`createModel(component,modelName)`

Description

`createModel(component,modelName)` creates Simulink implementation model `modelName` with the same interface as the specified AUTOSAR architecture component and links the component to the implementation model. The `component` argument is a component handle returned by a previous call to `addComponent`. If not specified, `modelName` defaults to the name of the component.

Examples

Create Implementation Model for AUTOSAR Architecture Component

For an AUTOSAR component in an architecture model, create a Simulink implementation model with a matching interface. The function call links the component to the implementation model. By default, the implementation model has the same name as the component.

```
% Create AUTOSAR architecture model
modelName = 'myArchModel';
archModel = autosar.arch.createModel(modelName);

% Add component inside the architecture model
component = addComponent(archModel,'SWC1');
addPort(component,'Sender',{'PPort1','PPort2'});

% Create and link matching Simulink implementation model
createModel(component);
```

Input Arguments

`component` — Architecture component
handle

AUTOSAR architecture component from which to create a matching Simulink implementation model. The argument is a component handle returned by a previous call to `addComponent`.

Example: `component`

`modelName` — Implementation model name
character vector | string scalar

Name of the Simulink implementation model to create, based on the specified AUTOSAR architecture component. If not specified, `modelName` defaults to the name of the component.

Example: `'SWC1'`

See Also

`linkToModel`

Topics

"Configure AUTOSAR Architecture Model Programmatically"
"Define AUTOSAR Component Behavior by Creating or Linking Models"
"Author AUTOSAR Compositions and Components in Architecture Model"

Introduced in R2020a

createNumericType

Package: `autosar.api`

Create Simulink numeric data type definition from imported AUTOSAR data elements

Syntax

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

Description

`createNumericType(arProps,name,applicationDataTypePath)` creates a `Simulink.NumericType` object from an AUTOSAR application data type. The function can be used to work with AUTOSAR elements that you imported by using `updateAUTOSARProperties`.

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

Examples

Create Numeric Data Type from AUTOSAR Application Data Type

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

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

Create Numeric Data Type from AUTOSAR Implementation Data Type and CompuMethod

Create a Simulink numeric data type with the name `myDataType` from the AUTOSAR 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 AUTOSAR 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 AUTOSAR CompuMethod, which is used to translate between the numeric implementation data type and 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 AUTOSAR implementation data type for created Simulink numeric 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/uint32'`

Data Types: `char` | `string`

See Also

`autosar.api.getAUTOSARProperties` | `updateAUTOSARProperties` | `createEnumeration`

Topics

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

“AUTOSAR Component Configuration”
“Model AUTOSAR Data Types”

Introduced in R2019a

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.

```
addpath(fullfile(matlabroot,'/examples/autosarblockset/main'));
hModel = 'autosar_swc_expfcns';
open_system(hModel);
arProps = autosar.api.getAUTOSARProperties(hModel);

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

ifPaths =
  1×3 cell array
    {'/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 =
  1×2 cell array
    {'/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
`autosar.api.getAUTOSARProperties` | `add`

Topics
"AUTOSAR Property and Map Function Examples"
"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 ARXML code. This function does not remove calibration components.

Examples

Remove Unmapped Atomic Software Components From AUTOSAR Model

After importing AUTOSAR information from ARXML files and configuring a model for AUTOSAR, remove atomic software components that were imported but are not mapped to the model. This prevents unmapped components from being exported back to ARXML.

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

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

`autosar.api.getAUTOSARProperties` | `arxml.importer`

Topics
"AUTOSAR Property and Map Function Examples"
"Configure and Map AUTOSAR Component Programmatically"
"Import AUTOSAR XML Descriptions Into Simulink"
"AUTOSAR Component Configuration"

Introduced in R2014b

destroy

Package: `autosar.arch`

Remove and delete AUTOSAR architecture element

Syntax

`destroy(archElement)`

Description

`destroy(archElement)` removes and deletes architecture element `archElement` from an architecture model. The `archElement` argument is a component, composition, port, or connector handle returned by a previous call to `addComponent`, `addComposition`, `addPort`, `connect`, or `find`.

Examples

Remove and Delete Composition from AUTOSAR Architecture Model

In an AUTOSAR architecture model, find, remove, and delete a software composition.

```
% Create AUTOSAR architecture model
modelName = 'myArchModel';
archModel = autosar.arch.createModel(modelName);

% Add a composition
addComposition(archModel,'Sensors2');

% Find and destroy the composition
composition = find(archModel,'Composition');
destroy(composition);
```

Input Arguments

`archElement` — Architecture element

handle

Component, composition, port, or connector element to remove and delete. The argument is a component, composition, port, or connector handle returned by a previous call to `addComponent`, `addComposition`, `addPort`, `connect`, or `find`.

Example: `composition`

See Also

`addComponent` | `addComposition` | `addPort` | `connect` | `find` | `importFromARXML` | `layout`

Topics

“Configure AUTOSAR Architecture Model Programmatically”
“Add and Connect AUTOSAR Compositions and Components”

"Author AUTOSAR Compositions and Components in Architecture Model"

Introduced in R2020a

export

Package: `autosar.arch`

Export AUTOSAR architecture model ARXML and generate component code

Syntax

```
export(archCCM)
export(archCCM,Name,Value)
```

Description

`export(archCCM)` exports ARXML descriptions from AUTOSAR component, composition, or architecture model `archCCM`. The function also generates code for Simulink implementation models linked by AUTOSAR components within the export scope. The containing architecture model must be open or loaded. The `archCCM` argument is a component, composition, or architecture model handle returned by a previous call to `addComponent`, `addComposition`, `autosar.arch.createModel`, or `autosar.arch.loadModel`.

`export(archCCM,Name,Value)` specifies additional export options with one or more `Name,Value` pair arguments. For example, you can specify a ZIP file in which generated files are packaged.

Examples

Generate ARXML Descriptions and Code for Architecture Model

Export composition XML descriptions and generate component code for an AUTOSAR architecture model.

```
% Load AUTOSAR architecture model
modelName = 'autosar_tpc_composition';
addpath(fullfile(matlabroot,'/examples/autosarblockset/main'));
archModel = autosar.arch.loadModel(modelName);
% Export composition XML descriptions and generate component code
export(archModel);
```

Generate ARXML Descriptions and Code for Nested Composition

Export XML descriptions and generate component code for a composition nested in an AUTOSAR architecture model.

```
% Load AUTOSAR architecture model
modelName = 'autosar_tpc_composition';
addpath(fullfile(matlabroot,'/examples/autosarblockset/main'));
archModel = autosar.arch.loadModel(modelName);
% Export nested Sensors composition
export(archModel.Compositions(1));
```

Generate ARXML Descriptions and Code In ZIP File

Export XML descriptions and generate component code for an AUTOSAR architecture model. In the `PackageCodeAndArxml` value argument, specify the name of a ZIP file in which to package the generated files.

```
% Load AUTOSAR architecture model
modelName = 'autosar_tpc_composition';
addpath(fullfile(matlabroot,'/examples/autosarblockset/main'));
archModel = autosar.arch.loadModel(modelName);
% Export ARXML descriptions and code into ZIP file
export(archModel,'PackageCodeAndARXML','myArchModel.zip');
```

Generate ECU Extract for Architecture Model

Export composition XML descriptions and generate component code for an AUTOSAR architecture model. As part of composition XML export, generate an ECU extract into the file `System.arxml`, which is located in the composition folder. The ECU extract for example model `autosar_tpc_composition` maps software components from both the top-level composition and a nested `Sensors` composition to one ECU.

```
% Load AUTOSAR architecture model
modelName = 'autosar_tpc_composition';
addpath(fullfile(matlabroot,'/examples/autosarblockset/main'));
archModel = autosar.arch.loadModel(modelName);
% Export ECU extract into composition folder
export(archModel,'ExportECUExtract',true);
```

Input Arguments

`archCCM` — Component, composition, or architecture model
handle

AUTOSAR component, composition, or architecture model for which to export ARXML descriptions and generate component code. The argument is a component, composition, or architecture model handle returned by a previous call to `addComponent`, `addComposition`, `autosar.arch.createModel`, or `autosar.arch.loadModel`.

Example: `archModel`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `'PackageCodeAndARXML','SensorsComposition.zip'` specifies the name of a ZIP file that packages the generated files.

`ExportECUExtract` — Export ECU extract
`false` (default) | `true`

As part of XML export, generate an ECU extract into the file `System.arxml`, which is located in the composition folder. The ECU extract for a composition or architecture model maps software

components from the top composition and any nested compositions to one ECU. For more information, see “Export Composition ECU Extract”.

Example: `'ExportECUExtract',true`

`ExportedARXMLFolder` — Folder location for exported ARXML files
character vector | string scalar

Full path to a folder in which to place exported ARXML description files.

Example: `'ExportedARXMLFolder','C:\temp\arxml'`

`PackageCodeAndARXML` — Name of ZIP file in which to package generated files
character vector | string scalar

Name of a ZIP file in which to package the generated files, including generated code and exported ARXML descriptions.

Example: `'PackageCodeAndARXML','SensorsComposition.zip'`

See Also

`getXmlOptions` | `setXmlOptions`

Topics

“Configure AUTOSAR Architecture Model Programmatically”
“Generate and Package AUTOSAR Composition XML Descriptions and Component Code”
“Author AUTOSAR Compositions and Components in Architecture Model”

Introduced in R2020a

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.

```
addpath(fullfile(matlabroot,'/examples/autosarblockset/main'));
hModel = 'autosar_swc_expfcns';
open_system(hModel);
arProps = autosar.api.getAUTOSARProperties(hModel);
ifPaths = find(arProps,[],'SenderReceiverInterface',...
  'IsService',false,'PathType','FullyQualified')
```

```
ifPaths =
  1×2 cell array
    {'/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 =
  1×3 cell array
    {'/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

`autosar.api.getAUTOSARProperties` | `add` | `delete` | `get` | `set`

Topics

"AUTOSAR Property and Map Function Examples"
"Configure and Map AUTOSAR Component Programmatically"
"AUTOSAR Component Configuration"

Introduced in R2013b

find

Package: `autosar.arch`

Find AUTOSAR architecture elements

Syntax

```
archElements = find(archCCM,category)
archElements = find(archCCM,category,'AllLevels',value)
archElements = find(archCCM,category,property,value)
```

Description

`archElements = find(archCCM,category)` searches AUTOSAR component, composition, or architecture model `archCCM` for architecture elements that match the specified category. The `archElements` output argument returns handles for the architecture elements found. Valid values for `category` are `Component`, `Composition`, `Port`, or `Connector`. The `archCCM` argument is a component, composition, or architecture model handle returned by a previous call to `addComponent`, `addComposition`, `autosar.arch.createModel`, or `autosar.arch.loadModel`. The default scope of `find` is the top level of the specified composition or architecture model, not all levels of the model hierarchy.

`archElements = find(archCCM,category,'AllLevels',value)` allows you to extend the search for AUTOSAR architecture elements to all levels of an AUTOSAR composition or architecture model hierarchy. To search all levels, specify `value` as `true`.

`archElements = find(archCCM,category,property,value)` specifies a constraining value on a property of the specified category of elements, narrowing the search.

Examples

Find Elements at Different Levels of AUTOSAR Architecture Model

In AUTOSAR architecture model `myArchModel`:

- Find components that are located only in the architecture model top level.
- Find components located in all levels of the model hierarchy.
- Find composition block ports and list their `Kind` and `Name` values.

```
% Create AUTOSAR architecture model
modelName = 'myArchModel';
archModel = autosar.arch.createModel(modelName);

% Add a composition
composition = addComposition(archModel,'Sensors');

% Add 2 components inside Sensors composition
names = {'PedalSnsr','ThrottleSnsr'};
sensorSWCs = addComponent(composition,names,'Kind','SensorActuator');
layout(composition); % Auto-arrange layout
```

```
% Add components at architecture model top level
addComponent(archModel,'Controller1');
actuator = addComponent(archModel,'Actuator');
set(actuator,'Kind','SensorActuator');

% Add architecture ports
addPort(archModel,'Receiver',{'TPS_Hw','APP_Hw'});
addPort(archModel,'Sender','ThrCmd_Hw');

% Add composition ports
addPort(composition,'Receiver',{'TPS_Hw','APP_Hw'});
addPort(composition,'Sender',{'TPS_Perc','APP_Perc'});

% Add component ports
controller = find(archModel,'Component','Name','Controller1');
addPort(controller,'Receiver',{'TPS_Perc','APP_Perc'});
addPort(controller,'Sender','ThrCmd_Perc');
addPort(actuator,'Receiver','ThrCmd_Perc');
addPort(actuator,'Sender','ThrCmd_Hw');

% At top level, connect composition and components based on matching port names
connect(archModel,composition,controller);
connect(archModel,controller,actuator);

% Connect specified arch root ports to specified composition and component ports
connect(archModel,archModel.Ports(1),composition.Ports(1));
% Use find to construct port specifications
connect(archModel,...
    find(archModel,'Port','Name','APP_Hw'),...
    find(composition,'Port','Name','APP_Hw'));
connect(archModel,actuator.Ports(2),archModel.Ports(3));

layout(archModel);  % Auto-arrange layout

% Find components in architecture model top level only
components_in_arch_top_level = find(archModel,'Component')
% Find components in all hierarchy
components_in_all_hierarchy = find(archModel,'Component','AllLevels',true)
% Find ports for composition block only
composition_ports = find(composition,'Port')

% List Kind and Name property values for composition ports
for ii=1:length(composition_ports)
    Port = composition_ports(ii);
    portName = get(Port,'Name');
    portKind = get(Port,'Kind');
    fprintf('%s port %s\n',portKind,portName);
end
```

```
components_in_arch_top_level =
  2×1 Component array with properties:
    Name
    Kind
    Ports
    ReferenceName
    Parent
    SimulinkHandle

components_in_all_hierarchy =
  4×1 Component array with properties:
    Name
    Kind
    Ports
    ReferenceName
    Parent
    SimulinkHandle

composition_ports =
  4×1 CompPort array with properties:
    Kind
    Connected
    Name
```

```
    Parent
    SimulinkHandle

Receiver port TPS_Hw
Receiver port APP_Hw
Sender port TPS_Perc
Sender port APP_Perc
```

Input Arguments

`archCCM` — Component, composition, or architecture model

handle

AUTOSAR component, composition, or architecture model in which to search for specified architecture elements. The argument is a component, composition, or architecture model handle returned by a previous call to `addComponent`, `addComposition`, `autosar.arch.createModel`, or `autosar.arch.loadModel`.

Example: `archModel`

`category` — Type of architecture element

character vector | string scalar

Type of AUTOSAR architecture element to find. Valid categories are `Component`, `Composition`, `Port`, or `Connector`.

Example: `'Component'`

`'AllLevels',value` — Whether to search all levels of model hierarchy

`false` (default) | `true`

Specify `true` to search all levels of an AUTOSAR composition or architecture model hierarchy for the specified architecture elements. The default scope of `find` is the top level of the specified composition or architecture model, not all levels of the model hierarchy.

Example: `'AllLevels',true`

`property,value` — Property and value

name (character vector or string scalar), value

Valid property of the specified category of architecture elements, and a value to match for that property in the search.

Example: `'Name','APP_Hw'`

Output Arguments

`archElements` — Elements found

handle | array of handles

Returns one or more handles for the architecture elements found.

See Also

`get` | `set`

Topics

"Configure AUTOSAR Architecture Model Programmatically"

“Author AUTOSAR Compositions and Components in Architecture Model”

Introduced in R2020a

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 specified 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.

```
addpath(fullfile(matlabroot,'/examples/autosarblockset/main'));
hModel = 'autosar_swc_expfcns';
open_system(hModel);
arProps = autosar.api.getAUTOSARProperties(hModel);
isService = get(arProps,'Interface1','IsService')

isService =
  logical
   0
```

Get Component Qualified Name and Runnable Symbol Name

For an AUTOSAR model, to prepare for setting the `symbol` property for runnable `Runnable1` to `test_symbol`, get the AUTOSAR component qualified name and the existing runnable symbol name.

```
addpath(fullfile(matlabroot,'/examples/autosarblockset/main'));
hModel = 'autosar_swc_expfcns';
open_system(hModel);
arProps = autosar.api.getAUTOSARProperties(hModel);
compQName = get(arProps,'XmlOptions','ComponentQualifiedName');
runnables = find(arProps,compQName,'Runnable','PathType','FullyQualified');
runnables(2)

ans =
  1×1 cell array
    {'/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 AUTOSAR element

character vector | string scalar

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

Example: `'Input'`

`property` — Element property

character vector | string scalar

Property for which to return a value, among valid properties of 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

`autosar.api.getAUTOSARProperties` | `set`

Topics

"AUTOSAR Property and Map Function Examples"
"Configure and Map AUTOSAR Component Programmatically"
"AUTOSAR Component Configuration"

Introduced in R2013b

get

Package: `autosar.arch`

Get property of AUTOSAR architecture element

Syntax

`pValue = get(archElement,property)`

Description

`pValue = get(archElement,property)` returns the current value `pValue` of the specified property for AUTOSAR architecture element `archElement`. The `archElement` argument is a component, composition, port, or connector handle returned by a previous call to `addComponent`, `addComposition`, `addPort`, `connect`, or `find`.

Examples

Get and List Properties of AUTOSAR Architecture Elements

In an AUTOSAR architecture model, find ports located in all levels of the model hierarchy. Get and list their `Kind` and `Name` property values.

```
% Create AUTOSAR architecture model
modelName = 'myArchModel';
archModel = autosar.arch.createModel(modelName);

% Add composition and component at architecture model top level
composition = addComposition(archModel,'Sensors');
addComponent(archModel,'Controller1');

% Add composition ports
addPort(composition,'Receiver',{'TPS_Hw','APP_Hw'});
addPort(composition,'Sender',{'TPS_Perc','APP_Perc'});

% Add component ports
controller = find(archModel,'Component','Name','Controller1');
addPort(controller,'Receiver',{'TPS_Perc','APP_Perc'});
addPort(controller,'Sender','ThrCmd_Perc');

% Connect composition and component based on matching port names
connect(archModel,composition,controller);

% Create implementation model for component
createModel(controller);

layout(archModel);  % Auto-arrange layout

% Set properties
set(composition.Ports(1),'Name','NewPortName1');  % Rename 2 composition ports
set(composition.Ports(3),'Name','NewPortName2');
set(find(controller,'Port','Name','TPS_Perc'),...
    'Name','NewPortName3');  % Rename port for Controller1 component & implementation
set(controller,'Kind','ServiceProxy');  % Component type for Controller1 component
set(controller,'Name','Instance1');     % Name for Controller1 component

% Find ports in architecture model hierarchy
ports_in_hierarchy = find(archModel,'Port','AllLevels',true)
```

```
% List Kind and Name property values for each port
for ii=1:length(ports_in_hierarchy)
    port = ports_in_hierarchy(ii);
    portName = get(port,'Name');
    portKind = get(port,'Kind');
    fprintf('%s port %s\n',portKind,portName);
end
```

```
ports_in_hierarchy =
  7×1 CompPort array with properties:
    Kind
    Connected
    Name
    Parent
    SimulinkHandle

Receiver port NewPortName1
Receiver port APP_Hw
Sender port NewPortName2
Sender port APP_Perc
Sender port ThrCmd_Perc
Receiver port NewPortName3
Receiver port APP_Perc
```

Input Arguments

`archElement` — Architecture element

handle

AUTOSAR architecture element for which to return the current value of a property. The argument is a component, composition, port, or connector handle returned by a previous call to `addComponent`, `addComposition`, `addPort`, `connect`, or `find`.

Example: `port`

`property` — Element property

character vector | string scalar

Property for which to return a value, among valid properties of the AUTOSAR architecture element.

Example: `'Name'`

Output Arguments

`pValue` — Property value

value of property

Returns the value of the specified property of the specified AUTOSAR architecture element.

See Also

`find` | `set`

Topics

"Configure AUTOSAR Architecture Model Programmatically"
"Author AUTOSAR Compositions and Components in Architecture Model"

Introduced in R2020a

getClassName

Get class name of model

Syntax

```
name = getClassName(slMap)
```

Description

`name = getClassName(slMap)` returns the class name of the model.

Examples

Get Class Name of Model

Open the model. To access the mapping information associated with the model, `slMap`, use the `autosar.api.getSimulinkMapping` function.

```
%% Open an adaptive AUTOSAR model
hModel = 'autosar_LaneGuidance';
addpath(fullfile(matlabroot,'/examples/autosarblockset/main'));
open_system(hModel);

%% Access the mapping information
slMap = autosar.api.getSimulinkMapping(hModel);
```

To access the class name of the model, use the `getClassName` function. If you did not specify a class name for the model, the `getClassName` function returns an empty character vector and the class name in the generated code uses the model name as the default class name.

```
name = getClassName(slMap)
```

```
name =

    0x0 empty char array
```

Specify a class name for the model by using the `setClassName` function.

```
setClassName(slMap, 'myClassName');
```

The `getClassName` function now returns the specified class name.

```
name = getClassName(slMap)
```

```
name =

    'myClassName'
```

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`

Output Arguments

`name` — Class name of model
character vector

Class name of model returned as a character vector. If you do not specify a class name, the class name in the generated code uses the model name as the default class name.

See Also

`autosar.api.getSimulinkMapping` | `setClassName` | `getClassNamespace` | `setClassName`

Topics
"Configure AUTOSAR Adaptive Code Generation"

Introduced in R2021a

getClassNamespace

Get class namespace for a model

Syntax

```
namespace = getClassNamespace(slMap)
```

Description

`namespace = getClassNamespace(slMap)` returns the class namespace specified for the model. Class namespaces can help to prevent name conflicts in large projects.

Examples

Access Class Namespace for Model

Open the model. To access the mapping information associated with the model, `slMap`, use the `autosar.api.getSimulinkMapping` function.

```
%% Open an adaptive AUTOSAR model
hModel = 'autosar_LaneGuidance';
addpath(fullfile(matlabroot,'/examples/autosarblockset/main'));
open_system(hModel);

%% Access the mapping information
slMap = autosar.api.getSimulinkMapping(hModel);
```

To access the namespace of the model, use the `getClassNamespace` function. If you did not specify a namespace for the model, the `getClassNamespace` function returns an empty character vector.

```
name = getClassNamespace(slMap)
```

```
name =

  0x0 empty char array
```

Specify a namespace for the model by using the `setClassNamespace` function.

```
setClassNamespace(slMap, 'myClassNamespace');
```

The `getClassNamespace` function now returns the specified class namespace.

```
name = getClassNamespace(slMap)
```

```
name =

    'myClassNamespace'
```

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`

Output Arguments

`namespace` — Class namespace of model
character vector

Class namespace of model returned as a character vector. If you did not specify a namespace for the model, the `getClassNamespace` function returns an empty character vector.

See Also

`autosar.api.getSimulinkMapping` | `setClassNamespace` | `getClassName` | `setClassName`

Topics
"Configure AUTOSAR Adaptive Code Generation"

Introduced in R2021a

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 File

Get the names of AUTOSAR atomic software components present in an ARXML file. The ARXML file is located at *`matlabroot`*`/examples/autosarblockset/data`, which is on the default MATLAB search path.

Create an initial Simulink representation of the `Controller` composition.

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

names =
  5×1 cell array
    {'/Company/Components/Controller'                  }
    {'/Company/Components/ThrottlePositionMonitor'     }
    {'/Company/Components/AccelerationPedalPositionSensor'}
    {'/Company/Components/ThrottlePositionActuator'    }
    {'/Company/Components/ThrottlePositionSensor'      }

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

Get AUTOSAR Sensor-Actuator Software Component Names from ARXML File

Get the names of AUTOSAR sensor-actuator software components present in an ARXML file. The ARXML file is located at *`matlabroot`*`/examples/autosarblockset/data`, which is on the default MATLAB search path.

```
ar = arxml.importer('ThrottlePositionControlComposition.arxml');
names = getComponentNames(ar,'SensorActuator')

names =
  3×1 cell array
    {'/Company/Components/AccelerationPedalPositionSensor'}
    {'/Company/Components/ThrottlePositionActuator'       }
    {'/Company/Components/ThrottlePositionSensor'         }
```

Get AUTOSAR Software Composition Names from ARXML File

Get the names of AUTOSAR software compositions present in an ARXML file. The ARXML file is located at `matlabroot/examples/autosarblockset/data`, which is on the default MATLAB search path.

Create an initial Simulink representation of the listed composition.

```
ar = arxml.importer('ThrottlePositionControlComposition.arxml');
names = getComponentNames(ar,'Composition')

names =
  1×1 cell array
    {'/Company/Components/ThrottlePositionControlComposition'}

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

Input Arguments

ar — `arxml.importer` object
handle

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

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` | `createComponentAsModel` | `createCompositionAsModel`

Topics
"Import AUTOSAR XML Descriptions Into Simulink"
"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` for classic models and `Persistency` for adaptive models.

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

Examples

Get AUTOSAR Mapping Information for Simulink Data Stores

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

```
hModel = 'autosar_bsw_sensor1';
addpath(fullfile(matlabroot,'/examples/autosarblockset/main'));
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.

For AUTOSAR classic models, valid property names include `ShortName`, `SwAddrMethod`, `SwCalibrationAccess`, and `DisplayFormat`. For `ArTypedPerInstancememory`, you can specify `NeedsNVRAMAccess`. For `StaticMemory`, you can specify C type qualifier properties `IsVolatile` or `Qualifier` (AUTOSAR additional native type qualifier).

For AUTOSAR adaptive models, valid property names include `Port` and `DataElement`.

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
`autosar.api.getSimulinkMapping` | `mapDataStore` | Data Store Memory

Topics
"Map Data Stores to AUTOSAR Variables"
"Map Submodel Data Stores to AUTOSAR Variables"
"Map Data Stores to AUTOSAR Persistent Memory Ports and Data Elements"
"Configure AUTOSAR Per-Instance Memory"
"Configure AUTOSAR Static Memory"
"Model AUTOSAR Adaptive Persistent Memory"
"AUTOSAR Property and Map Function Examples"
"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_expfcns. The model has data transfer lines named irv1, irv2, irv3, and irv4.

```
hModel = 'autosar_swc_expfcns';
addpath(fullfile(matlabroot,'/examples/autosarblockset/main'));
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.

```
hModel = 'mMultitasking_4rates';
addpath(fullfile(matlabroot,'/help/toolbox/autosar/examples'));
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: `arDataAcessMode`

See Also

`autosar.api.getSimulinkMapping` | `mapDataTransfer`

Topics

"Map Data Transfers to AUTOSAR Inter-Runnable Variables"
"Model AUTOSAR Component Behavior"
"AUTOSAR Property and Map Function Examples"
"AUTOSAR Component Configuration"

Introduced in R2013b

getFunction

Package: autosar.api

Get AUTOSAR mapping information for Simulink entry-point function

Syntax

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

Description

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

`[arRunnableName,arRunnableSwAddrMethod,arInternalDataSwAddrMethod] = getFunction(slMap,slEntryPointFunction)` 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_swc`. The model has an initialize entry-point function named `Runnable_Init` and periodic entry-point functions named `Runnable_1s` and `Runnable_2s`.

```
hModel = 'autosar_swc';
addpath(fullfile(matlabroot,'/examples/autosarblockset/main'));
open_system(hModel);
slMap=autosar.api.getSimulinkMapping(hModel);
arRunnableName=getFunction(slMap,'Initialize')
```

```
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 single-tasking periodic entry-point function.

```
hModel = 'autosar_swc_counter';
addpath(fullfile(matlabroot,'/examples/autosarblockset/main'));
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 periodic function and internal data to myCODE and myVAR SwAddrMethods
slMap = autosar.api.getSimulinkMapping(hModel);
mapFunction(slMap,'Periodic','Runnable_Step',...
    'SwAddrMethod','myCODE','SwAddrMethodForInternalData','myVAR')

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

swAddrPaths =
  1×2 cell array
    {'/Company/Powertrain/DataTypes/SwAddrMethods/CODE'}
    {'/Company/Powertrain/DataTypes/SwAddrMethods/myCODE'}

swAddrPaths =
  1×2 cell array
    {'/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`

`slEntryPointFunction` — Simulink entry-point function

character vector | string scalar

Simulink entry-point function for which to return AUTOSAR mapping information. The value format is based on the function type.

Function Type	Value
Initialize	`'Initialize'`.
Reset	`'Reset:slIdentifier'`, where `slIdentifier` is the name of a reset function in the model.

Function Type	Value
Terminate	`'Terminate'`.
Single-tasking periodic	`'Periodic'`.
Periodic (implicit task)	`'Periodic:`*`slIdentifier`*`'`, where *`slIdentifier`* is the corresponding period annotation, as displayed in the Timing Legend. For example, `'Periodic:D1'`.
Partition (explicit task)	`'Partition:`*`slIdentifier`*`'`, where *`slIdentifier`* is the partition name, as displayed in the Schedule Editor. For example, `'Partition:P1'`.
Exported	`'ExportedFunction:`*`slIdentifier`*`'`, where *`slIdentifier`* is the name of the Inport block that drives the control port of the function-call subsystem. For example: • `'ExportedFunction:Trigger_1s'` in example model `autosar_swc_slfcns` • `'ExportedFunction:FunctionTrigger'` in example model `autosar_swc_fcncalls`
Simulink function in client-server configuration	`'SimulinkFunction:`*`slIdentifier`*`'`, where *`slIdentifier`* is the name of a global Simulink function in the model. For example, `'SimulinkFunction:readData'` in the example model in “Configure AUTOSAR Server”.

Example: `'Periodic:D1'`

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 object.

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
`autosar.api.getSimulinkMapping` | `mapFunction`

Topics

“Map Entry-Point Functions to AUTOSAR Runnables”
“Configure AUTOSAR Runnables and Events”
“AUTOSAR Property and Map Function Examples”
“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

`autosar.api.getSimulinkMapping` | `mapFunctionCaller`

Topics

"Map Function Callers to AUTOSAR Client-Server Ports and Operations"
"Configure AUTOSAR Client-Server Communication"
"AUTOSAR Property and Map Function Examples"
"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_expfcns. The model has an inport named RPort_DE1.

```
hModel = 'autosar_swc_expfcns';
addpath(fullfile(matlabroot,'/examples/autosarblockset/main'));
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

`autosar.api.getSimulinkMapping` | `mapInport`

Topics

"Map Inports and Outports to AUTOSAR Sender-Receiver Ports and Data Elements"
"Configure AUTOSAR Sender-Receiver Communication"
"Configure AUTOSAR Queued Sender-Receiver Communication"
"Map Inports and Outports to AUTOSAR Service Ports and Events"
"Model AUTOSAR Adaptive Service Communication"
"AUTOSAR Property and Map Function Examples"
"AUTOSAR Component Configuration"

Introduced in R2013b

getInternalDataPackaging

Package: `autosar.api`

Get default internal data packaging for AUTOSAR component model

Syntax

`pkgSetting = getInternalDataPackaging(slMap)`

Description

`pkgSetting = getInternalDataPackaging(slMap)` returns the default data packaging setting used for internal data stores, signals, and states in the generated code for an AUTOSAR component model. Valid setting values are:

- `Default` — Accept the default internal data packaging provided by the software. Use `Default` for multi-instance models and submodels referenced from AUTOSAR component models.
- `PrivateGlobal` — Package internal variable data without a `struct` and make it private (visible only to *`model`*`.c`).
- `PrivateStructure` — Package internal variable data in a `struct` and make it private (visible only to *`model`*`.c`).
- `PublicGlobal` — Package internal variable data without a `struct` and make it public (`extern` declaration in *`model`*`.h`).
- `PublicStructure` — Package internal variable data in a `struct` and make it public (`extern` declaration in *`model`*`.h`).

If the data packaging is set to `PrivateGlobal` or `PrivateStructure`, building the model generates header file *`model`*`_private.h`, even when model configuration parameter **File packaging format** is set to `Compact`.

If the model configuration option **Generate separate internal data per entry-point function** is set for the AUTOSAR model, task-based internal data grouping overrides the AUTOSAR internal data packaging setting. However, the AUTOSAR setting determines the public or private visibility of the generated internal data groups.

Examples

Get Default Internal Packaging Setting for AUTOSAR Model

Return and modify the default data packaging setting used for internal variables in the generated code for the AUTOSAR component model.

```
hModel = 'autosar_swc';
addpath(fullfile(matlabroot,'/examples/autosarblockset/main'));
open_system(hModel);
slMap = autosar.api.getSimulinkMapping(hModel);
pkgSetting1 = getInternalDataPackaging(slMap)
setInternalDataPackaging(slMap,'PrivateStructure')
pkgSetting2 = getInternalDataPackaging(slMap)
```

```
pkgSetting1 =
    'Default'

pkgSetting2 =
    'PrivateStructure'
```

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`

Output Arguments

pkgSetting — Default internal data packaging setting
character vector

Variable that returns the default data packaging setting used for internal variables in the generated code for the AUTOSAR component model. Valid setting values are `Default`, `PrivateGlobal`, `PrivateStructure`, `PublicGlobal`, and `PublicStructure`.

Example: `pkgSetting`

See Also

`setInternalDataPackaging` | `autosar.api.getSimulinkMapping`

Topics

"Map AUTOSAR Elements for Code Generation"
"AUTOSAR Component Configuration"

Introduced in R2021a

getOutport

Package: autosar.api

Get AUTOSAR mapping information for Simulink outport

Syntax

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

Description

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

Examples

Get AUTOSAR Mapping Information for Model Outport

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

```
hModel = 'autosar_swc_expfcns';
addpath(fullfile(matlabroot,'/examples/autosarblockset/main'));
open_system(hModel);
slMap=autosar.api.getSimulinkMapping(hModel);
[arPortName,arDataElementName,arDataAccessMode]=getOutport(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 outport

character vector | string scalar

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

Example: 'Output'

Output Arguments

arPortName — Name of AUTOSAR port

character vector

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

Example: arPortName

arDataElementName — Name of AUTOSAR data element

character vector

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

Example: arDataElementName

arDataAccessMode — Value of AUTOSAR data access mode

character vector

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

Example: arDataAcessMode

See Also

autosar.api.getSimulinkMapping | mapOutport

Topics

"Map Inports and Outports to AUTOSAR Sender-Receiver Ports and Data Elements"
"Configure AUTOSAR Sender-Receiver Communication"
"Configure AUTOSAR Queued Sender-Receiver Communication"
"Map Inports and Outports to AUTOSAR Service Ports and Events"
"Model AUTOSAR Adaptive Service Communication"
"AUTOSAR Property and Map Function Examples"
"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`, `ConstantMemory`, and `PortParameter`.

`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';
addpath(fullfile(matlabroot,'/examples/autosarblockset/main'));
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` — 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 `PortParameter`, you can also specify `Port` or `DataElement`. 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 parameter or the value of a parameter property.

Example: `arValue`

See Also

`autosar.api.getSimulinkMapping` | `mapParameter`

Topics

"Map Model Workspace Parameters to AUTOSAR Component Parameters"
"Map Submodel Parameters to AUTOSAR Component Parameters"
"Configure AUTOSAR Constant Memory"
"Configure AUTOSAR Shared or Per-Instance Parameters"
"Configure AUTOSAR Port Parameters for Communication with Parameter Component"
"AUTOSAR Property and Map Function Examples"
"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 outport 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_swc_counter`.

```
hModel = 'autosar_swc_counter';
addpath(fullfile(matlabroot,'/examples/autosarblockset/main'));
open_system(hModel);
slMap = autosar.api.getSimulinkMapping(hModel);

portHandles = get_param('autosar_swc_counter/RelOpt','portHandles');
outportHandle = portHandles.Outport;
mapSignal(slMap,outportHandle,'StaticMemory')
arMappedTo = getSignal(slMap,outportHandle)
arValue = getSignal(slMap,outportHandle,'SwCalibrationAccess')

portHandles = get_param('autosar_swc_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 outport port handle for a block signal

handle

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

```
portHandles = get_param('autosar_swc_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

`autosar.api.getSimulinkMapping` | `addSignal` | `mapSignal` | `removeSignal`

Topics

"Map Block Signals and States to AUTOSAR Variables"
"Map Submodel Signals and States to AUTOSAR Variables"
"Configure AUTOSAR Per-Instance Memory"
"Configure AUTOSAR Static Memory"
"AUTOSAR Property and Map Function Examples"

"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_swc_counter`. The state owner block has one state.

```
hModel = 'autosar_swc_counter';
addpath(fullfile(matlabroot,'/examples/autosarblockset/main'));
open_system(hModel);
slMap = autosar.api.getSimulinkMapping(hModel);

mapState(slMap,'autosar_swc_counter/X','','ArTypedPerInstanceMemory',...
    'SwCalibrationAccess','ReadWrite')
arMappedTo = getState(slMap,'autosar_swc_counter/X')
arValue = getState(slMap,'autosar_swc_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 to return AUTOSAR mapping information for.

Example: `'autosar_swc_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

`autosar.api.getSimulinkMapping` | `mapState`

Topics

"Map Block Signals and States to AUTOSAR Variables"
"Map Submodel Signals and States to AUTOSAR Variables"
"Configure AUTOSAR Per-Instance Memory"
"Configure AUTOSAR Static Memory"
"AUTOSAR Property and Map Function Examples"
"AUTOSAR Component Configuration"

Introduced in R2018b

getXmlOptions

Package: autosar.arch

Get XML option for AUTOSAR architecture model

Syntax

```
pValue = getXmlOptions(archModel,property)
```

Description

`pValue = getXmlOptions(archModel,property)` returns the current value `pValue` of XML option `property` in architecture model `archModel`. The `archModel` argument is a model handle returned by a previous call to `autosar.arch.createModel` or `autosar.arch.loadModel`. For more information about XML options, see “Configure AUTOSAR XML Options”.

Examples

Get Value of XML Option DataTypePackage for AUTOSAR Architecture Model

For a new AUTOSAR architecture model, get the initial value of the AUTOSAR XML data type package path.

```
archModel = autosar.arch.createModel('MyArchModel');
pValue = getXmlOptions(archModel,'DataTypePackage')
```

```
pValue =
    '/DataTypes'
```

Input Arguments

archModel — Architecture model

handle

AUTOSAR architecture model for which to return the current value of an XML option value. The argument is a model handle returned by a previous call to `autosar.arch.createModel` or `autosar.arch.loadModel`.

Example: `archModel`

property — XML option

character vector | string scalar

XML option for which to return a value. For more information about XML options, see “Configure AUTOSAR XML Options”.

Example: `'DataTypePackage'`

Output Arguments

`pValue` — XML option value
value of option

Returns the value of the specified XML option of the specified AUTOSAR architecture model.

See Also

`export` | `setXmlOptions`

Topics
"Configure AUTOSAR Architecture Model Programmatically"
"Generate and Package AUTOSAR Composition XML Descriptions and Component Code"
"Author AUTOSAR Compositions and Components in Architecture Model"

Introduced in R2020a

importFromARXML

Package: `autosar.arch`

Import composition from ARXML files into AUTOSAR architecture model

Syntax

```
importFromARXML(archModel,arxmlInput,compQName)
importFromARXML(archModel,arxmlInput,compQName,Name,Value)
```

Description

`importFromARXML(archModel,arxmlInput,compQName)` imports composition `compQName` from `arxmlInput` into architecture model `archModel`. The `archModel` argument is an architecture model handle returned by a previous call to `autosar.arch.createModel` or `autosar.arch.loadModel`. Composition import requires an open AUTOSAR architecture model with no functional content.

`importFromARXML(archModel,arxmlInput,compQName,Name,Value)` specifies additional import options with one or more `Name,Value` pair arguments. You can specify:

- Whether to include or exclude AUTOSAR software components, which define composition behavior. By default, the import includes components within the composition.
- Simulink data dictionary in which to place data objects for imported AUTOSAR data types.
- Names of existing Simulink behavior models to link to imported AUTOSAR software components.
- Component options to apply when creating Simulink behavior models for imported AUTOSAR software components. For example, how to model periodic runnables, or a `PredefinedVariant` or `SwSystemconstantValueSets` with which to resolve component variation points.

Examples

Import AUTOSAR Composition to Architecture Model

This example:

1. Creates AUTOSAR architecture model `myArchModel`.
2. Imports software composition `/Company/Components/ThrottlePositionControlComposition` from AUTOSAR example file `ThrottlePositionControlComposition.arxml` into the architecture model.

The ARXML file is located at *`matlabroot`*`/examples/autosarblockset/data`, which is on the default MATLAB search path.

```
% Create AUTOSAR architecture model
modelName = "myArchModel";
archModel = autosar.arch.createModel(modelName);

% Import composition from file ThrottlePositionControlComposition.arxml
importerObj = arxml.importer("ThrottlePositionControlComposition.arxml");  % Parse ARXML
```

```
importFromARXML(archModel,importerObj,...
  "/Company/Components/ThrottlePositionControlComposition");

Creating model 'ThrottlePositionSensor' for component 1 of 5:
  /Company/Components/ThrottlePositionSensor
Creating model 'ThrottlePositionMonitor' for component 2 of 5:
  /Company/Components/ThrottlePositionMonitor
Creating model 'Controller' for component 3 of 5:
  /Company/Components/Controller
Creating model 'AccelerationPedalPositionSensor' for component 4 of 5:
  /Company/Components/AccelerationPedalPositionSensor
Creating model 'ThrottlePositionActuator' for component 5 of 5:
  /Company/Components/ThrottlePositionActuator
Importing composition 1 of 1:
  /Company/Components/ThrottlePositionControlComposition
```

Import AUTOSAR Composition and Link Existing Component Models

This example shows the function call syntax to:

1. Create an AUTOSAR architecture model with no functional content.
2. Import AUTOSAR software composition `/pkg/rootComposition` from a file named `mySWCs.arxml` into the architecture model.
3. For software components `mySwc1` and `mySwc2` contained within the composition, link existing Simulink component models rather than creating new ones.

```
% Create AUTOSAR architecture model
modelName = 'myArchModel';
archModel = autosar.arch.createModel(modelName);

% Import composition from ARXML file and link existing component models
importFromARXML(archModel,'mySWCs.arxml','/pkg/rootComposition',...
  'ComponentModels',{'mySwc1','mySwc2'})
```

Import AUTOSAR Composition and Use Component `PredefinedVariant`

This example shows the function call syntax to:

1. Create an AUTOSAR architecture model with no functional content.
2. Import AUTOSAR software composition `/CompositionType/myComposition` from a file named `myComposition.arxml` into the architecture model.
3. For each software component contained within the composition, at component model creation time, use `PredefinedVariant` Senior to resolve variation points in the component.

```
% Create AUTOSAR architecture model
modelName = "myArchModel";
archModel = autosar.arch.createModel(modelName);

% Import composition from ARXML file and use PredefinedVariant for components
importerObj = arxml.importer("MyComposition.arxml"); % Import AUTOSAR information
importFromARXML(archModel,importerObj,"/CompositionType/myComposition",...
  "PredefinedVariant","/pkg/body/Variants/Senior"););
```

Input Arguments

`archModel` — Architecture model

handle

AUTOSAR architecture model into which to import the specified composition. The argument is an architecture model handle returned by a previous call to `autosar.arch.createModel` or `autosar.arch.loadModel`.

Example: `archModel`

`arxmlInput` — `arxml.importer` object or ARXML file names
handle | character vector | string scalar | cell array of character vectors | string array

ARXML files from which to import the specified composition, specified as one of the following:

- A handle to AUTOSAR information imported from ARXML files, previously returned by *`importerObj`* `= arxml.importer(`*`arxmlFiles`*`)`.
- One or more ARXML file names.

Example: `importerObj`, `"myComposition.arxml"`

`compQName` — Composition path
character vector | string scalar | cell array of character vectors | string array

Absolute short-name path (qualified name) of the composition to import into the specified composition or architecture model.

Example: `"/CompositionType/myComposition"`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `'DataDictionary','ardata.sldd'` directs the importer to place data objects corresponding to imported AUTOSAR data types in the specified Simulink data dictionary.

`ComponentModels` — Simulink component models
cell array of character vectors | string array

Names of existing atomic software component models to link when creating a Simulink representation of the composition. For components contained within the composition, link the specified component behavior models instead of creating new ones.

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

`DataDictionary` — Simulink data dictionary
character vector | string scalar

Simulink data dictionary in which to place data objects corresponding to imported AUTOSAR data types. If the specified dictionary does not exist, the importer creates it. The composition and its components are then associated with that data dictionary.

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

`ExcludeInternalBehavior` — Suppress component import
`false` (default) | `true`

Specify whether to allow (default) or suppress the import of software components that define the behavior of the composition. If component import is suppressed (`true`), the import still links models specified by the `ComponentModels` argument.

Example: `'ExcludeInternalBehavior',true`

`ModelPeriodicRunnablesAs` — For imported components, subsystem type for periodic runnables
`'AtomicSubsystem'` (default) | `'FunctionCallSubsystem'` | `'Auto'`

By default, when importing a software component contained within a composition, `importFromARXML` imports AUTOSAR periodic runnables found in the 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'`

`PredefinedVariant` — For imported components, path to AUTOSAR predefined variant
character vector | string scalar

Path to a `PredefinedVariant` defined in the ARXML files. 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 component 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` — For imported components, 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 ARXML files. 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 component 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'}'`

See Also

`addComponent` | `addComposition` | `addPort` | `connect` | `destroy` | `layout`

Topics

"Configure AUTOSAR Architecture Model Programmatically"
"Import AUTOSAR Composition from ARXML"
"Import AUTOSAR Composition into Architecture Model"
"Author AUTOSAR Compositions and Components in Architecture Model"

Introduced in R2020b

layout

Package: autosar.arch

Arrange AUTOSAR composition or architecture model layout based on heuristics

Syntax

layout(archCM)

Description

layout(archCM) automatically arranges the modeling elements inside composition or architecture model archCM based on a set of heuristics. The archM argument is a composition or architecture model handle returned by a previous call to addComposition, autosar.arch.createModel, or autosar.arch.loadModel.

Examples

Arrange Layout After Adding Blocks to AUTOSAR Architecture Model

In an AUTOSAR architecture model, add AUTOSAR components, and then update the arrangement of elements in the model layout.

```
% Create AUTOSAR architecture model
modelName = 'myArchModel';
archModel = autosar.arch.createModel(modelName);

% Add components and auto-arrange layout
addComponent(archModel,{'SWC1','SWC2','SWC3'});
layout(archModel);
```

Input Arguments

archCM — Composition or architecture model
handle

AUTOSAR composition or architecture model in which to arrange modeling elements based on a set of heuristics. The argument is a composition or architecture model handle returned by a previous call to addComposition, autosar.arch.createModel, or autosar.arch.loadModel.

Example: archModel

See Also
addComponent | addComposition | addPort | connect | destroy | importFromARXML

Topics
“Configure AUTOSAR Architecture Model Programmatically”
“Add and Connect AUTOSAR Compositions and Components”
“Author AUTOSAR Compositions and Components in Architecture Model”

Introduced in R2020a

linkToModel

Package: `autosar.arch`

Link AUTOSAR architecture component to Simulink implementation model

Syntax

`linkToModel(component,modelName)`

Description

`linkToModel(component,modelName)` links the specified AUTOSAR architecture component to existing Simulink implementation model `modelName`. The component inherits the interface of the linked implementation model. The `component` argument is a component handle returned by a previous call to `addComponent`.

Examples

Link AUTOSAR Architecture Component to Implementation Model

In an architecture model, link an AUTOSAR component to a Simulink implementation model. The component inherits the interface of the linked implementation model.

```
% Create AUTOSAR architecture model
modelName = 'myArchModel';
archModel = autosar.arch.createModel(modelName);

% Add component inside the architecture model
component = addComponent(archModel,'SWC1');

% Add path to implementation model
addpath(fullfile(matlabroot,'/examples/autosarblockset/main'));

% Link to Simulink implementation model and inherit its interface
linkToModel(component,'autosar_tpc_controller');
```

Input Arguments

`component` — Architecture component
handle

AUTOSAR architecture component to link to the specified Simulink implementation model. The argument is a component handle returned by a previous call to `addComponent`.

Example: `component`

`modelName` — Implementation model name
character vector | string scalar

Name of an existing Simulink implementation model to link from the specified AUTOSAR architecture component.

Example: `'autosar_tpc_controller'`

See Also

`createModel`

Topics

"Configure AUTOSAR Architecture Model Programmatically"
"Define AUTOSAR Component Behavior by Creating or Linking Models"
"Author AUTOSAR Compositions and Components in Architecture Model"

Introduced in R2020a

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` for classic models and `Persistency` for adaptive models.

`mapDataStore(slMap,slBlockHandle,arVarType,Name,Value)` specifies additional properties for an AUTOSAR `ArTypedPerInstanceMemory`, `StaticMemory`, or `Persistency` 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 the Simulink data store memory block `Data Store Memory` in example model `autosar_bsw_sensor1`.

```
hModel = 'autosar_bsw_sensor1';
addpath(fullfile(matlabroot,'/examples/autosarblockset/main'));
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 that you set AUTOSAR mapping information for.

Example: `'autosar_bsw_sensor1/Data Store Memory'`

`arVarType` — Type of AUTOSAR variable
character vector | string scalar

Type of AUTOSAR variable that you want to map the specified Simulink data store to. Valid AUTOSAR variable types include `ArTypedPerInstanceMemory`, `StaticMemory`, and `Auto` for classic models. Valid AUTOSAR variable types include `Persistency` and `Auto` for adaptive models. To accept software mapping defaults, specify `Auto`.

Example: `'StaticMemory'`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

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

`DataElement` — Parameter interface data element (`Persistency` only)
character vector | string scalar

Specify the data element of the persistency port associated with the AUTOSAR adaptive variable. `DataElement` can be set with `Port` only.

Example: `'Port','Perport','DataElement','Delement1'`

`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 calibration and measurement 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 requires 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. Specify

`RestoreAtStart` to `true` to read data from memory at the beginning of a program. Specify `StoreAtShutdown` to `true` to write data to memory at the end of a program.

Example: `'NeedsNVRAMAccess','true', 'RestoreAtStart', 'true', 'StoreAtShutdown', 'true'`

`Port` — Parameter receiver port (`Persistency` only)
character vector | string scalar

Specify the persistency port to associate with the AUTOSAR adaptive variable. `Port` can be set with `DataElement` only.

Example: `'Port','Perport','DataElement','Delement1'`

`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 calibration and measurement 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 calibration and measurement 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

`autosar.api.getSimulinkMapping` | `getDataStore` | Data Store Memory

Topics
“Map Data Stores to AUTOSAR Variables”
“Map Submodel Data Stores to AUTOSAR Variables”
“Map Data Stores to AUTOSAR Persistent Memory Ports and Data Elements”

“Configure AUTOSAR Per-Instance Memory”
“Configure AUTOSAR Static Memory”
“Model AUTOSAR Adaptive Persistent Memory”
“AUTOSAR Property and Map Function Examples”
“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_expfcns`. 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_expfcns';
addpath(fullfile(matlabroot,'/examples/autosarblockset/main'));
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`.

```
hModel = 'mMultitasking_4rates';
addpath(fullfile(matlabroot,'/help/toolbox/autosar/examples'));
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

`autosar.api.getSimulinkMapping` | `getDataTransfer`

Topics
"Map Data Transfers to AUTOSAR Inter-Runnable Variables"
"Model AUTOSAR Component Behavior"
"AUTOSAR Property and Map Function Examples"
"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,slEntryPointFunction,arRunnableName)
mapFunction(slMap,slEntryPointFunction,arRunnableName,Name,Value)

Description

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

mapFunction(slMap,slEntryPointFunction,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_swc. The model has an initialize entry-point function named Runnable_Init and periodic entry-point functions named Runnable_1s and Runnable_2s.

```
hModel = 'autosar_swc';
addpath(fullfile(matlabroot,'/examples/autosarblockset/main'));
open_system(hModel);
slMap=autosar.api.getSimulinkMapping(hModel);
mapFunction(slMap,'Initialize','Runnable_Init');
arRunnableName=getFunction(slMap,'Initialize')
```

```
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 single-tasking periodic entry-point step function.

```
hModel = 'autosar_swc_counter';
addpath(fullfile(matlabroot,'/examples/autosarblockset/main'));
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 periodic function and internal data to myCODE and myVAR SwAddrMethods
slMap = autosar.api.getSimulinkMapping(hModel);
mapFunction(slMap,'Periodic','Runnable_Step',...
    'SwAddrMethod','myCODE','SwAddrMethodForInternalData','myVAR')

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

swAddrPaths =
  1×2 cell array
    {'/Company/Powertrain/DataTypes/SwAddrMethods/CODE'}
    {'/Company/Powertrain/DataTypes/SwAddrMethods/myCODE'}

swAddrPaths =
  1×2 cell array
    {'/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`

slEntryPointFunction — Simulink entry-point function

character vector | string scalar

Simulink entry-point function for which to set AUTOSAR mapping information. The value format is based on the function type.

Function Type	Value
Initialize	`'Initialize'`.
Reset	`'Reset:slIdentifier'`, where `slIdentifier` is the name of a reset function in the model.
Terminate	`'Terminate'`.

Function Type	Value
Single-tasking periodic	`'Periodic'`.
Periodic (implicit task)	`'Periodic:`*`slIdentifier`*`'`, where *`slIdentifier`* is the corresponding period annotation, as displayed in the Timing Legend. For example, `'Periodic:D1'`.
Partition (explicit task)	`'Partition:`*`slIdentifier`*`'`, where *`slIdentifier`* is the partition name, as displayed in the Schedule Editor. For example, `'Partition:P1'`.
Exported	`'ExportedFunction:`*`slIdentifier`*`'`, where *`slIdentifier`* is the name of the Inport block that drives the control port of the function-call subsystem. For example: • `'ExportedFunction:Trigger_1s'` in example model `autosar_swc_slfcns` • `'ExportedFunction:FunctionTrigger'` in example model `autosar_swc_fcncalls`
Simulink function in client-server configuration	`'SimulinkFunction:`*`slIdentifier`*`'`, where *`slIdentifier`* is the name of a global Simulink function in the model. For example, `'SimulinkFunction:readData'` in the example model in "Configure AUTOSAR Server".

Example: `'Periodic:D1'`

`arRunnableName` — Name of AUTOSAR runnable
character vector | string scalar

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

Example: `'Runnable_2s'`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

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 `SwAddrMethod`s 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

`autosar.api.getSimulinkMapping` | `getFunction`

Topics

"Map Entry-Point Functions to AUTOSAR Runnables"
"Configure AUTOSAR Runnables and Events"
"AUTOSAR Property and Map Function Examples"
"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

`autosar.api.getSimulinkMapping` | `getFunctionCaller`

Topics
"Map Function Callers to AUTOSAR Client-Server Ports and Operations"
"Configure AUTOSAR Client-Server Communication"
"AUTOSAR Property and Map Function Examples"
"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_swc_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_swc_expfcns';
addpath(fullfile(matlabroot,'/examples/autosarblockset/main'));
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

`autosar.api.getSimulinkMapping` | `getInport`

Topics

"Map Inports and Outports to AUTOSAR Sender-Receiver Ports and Data Elements"
"Configure AUTOSAR Sender-Receiver Communication"
"Configure AUTOSAR Queued Sender-Receiver Communication"
"Map Inports and Outports to AUTOSAR Service Ports and Events"
"Model AUTOSAR Adaptive Service Communication"
"AUTOSAR Property and Map Function Examples"
"AUTOSAR Component Configuration"

Introduced in R2013b

mapOutport

Package: autosar.api

Map Simulink outport to AUTOSAR port

Syntax

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

Description

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

Examples

Set AUTOSAR Mapping Information for Model Outport

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

```
hModel = 'autosar_swc_expfcns';
addpath(fullfile(matlabroot,'/examples/autosarblockset/main'));
open_system(hModel);
slMap=autosar.api.getSimulinkMapping(hModel);
mapOutport(slMap,'PPort_DE1','PPort','DE1','ExplicitSend');
[arPortName,arDataElementName,arDataAccessMode]=getOutport(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 outport

character vector | string scalar

Name of the model outport 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 outport.

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 outport.

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 outport. The value can be `ImplicitSend`, `ImplicitSendByRef`, `ExplicitSend`, `EndToEndWrite`, `ModeSend`, or `QueuedExplicitSend`.

Example: `'ExplicitSend'`

See Also

`autosar.api.getSimulinkMapping` | `getOutport`

Topics

"Map Inports and Outports to AUTOSAR Sender-Receiver Ports and Data Elements"
"Configure AUTOSAR Sender-Receiver Communication"
"Configure AUTOSAR Queued Sender-Receiver Communication"
"Map Inports and Outports to AUTOSAR Service Ports and Events"
"Model AUTOSAR Adaptive Service Communication"
"AUTOSAR Property and Map Function Examples"
"AUTOSAR Component Configuration"

Introduced in R2013b

mapParameter

Package: `autosar.api`

Map Simulink model workspace parameter to AUTOSAR component 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`, `ConstantMemory`, and `PortParameter`.

`mapParameter(slMap,slParameter,arParamType,Name,Value)` specifies additional properties for an AUTOSAR `SharedParameter`, `PerInstanceParameter`, `ConstantMemory`, or `PortParameter` 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';
addpath(fullfile(matlabroot,'/examples/autosarblockset/main'));
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 parameter to which to map the specified Simulink model workspace parameter. Valid AUTOSAR parameter types include `SharedParameter`, `PerInstanceParameter`, `ConstantMemory`, `PortParameter`, and `Auto`. To accept software mapping defaults, specify `Auto`.

Example: `'SharedParameter'`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

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

`DataElement` — Parameter interface data element (`PortParameter` only)

character vector | string scalar

Specify the name of a parameter interface data element configured in the AUTOSAR Dictionary.

Example: `'DataElement','ParamElement1'`

`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 calibration and measurement 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'`

Port — Parameter receiver port (PortParameter only)
character vector | string scalar

Specify the name of a parameter receiver port configured in the AUTOSAR Dictionary.

Example: `'Port','myParamPort'`

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 calibration and measurement 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 calibration and measurement 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

`autosar.api.getSimulinkMapping` | `getParameter`

Topics

“Map Model Workspace Parameters to AUTOSAR Component Parameters”
“Map Submodel Parameters to AUTOSAR Component Parameters”
“Configure AUTOSAR Constant Memory”
“Configure AUTOSAR Shared or Per-Instance Parameters”

"Configure AUTOSAR Port Parameters for Communication with Parameter Component"
"AUTOSAR Property and Map Function Examples"
"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 outport 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_swc_counter`.

```
hModel = 'autosar_swc_counter';
addpath(fullfile(matlabroot,'/examples/autosarblockset/main'));
open_system(hModel);
slMap = autosar.api.getSimulinkMapping(hModel);

portHandles = get_param('autosar_swc_counter/RelOpt','portHandles');
outportHandle = portHandles.Outport;
mapSignal(slMap,outportHandle,'StaticMemory')
arMappedTo = getSignal(slMap,outportHandle)
arValue = getSignal(slMap,outportHandle,'SwCalibrationAccess')

portHandles = get_param('autosar_swc_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 outport port handle for a block signal

handle

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

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

Example: `outportHandle`

`arVarType` — Type of AUTOSAR variable

character vector | string scalar

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

Example: `'StaticMemory'`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

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 calibration and measurement 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 a short name for the AUTOSAR variable. If unspecified, ARXML export 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 calibration and measurement 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 calibration and measurement 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
`autosar.api.getSimulinkMapping` | `addSignal` | `getSignal` | `removeSignal`

Topics
“Map Block Signals and States to AUTOSAR Variables”
“Map Submodel Signals and States to AUTOSAR Variables”
“Configure AUTOSAR Per-Instance Memory”
“Configure AUTOSAR Static Memory”
“AUTOSAR Property and Map Function Examples”
“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_swc_counter';
addpath(fullfile(matlabroot,'/examples/autosarblockset/main'));
open_system(hModel);
slMap = autosar.api.getSimulinkMapping(hModel);

mapState(slMap,'autosar_swc_counter/X','','ArTypedPerInstanceMemory',...
    'SwCalibrationAccess','ReadWrite')
arMappedTo = getState(slMap,'autosar_swc_counter/X')
arValue = getState(slMap,'autosar_swc_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 to set AUTOSAR mapping information for.

Example: `'autosar_swc_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 map the specified Simulink block state to. Valid AUTOSAR variable types include `ArTypedPerInstanceMemory`, `StaticMemory`, and `Auto`. To accept software mapping defaults, specify `Auto`.

Example: `'ArTypedPerInstanceMemory'`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

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 calibration and measurement 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 a short name for the AUTOSAR variable. If unspecified, ARXML export 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 calibration and measurement 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 calibration and measurement 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
`autosar.api.getSimulinkMapping` | `getState`

Topics
“Map Block Signals and States to AUTOSAR Variables”
“Map Submodel Signals and States to AUTOSAR Variables”
“Configure AUTOSAR Per-Instance Memory”
“Configure AUTOSAR Static Memory”
“AUTOSAR Property and Map Function Examples”
“AUTOSAR Component Configuration”

Introduced in R2018b

open

Package: `autosar.arch`

Open AUTOSAR architecture model

Syntax

`open(archModel)`

Description

`open(archModel)` opens architecture model `archModel` in the editor. The `archModel` argument is a model handle returned by a previous call to `autosar.arch.createModel` or `autosar.arch.loadModel`. The model must be loaded.

Examples

Open AUTOSAR Architecture Model in the Editor

Open a loaded AUTOSAR architecture model in the editor. Add a composition, save the change, and close the model.

```
% Load AUTOSAR architecture model located in current folder or on MATLAB path
modelName = 'myArchModel';
archModel = autosar.arch.loadModel(modelName);

% Open loaded model in the editor
open(archModel);

% Add a composition
composition = addComposition(archModel,'Sensors2');
layout(archModel);  % Auto-arrange model layout

% Save the model
save(archModel);

% Close the model
close(archModel);
```

Input Arguments

`archModel` — Architecture model
handle

AUTOSAR architecture model to open in the editor. The argument is a model handle returned by a previous call to `autosar.arch.createModel` or `autosar.arch.loadModel`. The model must be loaded.

Example: `archModel`

See Also

`autosar.arch.createModel` | `autosar.arch.loadModel` | `close` | `save`

Topics
"Configure AUTOSAR Architecture Model Programmatically"
"Create AUTOSAR Architecture Models"
"Author AUTOSAR Compositions and Components in Architecture Model"

Introduced in R2020a

overflowed

Determine when a message queue overflows

Syntax

```
overflowed(message_name)
```

Description

`overflowed(message_name)` checks whether a message is lost because it was sent to a queue that was already full. In each time step, the value of this operator is set when a chart adds a message to, or removes a message from, a queue. It is invalid to use the `overflowed` operator before sending or retrieving a message in the same time step. To use the `overflowed` operator, set the model to an `autosar.tlc` target for both simulation and code generation and verify that the inport or outport message connects to an external queue.

- To check the overflow status of an input message queue, first remove a message from the queue.
- To check the overflow status of an output message queue, first add a message to the queue.
- To check the overflow status of a local message queue, first add a message to the queue or remove a message from the queue.

Examples

Check for Overflow in Transition

Check the input or local queue for message `M`. If a message is present and the queue has overflowed, transition occurs.

```
M[overflowed(M)]
```

Check for Overflow in State Action

Check the input or local queue for message `M`. If a message is present and the queue has overflowed, increment the value of `x`.

```
on M:
if overflowed(M) == true
    x = x+1;
end
```

Check for Overflow After Sending Message

Send message and check for overflow. If the queue overflows, increment the value of `x`.

```
entry:
M.data = 3;
```

```
send(M);
if overflowed(M) == true
    x = x+1;
end
```

Tips

- By default, when a message queue overflows, simulation stops with an error. To prevent a run-time error and allow the `overflowed` operator to dynamically react to dropped messages, set the value of the **Queue Overflow Diagnostic** property to `Warning` or `None`. For more information, see “Queue Overflow Diagnostic” (Stateflow).

See Also

length | receive | send

Topics

“Determine When a Queue Overflows”
“Control Message Activity in Stateflow Charts” (Stateflow)
“Set Properties for a Message” (Stateflow)

Introduced in R2018b

removeSignal

Package: `autosar.api`

Remove Simulink block signal from AUTOSAR mapping

Syntax

`removeSignal(slMap,slPortHandle)`

Description

`removeSignal(slMap,slPortHandle)` removes the Simulink block signal associated with outport port handle `slPortHandle` from AUTOSAR mapping.

Examples

Remove Simulink Block Signal from Mapping

In example model `autosar_swc_counter`, remove Simulink signal `equal_to_count`, which originates in the `RelOpt` block, from the AUTOSAR component signal mapping.

```
hModel = 'autosar_swc_counter';
addpath(fullfile(matlabroot,'/examples/autosarblockset/main'));
open_system(hModel);
slMap = autosar.api.getSimulinkMapping(hModel);

portHandles = get_param('autosar_swc_counter/RelOpt','portHandles');
outportHandle = portHandles.Outport;
removeSignal(slMap,outportHandle);
```

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 outport port handle for a block signal

handle

Outport port handle for a Simulink block signal to remove from AUTOSAR mapping. Use MATLAB commands to construct the outport port handle. For example, for a Relational Operator block named `RelOpt`:

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

Example: `outportHandle`

See Also

`autosar.api.getSimulinkMapping` | `addSignal` | `getSignal` | `mapSignal`

Topics

"Map Block Signals and States to AUTOSAR Variables"
"Map Submodel Signals and States to AUTOSAR Variables"
"Configure AUTOSAR Per-Instance Memory"
"Configure AUTOSAR Static Memory"
"AUTOSAR Property and Map Function Examples"
"AUTOSAR Component Configuration"

Introduced in R2020b

save

Package: `autosar.arch`

Save AUTOSAR architecture model

Syntax

`save(archModel)`

Description

`save(archModel)` saves architecture model `archModel`. The `archModel` argument is a model handle returned by a previous call to `autosar.arch.createModel` or `autosar.arch.loadModel`. The model must be open or loaded.

Examples

Save AUTOSAR Architecture Model After Making Change

Create an AUTOSAR architecture model, add a composition, and save the model with the change. Close the model.

```
% Create AUTOSAR architecture model
modelName = 'myArchModel';
archModel = autosar.arch.createModel(modelName);

% Add a composition
composition = addComposition(archModel,'Sensors2');

% Save the model
save(archModel);

% Close the model
close(archModel);
```

Input Arguments

`archModel` — Architecture model

handle

AUTOSAR architecture model to save. The argument is a model handle returned by a previous call to `autosar.arch.createModel` or `autosar.arch.loadModel`. The model must be open or loaded.

Example: `archModel`

See Also

`autosar.arch.createModel` | `autosar.arch.loadModel` | `close` | `open`

Topics

"Configure AUTOSAR Architecture Model Programmatically"
"Create AUTOSAR Architecture Models"

"Author AUTOSAR Compositions and Components in Architecture Model"

Introduced in R2020a

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 a 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 `Interface1` to `true` (1), indicating that the port interface is used for AUTOSAR services.

```
addpath(fullfile(matlabroot,'/examples/autosarblockset/main'));
hModel = 'autosar_swc_expfcns';
open_system(hModel);
arProps = autosar.api.getAUTOSARProperties(hModel);
set(arProps,'Interface1','IsService',true);
isService = get(arProps,'Interface1','IsService')
```

```
isService =
  logical
   1
```

Set Runnable Symbol Name

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

```
addpath(fullfile(matlabroot,'/examples/autosarblockset/main'));
hModel = 'autosar_swc_expfcns';
open_system(hModel);
arProps = autosar.api.getAUTOSARProperties(hModel);
compQName = get(arProps,'XmlOptions','ComponentQualifiedName');
runnables = find(arProps,compQName,'Runnable','PathType','FullyQualified');
runnables(2)
```

```
ans =
  1×1 cell array
    {'/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` — Element property
character vector | string scalar

Property for which to set a value, among valid properties of 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

`autosar.api.getAUTOSARProperties` | `get`

Topics
"AUTOSAR Property and Map Function Examples"
"Configure and Map AUTOSAR Component Programmatically"
"AUTOSAR Component Configuration"

Introduced in R2013b

set

Package: `autosar.arch`

Set property of AUTOSAR architecture element

Syntax

`set(archElement,property,value)`

Description

`set(archElement,property,value)` sets the specified property to `value` for AUTOSAR architecture element `archElement`. The `archElement` argument is a component, composition, port, or connector handle returned by a previous call to `addComponent`, `addComposition`, `addPort`, `connect`, or `find`.

Examples

Set and List Properties of AUTOSAR Architecture Elements

In an AUTOSAR architecture model, use the `set` function to:

- Modify the `Name` property for two AUTOSAR composition ports.
- Modify the `Name` property for an AUTOSAR component port, which also renames the corresponding Simulink implementation model port.
- Modify the `Kind` and `Name` properties for the AUTOSAR component.

Then list the model port `Name` values, which reflect the port renames.

```
% Create AUTOSAR architecture model
modelName = 'myArchModel';
archModel = autosar.arch.createModel(modelName);

% Add composition and component at architecture model top level
composition = addComposition(archModel,'Sensors');
addComponent(archModel,'Controller1');

% Add composition ports
addPort(composition,'Receiver',{'TPS_Hw','APP_Hw'});
addPort(composition,'Sender',{'TPS_Perc','APP_Perc'});

% Add component ports
controller = find(archModel,'Component','Name','Controller1');
addPort(controller,'Receiver',{'TPS_Perc','APP_Perc'});
addPort(controller,'Sender','ThrCmd_Perc');

% Connect composition and component based on matching port names
connect(archModel,composition,controller);

% Create implementation model for component
createModel(controller);

layout(archModel);  % Auto-arrange layout

% Set properties
```

```
set(composition.Ports(1),'Name','NewPortName1');  % Rename 2 composition ports
set(composition.Ports(3),'Name','NewPortName2');
set(find(controller,'Port','Name','TPS_Perc'),...
    'Name','NewPortName3');  % Rename port for Controller1 component & implementation
set(controller,'Kind','ServiceProxy');  % Component type for Controller1 component
set(controller,'Name','Instance1');     % Name for Controller1 component

% Find ports in architecture model hierarchy
ports_in_hierarchy = find(archModel,'Port','AllLevels',true)
% List Kind and Name property values for each port
for ii=1:length(ports_in_hierarchy)
    port = ports_in_hierarchy(ii);
    portName = get(port,'Name');
    portKind = get(port,'Kind');
    fprintf('%s port %s\n',portKind,portName);
end
```

```
ports_in_hierarchy =
  7×1 CompPort array with properties:
    Kind
    Connected
    Name
    Parent
    SimulinkHandle

Receiver port NewPortName1
Receiver port APP_Hw
Sender port NewPortName2
Sender port APP_Perc
Sender port ThrCmd_Perc
Receiver port NewPortName3
Receiver port APP_Perc
```

Input Arguments

`archElement` — Architecture element

handle

AUTOSAR architecture element for which to set the value of a property. The argument is a component, composition, port, or connector handle returned by a previous call to `addComponent`, `addComposition`, `addPort`, `connect`, or `find`.

Example: `port`

`property` — Element property

character vector | string scalar

Property for which to set a value, among valid properties of the AUTOSAR architecture element.

Example: `'Name'`

`value` — Property value

value of property

Value to set for the specified property of the specified AUTOSAR architecture element.

Example: `'NewPortName1'`

See Also

`find` | `get`

Topics

"Configure AUTOSAR Architecture Model Programmatically"

"Author AUTOSAR Compositions and Components in Architecture Model"

Introduced in R2020a

setClassName

Set class name of model

Syntax

`setClassName(slMap, name)`

Description

`setClassName(slMap, name)` sets the class name of the model in the generated code.

Examples

Set Class Name of Model

Open the model. To access the mapping information associated with the model, `slMap`, use the `autosar.api.getSimulinkMapping` function.

```
%% Open an adaptive AUTOSAR model
hModel = 'autosar_LaneGuidance';
addpath(fullfile(matlabroot,'/examples/autosarblockset/main'));
open_system(hModel);

%% Access the mapping information
slMap = autosar.api.getSimulinkMapping(hModel);
```

Specify a class name for the model by using the `setClassName` function.

```
setClassName(slMap, 'myClassName');
```

The `getClassName` function now returns the specified class name.

```
name = getClassName(slMap)
```

```
name =

    'myClassName'
```

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`

name — Class name of model
character vector

Class name of model in the generated code specified as a character vector. If you do not specify a class name, the class name of the model in the generated code is set to the name of the model.

Data Types: `char` | `string`

See Also

`autosar.api.getSimulinkMapping` | `getClassName` | `getClassNamespace` | `setClassNamespace`

Topics

"Configure AUTOSAR Adaptive Code Generation"

Introduced in R2021b

setClassNamespace

Set class namespace of model

Syntax

```
setClassNamespace(slMap, namespace)
```

Description

`setClassNamespace(slMap, namespace)` sets the class namespace of the model in the generated code. Control the scope of the generated code by specifying a namespace for the generated class. In systems that use a model hierarchy, you can specify a different namespace for each model in the hierarchy.

Examples

Set Class Namespace for Model

Open the model. To access the mapping information associated with the model, `slMap`, use the `autosar.api.getSimulinkMapping` function.

```
%% Open an adaptive AUTOSAR model
hModel = 'autosar_LaneGuidance';
addpath(fullfile(matlabroot,'/examples/autosarblockset/main'));
open_system(hModel);

%% Access the mapping information
slMap = autosar.api.getSimulinkMapping(hModel);
```

To specify a namespace for the model in the generated code, use the `setClassNamespace` function.

```
setClassNamespace(slMap, 'myClassNamespace');
```

To configure a nested namespace, use the scope resolution operator `::` to specify scope.

```
setClassNamespace(slMap, 'myNestedClassNamespace1::ns2::ns3');
```

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`

`namespace` — Class namespace of model
character vector

Class namespace of model in the generated code specified as a character vector. If you do not specify a class namespace, the code generated for the model does not use a namespace.

Data Types: `char` | `string`

See Also

`autosar.api.getSimulinkMapping` | `getClassNamespace` | `setClassName` | `getClassName`

Topics

"Configure AUTOSAR Adaptive Code Generation"

Introduced in R2021b

setInternalDataPackaging

Package: autosar.api

Set default internal data packaging for AUTOSAR component model

Syntax

setInternalDataPackaging(slMap,pkgSetting)

Description

setInternalDataPackaging(slMap,pkgSetting) sets the default data packaging to use for internal data stores, signals, and states in the generated code for an AUTOSAR component model. Valid setting values are:

- Default — Accept the default internal data packaging provided by the software. Use Default for multi-instance models and submodels referenced from AUTOSAR component models.
- PrivateGlobal — Package internal variable data without a struct and make it private (visible only to *model*.c).
- PrivateStructure — Package internal variable data in a struct and make it private (visible only to *model*.c).
- PublicGlobal — Package internal variable data without a struct and make it public (extern declaration in *model*.h).
- PublicStructure — Package internal variable data in a struct and make it public (extern declaration in *model*.h).

If the data packaging is set to PrivateGlobal or PrivateStructure, building the model generates header file *model*_private.h, even when model configuration parameter **File packaging format** is set to Compact.

If the model configuration option **Generate separate internal data per entry-point function** is set for the AUTOSAR model, task-based internal data grouping overrides the AUTOSAR internal data packaging setting. However, the AUTOSAR setting determines the public or private visibility of the generated task-based internal data groups.

Examples

Specify Private, Structure-Based Internal Data Packaging for AUTOSAR Model

Return and modify the default data packaging setting used for internal variables in the generated code for the AUTOSAR component model. Specify to package the internal variable data in a struct and make it private.

```
hModel = 'autosar_swc';
addpath(fullfile(matlabroot,'/examples/autosarblockset/main'));
open_system(hModel);
slMap = autosar.api.getSimulinkMapping(hModel);
pkgSetting1 = getInternalDataPackaging(slMap)
```

```
setInternalDataPackaging(slMap,'PrivateStructure')
pkgSetting2 = getInternalDataPackaging(slMap)

pkgSetting1 =
    'Default'

pkgSetting2 =
    'PrivateStructure'
```

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`

`pkgSetting` — Default internal data packaging setting

character vector | string scalar

Value specifying the default data packaging to use for internal variables in the generated code for the AUTOSAR component model. Valid setting values are `Default`, `PrivateGlobal`, `PrivateStructure`, `PublicGlobal`, and `PublicStructure`.

Example: `'PrivateStructure'`

See Also

`getInternalDataPackaging` | `autosar.api.getSimulinkMapping`

Topics

"Map AUTOSAR Elements for Code Generation"
"AUTOSAR Component Configuration"

Introduced in R2021a

setXmlOptions

Package: `autosar.arch`

Set XML option for AUTOSAR architecture model

Syntax

`setXmlOptions(archModel,property,value)`

Description

`setXmlOptions(archModel,property,value)` sets XML option `property` to `value` in architecture model `archModel`. The `archModel` argument is a model handle returned by a previous call to `autosar.arch.createModel` or `autosar.arch.loadModel`. For more information about XML options, see “Configure AUTOSAR XML Options”.

Examples

Set Value of XML Option DataTypePackage for AUTOSAR Architecture Model

For a new AUTOSAR architecture model, modify the value of the AUTOSAR XML data type package path from `/DataTypes` to `/MyDataTypes`.

```
archModel = autosar.arch.createModel('MyArchModel');
setXmlOptions(archModel,'DataTypePackage','/MyDataTypes');
pValue = getXmlOptions(archModel,'DataTypePackage')
```

```
pValue =
    '/MyDataTypes'
```

Input Arguments

`archModel` — Architecture model

handle

AUTOSAR architecture model for which to set the value of an XML option. The argument is a model handle returned by a previous call to `autosar.arch.createModel` or `autosar.arch.loadModel`.

Example: `archModel`

`property` — XML option

character vector | string scalar

XML option for which to set a value. For more information about XML options, see “Configure AUTOSAR XML Options”.

Example: `'DataTypePackage'`

`value` — XML option value

value of option

Value to set for the specified XML option of the specified AUTOSAR architecture model.

Example: `'/MyDataTypes'`

See Also

`export` | `getXmlOptions`

Topics

"Configure AUTOSAR Architecture Model Programmatically"
"Generate and Package AUTOSAR Composition XML Descriptions and Component Code"
"Author AUTOSAR Compositions and Components in Architecture Model"

Introduced in R2020a

updateAUTOSARProperties

Package: `arxml`

Update model with ARXML definitions from AUTOSAR element packages

Syntax

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

Description

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

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`.

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

If you import AUTOSAR numeric or enumeration data types, you can use the `createNumericType` and `createEnumeration` functions to create corresponding Simulink data type objects.

`updateAUTOSARProperties(ar,modelname,Name,Value)` updates the specified open model with AUTOSAR elements by 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 shows how to import definitions from the example shared descriptions file `SwAddrMethods.arxml` into the example model `autosar_swc` and generate an update report.

The ARXML file is located at `*matlabroot*/examples/autosarblockset/data`, which is on the default MATLAB search path.

```
addpath(fullfile(matlabroot,'/examples/autosarblockset/main'));  % Add path to model
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

This example shows the function call syntax to update a model 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

This example shows the function call syntax to 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

This example shows the function call syntax to update a model 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

This example shows the function call syntax to update a model 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

handle

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

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 pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

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 | createEnumeration | createNumericType

Topics

"Import and Reference Shared AUTOSAR Element Definitions"
"Import AUTOSAR Package into Component Model"
"Import AUTOSAR Package into Adaptive Component Model"
"Import AUTOSAR XML Descriptions Into Simulink"
"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 found in the XML files associated with `arxml.importer` object `ar`. The XML files must contain the AUTOSAR software component mapped by the model.

When comparing the current version of the XML file with the previous version, the comparison routine applies these rules in order:

1. If elements have the same UUID and type, the elements match. The function does not update the model.
2. If elements have different UUIDs, the elements do not match. The function updates the model with the ARXML change.
3. If elements have the same qualified name, the elements match. The function does not update the model.
4. Otherwise, elements do not match. The function updates the model with the ARXML changes.

The update generates and opens a report that details the changes made to the model, and required changes that were not made by the function.

AUTOSAR package structure updates affect the stored AR-PACKAGE structure and are applied to future exports. But imported package structure updates do not affect AUTOSAR Dictionary package path XML options. The XML package path options apply to AUTOSAR elements created in Simulink rather than to imported elements.

Examples

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
handle

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

`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` | `updateAUTOSARProperties`

Topics

"Import AUTOSAR Software Component Updates"
"Import AUTOSAR Component to Simulink"
"Import AUTOSAR Composition to Simulink"
"Import AUTOSAR XML Descriptions Into Simulink"
"AUTOSAR ARXML Importer"

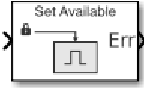
Introduced in R2014a

Blocks

Control Function Available Caller

Call AUTOSAR Function Inhibition Manager (FiM) service interface `ControlFunctionAvailable`

Library: AUTOSAR Blockset / Classic Platform / Basic Software / Function Inhibition Manager (FiM)



Description

For the AUTOSAR Classic Platform, the AUTOSAR standard defines important services as part of Basic Software (BSW) that runs in the AUTOSAR Runtime Environment (RTE). Examples include services provided by the Diagnostic Event Manager (Dem), the Function Inhibition Manager (FiM), and the NVRAM Manager (NvM). In the AUTOSAR RTE, AUTOSAR software components typically access BSW services using client-server communication.

To support system-level modeling and simulation of AUTOSAR components and services, AUTOSAR Blockset provides an AUTOSAR Basic Software block library. The library contains preconfigured blocks for modeling component calls to AUTOSAR BSW services and reference implementations of the BSW services.

As defined in the AUTOSAR specification, the Function Inhibition Manager provides a control mechanism for selectively inhibiting (deactivating) function execution in software component runnables based on function identifiers (FIDs) with inhibit conditions.

The Function Inhibition Manager is closely related to the Diagnostic Event Manager because inhibiting conditions can be based on the status of diagnostic events. The Control Function Available Caller block calls the FiM service interface `ControlFunctionAvailable` to initiate the `SetFunctionAvailable` operation.

Parameters

`Client port name` — Name of client port AUTOSAR component uses to call FiM service interface `ControlFunctionAvailable`

`FiM_ControlFunctionAvailable` (default) | character vector

Enter the name of the client port the AUTOSAR software component uses to call the FiM service interface `FiM_ControlFunctionAvailable`.

`Operation` — Specify operation defined in FiM service interface `ControlFunctionAvailable`

`SetFunctionAvailable` (default)

This block supports the FiM operation `SetFunctionAvailable` and generates inports and outports for the operation. Passing a true value marks the function associated with the client port as available, a false value marks the function as not available. A `GetPermission` operation (Function Inhibition Caller block) associated with a function that is not available returns false.

`Sample time` — Block sample time

-1 (default) | scalar

Block sample time. The default sets the block to inherit its sample time from the model.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

See Also

Function Inhibition Caller | DiagnosticOperationCycleCaller | Diagnostic Service Component

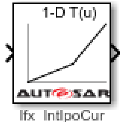
Topics
“Configure Calls to AUTOSAR Function Inhibition Manager Service”
“Model AUTOSAR Basic Software Service Calls”

Introduced in R2020a

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" 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”.

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 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'`

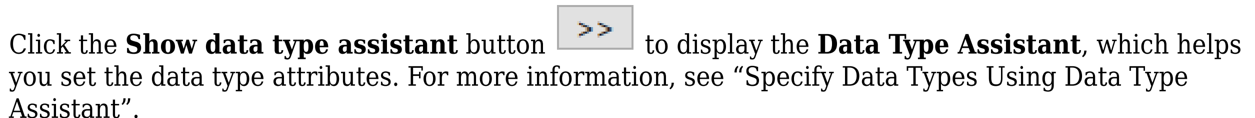
Data Types

`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”.

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'`

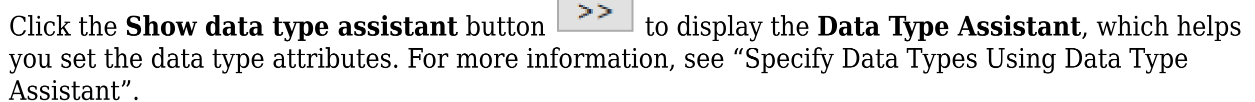
`Breakpoints` — Breakpoint data type

`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 date. 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”.

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

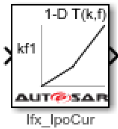
“Configure Lookup Tables for AUTOSAR Calibration and Measurement”
“Code Generation with AUTOSAR Code Replacement Library”

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`

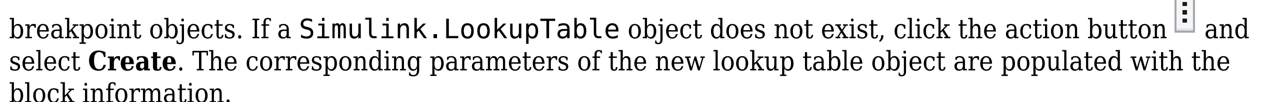
Specify whether to enter table data directly or use a lookup table object. If you set this parameter to:

- `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” 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”.

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'`

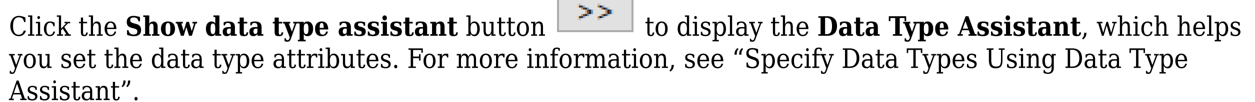
Data Types

`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”.

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

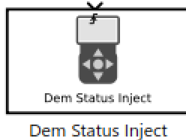
Curve | Map | Map Using Prelookup | Prelookup

Topics
"Configure Lookup Tables for AUTOSAR Calibration and Measurement"
"Code Generation with AUTOSAR Code Replacement Library"

Introduced in R2019a

Dem Status Inject

Inject an event failure to test recovery
Library: AUTOSAR Blockset / Classic Platform / Basic Software / Diagnostic Event Manager (Dem)



Dem Status Inject

Description

The Dem Status Inject block can be configured to instantaneously set the diagnostic status for an AUTOSAR event simulated in the Diagnostic Service Component block. This status value can be configured according to the Unified Diagnostic Services (UDS) standard. Specifically, you can use this block to inject a transient failure into a system to test its ability to recover. This block simulates and responds to other blocks affecting its status to show system recovery.

Parameters

`EventID` — Specify the event

0 (default) | integer

Specify the AUTOSAR event that you want to override by using this block.

`Fault type` — Specify the type of fault

Event Fail (default) | Event Pass | Operation Cycle Start | Operation Cycle End | Fault Record Overwritten | Fault Maturation | Clear Diagnostic | Aging | Healing | Indicator Conditions Met

Specify the type of diagnostic event that you want to inject into the system.

`Trigger type` — Specify the inject condition

rising (default) | falling | either | function-call | message

Specify when to inject the diagnostic event into the system.

`Test Failed` — Indicates the result of the most recently performed test

Clear (default) | Set

Test failed the last time it was checked.

This read-only property is set by the **Fault type**.

`Test Failed this Operation Cycle` — Indicates whether a diagnostic test has reported a failure during the current operation cycle

Clear (default) | Set

Test failed during the current operation cycle.

This read-only property is set by the **Fault type**.

`Pending DTC` — Indicates whether a diagnostic test has reported a failure during the current or last completed operation cycle
Clear (default) | Set

Test failed during the current or previous operation cycle.

This read-only property is set by the **Fault type**.

`Confirmed DTC` — Indicates whether a malfunction was detected enough times to warrant that the DTC should be stored in long-term memory
Clear (default) | Set

Test failure confirmed at the time of the request.

This read-only property is set by the **Fault type**.

`Test Not Completed Since Last Clear` — Indicates whether a test has run and completed since the last time a call was made to ClearDiagnosticInformation
Clear (default) | Set

Test not performed since the last code clear.

This read-only property is set by the **Fault type**.

`Test Failed Since Last Clear` — Indicates whether a test has failed since the last time a call was made to ClearDiagnosticInformation
Cear (default) | Set

Test failed at least once since last code clear.

This read-only property is set by the **Fault type**.

`Test Not Completed This Operation Cycle` — Test not performed during the current operation cycle
Clear (default) | Set

Test not completed during this operation cycle.

This read-only property is set by the **Fault type**.

`Warning Indicator Requested` — Indicates the status of any warning indicators associated with a particular DTC
Clear (default) | Set

Test failure so severe that it alerts the server.

This read-only property is set by the **Fault type**.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

See Also

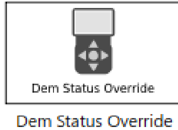
Dem Status Override

Introduced in R2022a

Dem Status Override

Override an event to simulate and verify behavior

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



Dem Status Override

Description

The Dem Status Override block can be configured to set the diagnostic status for an AUTOSAR event simulated in the Diagnostic Service Component block. This status value can be configured according to the Unified Diagnostic Services (UDS) standard. Specifically, you can use this block to set the return value of `GetEventStatus` calls for a specific event regardless of the behavior of the other events in the model. Other blocks that attempt to modify the event status during simulation are ignored. This block allows a predictable entry-point for testing specific behavior that you can use as a low-cost method to quickly gain coverage of a component model.

Parameters

`EventID` — Specify the event
0 (default) | integer

Specify the AUTOSAR event that you want to override by using this block.

`Test Failed` — Indicates the result of the most recently performed test
Clear (default) | Set

Test failed the last time it was checked.

To set this property, select `Dialog` to manually configure the status or `Input port` to have the status configured dynamically in response to the behavior of a connected input.

`Test Failed this Operation Cycle` — Indicates whether a diagnostic test has reported a failure during the current operation cycle
Clear (default) | Set

Test failed during the current operation cycle.

To set this property, select `Dialog` to manually configure the status or `Input port` to have the status configured dynamically in response to the behavior of a connected input.

`Pending DTC` — Indicates whether a diagnostic test has reported a failure during the current or last completed operation cycle
Clear (default) | Set

Test failed during the current or previous operation cycle.

To set this property, select `Dialog` to manually configure the status or `Input port` to have the status configured dynamically in response to the behavior of a connected input.

`Confirmed DTC` — Indicates whether a malfunction was detected enough times to warrant that the DTC should be stored in long-term memory
Clear (default) | Set

Test failure confirmed at the time of the request.

To set this property, select `Dialog` to manually configure the status or `Input port` to have the status configured dynamically in response to the behavior of a connected input.

`Test Not Completed Since Last Clear` — Indicates whether a test has run and completed since the last time a call was made to ClearDiagnosticInformation
Clear (default) | Set

Test not performed since the last code clear.

To set this property, select `Dialog` to manually configure the status or `Input port` to have the status configured dynamically in response to the behavior of a connected input.

`Test Failed Since Last Clear` — Indicates whether a test has failed since the last time a call was made to ClearDiagnosticInformation
Cear (default) | Set

Test failed at least once since last code clear.

To set this property, select `Dialog` to manually configure the status or `Input port` to have the status configured dynamically in response to the behavior of a connected input.

`Test Not Completed This Operation Cycle` — Test not performed during the current operation cycle
Clear (default) | Set

Test not completed during this operation cycle.

To set this property, select `Dialog` to manually configure the status or `Input port` to have the status configured dynamically in response to the behavior of a connected input.

`Warning Indicator Requested` — Indicates the status of any warning indicators associated with a particular DTC
Clear (default) | Set

Test failure so severe that it alerts the server.

To set this property, select `Dialog` to manually configure the status or `Input port` to have the status configured dynamically in response to the behavior of a connected input.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

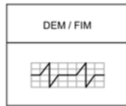
See Also

Dem Status Inject

Introduced in R2022a

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) and Function Inhibition Manager (FiM) services supported by AUTOSAR Basic Software (BSW) caller blocks. When coupled with Dem and FiM caller blocks, the reference implementations enable you to configure and run system-level or composition-level simulations of AUTOSAR Dem and FiM service calls.

The block has prepopulated parameters, including RTE service ID parameters, Dem **Counter-Based Debouncing** parameters, and FiM inhibition condition parameters. Examine the parameter settings and, if necessary, make modifications based on how you are using the Dem or FiM service operations.

The RTE tab lists component client ports and their mapping to Dem or FiM service IDs for events, operation cycles, or functions with inhibition conditions. Each row in the table represents a call into Dem or FiM services from a Basic Software caller block, for which you can modify an ID value.

The Dem tab **Counter-Based Debouncing** parameters control the counter-based debounce algorithm provided by the Dem service reference implementations. During multiple simulations, you can adjust event step size and threshold parameters and observe the effects.

Use the counter-based debouncing parameters to determine when a monitored event has passed or failed. For each event ID, the software maintains a counter. When PREFAIL events arrive, the event ID counter increments by the **Increment step size** (default 1). When PREPASS events arrive, the event ID counter decrements by the **Decrement step size** (default 1). To determine the event ID counter thresholds at which an event fails or passes, use block parameters **Failed threshold** (default 2) and **Passed threshold** (default -1).

In the Dem reference implementations, the step size and threshold parameters apply globally to event IDs, not to individual IDs.

The FIM tab lists function identifiers (FIDs) and their associated inhibition conditions and client ports. The tab provides graphical controls for adding or removing inhibition conditions for a selected FID. For each inhibition condition, select ID and mask values.

Parameters

`ID (RTE tab)` — ID that identifies service event, operation cycle, or function with inhibition condition

1 (default) | scalar

Each row in the RTE tab table represents a call into Dem or FiM services from a Basic Software caller block. Check the ID mappings for events, operation cycles, and functions with inhibition conditions.

For events, calls that act on the same event use the same event ID. For an example of mapping Dem client ports to shared event IDs, see “Simulate AUTOSAR Basic Software Services and Run-Time Environment”.

`Increment step size` — Fixed-step value to increment event ID counter when PREFAIL events arrive
1 (default) | scalar (1 to 32767)

Specify a fixed-step value that the Dem event ID counter increments by when PREFAIL events arrive.

`Decrement step size` — Fixed-step value to decrement the event ID counter when PREPASS events arrive
1 (default) | scalar (1 to 32767)

Specify a fixed-step value that the Dem event ID counter decrements by when PREPASS events arrive.

`Failed threshold` — Dem event ID counter threshold that represents a failed status
2 (default) | scalar (1 to 32767)

Specify a Dem event ID counter threshold value to represent failed status. Events that reach this threshold are considered to have failed.

`Passed threshold` — Dem event ID counter threshold that represents passed status
-1 (default) | scalar (-32768 to -1)

Specify a Dem event ID counter threshold value to represent passed status. Events that reach this threshold are considered to have passed.

`ID (FiM tab)` — Inhibition condition ID
1 (default) | scalar

In the FiM tab table, each row grouped under an FID represents an inhibition condition with an ID, one or more component client ports associated with the ID, and a mask. For each inhibition condition, you can modify the ID value. For examples of inhibition condition configuration, see “Configure and Simulate AUTOSAR Function Inhibition Service Calls”.

`Mask` — Inhibition condition mask
LAST_FAILED | NOT_TESTED | TESTED | TESTED_AND_FAILED

In the FiM tab table, each row grouped under an FID represents an inhibition condition with an ID, one or more component client ports associated with the ID, and a mask. For each inhibition condition, you can modify the mask value. For examples of inhibition condition configuration, see “Configure and Simulate AUTOSAR Function Inhibition Service Calls”.

See Also

DiagnosticInfoCaller | DiagnosticMonitorCaller | DiagnosticEventAvailableCaller | DiagnosticOperationCycleCaller | Function Inhibition Caller | Control Function Available Caller

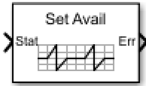
Topics

“Configure Calls to AUTOSAR Diagnostic Event Manager Service”
“Configure Calls to AUTOSAR Function Inhibition Manager Service”
“Configure AUTOSAR Basic Software Service Implementations for Simulation”

Introduced in R2017b

DiagnosticEventAvailableCaller

Call AUTOSAR Diagnostic Event Manager (Dem) service interface `EventAvailable`
Library: AUTOSAR Blockset / Classic Platform / Basic Software / Diagnostic Event Manager (Dem)



Description

For the AUTOSAR Classic Platform, the AUTOSAR standard defines important services as part of Basic Software (BSW) that runs in the AUTOSAR Runtime Environment (RTE). Examples include services provided by the Diagnostic Event Manager (Dem), the Function Inhibition Manager (FiM), and the NVRAM Manager (NvM). In the AUTOSAR RTE, AUTOSAR software components typically access BSW services by using client-server communication.

To support system-level modeling and simulation of AUTOSAR components and services, AUTOSAR Blockset provides an AUTOSAR Basic Software block library. The library contains preconfigured blocks for modeling component calls to AUTOSAR BSW services and reference implementations of the BSW services.

The DiagnosticEventAvailableCaller block calls the Dem service interface `EventAvailable` to initiate the `SetEventAvailable` operation. A component uses `SetEventAvailable` to temporarily disable and enable a specific event, for example, an event of the same name associated with an existing Dem `SetEventStatus` caller block. Typically you connect a true/false Boolean constant block to the `SetEventAvailable` input, so that you can switch the event off (false) or on (true). When disabled, the event fired by the `SetEventStatus` block has no effect.

Parameters

`Client port name` — Name of client port AUTOSAR component uses to call Dem service interface `EventAvailable`
`EventAvailable` (default) | character vector

Enter the name of the client port the AUTOSAR software component uses to call the Dem service interface `EventAvailable`.

`Operation` — Specify operation defined in Dem service interface `EventAvailable`
`SetEventAvailable` (default)

This block supports the Dem operation `SetEventAvailable` and generates inports and outports for the operation. You can use this operation to configure events as unavailable. An unavailable event is treated as if it is not configured in the system and returns E_NOT_OK when accessed by other operations.

`Sample time` — Block sample time
-1 (default) | scalar

Block sample time. The default sets the block to inherit its sample time from the model.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

See Also

DiagnosticInfoCaller | DiagnosticMonitorCaller | DiagnosticOperationCycleCaller | Diagnostic Service Component

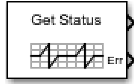
Topics
"Configure Calls to AUTOSAR Diagnostic Event Manager Service"
"Model AUTOSAR Basic Software Service Calls"

Introduced in R2020a

DiagnosticInfoCaller

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

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



Description

The AUTOSAR standard defines a Diagnostic Event Manager (Dem) service as a part of Basic Software (BSW) that runs in the AUTOSAR Runtime Environment (RTE). AUTOSAR software components access Dem services through client-server calls. The DiagnosticInfoCaller block calls the Dem service interface `DiagnosticInfo` to initiate a specified operation.

Parameters

`Client port name` — Name of client port AUTOSAR component uses to call Dem service interface `DiagnosticInfo`

`DiagnosticInfo` (default) | character vector

Enter the name of the client port the AUTOSAR software component uses to call the Dem service interface `DiagnosticInfo`.

`Operation` — Specify operation defined in Dem service interface `DiagnosticInfo`

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

Select the operation that the AUTOSAR software component calls from the Dem service interface `DiagnosticInfo`. The AUTOSAR standard defines the operations. After you select the operation, the inports and outports for the block are generated to support the operation.

`Data type for FormatStatus` — Specify data type to represent a Dem format type

`Enum:Dem_DTCFormatType` (default)

Specify an enumerated data type to represent a Dem format type required for some operations. For more information, see the AUTOSAR standard *Specification of Diagnostic Event Manager*.

Dependencies

Specify this parameter when **Operation** is set to `GetDTCOfEvent`.

`Sample time` — Block sample time

-1 (default) | scalar

Block sample time. The default sets the block to inherit its sample time from the model.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

See Also

DiagnosticMonitorCaller | DiagnosticOperationCycleCaller | DiagnosticEventAvailableCaller | Diagnostic Service Component

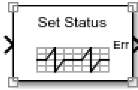
Topics
"Configure Calls to AUTOSAR Diagnostic Event Manager Service"
"Configure AUTOSAR Basic Software Service Implementations for Simulation"

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

The AUTOSAR standard defines a Diagnostic Event Manager (Dem) service as a part of Basic Software (BSW) that runs in the AUTOSAR Runtime Environment (RTE). AUTOSAR software components access Dem services through client-server calls. The DiagnosticMonitorCaller block calls the Dem service interface `DiagnosticMonitor` to initiate a specified operation.

Parameters

`Client port name` — Name of client port AUTOSAR component uses to call Dem service interface `DiagnosticMonitor`

`DiagnosticMonitor` (default) | character vector

Enter the name of the client port the AUTOSAR software component uses to call the Dem service interface `DiagnosticMonitor`.

`Operation` — Specify operation defined in Dem service interface `DiagnosticMonitor`

`SetEventStatus` (default) | `ResetEventStatus` | `PrestoreFreezeFrame` | `ClearPrestoredFreezeFrame` | `SetEventDisabled`

Select the operation that the AUTOSAR software component calls from the Dem service interface `DiagnosticMonitor`. The operations are defined by the AUTOSAR standard. After the operation is selected, the inports and outports for the block are generated to support the operation.

`Data type for EventStatus` — Specify a data type to represent a Dem event type

`Enum: Dem_EventStatusType` (default)

Specify an enumerated data type to represent a Dem event type required for some operations. For more information, see the AUTOSAR standard *Specification of Diagnostic Event Manager*.

Dependencies

Specify this parameter when **Operation** is set to `SetEventStatus`.

`Sample time` — Block sample time

-1 (default) | scalar

Block sample time. The default sets the block to inherit its sample time from the model.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

See Also

DiagnosticInfoCaller | DiagnosticOperationCycleCaller | DiagnosticEventAvailableCaller | Diagnostic Service Component

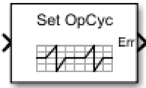
Topics
"Configure Calls to AUTOSAR Diagnostic Event Manager Service"
"Configure AUTOSAR Basic Software Service Implementations for Simulation"

Introduced in R2016b

DiagnosticOperationCycleCaller

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

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



Description

For the AUTOSAR Classic Platform, the AUTOSAR standard defines important services as part of Basic Software (BSW) that runs in the AUTOSAR Runtime Environment (RTE). Examples include services provided by the Diagnostic Event Manager (Dem), the Function Inhibition Manager (FiM), and the NVRAM Manager (NvM). In the AUTOSAR RTE, AUTOSAR software components typically access BSW services by using client-server communication.

To support system-level modeling and simulation of AUTOSAR components and services, AUTOSAR Blockset provides an AUTOSAR Basic Software block library. The library contains preconfigured blocks for modeling component calls to AUTOSAR BSW services and reference implementations of the BSW services.

As defined in the AUTOSAR specification, the Function Inhibition Manager provides a control mechanism for selectively inhibiting (deactivating) function execution in software component runnables based on function identifiers (FIDs) with inhibit conditions.

The Function Inhibition Manager is closely related to the Diagnostic Event Manager because inhibiting conditions can be based on the status of diagnostic events. An operation cycle affects events that share the same Diagnostic Service Component. The DiagnosticOperationCycleCaller block calls the Dem service interface `OperationCycle` to control operation cycles.

Parameters

`Client port name` — Name of client port AUTOSAR component uses to call Dem service interface `OperationCycle`

`OperationCycle` (default) | character vector

Enter the name of the client port the AUTOSAR software component uses to call the Dem service interface `OperationCycle`.

`Operation` — Specify operation defined in Dem service interface `OperationCycle`

`SetOperationCycleState` (default) | `GetOperationCycleState`

Select a Dem operation to control or monitor operation cycles. To start and stop operation cycles, select `SetOperationCycleState`. To query the current state of an operation cycle, select `GetOperationCycleState`. After you select an operation, the inports and outports for the block are generated to support that operation.

`Data type for CycleState` — Specify to start or stop operation cycles

`Enum:Dem_OperationCycleStateType.DEM_CYCLE_STATE_START` (default) |
`Enum:Dem_OperationCycleStateType.DEM_CYCLE_STATE_END`

Enter a value to control the start or stop of component operation cycles. To start operation cycles, enter the value `Enum:Dem_OperationCycleStateType.DEM_CYCLE_STATE_START`. To end operation cycles, enter the value `Enum:Dem_OperationCycleStateType.DEM_CYCLE_STATE_END`.

Dependencies

Specify this parameter when **Operation** is set to `SetOperationCycleState`.

`Sample time` — Block sample time
-1 (default) | scalar

Block sample time. The default sets the block to inherit its sample time from the model.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

See Also
Function Inhibition Caller | Control Function Available Caller | Diagnostic Service Component

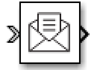
Topics
"Configure Calls to AUTOSAR Function Inhibition Manager Service"
"Model AUTOSAR Basic Software Service Calls"

Introduced in R2020a

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.

Ports

Input

`Port_1` — Input event
scalar | vector | matrix

The input port for the Event Receive block to accept an event.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `Boolean` | `enumerated` | `bus`

Output

`Port_1` — Output signal
scalar | vector | matrix

The output port for the Event Receive block to output a signal.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `Boolean` | `enumerated` | `bus`

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

See Also

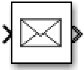
Event Send

Topics
“Configure AUTOSAR Adaptive Software Components”
“Create and Configure AUTOSAR Adaptive Software Component”

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 outport, add an Event Send block which converts an input signal to an event, while preserving the signal values and data type.

Ports

Input

`Port_1` — Input signal
scalar | vector | matrix

The input port for the Event Send block to receive inputs of any type that AUTOSAR Blockset supports.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `Boolean` | `enumerated` | `bus`

Output

`Port_1` — Output event
scalar | vector | matrix

The output port for the Event Send block to output an event.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `Boolean` | `enumerated` | `bus`

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

See Also

Event Receive

Topics
“Configure AUTOSAR Adaptive Software Components”

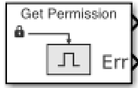
“Create and Configure AUTOSAR Adaptive Software Component”

Introduced in R2019a

Function Inhibition Caller

Call AUTOSAR Function Inhibition Manager (FiM) service interface FunctionInhibition

Library: AUTOSAR Blockset / Classic Platform / Basic Software / Function Inhibition Manager (FiM)



Description

For the AUTOSAR Classic Platform, the AUTOSAR standard defines important services as part of Basic Software (BSW) that runs in the AUTOSAR Runtime Environment (RTE). Examples include services provided by the Diagnostic Event Manager (Dem), the Function Inhibition Manager (FiM), and the NVRAM Manager (NvM). In the AUTOSAR RTE, AUTOSAR software components typically access BSW services by using client-server communication.

To support system-level modeling and simulation of AUTOSAR components and services, AUTOSAR Blockset provides an AUTOSAR Basic Software block library. The library contains preconfigured blocks for modeling component calls to AUTOSAR BSW services and reference implementations of the BSW services.

As defined in the AUTOSAR specification, the Function Inhibition Manager provides a control mechanism for selectively inhibiting (deactivating) function execution in software component runnables based on function identifiers (FIDs) with inhibit conditions.

The Function Inhibition Manager is closely related to the Diagnostic Event Manager because inhibiting conditions can be based on the status of diagnostic events. The Function Inhibition Caller block calls the FiM service interface FunctionInhibition to initiate the GetFunctionPermission operation.

Parameters

Client port name — Name of client port AUTOSAR component uses to call FiM service interface FunctionInhibition

FiM_FunctionInhibition (default) | character vector

Enter the name of the client port the AUTOSAR software component uses to call the FiM service interface FiM_FunctionInhibition.

Operation — Specify operation defined in FiM service interface FunctionInhibition

GetFunctionPermission (default)

This block supports the FiM operation GetFunctionPermission and generates inports and outports for this operation. This operation queries the Function Inhibition Manager to check if it has permission to run associated functionality. Permissions are based on the inhibition configuration created by using the Diagnostic Service Component block. The operation returns true if the functionality has permission or false if the functionality is inhibited.

Sample time — Block sample time

-1 (default) | scalar

Block sample time. The default sets the block to inherit its sample time from the model.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

See Also

Control Function Available Caller | DiagnosticOperationCycleCaller | Diagnostic Service Component

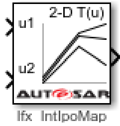
Topics
"Configure Calls to AUTOSAR Function Inhibition Manager Service"
"Model AUTOSAR Basic Software Service Calls"

Introduced in R2020a

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” 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”.

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

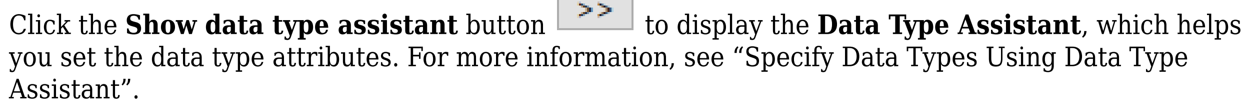
`Table data` — Data type of table data

`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”.

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'

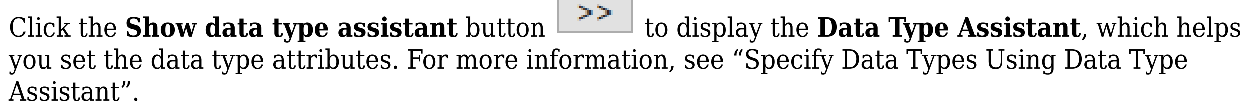
Breakpoints — Breakpoint data type

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 date. 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”.

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

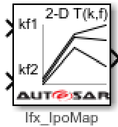
"Configure Lookup Tables for AUTOSAR Calibration and Measurement"
"Code Generation with AUTOSAR Code Replacement Library"

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`

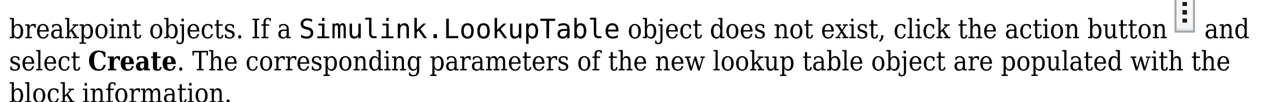
Specify whether to enter table data directly or use a lookup table object. If you set this parameter to:

- `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” 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”.

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'`

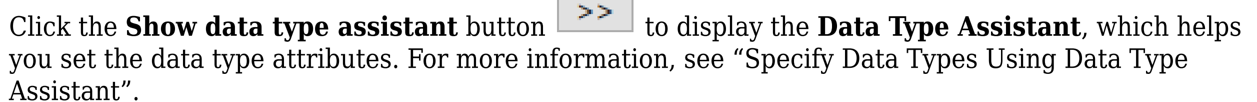
Data Types

`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".

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

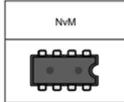
Topics
“Configure Lookup Tables for AUTOSAR Calibration and Measurement”
“Code Generation with AUTOSAR Code Replacement Library”

Introduced in R2019a

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 enable you to configure and run system-level or composition-level simulations of AUTOSAR NvM service calls.

The block has prepopulated parameters, including RTE block ID parameters and NvM **NVRAM Properties** parameters. Examine the parameter settings and consider if modifications are required based on how you are using the NvM service operations.

Parameters

`Block ID` — ID that identifies service block

1 (default) | scalar

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 use the same block ID. Check the block ID mappings. For examples of mapping NvM client ports to block IDs, see “Simulate AUTOSAR Basic Software Services and Run-Time Environment”.

`Maximum number of memory blocks` — Maximum number of memory blocks to use in NvM service operations

10 (default) | scalar

Specify maximum number of memory blocks to use in NvM service operations.

See Also

NvMAdminCaller | NvMServiceCaller

Topics

“Configure Calls to AUTOSAR NVRAM Manager Service”
“Configure AUTOSAR Basic Software Service Implementations for Simulation”

Introduced in R2017b

NvMAdminCaller

Call AUTOSAR NVRAM Manager (NvM) service interface `NvMadmin`

Library: AUTOSAR Blockset / Classic Platform / Basic Software / NVRAM Manager (NvM)

Description

The AUTOSAR standard defines a NVRAM Manager (NvM) service as a part of Basic Software (BSW) that runs in the AUTOSAR Runtime Environment (RTE). AUTOSAR software components access NvM services through client-server calls. The NvMAdminCaller block calls the AUTOSAR NvM service interface `NvMAdmin` to initiate a specified operation.

Parameters

`Client port name` — Name of client port AUTOSAR component uses to call NvM service interface `NvMAdmin`
`NvMAdmin` (default) | character vector

Enter the name of the client port the AUTOSAR software component uses to call the NvM service interface `NvMAdmin`.

`Operation` — Operation defined in NvM service interface `NvMAdmin`
`SetBlockProtection` (default)

Select the operation that the AUTOSAR software component calls from the NvM service interface `NvMAdmin`. The AUTOSAR standard defines the operations. After you select the operation, the inports and outports for the block are generated to support the operation. One operation is supported: `SetBlockProtection`.

`Sample time` — Block sample time
-1 (default) | scalar

Block sample time. The default sets the block to inherit its sample time from the model.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

See Also

NvMServiceCaller | NVRAM Service Component

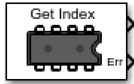
Topics
“Configure Calls to AUTOSAR NVRAM Manager Service”
“Configure AUTOSAR Basic Software Service Implementations for Simulation”

Introduced in R2016b

NvMServiceCaller

Call AUTOSAR NVRAM Manager (NvM) service interface `NvMService`

Library: AUTOSAR Blockset / Classic Platform / Basic Software / NVRAM Manager (NvM)



Description

The AUTOSAR standard defines a NVRAM Manager (NvM) service as a part of Basic Software (BSW) that runs in the AUTOSAR Runtime Environment (RTE). AUTOSAR software components access NvM services through client-server calls. The NvMServiceCaller block calls the AUTOSAR NvM service interface `NvMService` by to initiate specified operation.

Parameters

`Client port name` — Name of client port AUTOSAR component uses to call NvM service interface `NvMService`

`NvMService` (default) | character vector

Enter the name of the client port the AUTOSAR software component uses to call the NvM service interface `NvMService`.

`Operation` — Specify operation defined in NvM service interface `NvMService`

`GetDataIndex` (default) | `GetErrorStatus` | `EraseNvBlock` | `InvalidateNvBlock` | `ReadBlock` | `RestoreBlockDefaults` | `SetDataIndex` | `SetRamBlockStatus` | `WriteBlock`

Select the operation that the AUTOSAR software component calls from the NvM service interface `NvMService`. The AUTOSAR standard defines the operations. After you select the operation, the inports and outports for the block are generated to support the operation.

`Argument specification` — Specify data type and dimensions for operation read or write access

`uint8(1)` (default) | `single()` | `double()` | `int8()` | `int16()` | `int32()` | `int64()` | `uint16()` | `uint32()` | `uint64()` | `Boolean()` | `enumerated()` | `bus()`

A MATLAB expression that specifies data type and dimensions for data to be read or written by the operation.

- To specify a multidimensional data type, you can use array syntax, such as `int8([1 1; 1 1])`.
- To specify a structured data type, you can create a `Simulink.Parameter` data object, type it with a `Simulink.Bus` object, and reference the parameter name.

For examples, see "Argument Specification for Simulink Function Blocks".

Dependencies

Specify this parameter when **Operation** is set to `ReadBlock`, `RestoreBlockDefaults`, or `WriteBlock`.

`Sample time` — Block sample time
-1 (default) | scalar

Block sample time. The default sets the block to inherit its sample time from the model.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

See Also
NvMAdminCaller | NVRAM Service Component

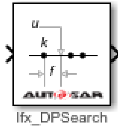
Topics
"Configure Calls to AUTOSAR NVRAM Manager Service"
"Configure AUTOSAR Basic Software Service Implementations for Simulation"

Introduced in R2016b

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)

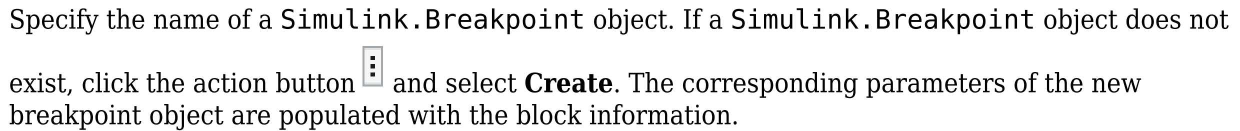
Explicitly specify the breakpoint data. Each breakpoint data set must be a strictly monotonically increasing vector that contains two or more elements.

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

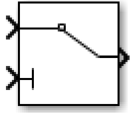
"Configure Lookup Tables for AUTOSAR Calibration and Measurement"
"Code Generation with AUTOSAR Code Replacement Library"

Introduced in R2019a

Signal Invalidation

Control AUTOSAR root outport 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 outport 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

Software Component

Model software component in AUTOSAR architecture model
Library: AUTOSAR Blockset

Description

In an AUTOSAR architecture model, you use the composition editor and the Simulink Toolstrip **Modeling** tab to add and connect compositions and components. Use the Software Component block to add a software component to an AUTOSAR software design.

To add and connect AUTOSAR components:

- For each component required by the design, from the **Modeling** tab or the palette, add a Software Component block. You can use the Property Inspector to set the component **Kind** — `Application`, `ComplexDeviceDriver`, `EcuAbstraction`, `SensorAccuator`, or `ServiceProxy`. (`Application` and `SensorAccuator` are common.)
- Add component require and provide ports. To add each component port, click an edge of a Software Component block. When port controls appear, select **Input** for a require port or **Output** for a provide port.
- To connect the Software Component blocks to other blocks, connect the block ports with signal lines.
- To connect the Software Component blocks to architecture or composition model root ports, drag from the component ports to the containing model boundary. When you release the connection, a root port is created at the boundary.
- Configure additional AUTOSAR properties using the Property Inspector.

After you add and connect AUTOSAR components, add Simulink behavior to the AUTOSAR components by creating, importing, or linking models.

If you have Requirements Toolbox™ software, you can link components in an AUTOSAR architecture model to Simulink requirements.

Ports

Input

`Input port` — Component require port

scalar | vector | matrix

Require port in the AUTOSAR software component port interface.

If you link the component block to an implementation model, the port interfaces of the block and model, including the number of ports, match.

Output

`Output port` — Component provide port
scalar | vector | matrix

Provide port in the AUTOSAR software component port interface.

If you link the component block to an implementation model, the port interfaces of the block and model, including the number of ports, match.

See Also

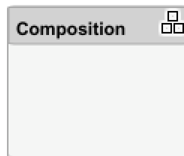
Software Composition | Diagnostic Service Component | NVRAM Service Component

Topics
“Add and Connect AUTOSAR Compositions and Components”
“Author AUTOSAR Compositions and Components in Architecture Model”
“Define AUTOSAR Component Behavior by Creating or Linking Models”
“Create AUTOSAR Architecture Views for Analysis”
“Link AUTOSAR Components to Simulink Requirements”

Introduced in R2019b

Software Composition

Model software composition in AUTOSAR architecture model
Library: AUTOSAR Blockset



Description

In an AUTOSAR architecture model, you use the composition editor and the Simulink Toolstrip **Modeling** tab to add and connect compositions and components. Use the Software Composition block to add a nested software composition to an AUTOSAR software design.

To add and connect a nested AUTOSAR composition:

- From the **Modeling** tab or the palette, add a Software Composition block.
- Add composition require and provide ports. To add each composition port, click an edge of the Software Composition block. When port controls appear, select **Input** for a require port or **Output** for a provide port.

  Alternatively, open the Software Composition block. To add each composition port, click the boundary of the composition diagram. When port controls appear, select **Input** for a require port or **Output** for a provide port.
- To connect the Software Composition block to other blocks, connect the block ports with signal lines.
- To connect the Software Composition block to architecture or composition model root ports, drag from the composition ports to the containing model boundary. When you release the connection, a root port is created at the boundary.
- Configure additional AUTOSAR properties using the Property Inspector.

An AUTOSAR composition contains a set of AUTOSAR components and compositions with a shared purpose. To populate a composition, open the Software Composition block and begin adding more Software Component and Software Composition blocks.

Ports

Input

`Input port` — Composition require port
scalar | vector | matrix

Require port in the AUTOSAR software composition port interface.

Output

`Output port` — Composition provide port
scalar | vector | matrix

Provide port in the AUTOSAR software composition port interface.

See Also

Software Component | Diagnostic Service Component | NVRAM Service Component

Topics

“Add and Connect AUTOSAR Compositions and Components”
“Author AUTOSAR Compositions and Components in Architecture Model”
“Create AUTOSAR Architecture Views for Analysis”

Introduced in R2019b

3

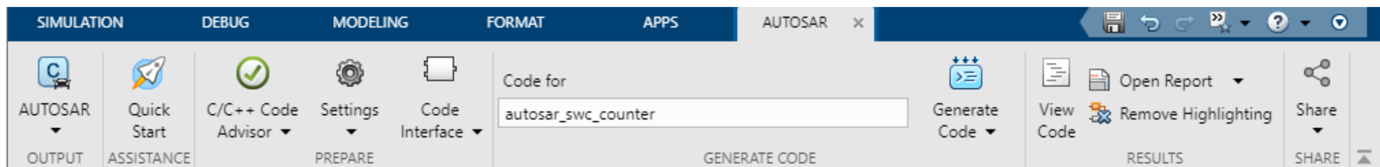
Apps

AUTOSAR Component Designer

Generate AUTOSAR compliant C or C++ code and export ARXML descriptions for processors used in automotive systems

Description

Use the **AUTOSAR Component Designer** app to generate C or C++ code and ARXML descriptions from an AUTOSAR component model. When you open the app, an **AUTOSAR** tab is added to the toolstrip. The **AUTOSAR** tab represents groups of tasks in the AUTOSAR Blockset workflow.



Use the app to perform these tasks:

- If you are new to AUTOSAR Blockset, use the Embedded Coder Quick Start to prepare your model for AUTOSAR code generation. The Quick Start chooses fundamental code generation settings based on your goals and application. Click **Quick Start**. For code generation output, select either **C code compliant with AUTOSAR** or **C++ code compliant with AUTOSAR Adaptive Platform**.
- Set code generation objectives and prepare your model for code generation. Click **C/C++ Code Advisor**.
- Configure model configuration parameters. Select **Settings > C/C++ Code generation settings** or **Settings > Hardware Implementation**.
- Opening the **AUTOSAR Component Designer** app opens the AUTOSAR Code perspective, which contains the Code Mappings editor. Use the Code Mappings editor to map model entry-point functions, data, and other elements to AUTOSAR elements and properties that are defined in the AUTOSAR standard. Select **Code Interface > Individual Element Code Mappings**.
- Configure AUTOSAR elements from an AUTOSAR component perspective by using the AUTOSAR Dictionary. Select **Code Interface > AUTOSAR Dictionary** or, from the Code Mappings editor, click . In the **XML Options** view, configure settings for ARXML export.
- Generate AUTOSAR code and ARXML descriptions for testing or integration. Click **Generate Code**.
- Open the Code view to view the generated code alongside your model. Click **View Code**. In the Code view, you can trace between model elements and the code by clicking hyperlinked lines of code. To remove traceability highlighting, click **Remove Highlighting**.
- Open the latest HTML code generation report by clicking **Open Report**. To configure HTML report generation options, from the **Open Report** menu, select **Report Options**.
- Create a protected model to share with a third party for simulation and code generation. Select **Share > Generate Protected Model**.
- Package the model code and build artifacts in a ZIP file, for example, for relocation and integration. Select **Share > Generate Code and Package**. Optionally, you can modify the name of the generated ZIP file.

Open the AUTOSAR Component Designer App

In the **Apps** gallery, under **Code Generation**, click **AUTOSAR Component Designer**. The **AUTOSAR** tab opens.

Examples

- "Create AUTOSAR Software Component in Simulink"
- "Design and Simulate AUTOSAR Components and Generate Code"
- Code Mappings Editor

Tips

- If you are working with a model hierarchy, open the **AUTOSAR Component Designer** app in the Simulink Editor window for the top model of the hierarchy that you are generating code for. On

the **AUTOSAR** tab, the functionalities apply to the top model of the hierarchy that is open in the editor.
- To configure and view code for a referenced model, navigate to the model in the hierarchy and use the AUTOSAR Dictionary, Code Mappings editor, and Code view. These views apply to the active model, which can be the top model or a referenced model.

See Also

Functions
`autosar.api.create` | `autosar_ui_launch`

Topics
"Create AUTOSAR Software Component in Simulink"
"Design and Simulate AUTOSAR Components and Generate Code"
Code Mappings Editor

Introduced in R2019b

Tools

Code Mappings Editor

Map AUTOSAR elements for code generation

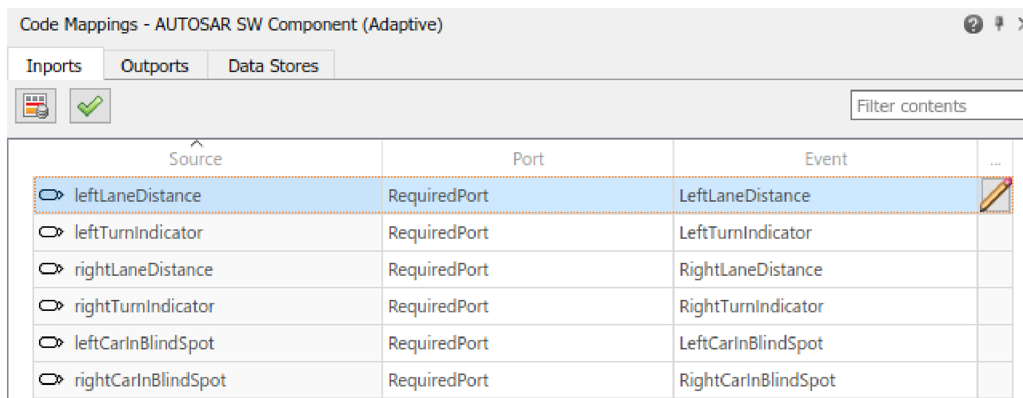
Description

The 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, the 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 or C++ code and exported ARXML descriptions.

For more information, see:

- “Map AUTOSAR Elements for Code Generation”
- “Map Calibration Data for Submodels Referenced from AUTOSAR Component Models”
- “Map AUTOSAR Adaptive Elements for Code Generation”



The 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 entry-point functions, parameters, signals, data transfers, or function callers.
- **Edit** icon – Open a properties dialog box to view and modify additional AUTOSAR code and calibration attributes for the currently-selected element.

Open the Code Mappings Editor

- If your model already has a mapped AUTOSAR software component, in the model window, do one of the following:

  - From the **Apps** tab, open the AUTOSAR Component Designer app.
  - Click the perspective control in the lower-right corner and select **Code**.

  The model opens in the AUTOSAR code perspective, which includes the Code Mappings editor.
- If your model does not have a mapped AUTOSAR component, in the model window, do one of the following:

  - Use the AUTOSAR Component Quick Start.

    1. On the **Modeling** tab, select **Model Settings**. In the Configuration Parameters dialog box, **Code Generation** pane, set the system target file to either `autosar.tlc` or `autosar_adaptive.tlc`. Click **OK**.
    2. On the **Apps** tab, click **AUTOSAR Component Designer**. The AUTOSAR Component Quick Start opens.

3. Work through the component quick-start procedure and click **Finish**.

- For an Embedded Coder model, you can use the Embedded Coder Quick Start.

  1. With the Embedded Coder app open, on the **C Code** tab, select **Quick Start**. The Embedded Coder Quick Start opens.
  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 the 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 Entry-Point Functions to AUTOSAR Runnables”
- “Map Inports and Outports to AUTOSAR Sender-Receiver Ports and Data Elements”
- “Map Model Workspace Parameters to AUTOSAR Component Parameters”
- “Map Data Stores to AUTOSAR Variables”
- “Map Block Signals and States to AUTOSAR Variables”
- “Map Data Transfers to AUTOSAR Inter-Runnable Variables”
- “Map Function Callers to AUTOSAR Client-Server Ports and Operations”
- “Map Submodel Parameters to AUTOSAR Component Parameters”
- “Map Submodel Data Stores to AUTOSAR Variables”
- “Map Submodel Signals and States to AUTOSAR Variables”

If you are modeling for the AUTOSAR Adaptive Platform, navigate Code Mappings editor tabs to:

- “Map Inports and Outports to AUTOSAR Service Ports and Events”
- “Map Data Stores to AUTOSAR Persistent Memory Ports and Data Elements”

- “Map AUTOSAR Elements for Code Generation”
- “Map Calibration Data for Submodels Referenced from AUTOSAR Component Models”
- “Configure AUTOSAR Elements and Properties”
- “Map AUTOSAR Adaptive Elements for Code Generation”
- “Configure AUTOSAR Adaptive Elements and Properties”
- “Configure and Map AUTOSAR Component Programmatically”

See Also

Topics
“Map AUTOSAR Elements for Code Generation”
“Map Calibration Data for Submodels Referenced from AUTOSAR Component Models”

“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

AUTOSAR Blockset Checks

In this section...

"MathWorks Automotive Advisory Board Checks" on page 5-2
"Check model configuration parameters for AUTOSAR compliance" on page 5-2
"Check compatibility of AUTOSAR Interpolation Routines" on page 5-3

MathWorks Automotive Advisory Board Checks

Use AUTOSAR Blockset Model Advisor checks to configure your model for AUTOSAR standard compatibility.

See Also

- "Run Model Advisor Checks"
- "AUTOSAR Blockset Checks" on page 5-2
- "Embedded Coder Checks" (Embedded Coder)

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 set the parameters by using the Auto-Fix option.

Following are the model parameters the check examines, provided that **AUTOSAR Compliance** is set to `on` by using a proper license (TLC file).

Parameter	Recommended Values	Auto Fix	Condition Dependencies
AutoInsertRateTranBlk	off	off	STC = STIndependent && SolverMode = SingleTasking
AutosarCompliant	On	On	
AutosarMaxShortNameLength	range(32,128)	128	~isAdaptiveAutosar

Parameter	Recommended Values	Auto Fix	Condition Dependencies
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.

See Also

- **AUTOSAR Component Designer**

Check compatibility of AUTOSAR Interpolation Routines

Check ID: `mathworks.autosar.lut_replacement_check`

Description

Identifies Simulink lookup table blocks that are incompatible with AUTOSAR Blockset Interpolation Routines.

Available with AUTOSAR Blockset.

Results and Recommended Actions

Condition	Recommended Action
Model configuration parameter `CodeReplacementLibrary` is set to `None`.	Model configuration parameter `CodeReplacementLibrary` should not be set to `None`.

BlockType	Condition	Recommended Action
Prelookup	Parameter `ExtrapMethod` is set to `Clip`.	Consider using AUTOSAR Blockset Prelookup block for better compatibility.
n-D Lookup Table	Parameter `NumberOfTableDimensions` is set to `1` and parameter `ExtrapMethod` is set to `Clip`.	Consider using AUTOSAR Blockset Curve block for better compatibility.
	Parameter `NumberOfTableDimensions` is set to `2` and parameter `ExtrapMethod` is set to `Clip`.	Consider using AUTOSAR Blockset Map block for better compatibility.
Interpolation Using Prelookup	Parameter `NumberOfTableDimensions` is set to `1` and parameter `ExtrapMethod` is set to `Clip`.	Consider using AUTOSAR Blockset Curve Using Prelookup block for better compatibility.
	Parameter `NumberOfTableDimensions` is set to `2` and parameter `ExtrapMethod` is set to `Clip`.	Consider using AUTOSAR Blockset Map Using Prelookup block for better compatibility.

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.

See Also

- “Code Generation with AUTOSAR Code Replacement Library”
- “Configure Lookup Tables for AUTOSAR Calibration and Measurement”