

TargetLink

# Classic AUTOSAR Modeling Guide

For TargetLink 5.1

Release 2020-B – November 2020

**dSPACE**

## How to Contact dSPACE

Mail:	dSPACE GmbH Rathenaustraße 26 33102 Paderborn Germany
Tel.:	+49 5251 1638-0
Fax:	+49 5251 16198-0
E-mail:	<a href="mailto:info@dspace.de">info@dspace.de</a>
Web:	<a href="http://www.dspace.com">http://www.dspace.com</a>

## How to Contact dSPACE Support

If you encounter a problem when using dSPACE products, contact your local dSPACE representative:

- Local dSPACE companies and distributors: <http://www.dspace.com/go/locations>
- For countries not listed, contact dSPACE GmbH in Paderborn, Germany.  
Tel.: +49 5251 1638-941 or e-mail: [support@dspace.de](mailto:support@dspace.de)

You can also use the support request form: <http://www.dspace.com/go/supportrequest>. If you are logged on to mydSPACE, you are automatically identified and do not need to add your contact details manually.

If possible, always provide the relevant dSPACE License ID or the serial number of the CmContainer in your support request.

## Software Updates and Patches

dSPACE strongly recommends that you download and install the most recent patches for your current dSPACE installation. Visit <http://www.dspace.com/go/patches> for software updates and patches.

## Important Notice

This publication contains proprietary information that is protected by copyright. All rights are reserved. The publication may be printed for personal or internal use provided all the proprietary markings are retained on all printed copies. In all other cases, the publication must not be copied, photocopied, reproduced, translated, or reduced to any electronic medium or machine-readable form, in whole or in part, without the prior written consent of dSPACE GmbH.

© 2004 - 2020 by:  
dSPACE GmbH  
Rathenaustraße 26  
33102 Paderborn  
Germany

This publication and the contents hereof are subject to change without notice.

AUTERA, ConfigurationDesk, ControlDesk, MicroAutoBox, MicroLabBox, SCALEXIO, SIMPHERA, SYNECT, SystemDesk, TargetLink and VEOS are registered trademarks of dSPACE GmbH in the United States or other countries, or both. Other brand names or product names are trademarks or registered trademarks of their respective companies or organizations.

The ability of dSPACE TargetLink to generate C code from certain MATLAB code in Simulink®/Stateflow® models is provided subject to a license granted to dSPACE by The MathWorks, Inc. MATLAB, Simulink, and Stateflow are trademarks or registered trademarks of The MathWorks, Inc. in the United States of America or in other countries or both.

# Contents

About This Guide	11
------------------	----

Introduction to AUTOSAR	15
-------------------------	----

About the AUTOSAR Development Partnership.....	15
ECU Software Architecture (Classic AUTOSAR).....	16
Basics on Tools for Developing ECU Software as Described by Classic AUTOSAR.....	18

Introduction to Using Classic AUTOSAR in TargetLink	21
---	----

Overview of the TargetLink Classic AUTOSAR Block Library.....	22
Basic Functionality of the TargetLink Classic AUTOSAR Module.....	24
Development Approaches with the TargetLink Classic AUTOSAR Module.....	27
How to Start Modeling from Scratch.....	29

Defining Types, Scalings, and Constrained Range Limits (Classic AUTOSAR)	31
---	----

Introduction to Data Types in Classic AUTOSAR.....	32
Basics on Classic AUTOSAR Data Types in the Data Dictionary.....	32
Basics on Implementation and Application Data Types (Classic AUTOSAR).....	34
Basics on Working with Implementation and Application Data Types (Classic AUTOSAR).....	35
Basics on Working With Array of Structs When Modeling Classic AUTOSAR in TargetLink.....	37
Defining Scalings and Constrained Range Limits (Classic AUTOSAR).....	39
Basics on Scalings and Constrained Range Limits (Classic AUTOSAR).....	39
Creating Implementation Data Types from Scratch (Classic AUTOSAR).....	41
How to Define Scalar Implementation Data Types (Classic AUTOSAR).....	41
How to Define Array Implementation Data Types (Classic AUTOSAR).....	43
How to Define 2-D Array Implementation Data Types (Classic AUTOSAR).....	44
How to Define Structured Implementation Data Types (Classic AUTOSAR).....	46

Creating Application Data Types from Scratch (Classic AUTOSAR).....	49
How to Create Primitive Application Data Types (Classic AUTOSAR).....	49
How to Create Array Application Data Types (Classic AUTOSAR).....	51
How to Create 2-D Array Application Data Types Classic (AUTOSAR).....	53
How to Create Record Application Data Types (Classic AUTOSAR).....	54
Using TargetLink's Wizards to Create Application Data Types or Implementation Data Types (Classic AUTOSAR).....	57
How to Create Application Data Types for Existing Implementation Data Types (Classic AUTOSAR).....	57
How to Create Implementation Data Types for Existing Application Data Types (Classic AUTOSAR).....	59
<b>Modeling Software Components (SWCs)</b>	<b>63</b>
Basics on Software Components.....	63
How to Create Software Components.....	66
<b>Modeling Runnables</b>	<b>69</b>
Basics on Runnables.....	69
How to Model Runnables.....	71
Basics on RTE Events.....	74
How to Specify an Event for a Runnable.....	76
How to Execute a Runnable in an Exclusive Area.....	79
Basics on Activation Reasons.....	80
How To Model a Runnable's Activation Reasons.....	82
Basics on Initializing Runnable-Specific Variables.....	84
How to Initialize Runnable-Specific Variables in a Restart Runnable.....	86
<b>Modeling Communication According to Classic AUTOSAR</b>	<b>89</b>
Introduction to Communication According to Classic AUTOSAR.....	90
Overview of Communication According to Classic AUTOSAR.....	90
Modeling Classic AUTOSAR Communication via Data Stores.....	93
Basics on Modeling AUTOSAR Communication via Data Stores.....	93
Dynamically Accessing AUTOSAR Data Stores via Custom Code (Type II) Blocks.....	96
Creating Interfaces, Ports, and Communication Subjects.....	100
How To Create Interfaces.....	100

How to Create Ports.....	101
How to Create Communication Subjects for Classic AUTOSAR Communication.....	103
Modeling Sender-Receiver Communication.....	108
Basics on Sender-Receiver Communication.....	108
Example of Modeling Sender-Receiver Communication via Port Blocks.....	112
Example of Modeling Sender-Receiver Communication via Data Store Blocks.....	115
Basics on Communication Attributes and Acknowledgments.....	117
Basics on Signal Invalidation.....	119
How to Model Invalidation, Acknowledgment Notifications and Access to the RTE Status.....	122
How to Model Initialization of Data Elements.....	126
Basics on Checking the Update Flag of Data Elements in Explicit Sender-Receiver Communication.....	127
How to Model Checks of the Update Flag in Explicit Sender-Receiver Communication.....	130
Modeling Client-Server Communication.....	133
Introduction.....	133
Basics on Client-Server Communication.....	133
Modeling the Client Side of Client-Server Communication.....	135
Introduction to Modeling the Client Side of Client-Server Communication.....	136
Basics on Modeling Operation Calls on the Client Side of Client- Server Communication.....	136
Modeling Unidirectional Get or Set Operations (Synchronous Operation Calls via Port Blocks).....	140
Basics on Modeling Unidirectional Get or Set Operations.....	140
How to Model a Unidirectional Get or Set Operation in Synchronous Client-Server Communication via a Port Block.....	141
Modeling Synchronous Client-Server Communication.....	144
How to Model Operation Calls via Synchronous Operation Call Subsystems.....	144
Modeling Asynchronous Client-Server Communication.....	146
How to Model Operation Calls via Asynchronous Operation Call Subsystems.....	146
How to Model Operation Results via Operation Result Provider Subsystems.....	149

Modeling the Server Side of Client-Server Communication.....	151
Introduction to Modeling the Server Side of Client-Server Communication.....	152
Basics on Modeling the Server Side of Client-Server Communication.....	152
How to Model the Implementation of a Server Operation.....	153
Basics on Combining a Synchronous Operation Call with a Server Operation's Implementation.....	156
How to Combine a Synchronous Operation Call with a Server Operation's Implementation.....	157
Modeling Port-Defined Argument Values.....	160
Basics on Port-Defined Argument Values.....	161
How to Model Port-Defined Argument Values.....	162
Modeling Communication Specifications.....	164
How to Create Communication Specifications for Operations.....	164
Modeling Application Errors for Operations.....	166
How to Model Application Errors of Operations.....	166
Specifying Code Optimization Settings for Classic AUTOSAR Operations.....	168
How to Specify Code Optimization Settings for Classic AUTOSAR Operations.....	168
Modeling Interrunnable Communication.....	171
Basics on Interrunnable Communication.....	171
How to Create Interrunnable Variables.....	173
How to Model Interrunnable Communication.....	175
Modeling NvData Communication.....	178
NvData Communication.....	178
Basics on NvData Communication.....	179
Example of Modeling NvData Communication via Port Blocks.....	182
Example of Modeling NvData Communication via Data Store Memory Blocks.....	185
Example of Modeling NvData Communication via Block Parameters.....	188
Reducing Rewrites to NVRAM.....	190
Basics on Reducing Rewrites to NVRAM.....	190
How to Reduce Write Accesses to the NVRAM.....	192
Modeling Styles for Reducing Rewrites to NVRAM.....	194
Reducing Write Accesses to NVRAM when Accessing Elements of Vector or Matrix Variables.....	194
Reducing Write Accesses to NVRAM when Accessing Components of a Struct.....	197

Reducing Write Accesses to NVRAM when Accessing Elements of Non-Scalar Struct Components (Vector or Matrix Variables).....	199
Reducing Write Accesses to NVRAM by Modeling With Data Store Blocks.....	203
Modeling Transformer Error Logic.....	205
Basics on Data Transformation.....	205
Details on Data Transformation for Sender-Receiver Communication.....	208
Details on Data Transformation for Client-Server Communication.....	209
<b>Modeling Per Instance Memories</b>	<b>213</b>
Basics on Per Instance Memories.....	213
How to Model Per Instance Memories.....	214
<b>Modeling Modes</b>	<b>217</b>
Basics on Modes.....	217
How to Create Mode Declarations.....	219
How to Create Interfaces and Ports for Mode Communication.....	220
How to Model Mode Communication.....	222
How to Model Mode Disabling Dependencies.....	224
<b>Preparing SWCs for Multiple Instantiation</b>	<b>227</b>
Basics on Preparing SWCs for Multiple Instantiation.....	227
How to Prepare SWCs for Multiple Instantiation.....	231
<b>Preparing SWCs for Measurement and Calibration</b>	<b>235</b>
Basics on Preparing SWCs for Measurement and Calibration.....	235
How to Model Per Instance Parameters for Calibration.....	239
How to Model Shared Parameters for Calibration.....	241
How to Model Parameter Communication for Calibration.....	244
Basics on Static and Constant Memories.....	247
How to Model Static Memories for Measurement.....	250
How to Model Constant Memories for Calibration.....	252
Basics on Preparing Look-Up Tables for Measurement and Calibration (Classic AUTOSAR).....	254
Example of Preparing a Look-Up Table for Measurement and Calibration.....	260

<b>Using TargetLink's Modeling Features</b>	<b>269</b>
Basics on Modeling Classic AUTOSAR and non-AUTOSAR Controllers in one Model.....	269
Partitioning Model and Code for Classic AUTOSAR.....	271
<b>Generating Classic AUTOSAR-Compliant Code</b>	<b>275</b>
Introduction.....	276
Basics on Classic AUTOSAR Code Generation.....	276
Basics on Generated Code.....	276
How to Generate Classic AUTOSAR-Compliant Code via the TargetLink Main Dialog.....	278
Compiler Abstraction.....	280
Basics on Compiler Abstraction According to Classic AUTOSAR.....	280
Mapping Variables and Functions to Memory Sections.....	282
Basics on Memory Mapping in Classic AUTOSAR.....	282
Example of Adapting the AR_POSCONTROL Demo Model to use Memory Sections (Classic AUTOSAR).....	287
<b>Simulating SWCs</b>	<b>295</b>
Basics on Simulating Classic-AUTOSAR-Compliant SWCs.....	296
How to Simulate the RTE Status.....	299
How to Simulate Acknowledgment Notifications.....	301
How to Simulate Signal Invalidiation.....	303
How to Model and Simulate Transformer Error Logic in Sender- Receiver Communication.....	305
How to Simulate Operation Calls in Asynchronous Client-Server Communication.....	309
<b>Using a Software Architecture Tool Together with TargetLink</b>	<b>313</b>
Generating/Updating a Frame Model from Classic AUTOSAR Data.....	314
Basics on Generating/Updating a Frame Model from Classic AUTOSAR Data.....	314
How to Generate a Frame Model from Classic AUTOSAR Data.....	316
How to Update an Existing Frame Model from Classic AUTOSAR Data.....	318
Example of Generating a Frame Model from an AUTOSAR File.....	319

Mode Management with SystemDesk and TargetLink.....	322
Introduction to Mode Management.....	322
Basics on Mode Management.....	322
Mode Management in the AR_Fuelsys Demo.....	324
Modeling and Implementing Mode Management.....	327
Workflow for Modeling Mode Management.....	327
Modeling the Software Architecture (SystemDesk).....	329
Defining the SWC Internal Behaviors (SystemDesk).....	332
Exporting a Container for each Controller SWC (SystemDesk).....	334
Importing the Controller SWCs (TargetLink).....	336
Implementing the Controller SWCs (TargetLink).....	338
<b>Glossary</b>	<b>343</b>
<b>Appendix</b>	<b>373</b>
Comprehensive Modeling of RTE Code Pattern.....	374
Overview of RTE API Functions.....	374
Calibration and Measurement.....	375
Client-Server Communication.....	376
Interrunnable Communication.....	377
Mode management.....	380
NvData Communication (Explicit).....	380
NvData Communication (Implicit).....	381
Per Instance Memory.....	383
Sender-Receiver Communication (Explicit).....	383
Sender-Receiver Communication (Implicit).....	385
AUTOSAR-Related Code Generator Options.....	387
AllowDuplicationOfImplicitAUTOSARDataAccess.....	387
AssumeFunctionCallSemanticsForRteAPIArguments.....	388
AssumeOperationCallsHaveNoUnknownDataFlow.....	389
GenerateIWriteForNonScalarIrvs.....	390
ARDATAPrototypeActualParamName.....	391
StrictRunnableInterfaceChecks.....	392
SuppressNoRunnableRestartCodeError.....	392
<b>Index</b>	<b>395</b>



# About This Guide

## Contents

This guide explains how to model and generate code for software components that are compliant with [classic AUTOSAR](#) by using TargetLink.

### Note

A separate license is required for generating [classic AUTOSAR](#) software components (SWCs) containing function code for ECUs and for modeling and simulating them.

**Orientation and Overview Guide** For an introduction to the use cases and the TargetLink features that are related to them, refer to the  [TargetLink Orientation and Overview Guide](#).

## Required knowledge

This guide is most useful to function developers and software specialists. It is assumed that you know how to work with TargetLink. Knowledge in handling the host PC, the Microsoft Windows operating system and MATLAB is also assumed.

## Symbols

dSPACE user documentation uses the following symbols:

Symbol	Description
	Indicates a hazardous situation that, if not avoided, will result in death or serious injury.
	Indicates a hazardous situation that, if not avoided, could result in death or serious injury.
	Indicates a hazardous situation that, if not avoided, could result in minor or moderate injury.
	Indicates a hazard that, if not avoided, could result in property damage.
	Indicates important information that you should take into account to avoid malfunctions.
	Indicates tips that can make your work easier.
	Indicates a link that refers to a definition in the glossary, which you can find at the end of the document unless stated otherwise.
	Precedes the document title in a link that refers to another document.

## Naming conventions

dSPACE user documentation uses the following naming conventions:

**%name%** Names enclosed in percent signs refer to environment variables for file and path names.

**< >** Angle brackets contain wildcard characters or placeholders for variable file and path names, etc.

## Special folders

Some software products use the following special folders:

**Common Program Data folder** A standard folder for application-specific configuration data that is used by all users.

`%PROGRAMDATA%\dSPACE\<InstallationGUID>\<ProductName>`

or

`%PROGRAMDATA%\dSPACE\<ProductName>\<VersionNumber>`

**Documents folder** A standard folder for user-specific documents.

`%USERPROFILE%\Documents\dSPACE\<ProductName>\<VersionNumber>`

**Local Program Data folder** A standard folder for application-specific configuration data that is used by the current, non-roaming user.

`%USERPROFILE%\AppData\Local\dSPACE\<InstallationGUID>\<ProductName>`

## Accessing dSPACE Help and PDF Files

After you install and decrypt dSPACE software, the documentation for the installed products is available in dSPACE Help and as Adobe® PDF files.

**dSPACE Help (local)** You can open your local installation of dSPACE Help:

- On its home page via Windows Start Menu
- On specific content using context-sensitive help via **F1**

**dSPACE Help (Web)** You can access the Web version of dSPACE Help at [www.dspace.com](http://www.dspace.com).

To access the Web version, you must have a *mydSPACE* account.

**PDF files** You can access PDF files via the  icon in dSPACE Help. The PDF opens on the first page.



# Introduction to AUTOSAR

## Where to go from here

## Information in this section

About the AUTOSAR Development Partnership.....	15
ECU Software Architecture (Classic AUTOSAR).....	16
Basics on Tools for Developing ECU Software as Described by Classic AUTOSAR.....	18

## About the AUTOSAR Development Partnership

### AUTOSAR

AUTOSAR stands for *AUTomotive Open System ARchitecture*. AUTOSAR is a development partnership of original equipment manufacturers, suppliers, and other industrial partners.

### Goal of the AUTOSAR development partnership

The goal of the AUTOSAR development partnership is to meet the increasing complexity of automotive software by an open standard for automotive electric/electronic architectures. It also aims at making automotive software more reusable.

### ECU software architecture

With ↗ [Classic AUTOSAR](#), the AUTOSAR consortium has developed a software architecture for ECUs. For detailed information, refer to <https://www.autosar.org/standards/classic-platform/>.

The architecture defines the following partitioning of the ECU software.

- There are hardware-abstracted and therefore reusable parts.
- There are parts that must be configured for specific hardware, but can be generated automatically.

- There are hardware-dependent parts that have to be designed and implemented for specific hardware.

This partitioning is explained in more detail in [ECU Software Architecture \(Classic AUTOSAR\)](#) on page 16.

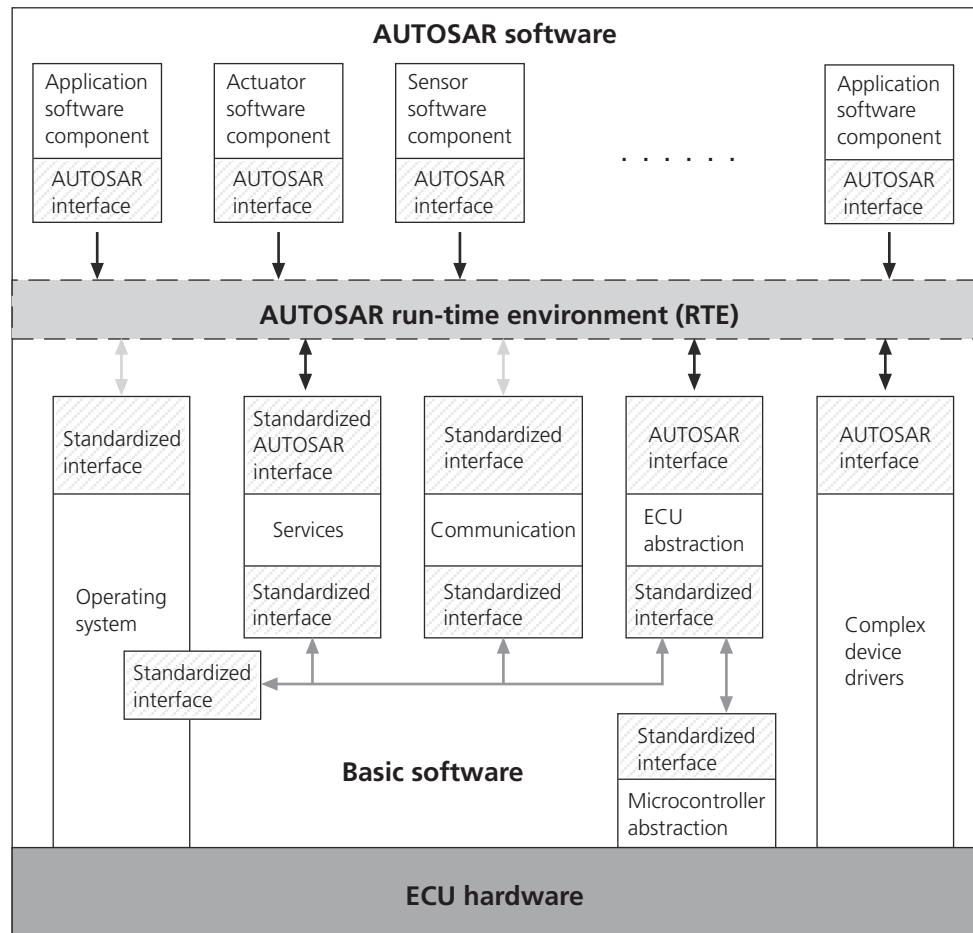
## ECU Software Architecture (Classic AUTOSAR)

### ECU software architecture

The AUTOSAR development partnership created [Classic AUTOSAR](#) to define a generic ECU software architecture that facilitates reusability.

### ECU overview

The illustration below is a schematic of one ECU and its software architecture according to [Classic AUTOSAR](#).



---

**ECU software overview**

The ECU software as described by [Classic AUTOSAR](#) is partitioned into the following three parts:

- The *Application layer* is the functional part of the ECU software, for example, the controller code. It consists of software components that are hardware-abstracted.
- The *AUTOSAR run-time environment (RTE)* is middleware software that allows ECU function development independently of the ECU hardware. In [Classic AUTOSAR](#), the RTE is ECU-specific. Each ECU has an RTE of its own.
- The *basic software* contains hardware-dependent parts of the software as well as the operating system, communication drivers, and [Classic AUTOSAR](#) services.

[Classic AUTOSAR](#) defines the interfaces of the basic software and all the software components of the AUTOSAR software, as can be seen in the illustration above. It also defines a methodology for the development process of an ECU executable. For details, refer to the *Technical Overview* document at <https://www.autosar.org/standards/classic-platform/>.

---

**Application layer**

Each SWC in the application layer encapsulates a functionality.

The SWCs are relocatable, i.e.:

- SWCs are hardware-abstracted and can thus be transferred from one ECU to another in a network consisting of several different ECUs.
- SWCs are independent of each other and can thus be transferred from one ECU to another without rearranging other SWCs.

TargetLink can be used to model SWCs that are compliant with [Classic AUTOSAR](#). An implementation of SWCs can be generated using TargetLink.

---

**Classic AUTOSAR RTE**

The RTE manages communication within the ECU software – in the application layer, and between the application layer and the basic software.

Handshaking between the communication participants is done via interfaces. The RTE references the interfaces to manage the information on which communication participants are connected to each other. The RTE delivers the communication messages.

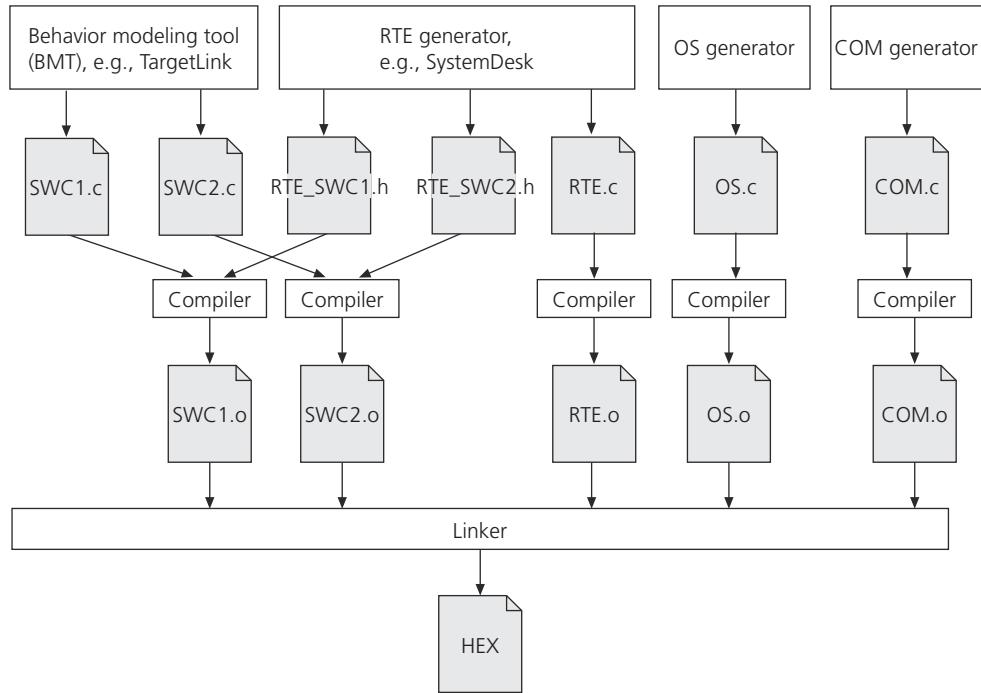
---

**Basic software**

The basic software comprises all hardware-specific ECU software. It consists of the operating system (OS), the communication layer (COM), and other service-oriented software. [Classic AUTOSAR](#) describes the interfaces that the basic software uses to provide services and access to data of sensors and actuators connected to the ECU.

**ECU executable generation**

The final step in developing ECU software is to generate an executable for the ECU. The illustration below is a schematic of ECU executable generation.



The code files which are generated by a behavior modeling tool like TargetLink, an RTE generator like dSPACE's SystemDesk, an OS generator, and a COM generator, are compiled against the corresponding header files. The resulting object files are linked to an executable.

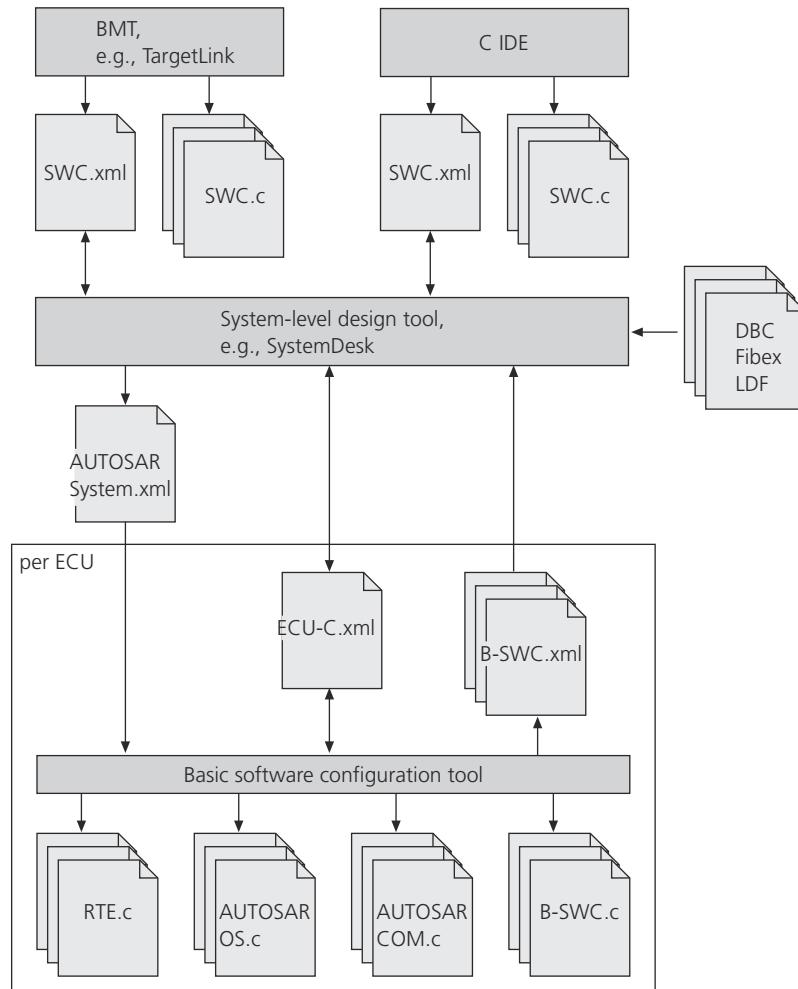
## Basics on Tools for Developing ECU Software as Described by Classic AUTOSAR

**Developing ECU software**

Developing ECU software according to [Classic AUTOSAR](#) makes ECU software reusable and reduces complexity, especially for ECU network development. The AUTOSAR development partnership has defined a software architecture consisting of application software components, run-time environment, and basic software for this purpose. The development partnership has also defined AUTOSAR file formats and AUTOSAR tools to standardize the ECU software development. For basic information on the ECU software architecture according to [Classic AUTOSAR](#), refer to [ECU Software Architecture \(Classic AUTOSAR\)](#) on page 16.

## Tools and dependencies in Classic-AUTOSAR-compliant ECU software development

The following illustration shows tools involved in developing ECU software that is compliant with [Classic AUTOSAR](#), AUTOSAR file formats, and their dependencies.



**Behavior modeling tools (BMTs) and C IDEs** The upper part of the illustration represents the development of reusable software components (SWCs) by using either behavior modeling tools (BMTs) such as Simulink or TargetLink, or manually written C code. Each SWC consists of a description file (SWC.xml) and corresponding C code (SWC.c).

dSPACE provides the *TargetLink AUTOSAR Module* for behavior modeling. It allows the import and export of SWC descriptions and generates C code in production quality, that is compliant with [Classic AUTOSAR](#).

**System-level design tools** The SWC descriptions are imported and exported by system-level design tools. The software architectures of complete systems or subsystems are designed in these tools. This involves tasks like assembling and connecting software components, specifying hardware topologies, handling

network communication, mapping software components onto ECUs, mapping data elements onto network signals, etc.

The result of system-level design is a description of the overall system in the form of an AUTOSAR system description file (AUTOSAR System.xml). A system-level design tool may also export ECU configuration files (ECU-C.xml) for each individual ECU. As a rule, this ECU configuration is not complete but contains only the parameters which can be derived from the system level.

dSPACE provides SystemDesk, which is a system-level design tool that covers all the above characteristics.

**Tip**

For detailed information on working with the dSPACE tool chain for [Classic AUTOSAR](#) and the interaction of SystemDesk and TargetLink, refer to the *AUTOSAR Architecture and Behavior Modeling* document, which you can find at [http://www.dspace.de/goto?sdlt\\_en](http://www.dspace.de/goto?sdlt_en).

**ECU-centric tools** Each individual ECU has to be configured, and finally the HEX file has to be generated. ECU-centric tools are used for this purpose. They read system descriptions and extract the aspects that are relevant to the ECU concerned. They also read and write the ECU configuration files (ECU-C.xml) containing configuration information for all the basic software components of one ECU, including the RTE configuration. After successful ECU configuration and validation, ECU-centric tools generate C code for basic software components and the RTE (AUTOSAR OS.c, AUTOSAR COM.c, B-SWC.c, RTE.c). In some cases, ECU-centric tools also export basic software module descriptions for basic software components (B-SWC.xml).

dSPACE's SystemDesk covers some features of ECU-centric tools such as OS, COM, and RTE configuration and RTE generation.

# Introduction to Using Classic AUTOSAR in TargetLink

## Where to go from here

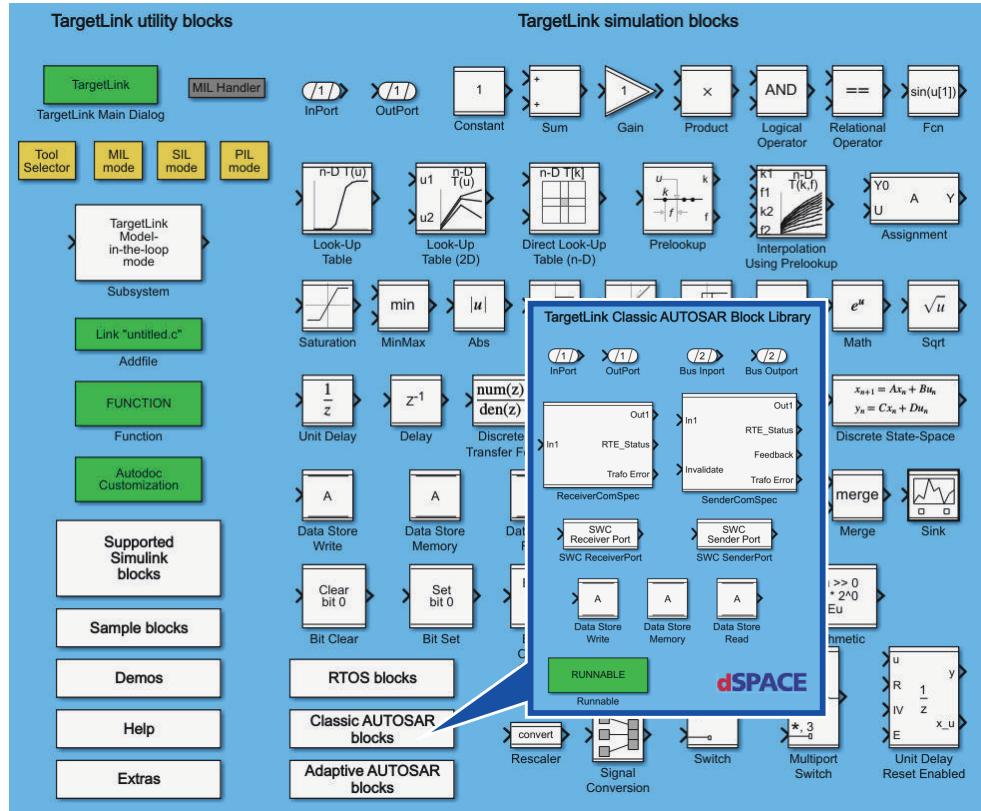
## Information in this section

Overview of the TargetLink Classic AUTOSAR Block Library.....	22
Basic Functionality of the TargetLink Classic AUTOSAR Module.....	24
Development Approaches with the TargetLink Classic AUTOSAR Module.....	27
How to Start Modeling from Scratch.....	29

## Overview of the TargetLink Classic AUTOSAR Block Library

### TargetLink Classic AUTOSAR Block Library

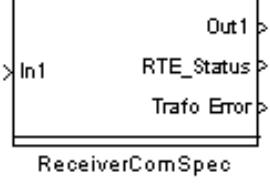
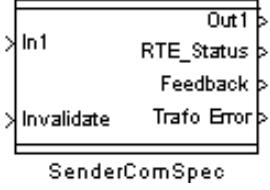
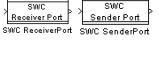
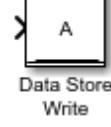
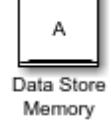
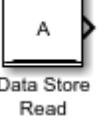
The illustration below shows the TargetLink Classic AUTOSAR Block Library, which is accessible via the MATLAB Command Window by the **tllib** command.



The TargetLink Classic AUTOSAR Block Library is accessible via the Classic AUTOSAR blocks button. The demos for the TargetLink Classic AUTOSAR Block Library are at the bottom of the Demos page, which is accessible via the Demos button.

**Blocks for modeling Classic AUTOSAR software components**

You can use the following blocks to model [Classic AUTOSAR](#) software components:

Block	Description
 or 	The Runnable block lets you model runnables as subsystems of the TargetLink subsystem in a Simulink model. Runnables are implemented by C functions.
 InPort Bus Import	The InPort/Bus Import blocks let you specify AUTOSAR communication mechanisms for signals entering a runnable subsystem. You can also specify logging settings to use TargetLink's data logging and analysis features.
 OutPort Bus Outport	The OutPort/Bus Outport blocks let you specify AUTOSAR communication mechanisms for signals leaving a runnable subsystem. You can also specify logging settings to use TargetLink's data logging and analysis features.
	The ReceiverComSpec block lets you generate code for accessing the RTE status of sender-receiver communication.
	The SenderComSpec block lets you generate code for invalidating data elements, handling acknowledgment notifications, and accessing the RTE status for sender-receiver communication.
	The SWC ReceiverPort/SWC SenderPort blocks let you visualize a software component's ports for sender-receiver communication in a model. These blocks are ignored during code generation.
 Data Store Write  Data Store Memory  Data Store Read	<p>You can use data store blocks in the following contexts:</p> <ul style="list-style-type: none"> <li>▪ Modeling NvData Communication</li> <li>▪ Modeling Interrunnable Communication</li> <li>▪ Modeling Sender-Receiver Communication</li> </ul>

**Demos for the TargetLink Classic AUTOSAR Module**

The TargetLink installation contains several Classic AUTOSAR demo models.

These show a variety of important aspects of modeling [Classic AUTOSAR](#) software components and are used as examples throughout this guide.

**Tip**

You can access the demos via the TargetLink block library or by typing `t1_demos` in MATLAB's Command Window.

---

**Related topics**

Basics

[AUTOSAR Software Components](#) ( [TargetLink Demo Models](#))

## Basic Functionality of the TargetLink Classic AUTOSAR Module

---

**Introduction**

You can use the TargetLink [Classic AUTOSAR](#) Module to develop and generate code for [Classic AUTOSAR](#) software components.

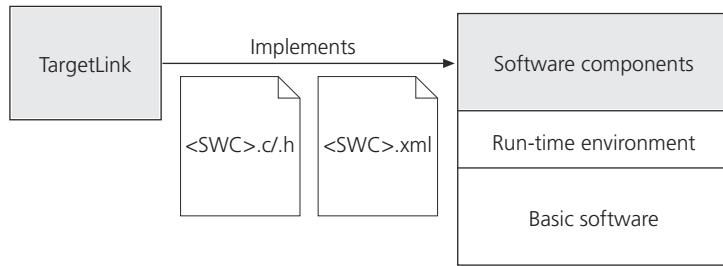
---

**Supported Classic AUTOSAR releases**

Classic AUTOSAR Release	Revision
R19-11	19-11
4.4	4.4.0
4.3	4.3.1 4.3.0
4.2	4.2.2 4.2.1
4.1	4.1.3 4.1.2 4.1.1
4.0	4.0.3 4.0.2

**Implementing software components**

You can implement software components with TargetLink as shown in the illustration below.



You can generate C code files and a description of the software component. The description is a standardized XML file for software component data exchange between tools that are compliant with [Classic AUTOSAR](#).

**Modeling according to Classic AUTOSAR**

**Modeling software components and runnables** TargetLink lets you model atomic software components and their internal behaviors including runnables as described by [Classic AUTOSAR](#). You can define all the required AUTOSAR elements such as types, constants, compu methods (scalings), RTE events, per instance memories, and exclusive areas in the Data Dictionary.

**Modeling AUTOSAR communication** You can model communication as described by [Classic AUTOSAR](#) between software components or runnables, i.e., via ports and interfaces or interrunnable variables.

**Preparing software components for measurement and calibration** TargetLink lets you prepare [Classic AUTOSAR](#) software components for measurement and calibration to optimize their performance. You can provide calibration information for block parameters used in one or more software components. TargetLink also lets you prepare interrunnable variables for measurement.

**Modeling Classic-AUTOSAR and non-AUTOSAR controllers within the same model** Dual block data at Function, InPort, and OutPort blocks allows you to use the same model for [Classic AUTOSAR](#) and non-AUTOSAR controller development, and easily migrate models to the [Classic AUTOSAR](#) use case.

**Modeling modes** TargetLink lets you model requests for mode switches and getting the currently active modes via ports and mode switch interfaces.

**Generating production code**

You can generate production code that is compliant with [Classic AUTOSAR](#) with RTE API calls for software components. This allows you to implement software components to an ECU.

TargetLink allows you to export production code compliant with [Classic AUTOSAR](#)

- As source code
  - As object code in compatibility mode
- 

## Exchanging software components

**Importing and exporting software component descriptions** You can import software component descriptions, i.e., AUTOSAR files to the Data Dictionary. After code generation the Data Dictionary lets you export the description of a software component so that it can be exchanged with architecture tools.

**Exchanging software components using containers** A container is a bundle of files that is related to a software component. A container can contain files such as code files (.c/.h), AUTOSAR files (.arxml), and variable description files (.a21).

Exchanging data with dSPACE's architecture tool SystemDesk using software component containers has the following benefits:

- Exchanging data safely, easily, and with a defined workflow.
  - Integrating software component into SystemDesk's simulation feature.
  - Managing additional related files such as feature specifications, test specifications, etc.
- 

## Generating a frame model from AUTOSAR data

You can generate a frame model based on [Classic AUTOSAR](#) data in AUTOSAR files or in the Data Dictionary. The result of the generation is a model that contains subsystems for software components and runnables with TargetLink blocks that reference the AUTOSAR data. To generate code, you have to insert the control algorithm in the frame model.

## Simulating software components

**Simulating in MIL/SIL/PIL simulation modes** You can use the TargetLink MIL, SIL, and PIL simulation modes for simulating [Classic AUTOSAR](#) software components. You can use TargetLink's data logging and analysis features for developing [Classic AUTOSAR](#) software components.

**Simulating the impact of the RTE on software components** [Classic AUTOSAR](#) communication and runnable triggering are impacts of the RTE on software components. TargetLink does not generate a fully Classic-AUTOSAR-compliant RTE, but you can use TargetLink's stub RTE to simulate RTE features of [Classic AUTOSAR](#) communication such as data invalidation, acknowledgment notifications, and RTE status signals. You can also use Simulink/Stateflow in conjunction with triggered subsystems for simulating runnable triggering by RTE events such as timing events.

## Development Approaches with the TargetLink Classic AUTOSAR Module

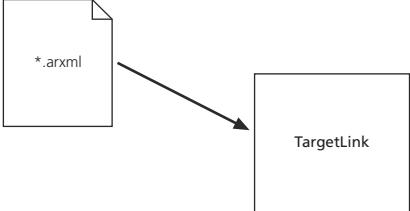
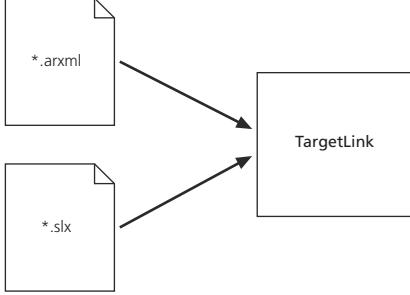
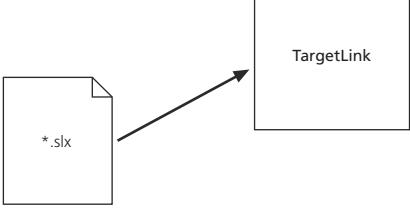
---

### Different modeling approaches

You can use different development approaches for a [Classic AUTOSAR](#) software component because the software component's behavior, i.e., the controller algorithm or the software component's description, can already be available.

## Development approaches for software components

The table below summarizes typical development approaches for [Classic AUTOSAR](#) software components.

Development Approach	Modeling Process with TargetLink
<p>Developing top-down</p> 	<ol style="list-style-type: none"> <li>1. Generate a frame model from your <a href="#">Classic AUTOSAR</a> file to start modeling with a TargetLink model according to the file. For instructions, refer to <a href="#">How to Generate a Frame Model from Classic AUTOSAR Data</a> on page 316.</li> <li>2. Model the functional behavior of your model with TargetLink blocks.</li> <li>3. Generate code for your model. For instructions, refer to <a href="#">How to Generate Classic AUTOSAR-Compliant Code via the TargetLink Main Dialog</a> on page 278.</li> <li>4. Export the <a href="#">Classic AUTOSAR</a> file of your model. For instructions, refer to <a href="#">Basics on Exporting AUTOSAR Files</a> (<a href="#">TargetLink Interoperation and Exchange Guide</a>).</li> </ol>
<p>Reusing non-AUTOSAR model</p> 	<ol style="list-style-type: none"> <li>1. Generate a frame model from your <a href="#">Classic AUTOSAR</a> file to start modeling with a TargetLink model according to the file. For instructions, refer to <a href="#">How to Generate a Frame Model from Classic AUTOSAR Data</a> on page 316.</li> <li>2. Insert the functional behavior of your controller model in the new frame model.</li> <li>3. Generate code for your model. For instructions, refer to <a href="#">How to Generate Classic AUTOSAR-Compliant Code via the TargetLink Main Dialog</a> on page 278.</li> <li>4. Export the <a href="#">Classic AUTOSAR</a> file of your model. For instructions, refer to <a href="#">Basics on Exporting AUTOSAR Files</a> (<a href="#">TargetLink Interoperation and Exchange Guide</a>).</li> </ol>
<p>Migrating non-AUTOSAR model</p> 	<p>You can use the <i>TargetLink AUTOSAR Migration Tool</i> for converting non-AUTOSAR TargetLink models to <a href="#">Classic AUTOSAR</a> at the click of a button. For further information, refer to <a href="http://www.dspace.de/goto?tl_ar_migration">http://www.dspace.de/goto?tl_ar_migration</a>.</p> <p>To convert a model by hand, follow the instructions below:</p> <ol style="list-style-type: none"> <li>1. Merge your model's DD project file with the dsdd_master_autosar4.dd [System] system template in order to use standardized <a href="#">Classic AUTOSAR</a> types in your model. For instructions, refer to <a href="#">How to Merge or Replace DD Objects</a> (<a href="#">TargetLink Data Dictionary Basic Concepts Guide</a>).</li> <li>2. Make your model compliant with <a href="#">Classic AUTOSAR</a>. You have to execute the following process steps: <ul style="list-style-type: none"> <li>▪ Specify the software components of your model and structure your model according to them. Refer to <a href="#">Modeling Software Components (SWCs)</a> on page 63.</li> <li>▪ Specify the runnables of your model and structure your model according to them. Refer to <a href="#">Modeling Runnables</a> on page 69.</li> <li>▪ Model SWCs' and runnables' ports and interfaces. Refer to <a href="#">Modeling Communication According to Classic AUTOSAR</a> on page 89.</li> </ul> </li> <li>3. Generate code for your model. For instructions, refer to <a href="#">How to Generate Classic AUTOSAR-Compliant Code via the TargetLink Main Dialog</a> on page 278.</li> <li>4. Export the <a href="#">Classic AUTOSAR</a> file of your model. For instructions, refer to <a href="#">Basics on Exporting AUTOSAR Files</a> (<a href="#">TargetLink Interoperation and Exchange Guide</a>).</li> </ol>
<p>Developing from scratch</p> 	<ol style="list-style-type: none"> <li>1. Create a new TargetLink model. For instructions, refer to <a href="#">How to Start Modeling from Scratch</a> on page 29.</li> <li>2. Model the functional behavior using TargetLink blocks.</li> <li>3. Generate code for your model. For instructions, refer to <a href="#">How to Generate Classic AUTOSAR-Compliant Code via the TargetLink Main Dialog</a> on page 278.</li> <li>4. Export the <a href="#">Classic AUTOSAR</a> file of your model. For instructions, refer to <a href="#">Basics on Exporting AUTOSAR Files</a> (<a href="#">TargetLink Interoperation and Exchange Guide</a>).</li> </ol>

# How to Start Modeling from Scratch

**Objective** To start modeling according to [Classic AUTOSAR](#) from scratch by creating a TargetLink model and a proper DD Project file.

**Preconditions** The following preconditions must be fulfilled:

- You have installed TargetLink including the TargetLink AUTOSAR Module.
- You created a TargetLink model as described in [How to Create a Model from Scratch](#) ([TargetLink Orientation and Overview Guide](#)).

**Method**

**To start modeling from scratch**

- 1 From the model's TargetLink menu, choose Data Dictionary Manager.  
The TargetLink Data Dictionary Manager opens.
- 2 From the File menu of the TargetLink Data Dictionary Manager, choose New  
- Use Current DD Workspace and create a DD project file based on the dsdd\_master\_autosar4.dd [System] system template.
- 3 From the File menu of the Data Dictionary Manager, choose Save As ... and save the DD project file.
- 4 Open the TargetLink Main Dialog and specify the following settings:

Property	Value
<b>CodeGenerationMode (Model element)</b> ( <a href="#">TargetLink Model Element Reference</a> ) (on the Code Generation page of the TargetLink Main Dialog Block)	Classic AUTOSAR
Associate model with fixed DD project file (on the Options page of the TargetLink Main Dialog Block)	Specify the DD project file created and saved above.

**Result** You have created a TargetLink subsystem together with a DD project file for modeling according to [Classic AUTOSAR](#).

**Next steps**

- Connect imports of the TargetLink subsystem with a source such as a Pulse Generator block.
- Connect outports of the TargetLink subsystem with a sink such as a Terminator block.

- Create a DD Module object for defining code file names of [Classic AUTOSAR](#) software components. For instructions, refer to [How to Create Module Specifications in the Data Dictionary](#) ( [TargetLink Customization and Optimization Guide](#)).
- Create a DD Module object for defining code file names of software components.

---

**Related topics**

HowTos

[How to Create Module Specifications in the Data Dictionary](#) ( [TargetLink Customization and Optimization Guide](#))

References

[TargetLink Main Dialog Block](#) ( [TargetLink Model Element Reference](#))

# Defining Types, Scalings, and Constrained Range Limits (Classic AUTOSAR)

## Where to go from here

## Information in this section

Introduction to Data Types in Classic AUTOSAR.....	32
Defining Scalings and Constrained Range Limits (Classic AUTOSAR).....	39
Creating Implementation Data Types from Scratch (Classic AUTOSAR).....	41
Creating Application Data Types from Scratch (Classic AUTOSAR).....	49
Using TargetLink's Wizards to Create Application Data Types or Implementation Data Types (Classic AUTOSAR).....	57

# Introduction to Data Types in Classic AUTOSAR

## Where to go from here

## Information in this section

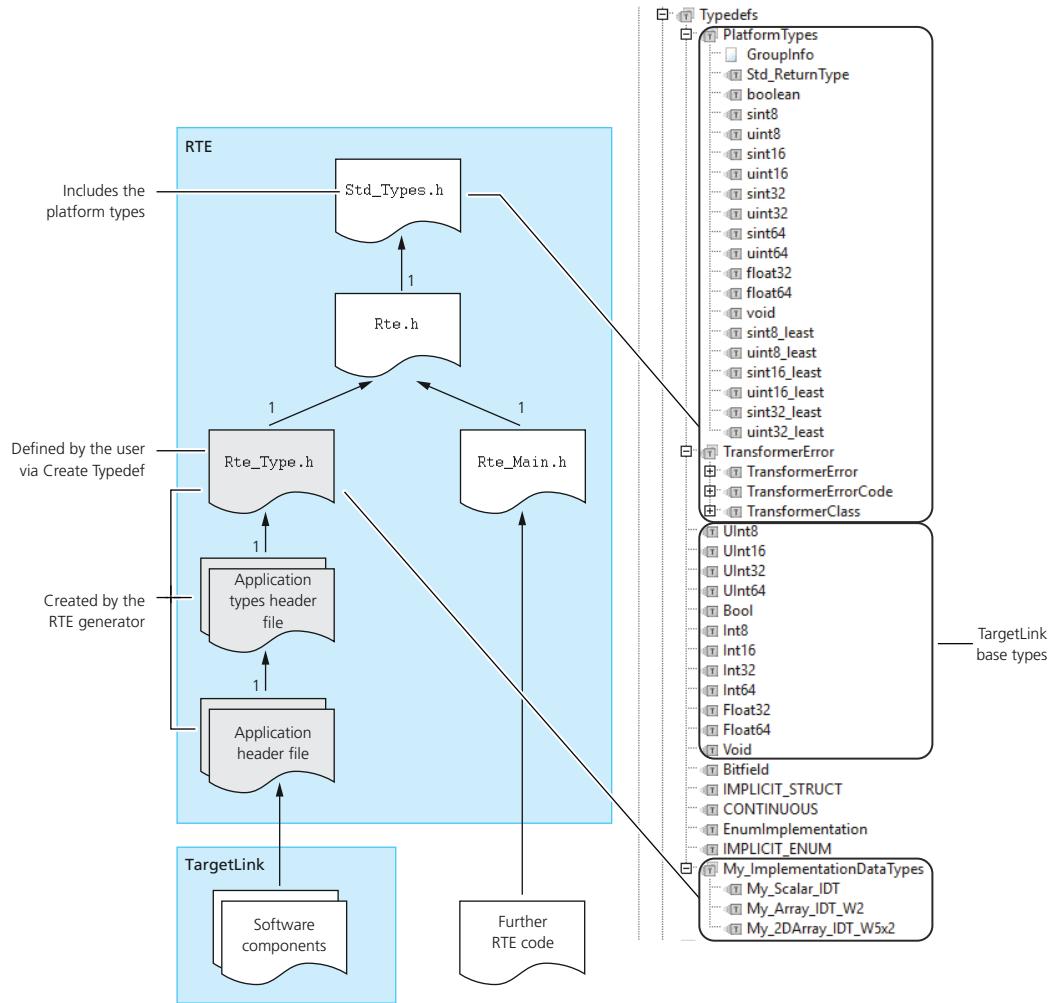
Basics on Classic AUTOSAR Data Types in the Data Dictionary.....	32
Basics on Implementation and Application Data Types (Classic AUTOSAR).....	34
Basics on Working with Implementation and Application Data Types (Classic AUTOSAR).....	35
Basics on Working With Array of Structs When Modeling Classic AUTOSAR in TargetLink.....	37

## Basics on Classic AUTOSAR Data Types in the Data Dictionary

### Classic AUTOSAR types in TargetLink

Classic AUTOSAR has defined modules for type definitions of the RTE. The RTE code generator has to define types at standardized locations.

The following illustration shows type definition files according to Classic AUTOSAR, and shows you where to find or define the corresponding objects in DD project files that are based on the dsdd\_master\_autosar4.dd [System] system template:



The left side of the illustration shows modules with type definitions that are standardized or generated by an RTE generator. The right side shows where the standardized types are located in the Data Dictionary.

#### Note

Do not change the predefined DD **TypedefGroup** and DD **Typedef** objects. Instead, create your own application and implementation data types. Refer to [Basics on Implementation and Application Data Types \(Classic AUTOSAR\)](#) on page 34 and [Basics on Working with Implementation and Application Data Types \(Classic AUTOSAR\)](#) on page 35.

**Related topics****Basics**

Basics on Implementation and Application Data Types (Classic AUTOSAR).....	34
Basics on Working with Implementation and Application Data Types (Classic AUTOSAR).....	35
Creating Application Data Types from Scratch (Classic AUTOSAR).....	49
Creating Implementation Data Types from Scratch (Classic AUTOSAR).....	41
Defining Scalings and Constrained Range Limits (Classic AUTOSAR).....	39
Using TargetLink's Wizards to Create Application Data Types or Implementation Data Types (Classic AUTOSAR).....	57

## Basics on Implementation and Application Data Types (Classic AUTOSAR)

**Introduction**

With [Classic AUTOSAR](#) Release 4.x, *implementation* and *application* data types are introduced to allow for a more abstract definition of software components.

**Application data types**

**According to Classic AUTOSAR** Application data types are used for defining types on the application level of abstraction. From the application point of view, physical data and its numerical representation are of concern. Accordingly, application data types have physical semantics and do not consider implementation details such as bit width or endianness.

Application data types can be constrained to change the resolution of the physical data's representation or define a range that is to be considered.

**Application data types in TargetLink**

- Are represented by DD ApplicationDataType objects in the /Pool/Autosar/ApplicationDataTypes/ subtree
- Are not considered during code generation
- Can be created, specified, imported, and exported
- Can be referenced at [data prototypes](#)

**Implementation data types**

**According to Classic AUTOSAR** Implementation data types are used for defining types on the implementation level of abstraction. From the implementation point of view, the storage and manipulation of digitally represented data is of concern. Accordingly, implementation data types have data semantics and do consider implementation details, such as data type.

Implementation data types can be constrained to change the resolution of the digital representation or define a range that is to be considered. Typically, they correspond to typedef statements in C code and still abstract from platform specific details like endianness.

### Implementation data types in TargetLink

- Are represented by DD Typedef objects that are defined by the user
- Are considered during code generation
- Are imported to (exported from) DD Typedef objects for [Classic AUTOSAR](#)

#### Related topics

##### Basics

<a href="#">Basics on Working with Implementation and Application Data Types (Classic AUTOSAR)</a> .....	35
--	----

##### HowTos

<a href="#">How to Create Application Data Types for Existing Implementation Data Types (Classic AUTOSAR)</a> .....	57
<a href="#">How to Create Implementation Data Types for Existing Application Data Types (Classic AUTOSAR)</a> .....	59

## Basics on Working with Implementation and Application Data Types (Classic AUTOSAR)

#### Introduction

To work with [implementation data types \(IDTs\)](#) and [application data types \(ADTs\)](#) as introduced with [Classic AUTOSAR](#) Release 4.x.

#### Typical use cases

When modeling with TargetLink according to [Classic AUTOSAR](#), three typical use cases exist:

- You want to add ADTs to a data dictionary *already containing* IDTs (typedefs).
- You want to add IDTs (typedefs) to a data dictionary *already containing* ADTs.
- You want to add ADTs to a data dictionary *not yet containing* IDTs (typedefs).

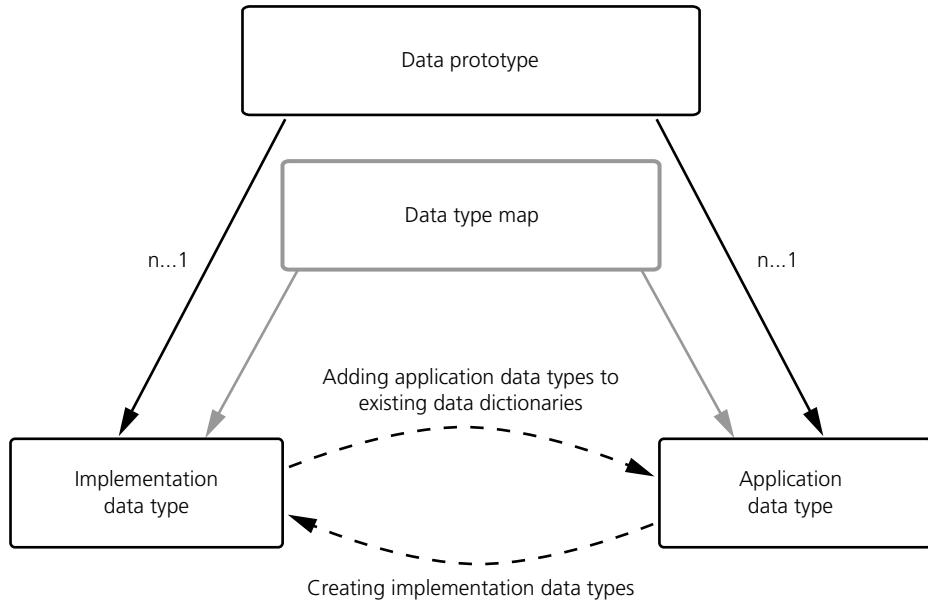
You have to understand the interplay between ADTs and IDTs in order to model correctly.

#### Interplay between ADTs and IDTs

According to [Classic AUTOSAR](#), ADTs are referenced at [data prototypes](#) and are mapped to IDTs via data type maps.

Working according to this framework allows you to streamline the number of different data types.

The figure below shows the context of the typical use cases and the interplay of ADTs and IDTs at [data prototypes](#) and data type maps:



As you can see, the relation between [data prototypes](#) and ADTs/IDTs is *n:1*. Each of the *n* [data prototypes](#) references exactly one ADT and one IDT.

The ADT and IDT are mapped to each other in a data type map, i.e., they stand in an *n:1* relationship.

## Data type maps

**According to Classic AUTOSAR** [Classic AUTOSAR](#) has defined data type maps that define a mapping of implementation and application data types. This supports abstraction of software design from implementation and is required to generate RTE code.

Data type maps are part of a data type mapping set that summarizes all the type mappings of a software component.

### Data type maps in TargetLink

- Are represented by a DD `DataTypeMaps` object.
- Contain DD `DataTypeMap` objects that define one mapping of an implementation data type (represented in TargetLink by a DD `TypeDef` object) and an application data type.
- Are part of a data type mapping set that is represented by a DD `DataTypeMappingSet` object  
(/Pool/Autosar/DataTypeMappingSets/<DataTypeMappingSet> DD path).

## Instructions for different use cases

Use Case	Instructions
You want to add ADTs to a data dictionary <i>already containing</i> IDTs (typedefs).	TargetLink assists you via its ApplicationDataType Creation Wizard. Refer to <a href="#">How to Create Application Data Types for Existing Implementation Data Types (Classic AUTOSAR)</a> on page 57.
You want to add IDTs (typedefs) to a data dictionary <i>already containing</i> ADTs.	TargetLink assists you via its ImplementationDataType Creation Wizard. Refer to <a href="#">How to Create Implementation Data Types for Existing Application Data Types (Classic AUTOSAR)</a> on page 59.
You want to specify ADTs from scratch.	<a href="#">Creating Application Data Types from Scratch (Classic AUTOSAR)</a> on page 49
You want to specify IDTs from scratch.	<a href="#">Creating Implementation Data Types from Scratch (Classic AUTOSAR)</a> on page 41

## Related topics

### Basics

[Basics on Implementation and Application Data Types \(Classic AUTOSAR\)](#) ..... 34

### HowTos

<a href="#">How to Create Application Data Types for Existing Implementation Data Types (Classic AUTOSAR)</a> .....	57
<a href="#">How to Create Implementation Data Types for Existing Application Data Types (Classic AUTOSAR)</a> .....	59

## Basics on Working With Array of Structs When Modeling Classic AUTOSAR in TargetLink

### Introduction

When modeling for [Classic AUTOSAR](#) you can access [array-of-struct variables](#) at the interface of [Runnable Subsystems](#).

### Modeling access

You model the access to array-of-struct variables at the interface of runnable subsystems via special modeling patterns that are based on Custom Code (type II) and data store blocks. Refer to [Dynamically Accessing AUTOSAR Data Stores via Custom Code \(Type II\) Blocks](#) on page 96.

**Demo model**

The access to array-of-struct variables at the interface of runnable subsystems is shown in the [AR\\_ARRAY\\_OF\\_STRUCT\\_DATA](#) ( [TargetLink Demo Models](#)) demo model.

---

**Related topics**

Basics

<a href="#">AR_ARRAY_OF_STRUCT_DATA</a> ( <a href="#">TargetLink Demo Models</a> )
<a href="#">Dynamically Accessing AUTOSAR Data Stores via Custom Code (Type II) Blocks</a> .....96
<a href="#">Working with Array-of-Bus Signals</a> ( <a href="#">TargetLink Preparation and Simulation Guide</a> )

# Defining Scalings and Constrained Range Limits (Classic AUTOSAR)

## Basics on Scalings and Constrained Range Limits (Classic AUTOSAR)

### Introduction

To specify [scalings](#) and [constrained range limits](#) as defined by [Classic AUTOSAR](#).

### Specifying scalings compliant with Classic AUTOSAR

In [Classic AUTOSAR](#), scalings are defined in **CompuMethod** elements. The following table shows where to specify scalings when modeling for [Classic AUTOSAR](#) in TargetLink:

Classic AUTOSAR	TargetLink
Computation method of application data type	Reference a DD Scaling object at the DD ScalingRef property of the DD ApplicationDataType object.

TargetLink allows the redundant specification of *the same scaling* at both application data type and implementation data type (typedef). This lets you use the same data types at [Classic AUTOSAR](#) interfaces and at TargetLink blocks that do not represent [Classic AUTOSAR](#) interfaces. No CompuMethod element is created for the implementation data type during AUTOSAR export.

### Specifying constrained range limits compliant with Classic AUTOSAR

In [Classic AUTOSAR](#), constrained range limits are specified in **DataConstraint** elements. The following table shows where to specify constrained range limits when modeling for [Classic AUTOSAR](#) in TargetLink:

Classic AUTOSAR	TargetLink
<p>Data constraints of base types, implementation data types, application data types, and <a href="#">data prototypes</a>, as shown in the following illustration:</p> <p>However, according to AUTOSAR, you typically specify constraints at the application data type level. For details, refer to the <i>SoftwareComponentTemplate</i> document at <a href="https://www.autosar.org/standards/classic-platform/">https://www.autosar.org/standards/classic-platform/</a>.</p>	<p>Specify constrained range limits via the Min and Max properties of the DD ApplicationDataType object.</p> <p>Execute the Synchronize Application Data Type Settings command. Repeat synchronization after every change you make at an application data type or its associated implementation data types or <a href="#">data prototypes</a>.<sup>1)</sup></p> <p><b>Note</b></p> <p>Do not specify constrained limits at <a href="#">data prototypes</a>.</p>

<sup>1)</sup> This command does not overwrite constrained range limits that are already specified at [data prototypes](#).

**Related topics**

**Basics**

Creating Application Data Types from Scratch (Classic AUTOSAR).....	49
Creating Implementation Data Types from Scratch (Classic AUTOSAR).....	41
Introduction to Data Types in Classic AUTOSAR.....	32
Using TargetLink's Wizards to Create Application Data Types or Implementation Data Types (Classic AUTOSAR).....	57

**References**

Synchronize Application Data Type Settings ( TargetLink Data Dictionary  
Manager Reference)

# Creating Implementation Data Types from Scratch (Classic AUTOSAR)

## Where to go from here

## Information in this section

How to Define Scalar Implementation Data Types (Classic AUTOSAR).....	41
How to Define Array Implementation Data Types (Classic AUTOSAR).....	43
How to Define 2-D Array Implementation Data Types (Classic AUTOSAR).....	44
How to Define Structured Implementation Data Types (Classic AUTOSAR).....	46

## How to Define Scalar Implementation Data Types (Classic AUTOSAR)

### Introduction

To create scalar [implementation data types \(IDTs\)](#) via DD Typedef objects.

### Preconditions

DD project files that are used for modeling according to [Classic AUTOSAR](#) must be based on the `dsdd_master_autosar4.dd` [System] DD system template.

### Method

#### To define a scalar implementation data type

##### Note

In TargetLink, implementation data types are represented as DD Typedef objects.

- 1 Create a DD `TypedefGroup` object in `/Pool/Typedef` and name it as required, e.g., `ImplementationDataTypes`.
- 2 Add the `GroupInfo` subtree to the `TypedefGroup` and specify a package, such as `SharedElements/ImplementationDataTypes`, via its `Package` property. This lets you export the implementation data types in the group to the specified package. Also, implementation data types in AUTOSAR files located in this package can be imported to that location.

- 3 Add a DD Typedef object to the TypedefGroup and name it as required, e.g. `My_Scalar_IDT`.
- 4 In the Typedef object's Property Value List, specify the following settings:

Property	Value
BaseType	Select a base type from the list.
CreateTypedef	on
Is BaseType	off
InvalidValue	Provide an invalid value if you want to model sender-receiver communication with invalidation. Mutually exclusive with InvalidValueToken.
InvalidValueToken <sup>1)</sup>	Relevant only, if you want to model sender-receiver communication with invalidation. Provide a conversion string to be substituted by the corresponding element of a conversion table. Mutually exclusive with InvalidValue.

<sup>1)</sup> Restricted to plain or Boolean data types whose scaling object's ConversionType property is set to TAB\_VERB.

**Result**

You created a scalar [implementation data type \(IDT\)](#) via a DD Typedef object.

TargetLink generates local variable definitions with the created type to the software component code to process the elements:

```
/* SLLocal: Default storage class for local variables | Width: 16 */
My_Scalar_IDT My_Scalar_DE;
```

**Next steps**

After creating IDTs, use TargetLink's ApplicationDataType Creation Wizard. For details, refer to [How to Create Application Data Types for Existing Implementation Data Types \(Classic AUTOSAR\)](#) on page 57.

**Related topics****Basics**

Basics on Communication Attributes and Acknowledgments.....	117
Basics on Exporting AUTOSAR Files ( <a href="#">TargetLink Interoperation and Exchange Guide</a> )	
Basics on Importing AUTOSAR Files ( <a href="#">TargetLink Interoperation and Exchange Guide</a> )	
Basics on Packages ( <a href="#">TargetLink Interoperation and Exchange Guide</a> )	
Basics on Signal Invalidation.....	119

## How to Define Array Implementation Data Types (Classic AUTOSAR)

<b>Objective</b>	To define array implementation data types (IDTs) via DD Typedef objects.
------------------	--

<b>Preconditions</b>	<ul style="list-style-type: none"> <li>DD project files that are used for modeling according to <a href="#">Classic AUTOSAR</a> must be based on the dsdd_master_autosar4.dd [System] DD system template.</li> <li>You created a scalar implementation data type as described in <a href="#">How to Define Scalar Implementation Data Types (Classic AUTOSAR)</a> on page 41.</li> </ul>
----------------------	--

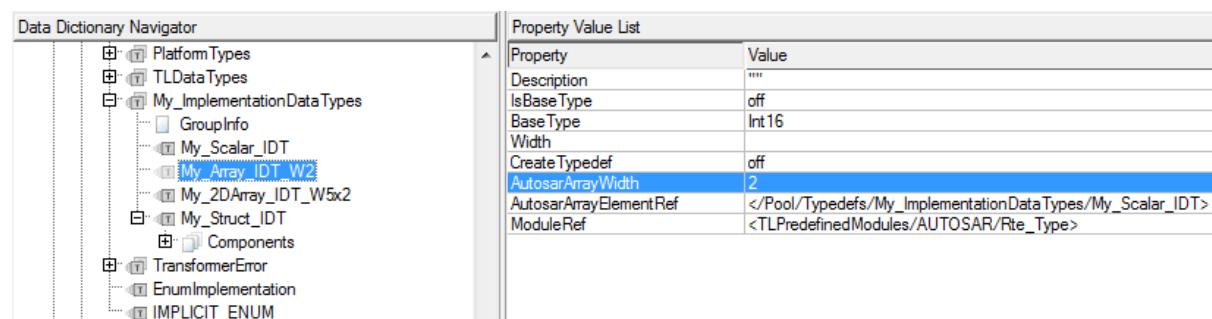
<b>Method</b>	<b>To define an array implementation data type</b>
---------------	--

### Note

In TargetLink, implementation data types are represented as DD Typedef objects.

- 1 Open the context menu of the DD TypedefGroup that contains your implementation data types.
- 2 From the AUTOSAR submenu, choose Create AUTOSAR Array Type and rename the resulting implementation data type as required, e.g., `My_Array_IDT_W2`.
- 3 In its Property Value List, specify the following settings:

Property	Value
BaseType	Select a base type from the list.
CreateTypedef	<code>off</code>
IsBaseType	<code>off</code>
AutosarArrayWidth	Specify the width of the array.
AutosarArrayElementRef	Select a scalar implementation data type (typedef) for the array elements that is compatible with the selected base type.
InvalidValue	Provide an invalid value if you want to model sender-receiver communication with invalidation.



**Note**

You have to specify a consistent width for interface elements and internal behavior elements that are of the defined array type. If you want to create a data element of the MyArray type, you have to specify 2 for the DataElement object's Width property.

**Result**

You created an array [implementation data type \(IDT\)](#) via a DD Typedef object.

TargetLink generates local variable definitions *not* with the created type but with its underlying base type:

```
/* SLLocal: Default storage class for Local variables | Width: 16 */
sint16 My_Array_DE[2];
```

**Next steps**

After creating IDTs, use TargetLink's ApplicationDataType Creation Wizard. For details, refer to [How to Create Application Data Types for Existing Implementation Data Types \(Classic AUTOSAR\)](#) on page 57.

**Related topics****Basics**

AUTOSAR ( <a href="#">TargetLink Data Dictionary Manager Reference</a> )	117
Basics on Communication Attributes and Acknowledgments.....	117
Basics on Exporting AUTOSAR Files ( <a href="#">TargetLink Interoperation and Exchange Guide</a> )	
Basics on Importing AUTOSAR Files ( <a href="#">TargetLink Interoperation and Exchange Guide</a> )	
Basics on Packages ( <a href="#">TargetLink Interoperation and Exchange Guide</a> )	
Basics on Signal Invalidation.....	119

## How to Define 2-D Array Implementation Data Types (Classic AUTOSAR)

**Introduction**

To define 2-D array [implementation data types \(IDTs\)](#) via DD Typedef objects.

**Preconditions**

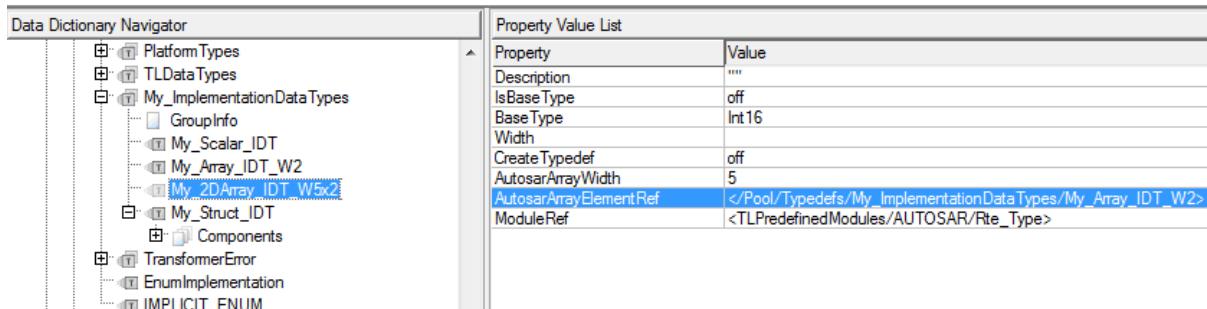
- DD project files that are used for modeling according to [Classic AUTOSAR](#) must be based on the dsdd\_master\_autosar4.dd [System] DD system template.
- You created a scalar [implementation data type \(IDT\)](#) as described in [How to Define Scalar Implementation Data Types \(Classic AUTOSAR\)](#) on page 41.
- You created an array [implementation data type \(IDT\)](#) as described in [How to Define Array Implementation Data Types \(Classic AUTOSAR\)](#) on page 43.

**Method****To create a 2-D array implementation data type****Note**

In TargetLink, implementation data types are represented as DD Typedef objects.

- 1 Open the context menu of the DD TypedefGroup that contains your implementation data types.
- 2 From the AUTOSAR submenu, choose Create AUTOSAR Array Type and rename the resulting implementation data type as required, e.g., My\_2DArray\_IDT\_W5x2.
- 3 In its Property Value List, specify the following settings:

Property	Value
BaseType	Select a base type from the list.
CreateTypedef	off
IsBaseType	off
AutosarArrayWidth	Specify the dimensionality of the array's first dimension as a scalar value. The second dimension results from the Typedef object's settings that is referenced via the AutosarArrayElementRef property.
AutosarArrayElementRef	Select an array implementation data type for the array elements that is compatible with the selected base type. This Typedef object's AutosarArrayWidth property determines the array's second dimension.
InvalidValue	Provide an invalid value if you want to model sender-receiver communication with invalidation.

**Note**

You have to specify a consistent width for interface elements and internal behavior elements that are of the defined array type. If you want to create a data element of the My\_2DArray\_IDT\_W5x2, you have to specify [5 2] for the DataElement object's Width property.

**Result**

You created an array [implementation data type \(IDT\)](#) via a DD Typedef object.

TargetLink generates local variable definitions *not* with the created type but with its underlying base type:

```
/* $ILocal: Default storage class for local variables | Width: 16 */
sint16 My_2DArray_DE[5][2];
```

## Next steps

After creating IDTs, use TargetLink's ApplicationDataType Creation Wizard. For details, refer to [How to Create Application Data Types for Existing Implementation Data Types \(Classic AUTOSAR\)](#) on page 57.

## Related topics

### Basics

Basics on Communication Attributes and Acknowledgments.....	117
Basics on Exporting AUTOSAR Files (TargetLink Interoperation and Exchange Guide)	
Basics on Importing AUTOSAR Files (TargetLink Interoperation and Exchange Guide)	
Basics on Packages (TargetLink Interoperation and Exchange Guide)	
Basics on Signal Invalidation.....	119

## How to Define Structured Implementation Data Types (Classic AUTOSAR)

### Objective

To define structured implementation data types.

### Precondition

DD project files that are used for modeling according to [Classic AUTOSAR](#) must be based on the `dsdd_master_autosar4.dd` [System] DD system template.

### Method

#### To define a structured implementation data type

##### Note

In TargetLink, implementation data types are represented as DD `Typedef` objects.

- 1 Create a DD `TypedefGroup` object in `/Pool/Typedef` and name it as required, e.g., `ImplementationDataTypes`.
- 2 Add the `GroupInfo` subtree to the `TypedefGroup` and specify a package, such as `SharedElements/ImplementationDataTypes`, via its `Package` property. This lets you export the implementation data types in the group to the specified package. Also, implementation data types in AUTOSAR files located in this package can be imported to that location.

- 3 Add a DD Typedef object to the TypedefGroup and name it as required, e.g., `My_Struct_IDT`.

- 4 In its Property Value List, specify the following settings:

Property	Value
BaseType	Struct
Is BaseType	off
CreateTypedef	on
InvalidValueRef	If required, you can reference a DD Variable object to specify this structured IDT's invalidation value.

- 5 Right-click the type definition object and choose **Create Components** from the context menu.
- 6 Right-click the **Components** object and choose **Create Component** from the context menu.
- 7 In the Property Value List, select a user-defined type.

The screenshot shows the Data Dictionary Navigator interface. On the left, the tree view displays the structure of the DDO (Data Dictionary Object) under the 'Pool' category. A blue arrow points from the 'ImplementationDataTypes' node to the 'Property Value List' window on the right. This window shows the configuration for the 'CombiSensorsStruct\_IDT' component, which is a user-defined type. Another blue arrow points from the 'Components' node in the tree view to the second 'Property Value List' window at the bottom, which shows the configuration for the 'SensorsStruct\_IDT' component, which is a type reference.

Property	Value
Description	"ImplementationDataType for ApplicationDataType CombiSensorsStruct"
Is BaseType	off
Base Type	Struct
Width	
CreateTypedef	on
Category	"STRUCTURE"
Uuid	"a4651e13-04eb-4250-9c02-e1b232e61e33"
ModuleRef	<TLPredefinedModules/AUTOSAR/Rte_Type>
AutosarArraySubElementName	<value not set>
BaseTypeCut	<value not set>
BaseTypeRef	<value not set>
BaseTypeRename	<value not set>
DefaultEnumElementRef	<value not set>
EnumElementNameTemplate	<value not set>
EnumImplementation	<value not set>
ExcludeFromCodeGeneration	<value not set>
InvalidValue	<value not set>
InvalidValueRef	<value not set>
InvalidValueToken	<value not set>
OmitCastOfMacroValue	<value not set>
Tag	<value not set>
UnderlyingEnumDataType	<value not set>

Property	Value
Description	"Element implementation for ApplicationRecordDataType SensorsStruct"
Type	<PlatformTypes/Unit16>
Width	
Volatile	off
Const	off
Category	"TYPE_REFERENCE"
Uuid	"aa403a32-4872-47bb-8aa3-9ca521102035"
Deposit	<value not set>
NumberOfBits	<value not set>
TypePrefix	<value not set>

#### Note

If you select an array type as a component of the struct type, you have to specify a consistent Width for the component.

**Result**

You created a type definition for a struct.

TargetLink generates local struct definitions for interface elements or elements of the internal behavior to the software component code:

```
/* SLLocal: Default storage class for Local variables | Width: N.A. */
My_Struct_IDT My_Struct_DE;
```

**Next steps**

After creating IDTs, use TargetLink's ApplicationDataType Creation Wizard.

For details, refer to [How to Create Application Data Types for Existing Implementation Data Types \(Classic AUTOSAR\)](#) on page 57.

**Related topics**

**Basics**

Basics on Communication Attributes and Acknowledgments.....	117
Basics on Exporting AUTOSAR Files (TargetLink Interoperation and Exchange Guide)	
Basics on Importing AUTOSAR Files (TargetLink Interoperation and Exchange Guide)	
Basics on Packages (TargetLink Interoperation and Exchange Guide)	
Basics on Signal Invalidation.....	119

# Creating Application Data Types from Scratch (Classic AUTOSAR)

## Where to go from here

## Information in this section

How to Create Primitive Application Data Types (Classic AUTOSAR).....	49
How to Create Array Application Data Types (Classic AUTOSAR).....	51
How to Create 2-D Array Application Data Types Classic (AUTOSAR).....	53
How to Create Record Application Data Types (Classic AUTOSAR).....	54

## How to Create Primitive Application Data Types (Classic AUTOSAR)

### Objective

To create primitive [application data types \(ADTs\)](#) that you can reference at [data prototypes](#) and other ADTs.

### Precondition

- DD project files that are used for modeling according to [Classic AUTOSAR](#) must be based on the dsdd\_master\_autosar4.dd [System] DD system template.
- You created the /Pool/Autosar/ApplicationDataTypes/ subtree.

### Method

#### To create ADTs of Primitive kind

- 1 In the Data Dictionary Navigator, right-click the /Pool/Autosar/ApplicationDataTypes/ subtree to open its context menu.
- 2 From the context menu, select **Create ApplicationDataTypeGroup**. A DD ApplicationDataTypeGroup object is created. Rename it as required.
- 3 Right-click the group object and choose **Create GroupInfo** from the context menu.  
In the Property Value List, enter a package name. This lets you export type definitions in the group to the specified package. Type definitions in AUTOSAR files located in the package can be imported to that location.
- 4 Right-click the group object and choose **Create ApplicationDataType** from the context menu.

- 5 In the Property Value List of the newly created ADT, specify the following settings:

Property	Value
ApplicationDataTypeKind	Primitive
Category	For a <a href="#">compound primitive data type</a> , specify the value as required: <ul style="list-style-type: none"> <li>▪ COM_AXIS - An axis that can be used in multiple <a href="#">characteristic tables</a></li> <li>▪ CURVE - A 1-D look-up table</li> <li>▪ MAP - A 2-D look-up table</li> </ul>
Min/Max	Specify upper/lower bounds if required.
ScalingRef	Specify a scaling if required.

**Result**

You have created a primitive application data type.

Property	Value
ApplicationDataTypeKind	Primitive
CalibrationAccess	NotAccessible
Description	"Scaled data type for throttle plate angle."
ScalingRef	<Shared/Sc_Throttle>
Max	108
Min	0

**Next steps**

After creating ADTs, use TargetLink's ImplementationDataType Creation Wizard. For details, refer to [How to Create Implementation Data Types for Existing Application Data Types \(Classic AUTOSAR\)](#) on page 59.

**Related topics****Basics**

Basics on Importing AUTOSAR Files (TargetLink Interoperation and Exchange Guide)	35
Basics on Packages (TargetLink Interoperation and Exchange Guide)	
Basics on Working with Implementation and Application Data Types (Classic AUTOSAR).....	35

**HowTos**

How to Create Array Application Data Types (Classic AUTOSAR).....	51
How to Create Implementation Data Types for Existing Application Data Types (Classic AUTOSAR).....	59
How to Create Record Application Data Types (Classic AUTOSAR).....	54

## How to Create Array Application Data Types (Classic AUTOSAR)

**Objective**

To create array application data types (ADTs) that you can reference at data prototypes and other ADTs.

**Precondition**

You created a primitive application data type as described in [How to Create Primitive Application Data Types \(Classic AUTOSAR\)](#) on page 49.

**Method****To create ADTs of Array kind**

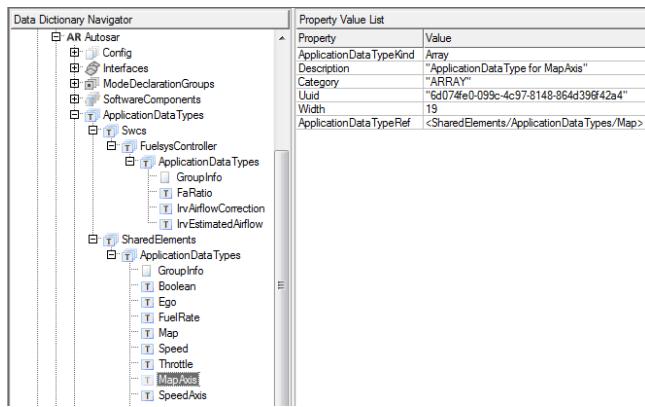
- 1 In the Data Dictionary Navigator, right-click the /Pool/Autosar/ApplicationDataTypes/ subtree to open its context menu.
- 2 From the context menu, select Create ApplicationDataTypeGroup. A DD ApplicationDataTypeGroup object is created. Rename it as required.
- 3 Right-click the group object and choose Create GroupInfo from the context menu.  
In the Property Value List, enter a package name. This lets you export type definitions in the group to the specified package. Type definitions in AUTOSAR files located in the package can be imported to that location.
- 4 Right-click the group object and choose Create ApplicationDataType from the context menu.

- 5 In the Property Value List of the newly created ADT specify the following settings:

Property	Value
ApplicationDataTypeKind	Array
ApplicationDataTypeRef	Specify the array's components by referencing an existing primitive application data type.
Width	Specify the array's width.

## Result

You created an array application data type.



## Next steps

After creating ADTs, use TargetLink's ImplementationDataType Creation Wizard. For details, refer to [How to Create Implementation Data Types for Existing Application Data Types \(Classic AUTOSAR\)](#) on page 59.

## Related topics

### Basics

[Basics on Importing AUTOSAR Files \(TargetLink Interoperation and Exchange Guide\)](#)

[Basics on Packages \(TargetLink Interoperation and Exchange Guide\)](#)

[Basics on Working with Implementation and Application Data Types \(Classic AUTOSAR\)](#).....35

### HowTos

[How to Create Implementation Data Types for Existing Application Data Types \(Classic AUTOSAR\)](#).....59

[How to Create Primitive Application Data Types \(Classic AUTOSAR\)](#).....49

[How to Create Record Application Data Types \(Classic AUTOSAR\)](#).....54

## How to Create 2-D Array Application Data Types Classic (AUTOSAR)

### Objective

To create 2-D array [application data types \(ADTs\)](#) that you can reference at [data prototypes](#) or other ADTs.

### Precondition

You created an array application data type as described in [How to Create Array Application Data Types \(Classic AUTOSAR\)](#) on page 51.

### Method

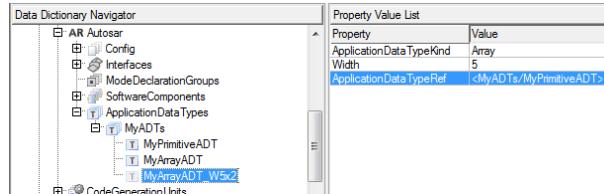
#### To create 2-D array ADTs

- 1 From the context menu of the /Pool/Autosar/ApplicationDataTypes/ object or a ApplicationDataTypeGroup object, select Create ApplicationDataType.
- 2 In the Property Value List of the newly created ADT, specify the following settings:

Property	Value
ApplicationDataTypeKind	Array
ApplicationDataTypeRef	Specify the array's components by referencing an existing array application data type.
Width	Specify the array's width.

### Result

You created an array application data type.



### Next steps

After creating ADTs, use TargetLink's ImplementationDataType Creation Wizard. For details, refer to [How to Create Implementation Data Types for Existing Application Data Types \(Classic AUTOSAR\)](#) on page 59.

**Related topics****Basics**

Basics on Importing AUTOSAR Files (Icon TargetLink Interoperation and Exchange Guide)	35
Basics on Packages (Icon TargetLink Interoperation and Exchange Guide)	
Basics on Working with Implementation and Application Data Types (Classic AUTOSAR).....	35

**HowTos**

How to Create Implementation Data Types for Existing Application Data Types (Classic AUTOSAR).....	59
How to Create Primitive Application Data Types (Classic AUTOSAR).....	49
How to Create Record Application Data Types (Classic AUTOSAR).....	54

## How to Create Record Application Data Types (Classic AUTOSAR)

**Objective**

To create [application data types \(ADTs\)](#) of type record that you can reference at [data prototypes](#) and other ADTs.

**Precondition**

The active DD workspace must be based on the dsdd\_master\_autosar4.dd [System] DD template.

The Data Dictionary must contain the /Pool/Autosar/ApplicationDataTypes/ subtree.

**Method****To create a record application data type**

- 1** In the Data Dictionary Navigator, right-click the /Pool/Autosar/ApplicationDataTypes/ subtree to open its context menu.
- 2** From the context menu, select Create ApplicationDataTypeGroup. A DD ApplicationDataTypeGroup object is created. Rename it as required.
- 3** Right-click the group object and choose Create GroupInfo from the context menu.  
In the Property Value List, enter a package name. This lets you export type definitions in the group to the specified package. Type definitions in AUTOSAR files located in the package can be imported to that location.
- 4** Right-click the group object and choose Create ApplicationDataType from the context menu.

- 5** In the Property Value List of the newly created ADT, specify the following settings:

Property	Value
ApplicationDataTypeKind	Record

- 6** From the context menu of the DD ApplicationDataType object, select Create Components.

A DD Components object is created.

- 7** From the context menu of the DD Components object, select Create Component.

A new DD Component object is created. Rename it as required.

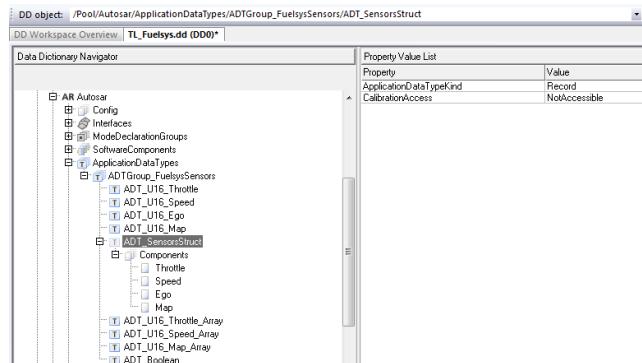
- 8** In the DD Component object's Property Value List, specify the following settings:

Property	Value
ApplicationDataTypeRef	Specify the record's component by referencing an existing application data type.

- 9** Repeat steps 7 and 8 for as many components as needed.

## Result

You created an ADT of record kind.



## Next steps

After creating ADTs, use TargetLink's ImplementationDataType Creation Wizard. Refer to [How to Create Implementation Data Types for Existing Application Data Types \(Classic AUTOSAR\)](#) on page 59.

**Related topics**

**Basics**

Basics on Importing AUTOSAR Files (Icon TargetLink Interoperation and Exchange Guide)	35
Basics on Packages (Icon TargetLink Interoperation and Exchange Guide)	
Basics on Working with Implementation and Application Data Types (Classic AUTOSAR).....	35

**HowTos**

How to Create Array Application Data Types (Classic AUTOSAR).....	51
How to Create Implementation Data Types for Existing Application Data Types (Classic AUTOSAR).....	59
How to Create Primitive Application Data Types (Classic AUTOSAR).....	49

# Using TargetLink's Wizards to Create Application Data Types or Implementation Data Types (Classic AUTOSAR)

## Where to go from here

## Information in this section

How to Create Application Data Types for Existing Implementation Data Types (Classic AUTOSAR).....	57
How to Create Implementation Data Types for Existing Application Data Types (Classic AUTOSAR).....	59

## How to Create Application Data Types for Existing Implementation Data Types (Classic AUTOSAR)

### Objective

To create [application data types \(ADTs\)](#), reference them at [data prototypes](#), and map the application data types to [implementation data types \(IDTs\)](#).

### Restriction

Creating application data types (ADTs) with TargetLink's [ApplicationDataType Creation Wizard](#) works only for data contained in DD0.

### Preconditions

You must provide a unique DD [DataTypeMappingSet](#) object when working with multiple software components.

### Part 1

#### To create and reference a data type mapping set

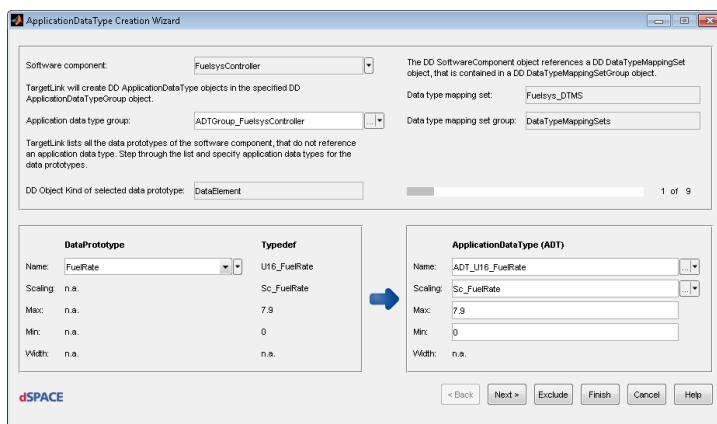
- 1 Right-click the DD Autosar object.
- 2 From the context menu, select [Create DataTypeMappingSets](#).  
A DD [DataTypeMappingSetGroup](#) object is created.
- 3 Create *exactly one* DD [DataTypeMappingSet](#) object.  
Rename it as required.
- 4 Select the DD [SoftwareComponent](#) object you want to create ADTs for.
- 5 In the Property Value List, locate the [DataTypeMappingSetRef](#) property and reference the DD [DataTypeMappingSet](#) object created in step 3.
- 6 Repeat steps 4 and 5 for all DD [SoftwareComponent](#) objects, you want to create ADTs for.

**Interim result**

You created a DD DataTypeMappingSet object and referenced it at DD SoftwareComponent objects you want to create ADTs for.

**Part 2****To create application data types for existing data dictionaries**

- 1** Right-click a DD SoftwareComponent object.
- 2** From the context menu, select AUTOSAR Tools - ApplicationDataType Creation Wizard.
- 3** In the wizard, step through the list of data prototypes and specify application data type settings as required.
- 4** Click Finish to close the wizard.



TargetLink performs the following:

- Creating the specified application data types in the specified application data type group.
  - Referencing the application data types at the respective data prototypes.
  - Creating data type maps between the types that are referenced at the data prototypes.
- 5** From the context menu of the DD ApplicationDataTypeGroup object, select Create GroupInfo.
- TargetLink creates a DD GroupInfo object.
- 6** In the DD GroupInfo object's Property Value List, enter a package name. This lets you export application type definitions in the group to the specified package. Type definitions in AUTOSAR files located in the package can be imported to that location.
  - 7** Repeat steps 1 to 6 for each software component you want to generate ADTs for.
  - 8** From the context menu of the Autosar subtree, select the Synchronize Application Data Type Settings command to synchronize and validate your AUTOSAR-specific data.

---

<b>Result</b>	You created application data types for the software components contained in your model. This lets you export an AUTOSAR file for <a href="#">Classic AUTOSAR</a> that contains ADTs, Typedefs (IDTs) and data type maps.
---------------	--

---

<b>Related topics</b>	<p>Basics</p> <p>Basics on Working with Implementation and Application Data Types (Classic AUTOSAR).....35</p> <p>References</p> <p>ApplicationDataType Creation Wizard Command ( <a href="#">TargetLink Data Dictionary Manager Reference</a>)</p>
-----------------------	---

## How to Create Implementation Data Types for Existing Application Data Types (Classic AUTOSAR)

---

<b>Objective</b>	To create <a href="#">implementation data types (IDTs)</a> , reference them at <a href="#">data prototypes</a> , and map the implementation data types to <a href="#">application data types (ADTs)</a> .
------------------	---

---

<b>Restriction</b>	Creating IDTs with TargetLink's ImplementationDataType Creation Wizard works only for data contained in DDO.
--------------------	--

---

<b>Preconditions</b>	You must provide a unique DD DataTypeMappingSet object when working with multiple software components.
----------------------	--

---

<b>Part 1</b>	<b>To create and reference a data type mapping set</b>
	<b>1</b> Right-click the DD Autosar object.
	<b>2</b> From the context menu, select Create DataTypeMappingSets. A DD DataTypeMappingSetGroup object is created.
	<b>3</b> Create exactly one DD DataTypeMappingSet object. Rename it as required.
	<b>4</b> Select the DD SoftwareComponent object you want to create IDTs for.
	<b>5</b> In the Property Value List, locate the DataTypeMappingSetRef property and reference the DD DataTypeMappingSet object created in step 3.

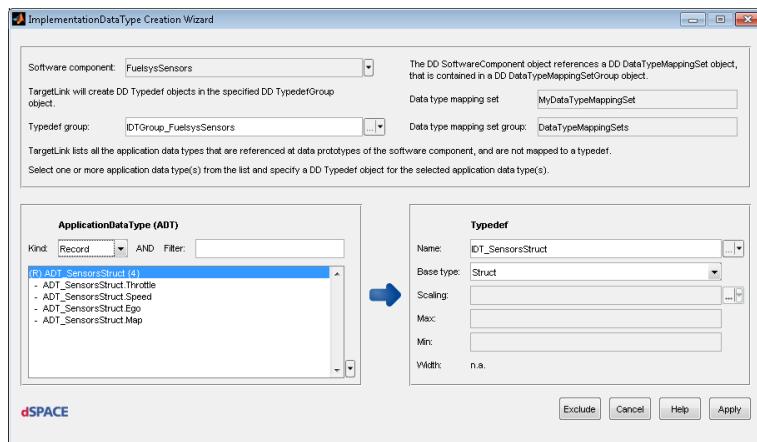
- 6** Repeat steps 4 and 5 for all DD SoftwareComponent objects you want to create IDTs for.

**Interim result**

You created a DD DataTypeMappingSet object and referenced it at DD SoftwareComponent objects you want to create IDTs for.

**Part 2****To create implementation data types for application data types**

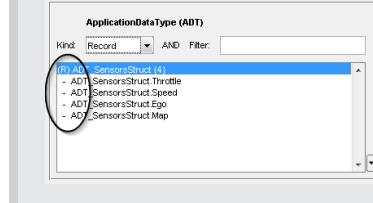
- 1** Right-click a DD SoftwareComponent object.
- 2** From the context menu, select AUTOSAR Tools - ImplementationDataType Creation Wizard.



- 3** In the wizard, select application data types by using the Kind list and a Filter expression as required.
- 4** In the wizard, specify Typedef settings as required.
- 5** Click Apply to create the specified implementation data type.

**Note**

When mapping composite ADTs of kind Record, you must map the parent element and all its components to suitable IDTs in one session.



TargetLink performs the following:

- Creating the DD Typedef object for the implementation data type.
- Referencing the typedef at the data prototypes that reference the selected application data types.

- Creating data type maps between the types that are referenced at the data prototypes.
- 6** On the context menu of the DD TypedefGroup object, select **Create GroupInfo**.
- TargetLink creates a DD GroupInfo object.
- 7** In the DD GroupInfo object's **Property Value List**, enter a package name. This lets you export application type definitions in the group to the specified package. Type definitions in AUTOSAR files located in the package can be imported to that location.
- 8** Repeat steps 1 to 7 for each software component you want to generate ADTs for.
- 9** From the context menu of the Autosar subtree, select the **Synchronize Application Data Type Settings** command to synchronize and validate your AUTOSAR-specific data.

---

**Result**

You created implementation data types for the software components contained in your model. This lets you export an AUTOSAR file for [Classic AUTOSAR](#) that contains ADTs, IDTs, and data type maps.

---

**Related topics****Basics**

[Basics on Working with Implementation and Application Data Types \(Classic AUTOSAR\)](#)..... 35

**References**

[ImplementationDataType Creation Wizard Command \(TargetLink Data Dictionary Manager Reference\)](#)



# Modeling Software Components (SWCs)

---

## Where to go from here

## Information in this section

Basics on Software Components.....	63
How to Create Software Components.....	66

---

## Basics on Software Components

---

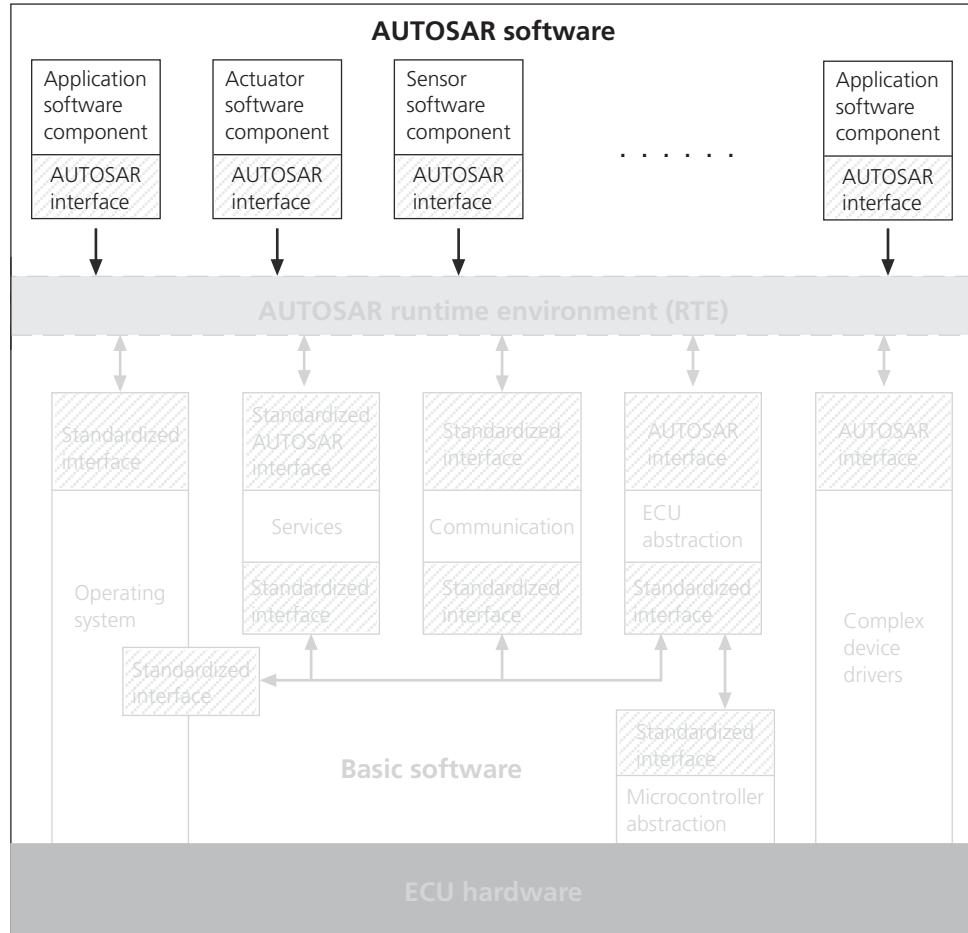
### Software components

Software components logically group and encapsulate single functionalities of ECUs.

---

### Application layer

In the illustration below, you can see that the topmost layer of the [Classic AUTOSAR](#) ECU software architecture, called the *application layer*, consists of software components (SWCs).



A software component encapsulates a certain functionality. A PI controller is modeled as an SWC in the AR\_POSCONTROL demo model as an example.

An SWC has the following characteristics:

- It is independent of the underlying hardware and relocatable.
- It is fully characterized by its reaction to an external stimulus.
- It has ports to communicate by and no hidden dependencies.

#### SWC elements according to Classic AUTOSAR

**SWC elements** SWCs have ports to provide and/or require information. Their functionality is represented by runnables.

An SWC can be a composition or an atomic SWC, which are defined as follows:

- A composition is an SWC that contains other SWCs. A composition has no functionality. It is merely a structural element.
- An atomic SWC is the basic SWC type that is not divisible and has to be implemented on one ECU. Atomic SWCs contain the functionality that is represented by runnables.

According to [Classic AUTOSAR](#), some SWCs are specialized. Examples are:

- **NvBlockSwComponent** - Manage access to the NVRAM in NvData communication.
- **ParameterSwComponent** - Provides shared calibration parameters in parameter communication.

For details, refer to the *Software Component Template* document at <https://www.autosar.org/standards/classic-platform/>.

## SWCs in TargetLink

**Creating SWCs** SWCs have to be defined in the /Pool/Autosar/SoftwareComponents/<SoftwareComponent> Data Dictionary subtree. They need no representation in a TargetLink model.

### Note

TargetLink does not support any of the following types of SWC:

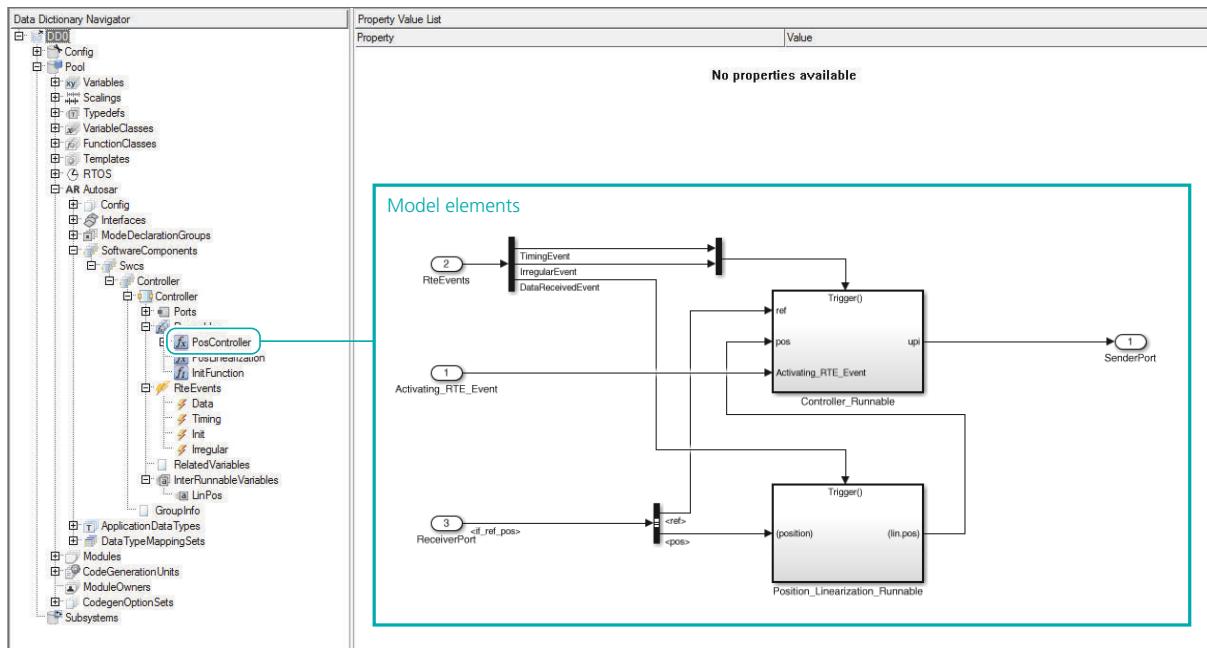
- Compositions
- ParameterSwComponentType
- NvBlockSwComponentType

**Modeling software components** Modeling software components with TargetLink basically means modeling a software component's runnables and specifying each runnable's communication. You can represent a software component by a subsystem.

The following strategies can be used to model SWCs using TargetLink:

- You can structure your TargetLink model so that a subsystem corresponds to an SWC. For example, you can model an SWC as a subsystem containing all the SWC's runnables.
- You can model only runnables and assign the runnables to SWCs defined in the Data Dictionary. Runnables of different SWCs can reside in one [TargetLink subsystem](#) of your model.

The illustration below shows a subsystem that represents a software component with two runnables, and the corresponding DD objects.



For instructions on creating SWCs, refer to [How to Create Software Components](#) on page 66.

## How to Create Software Components

### Objective

To model a functionality, you must define a software component.

### Preconditions

The following preconditions must be fulfilled:

- You have opened a TargetLink model with a Data Dictionary for the [Classic AUTOSAR](#) use case. Refer to [How to Start Modeling from Scratch](#) on page 29.
- You have created a DD Module object for creating software components and runnables. The module is required either globally for the software component code or locally for its runnables. Refer to [How to Create Module Specifications in the Data Dictionary](#) ([TargetLink Customization and Optimization Guide](#)). For an example refer to the AR\_POSCONTROL demo model.

### Method

#### To create software components

- 1 In the Data Dictionary Navigator, right-click the /Pool/Autosar/SoftwareComponents/ subtree.
- 2 Choose Create SoftwareComponentGroup from the context menu.

- 3 Enter a name for the software component group, for example, **ComponentType**.
- 4 Right-click the software component group and choose **Create GroupInfo** from the context menu.
- 5 In the **Property Value List**, enter a package name, for example, **ComponentType**. This lets you export software components in the group to the specified package. Software components in AUTOSAR files located in the package can be imported to that location.
- 6 Right-click the software component group and choose **Create SoftwareComponent** from the context menu.
- 7 Enter a name for the new software component, for example, **my\_SWC**.
- 8 In the **Property Value List**, specify the following settings that directly influence code generation:

Property	Value
AutoCreatedPIMNameTemplate	Optionally, you can control how TargetLink names the auto-created PIM. For each DD SoftwareComponent object, you can set the DD AutoCreatedPIMNameTemplate, which defaults to <b>ACP_\${I\$R}</b> . Relevant only if <b>SupportsMultipleInstantiation</b> is set to on.
ComponentType	Denotes the kind of the software component. The default is <b>ApplicationSoftwareComponent</b> . Change it only if you model special software components such as service or sensor/actuator software components.
ModuleRef	Lets you select a module for the software component code. If you do not select a module, you have to select modules for the software component's runnables.
SupportsMultipleInstantiation	Lets you instruct TargetLink to generate production code for this SWC that allows multiple instantiation.

The other properties are relevant for system-level design tools like SystemDesk, but out of scope in behavior modeling with TargetLink.

---

**Result** You have created a DD SoftwareComponent object in the Data Dictionary.

---

**Next steps** You can model the functionality of the software component with runnables. For instructions on creating runnables, refer to [How to Model Runnables](#) on page 71.

**Related topics**

**Basics**

AR_POSCONTROL (📖 TargetLink Demo Models)	
Basics on Packages (📖 TargetLink Interoperation and Exchange Guide)	
Basics on Preparing SWCs for Multiple Instantiation.....	227
Basics on Software Components.....	63

**HowTos**

How to Create Module Specifications in the Data Dictionary (📖 TargetLink Customization and Optimization Guide)	
How to Model Runnables.....	71

# Modeling Runnables

## Where to go from here

## Information in this section

Basics on Runnables.....	69
How to Model Runnables.....	71
Basics on RTE Events.....	74
How to Specify an Event for a Runnable.....	76
How to Execute a Runnable in an Exclusive Area.....	79
Basics on Activation Reasons.....	80
How To Model a Runnable's Activation Reasons.....	82
Basics on Initializing Runnable-Specific Variables.....	84
How to Initialize Runnable-Specific Variables in a Restart Runnable.....	86

## Basics on Runnables

### Runnables

Atomic software components contain runnables that are implemented as C functions by TargetLink.

### Runnables according to Classic AUTOSAR

**Runnable triggering** According to [Classic AUTOSAR](#), runnables are triggered by RTE events. These RTE events must be kept in mind when the runnables are mapped to tasks of the operating system for execution. For basic information on RTE events, refer to [Basics on RTE Events](#) on page 74.

**Exclusive areas** Runnable code can contain critical sections that must not be interrupted/preempted by other runnables. For example, to ensure data

consistency, runnable access to shared memory must not be interrupted by other runnables that access the same shared memory sections. [Classic AUTOSAR](#) has introduced the concept of exclusive areas for the mutual exclusion of runnables. According to [Classic AUTOSAR](#), a runnable can either always run in an exclusive area or dynamically enter and leave an exclusive area. Exclusive areas must be kept in mind when the runnables are mapped to tasks of the operating system for execution.

---

## Runnables in TargetLink

**Creating runnables** Runnables have to be defined in the following subtree of the Pool area of the Data Dictionary:

`Autosar/SoftwareComponents/<SoftwareComponent>/Runnables/`

If this subtree does not exist in your Data Dictionary, you can add it via the context menu of the DD SoftwareComponent object.

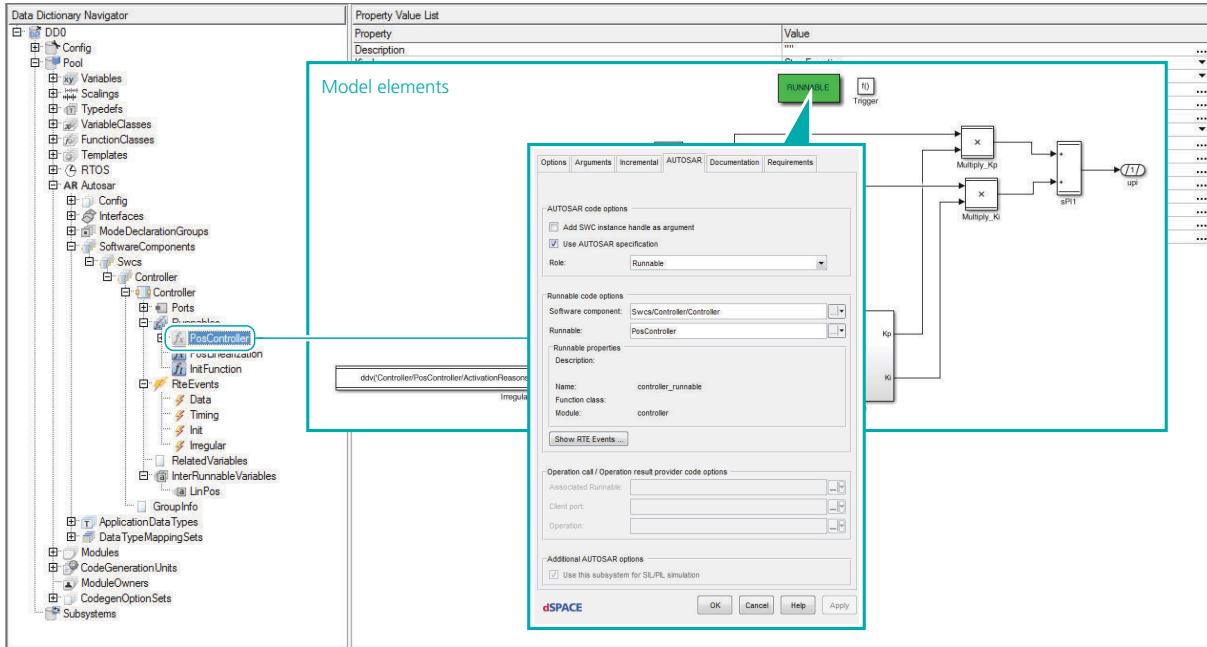
They are represented in a TargetLink model by an atomic subsystem with a Function block which is configured accordingly. For instructions on creating runnables, refer to [How to Model Runnables](#) on page 71.

**Triggering runnables** TargetLink lets you specify RTE events. You can export them to an AUTOSAR file and perform the runnable-to-task mapping with a system-level design tool such as SystemDesk. For instructions on specifying events for runnables, refer to [How to Specify an Event for a Runnable](#) on page 76.

However, in simulations in MIL/SIL/PIL simulation modes, the execution of the runnable subsystems is based on the data flow or, if specified, the subsystems are executed as function-call-triggered subsystems. For basic information on the control flow in TargetLink models, refer to [Controlling Subsystem Execution](#) ([TargetLink Customization and Optimization Guide](#)).

**Modeling runnables** You model each runnable as an atomic subsystem with a Function block.

The illustration below shows the modeling of a runnable.



## How to Model Runnables

### Objective

To model the functionality of an SWC, you have to model a runnable.

### Preconditions

The following preconditions must be fulfilled:

- You created an SWC. Refer to [How to Create Software Components](#) on page 66.
- You set TargetLink's Code generation mode mode to **Classic AUTOSAR** on the Code Generation page of the [TargetLink Main Dialog Block](#) ([TargetLink Model Element Reference](#)).

### Method

#### To model runnables

- 1 In the Data Dictionary Navigator, expand the /Pool/Autosar object and locate the following child object:  
**SoftwareComponents/<SoftwareComponent>/Runnables**  
If this subtree does not exist in your Data Dictionary, you can add it via the context menu of the DD SoftwareComponent object.
- 2 From its context menu, choose **Create Runnable**.
- 3 Enter a name for the new runnable, for example, **my\_runnable**.

- 4** In the Property Value List, specify the following settings:

Property	Value
Kind	Specifies whether you are modeling a TargetLink step function, restart function, or terminate function. The default is StepFunction. Change it only if you want to create the runnable for restarting or terminating the software component. For instructions on initializing runnable-specific variables in TargetLink's restart runnable, refer to <a href="#">How to Initialize Runnable-Specific Variables in a Restart Runnable</a> on page 86.
NameTemplate	Enter a name for the runnable in the generated code. You can use the \$D name macro. In the present context it evaluates to the DD Runnable object's name.
FunctionClassRef	<ul style="list-style-type: none"> <li>▪ Select AUTOSAR/RUNNABLE4 to generate code that supports the compiler abstraction defined by <a href="#">Classic AUTOSAR</a>.</li> <li>▪ Select a <a href="#">function class</a> with global scope such as GLOBAL_FCN to generate code without <a href="#">Classic AUTOSAR</a> compiler abstraction macros.</li> </ul>
ModuleRef	Lets you select a module for the runnable's code. If you do not select a module, you have to select a module for the software component.

- 5** To your model, add an atomic subsystem and open it.

- 6** Add a Function block and open its block dialog.

- 7** On the AUTOSAR page, make the following settings:

Property	Value
AUTOSAR mode	Classic
Role	Runnable
Software component	The DD SoftwareComponent object that contains the DD Runnable object that you want to model.
Runnable	The DD Runnable object that you want to model.

**Result**

You have created a runnable.

The screenshot shows the Data Dictionary Navigator on the left and the Property Value List on the right. In the Data Dictionary Navigator, under the 'SoftwareComponents' section, there is a 'Swcs' folder which contains a 'FuelsysController' component. Inside 'FuelsysController', there is a 'FuelsysSensors' component. Under 'FuelsysSensors', there is a 'Runnables' folder containing a 'DetectSensorFailures' function. This entire path is highlighted with a red rectangle. On the right, the 'Property Value List' window is open for the 'DetectSensorFailures' function. It shows various properties like Description, Kind, NoRestartCode, Uuid, CanBeInvokedConcurrently, MinimumStartInterval, and NameTemplate. Below these, the 'TargetLink Function: TL\_Fuelsys/DetectSensorFailures' tab is selected. The 'AUTOSAR' tab is also visible. The 'Role:' field is set to 'Runnable'. The 'Runnable code options' section shows 'Software component: Swcs/FuelsysSensors/FuelsysSensors' and 'Runnable: DetectSensorFailures'. The 'Runnable properties' section shows 'Description: Detects sensor failures and determines the fu...' and 'Name: DetectSensorFailures'. A second red rectangle highlights the 'Runnable code options' and 'Runnable properties' sections.

TargetLink lets you generate code such as the following to the specified module (the code shown below does not contain [Classic AUTOSAR compiler abstraction macros](#)):

```
...
extern void DetectSensorFailures(void);
...
void DetectSensorFailures(void)
{
    ...
}
```

**Next steps**

You can define an RTE event to activate the runnable. For instructions, refer to [How to Specify an Event for a Runnable](#) on page 76.

**Related topics****Basics**

Basics on Initializing Runnable-Specific Variables.....	84
Basics on Runnables.....	69
Basics on Using Name Macros (TargetLink Customization and Optimization Guide)	

**HowTos**

How to Create Software Components.....	66
How to Specify an Event for a Runnable.....	76

**References**

Function Block (TargetLink Model Element Reference)	
---	--

## Basics on RTE Events

**RTE events**

An RTE event defines the situations and conditions for starting or continuing the execution of a specific runnable.

**RTE events according to  
Classic AUTOSAR**

The following table shows a classification of the RTE events:

<b>RTE Event Kind</b>	<b>Description</b>
<b>Timing</b>	
TIMING_EVENT	Executes a runnable periodically. The runnable performs a task at constant intervals, for example, polling a variable. The runnable must not be triggered by any other RTE events.
<b>Client server communication</b>	
ASYNCHRONOUS_SERVER_CALL_RETURNS_EVENT	Executes a runnable after completion of a server operation that has been called asynchronously. The runnable processes the server result and possible application errors.
OPERATION_INVOKED_EVENT	Executes a runnable as a result of a client call. The runnable represents a server.
TRANSFORMER_HARD_ERROR_EVENT	Executes a runnable to react to a hard transformer error on the server side of client-server communication.

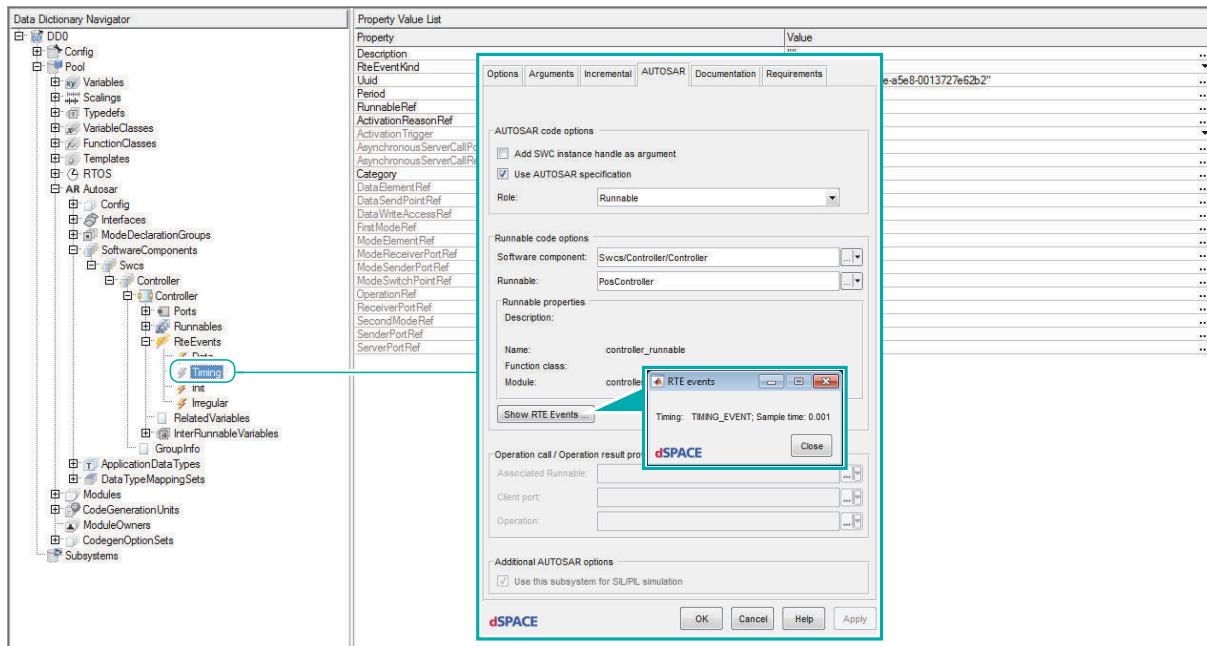
RTE Event Kind	Description
<b>Sender receiver communication</b>	
DATA_RECEIVED_EVENT	Executes a runnable after a data element or event message is provided by a sender. The runnable represents the receiver side of sender-receiver communication and processes the data element.
DATA_SEND_COMPLETED_EVENT	Executes a runnable as a result of a data sent acknowledgement. The runnable can handle any errors in the sending process or continue the application if the sending process was successful.
DATA_RECEIVE_ERROR_EVENT	Executes a runnable when the receiving of a data element or event message results in an error. The runnable handles any errors.
DATA_WRITE_COMPLETED_EVENT	Triggers runnable entities when an implicit write access was successful or an error occurred during writing. The triggered runnable entity can handle any errors in the writing process or continue the application if the writing process was successful.
<b>Mode management</b>	
MODE_SWITCHED_ACK_EVENT	Executes a runnable as a result of a successful mode switch. The runnable is a part of a mode managing software component.
MODE_SWITCH_EVENT	Executes a runnable as a result of a mode switch. The runnable can be triggered on entry to or exit from the mode, depending on the mode switch event's activation trigger.
<b>Others</b>	
BACKGROUND_EVENT	Triggers runnable entities that are executed in the background.
INIT_EVENT	Triggers runnable entities that are used for initialization purposes, i.e., for starting and restarting an SWC internal behavior.

## RTE events in TargetLink

**Creating RTE events** TargetLink lets you define RTE events in the Data Dictionary in the following subtree:  
`/Pool/Autosar/SoftwareComponents/<SoftwareComponent>/RteEvents/`  
 If this subtree does not exist in your Data Dictionary, you can add it via the context menu of the DD SoftwareComponent object.

**Modeling RTE events in simulations** TargetLink lets you specify RTE events for triggering runnables. In simulations in MIL/SIL/PIL simulation modes, runnable subsystems are executed either according to the data flow or as function-call-triggered subsystems. You can model RTE events using Stateflow charts as function calls for this.

The illustration below shows the RTE event as specified within the Data Dictionary and the block dialog of the Function block that resides in the corresponding [runnable subsystem](#):

**Related topics****HowTos**

[How to Specify an Event for a Runnable.....76](#)

## How to Specify an Event for a Runnable

**Objective**

You have to define an [RTE event](#) and assign it to a runnable.

**Preconditions**

The following preconditions must be fulfilled:

- You have created an SWC. For instructions on creating SWCs, refer to [How to Create Software Components](#) on page 66.
- You have created at least one runnable. For instructions on creating runnables, refer to [How to Model Runnables](#) on page 71.

**Method****To specify an event for a runnable**

- 1 In the Data Dictionary Navigator, right-click the `/Pool/Autosar/SoftwareComponents/<SoftwareComponent>/RteEvents/` subtree.

If this subtree does not exist in your Data Dictionary, you can add it via the context menu of the DD SoftwareComponent object.

- 2** Choose Create RteEvent from the context menu.  
TargetLink creates a new RTE event.
- 3** Enter a name for the RTE event, for example, **my\_RteEvent**.
- 4** In its Property Value List, specify the following properties:

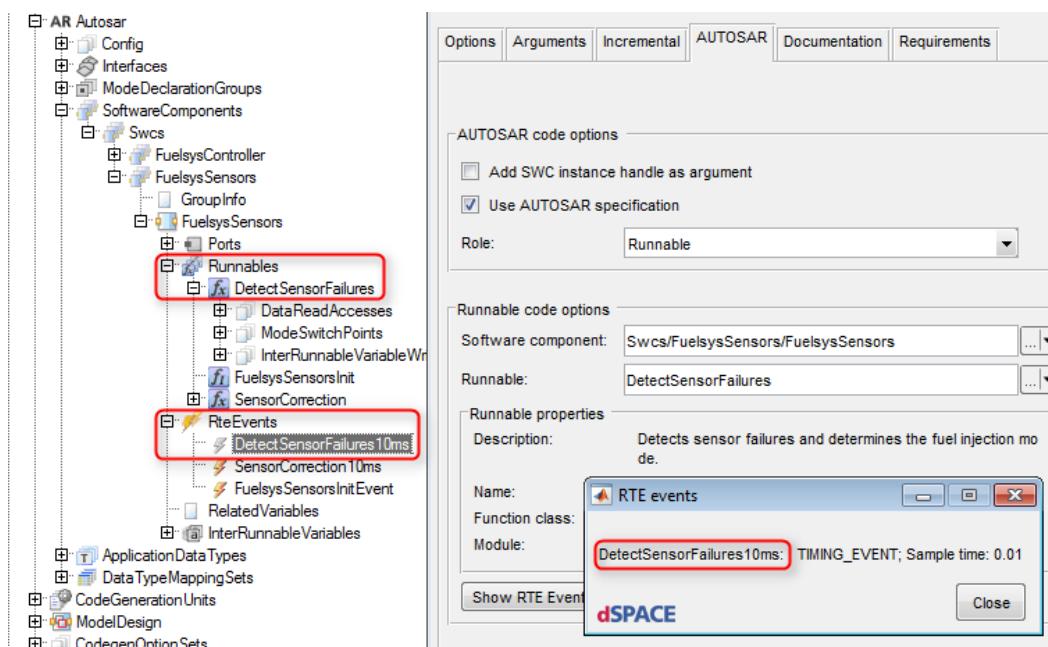
Property	Value
RunnableRef	Select the runnable to trigger.
RteEventKind	Select the kind of event you want to create.

- 5** For the following event kinds, you have to specify further settings:

Event Kind	Property	Value
ASYNCHRONOUS_SERVER_CALL RETURNS_EVENT	AsynchronousServerCallResultPointRef	Specify the access point.
DATA_RECEIVE_ERROR_EVENT	DataElementRef	Select a DD DataElement object of the DD SenderReceiverInterface object that is assigned to the selected DD ReceiverPort or SenderReceiverPort object.
DATA_RECEIVED_EVENT	DataElementRef	Select a DD DataElement object of the DD SenderReceiverInterface object that is assigned to the selected DD ReceiverPort or SenderReceiverPort object.
	ReceiverPortRef or SenderReceiverPortRef	Select a DD ReceiverPort or SenderReceiverPort object of the DD SoftwareComponent object.
DATA_SEND_COMPLETED_EVENT	DataSendPointRef	Select a data send point of the runnable. It must contain references to the data element and port.
DATA_WRITE_COMPLETED_EVENT	DataWriteAccessRef	Specify the access point that describes the write access which triggers this event when completed.
MODE_SWITCHED_ACK_EVENT	ModeSwitchPointRef	Select a mode switch point.
MODE_SWITCH_EVENT	ActivationTrigger	Select whether the runnable is to be triggered on entry to or exit from the mode.
	FirstModeRef	Select the initial mode of the mode declaration group.
	ModeElementRef	Select a mode element of a mode switch interface.
	ModeReceiverPortRef	Select the mode receiver port to read the mode element from.

Event Kind	Property	Value
OPERATION_INVOKED_EVENT	OperationRef	Select a DD Operation object of the DD ClientServerInterface object that is assigned to the selected DD ServerPort object.
	ServerPortRef	Select a DD ServerPort object of the DD SoftwareComponent object..
TIMING_EVENT	Period	Specify a period in s, e.g., 0.001. The RTE triggers the runnable at constant intervals with the specified period.
TRANSFORMER_HARD_ERROR_EVENT	OperationRef	Select a DD Operation object of the DD ClientServerInterface object that is assigned to the selected DD ServerPort object.
	ServerPortRef	Select a DD ServerPort object of the DD SoftwareComponent object..

- 6 Optionally, you can specify the runnable's activation reason via the ActivationReasonRef property.
- 7 In your model, open the runnable subsystem that implements the runnable and open the Function block's dialog to view the specified settings.

**Result**

You have defined an RTE event and assigned it to a runnable.

**Related topics****Basics**

Basics on Activation Reasons.....	80
Basics on RTE Events.....	74
Basics on Runnables.....	69

**HowTos**

How to Create Software Components.....	66
How To Model a Runnable's Activation Reasons.....	82
How to Model Runnables.....	71

## How to Execute a Runnable in an Exclusive Area

**Objective**

To create an exclusive area and specify runnables that must run under mutual exclusion.

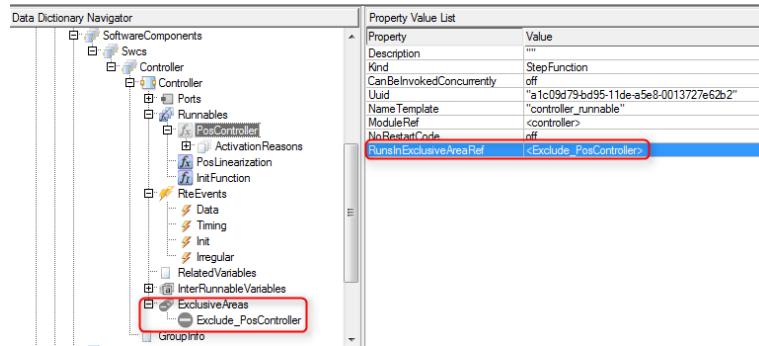
**Preconditions**

You have created a runnable. For instructions on creating runnables, refer to [How to Model Runnables](#) on page 71.

**Method****To execute a runnable in an exclusive area**

- 1** In the Data Dictionary Navigator, right-click the /Pool/Autosar/SoftwareComponents/<SoftwareComponent>/ExclusiveAreas subtree.  
If the ExclusiveAreas object does not exist you can create it via the context menu of the <SoftwareComponent> object.
- 2** From the context menu, choose Create Exclusive Area.
- 3** Enter a name for the exclusive area, for example, `my_ExclusiveArea`.
- 4** Left-click the runnable that you want to run in the exclusive area.
- 5** In the Property Value List, specify the following settings:

Property	Value
RunsInExclusiveAreaRef	Select the exclusive area.

**Result**

You have created an exclusive area and specified a runnable that runs in it.

**Related topics****Basics**

[Basics on Runnables](#)..... 69

**HowTos**

[How to Model Runnables](#)..... 71

## Basics on Activation Reasons

**Activation reasons**

An [activation reason](#) is an [activating RTE event](#) that triggered a runnable.

**Activation reasons according to Classic AUTOSAR**

According to [Classic AUTOSAR](#), each runnable has an event that triggered it. The RTE can inform the runnable about this event by passing the runnable its activation reason (one or more RTE events). Depending on the activation reason, conditional control flows can be modeled.

**Grouping RTE events** Several RTE events can be grouped by one activation reason. For example, different data received events or periodic events with different cycle times.

**Storing activation reasons** The runnable's activation reasons are stored as bits in a data type that is called activation vector.

**Activation reasons in TargetLink**

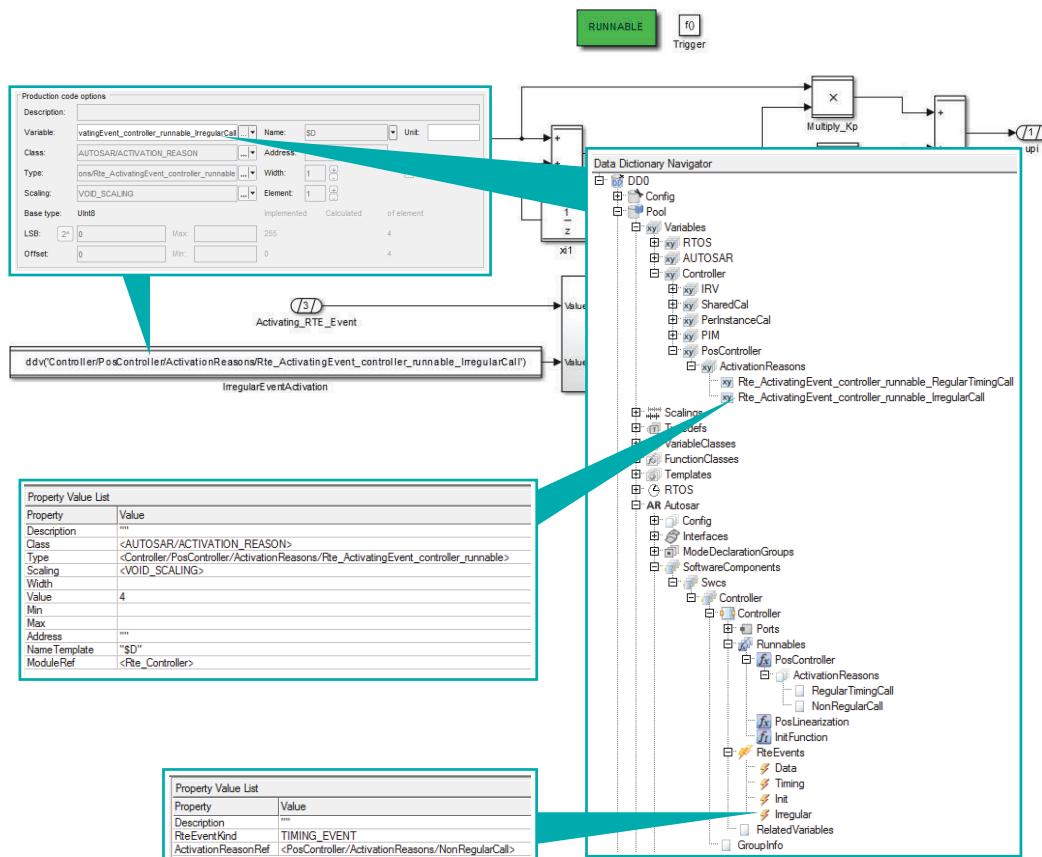
TargetLink lets you specify DD ActivationReason objects in a Runnable object's subtree.

Each ActivationReason object represents one activation reason and lets you specify its symbol and bit position within the runnable's activation vector.

### Note

Each DD ActivationReason object can be referenced at several DD RteEvent objects. This lets you group different activating RTE events.

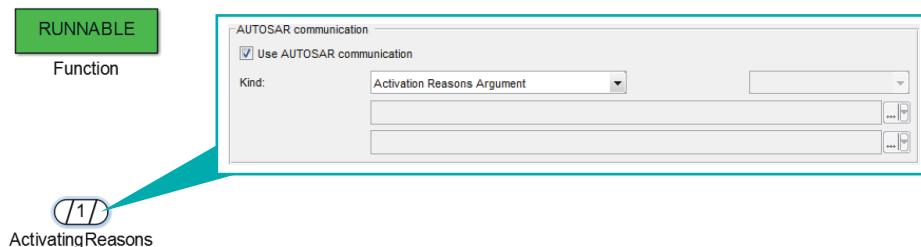
**Using activation reasons in the model** TargetLink lets you use activation reasons in the model. When you execute the Synchronize Activation Reason Settings command from the DD Runnable object's context menu, TargetLink automatically creates all the required DD Typedef objects and DD Variable objects. For an example refer to the [AR\\_POSCONTROL](#) ([TargetLink Demo Models](#)) demo model:



TargetLink makes sure that the resulting DD objects comply with [Classic AUTOSAR](#).

**Activation reasons in the model**

To your model you add a TargetLink InPort block to model the formal argument that represents the runnable's activation reason:



**Activation reasons in production code**

In production code, TargetLink generates activating reasons as formal arguments within the runnable function's signature:

```
FUNC(void, MySWC_CODE) MyRunnable(Rte_ActivatingEvent_MyRunnable activation);
```

**Limitation**

TargetLink does not support the runnable's access to its activation vector if the runnable is modeled as an [operation call with runnable implementation subsystem](#).

**Related topics**

**Basics**

Basics on RTE Events.....	74
Basics on Runnables.....	69
Synchronize Activation Reason Settings ( TargetLink Data Dictionary Manager Reference)	

**HowTos**

How To Model a Runnable's Activation Reasons.....	82
---	----

## How To Model a Runnable's Activation Reasons

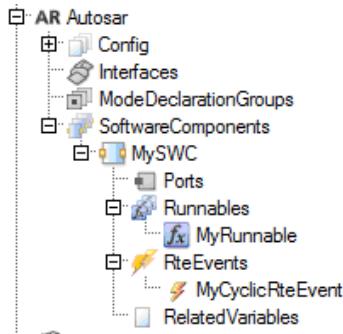
**Restriction**

TargetLink does not support the runnable's access to its activation vector if the runnable is modeled as an [operation call with runnable implementation subsystem](#).

**Precondition**

The following preconditions must be fulfilled:

- You created a DD SoftwareComponent object together with the following objects:
  - A DD Runnable object, such as **MyRunnable**.
  - A DD RteEvent object, such as **MyCyclicRteEvent**.

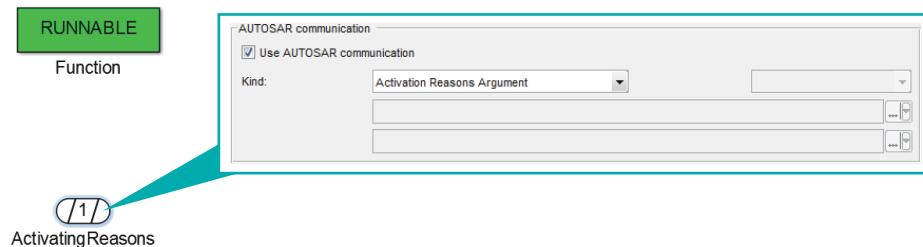
**Method****To model a runnable's activation reason**

- 1 From **MyRunnable**'s context menu select **Create ActivationReasons** to add the **ActivationReasons** object to the **Runnable** object's subtree.
- 2 Add a DD ActivationReason object such as **MyActivationReason** to **ActivationReasons'** subtree and specify its symbol and its bit position as required.
- 3 At the DD RteEvent objects, you want to group by the DD ActivationReason object created in step 2, make the following settings:

Property	Description
ActivationReasonRef	<b>MyRunnable/ActivationReasons/MyActivationReason</b>
RunnableRef	<b>MyRunnable</b>

- 4 If you want to use the activation reason's bit position to model a conditional control flow, execute the **Synchronize Activation Reason Settings** command from **MyRunnable**'s context menu.  
TargetLink automatically creates all the required DD Typedef objects and DD Variable objects so that you can use them in your model.
- 5 In the model, add a TargetLink InPort block to the root level of your runnable.

- 6** On the AUTOSAR page of the InPort block's block dialog make the following settings:



- 7** Generate code.

## Result

You modeled an activation reason that appears as formal parameter in the runnable function's signature:

```
FUNC(void, MySWC_CODE) MyRunnable(Rte_ActivatingEvent_MyRunnable activation);
```

## Related topics

### Basics

Basics on Activation Reasons.....	80
Synchronize Activation Reason Settings ( TargetLink Data Dictionary Manager Reference)	

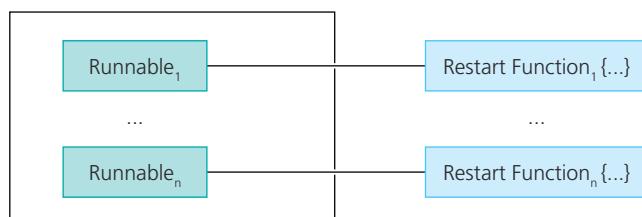
### HowTos

How to Create Software Components.....	66
How to Model Runnables.....	71
How to Specify an Event for a Runnable.....	76

## Basics on Initializing Runnable-Specific Variables

### Initializing runnable-specific variables

By default, TargetLink creates one restart function for each runnable. You can instruct TargetLink which of the runnable's variables you want to initialize in this restart function.



In production code, a typical restart function looks like this:

```
void RESTART_MySWC_MyRunnable(void)
{
    X_Sa1_Unit_Delay = 0;
}
```

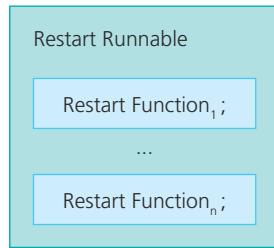
## Restart runnables

### Note

Restart runnables are TargetLink-specific. They are not described by [Classic AUTOSAR](#) as special runnable entities.

You must not reference a DD Runnable object whose Kind property is set to **RestartFunction** at Runnable blocks.

You can specify a restart runnable for each software component. This restart runnable then collects all the function calls to the other runnables' restart functions of the same SWC.



If all the software component's runnables are generated into the same module, TargetLink inlines each runnable's restart function within the restart runnable.

In production code, a typical restart runnable looks like this:

```
FUNC(void, MySWC_CODE) MyRestartRunnable(void)
{
    RESTART_MySWC_RunnableInOtherModule();
    X_Sa2_Unit_Delay = 0;
}
```

Note that the restart runnable's function body contains the call to another runnable's restart function, which indicates that this runnable was generated into another module than the restart runnable.

## Suppressing empty restart functions

You can suppress empty restart functions. This is done by setting the DD Runnable object's **NoRestartCode** property to **on**.

If you accidentally set the **NoRestartCode** property to **on** for a runnable whose restart function is not empty, TargetLink displays an error message.

You can use the **SuppressNoRunnableRestartCodeError** Code Generator option for debugging.

**Related topics****Basics**

Basics on Runnables.....	69
Modeling Communication According to Classic AUTOSAR.....	89
Modeling Software Components (SWCs).....	63

**HowTos**

How to Initialize Runnable-Specific Variables in a Restart Runnable.....	86
How to Model Runnables.....	71

**References**

SuppressNoRunnableRestartCodeError.....	392
---	-----

## How to Initialize Runnable-Specific Variables in a Restart Runnable

**Objective**

TargetLink lets you initialize the variables of a software component in a restart runnable.

**Note**

Restart runnables are TargetLink-specific. They are not described by [Classic AUTOSAR](#) as special runnable entities.

You must not reference a DD Runnable object whose Kind property is set to **RestartFunction** at Runnable blocks.

**Precondition**

The following preconditions must be fulfilled:

- You created a DD **SoftwareComponent** object such as **MySwc**.
- You created a DD **Module** object and referenced it at **MySwc's ModuleRef** property.

It is recommended that the SWC's runnables do not reference a different module so that TargetLink generates their code into the same module. This ensures that the calls to each runnable's restart function can be inlined correctly within the restart runnable's function body.

- You modeled one or more runnables, each of which contains at least one variable that you want to initialize in the SWC's restart runnable.

**Method****To initialize runnable-specific variables in a restart runnable**

- 1** Create a DD Runnable object in MySwc's subtree.
- 2** Rename the Runnable object as required and set its Kind property to **RestartFunction**.
- 3** Create a DD VariableClass object and rename it as **INIT\_IN\_RESTART\_RUNNABLE**.
- 4** In its Property Value List, make the following settings:

Property	Value
InitAtDefinition	off
RestartFunctionName	default

- 5** Reference the variable class created in step 3 at all the block variables you want to initialize in the restart runnable.
- 6** In all the other runnables of the software component, set the **NoRestartCode** property to **off**.
- 7** Generate code.

**Result**

You created a restart runnable in which to initialize the variables of all the software component's other runnables.

It looks like this:

```
FUNC(void, MySWC_CODE) MyRestartRunnable(void)
{
    X_Sa1_Unit_Delay = 0;
    X_Sa2_Unit_Delay = 0;
}
```

**Related topics****Basics**

Basics on Initializing Runnable-Specific Variables.....	84
Basics on Runnables.....	69
Modeling Communication According to Classic AUTOSAR.....	89
Modeling Software Components (SWCs).....	63

**HowTos**

How to Model Runnables.....	71
-----------------------------	----



# Modeling Communication According to Classic AUTOSAR

## Where to go from here

## Information in this section

Introduction to Communication According to Classic AUTOSAR.....	90
Modeling Classic AUTOSAR Communication via Data Stores.....	93
Creating Interfaces, Ports, and Communication Subjects.....	100
Modeling Sender-Receiver Communication.....	108
Modeling Client-Server Communication.....	133
Modeling Interrunnable Communication.....	171
Modeling NvData Communication.....	178
Modeling Transformer Error Logic.....	205

# Introduction to Communication According to Classic AUTOSAR

## Overview of Communication According to Classic AUTOSAR

### Communication kind

[Classic AUTOSAR](#) provides the following communication kinds:

AUTOSAR Communication Kind	Description
Sender-receiver communication	In sender-receiver communication, a sending runnable provides data and a receiving runnable requires it. The involved buffers depend on the kinds of ports that are used for communication: <ul style="list-style-type: none"> <li>▪ Separate buffer for reading and writing: <ul style="list-style-type: none"> <li>▪ Require port</li> <li>▪ Provide port</li> </ul> </li> <li>▪ Combined buffer for reading and writing: <ul style="list-style-type: none"> <li>▪ Provide-require port</li> </ul> </li> </ul>
Client-server communication	In client-server communication, a client runnable requests an operation from a server runnable. The operation can involve requesting a certain data item from a sensor, or it can be a complex computation. Operation arguments can be delivered to the server by the client, and in some cases, the result is transferred back.
Interrunnable communication	In interruptible communication, runnables of the same software component communicate via interruptible variables. Here, the communicating runnables reference the same interruptible variable for communication. Interruptible communication does not involve ports or interfaces.
Parameter communication	In parameter communication, software components of different types can share calibration parameters that are provided by a software component of the <code>ParameterSwComponentType</code> via provide ports for calibration parameter elements.
NvData communication	In NvData (nonvolatile data) communication, an atomic software component ( <code>ApplicationSwComponentType</code> ) can communicate via an NvData interface with another software component ( <code>NvBlockSwComponent</code> ) that manages the access to the NVRAM. The involved buffers depend on the kinds of ports that are used for communication: <ul style="list-style-type: none"> <li>▪ Separate buffer for reading and writing: <ul style="list-style-type: none"> <li>▪ Require port</li> <li>▪ Provide port</li> </ul> </li> <li>▪ Combined buffer for reading and writing: <ul style="list-style-type: none"> <li>▪ Provide-require port</li> </ul> </li> </ul>

**Interfaces and communication subjects** Each communication kind has its own communication subject. Some communication kinds also have interfaces. This is shown in the following table:

AUTOSAR Communication Kind	DD <Interface> Object	Communication Subject
Client-server communication	/Pool/Autosar/Interfaces/<ClientServerInterface>	Operation with optional OperationArgument
Interrunnable communication	-	InterRunnableVariable
NvData communication	/Pool/Autosar/Interfaces/<NvDataInterface>	NvDataElement
Parameter communication	/Pool/Autosar/Interfaces/<CalPrmInterface>	CalPrmElement
Sender-receiver communication	/Pool/Autosar/Interfaces/<SenderReceiverInterface>	DataElement

**Ports** AUTOSAR communication kinds that have interfaces also have ports. These are shown in the following table:

Mapping Between AUTOSAR Communication Kind, AUTOSAR Ports, and DD Port Objects			
AUTOSAR Communication Kind	Provide Port	Require Port	Provide-Require Port
Sender-receiver communication	DD SenderPort object	DD ReceiverPort object	DD SenderReceiverPort object
Client-server communication	DD ServerPort object	DD ClientPort object	Not supported by TargetLink.
NvData communication	DD NvSenderPort object	DD NvReceiverPort object	DD NvSenderReceiverPort object
Parameter communication	DD ProvideCalPrmPort object <sup>1)</sup>	DD RequireCalPrmPort object	Not defined by AUTOSAR

<sup>1)</sup> Out of scope for TargetLink as a behavior modeling tool.

**Communication attributes of a port** Communication attributes are stored in a communication specification at the port for each DataElement, NvDataElement, CalPrmElement, or OperationArgument. They specify the data flow with initial values, acknowledgement requests and queue lengths, for example.

**Creating interfaces, ports,  
and communication subjects  
in TargetLink**

<b>AUTOSAR Communication Kind</b>	<b>Instructions</b>		
	<b>Creating Interfaces</b>	<b>Creating Ports</b>	<b>Creating Communication Subjects</b>
Client-server communication	<a href="#">How To Create Interfaces on page 100</a>	<a href="#">How to Create Ports on page 101</a>	<a href="#">How to Create Communication Subjects for Classic AUTOSAR Communication on page 103</a>
NvData communication			
Parameter communication			
Sender-receiver communication			
Interrunnable communication	-	-	<a href="#">How to Create Interrunnable Variables on page 173</a>

**Related topics**

**Basics**

<a href="#">Creating Interfaces, Ports, and Communication Subjects.....</a>	100
<a href="#">Modeling Client-Server Communication.....</a>	133
<a href="#">Modeling Interrunnable Communication.....</a>	171
<a href="#">Modeling Sender-Receiver Communication.....</a>	108

**HowTos**

<a href="#">Modeling NvData Communication.....</a>	178
--	-----

# Modeling Classic AUTOSAR Communication via Data Stores

## Where to go from here

## Information in this section

Basics on Modeling AUTOSAR Communication via Data Stores.....	93
Dynamically Accessing AUTOSAR Data Stores via Custom Code (Type II) Blocks.....	96

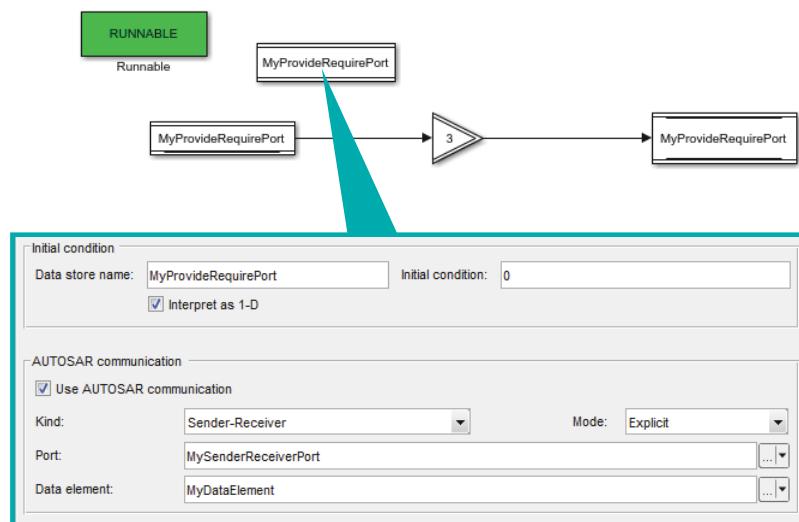
## Basics on Modeling AUTOSAR Communication via Data Stores

### Introduction

You can use Data Store Memory blocks or [global data stores](#) to model communication as described by AUTOSAR.

### Modeling with Data Store Memory blocks

When modeling with Data Store Memory blocks, you provide the AUTOSAR specification on the AUTOSAR page of the block dialog. The following illustration shows the specification of explicit sender-receiver communication as an example:



### Modeling with global data stores

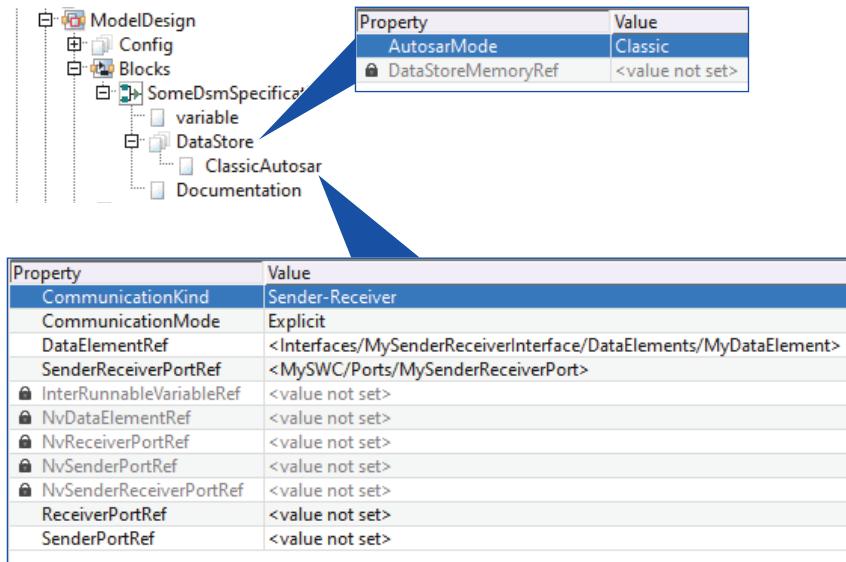
#### Note

Global data stores are not supported when modeling for [Adaptive AUTOSAR](#).

When modeling with a [global data store](#), you provide the AUTOSAR specification at the Block object that specifies the global data store. This is done in the following child objects:

- DataStore child object
- ClassicAutowar child object

The following illustration shows the specification of explicit sender-receiver communication as an example:

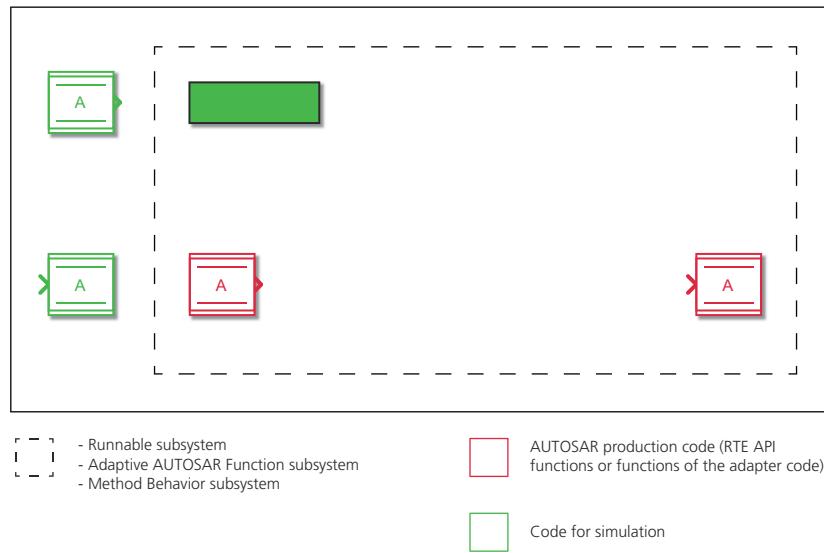


#### Code generated for data stores

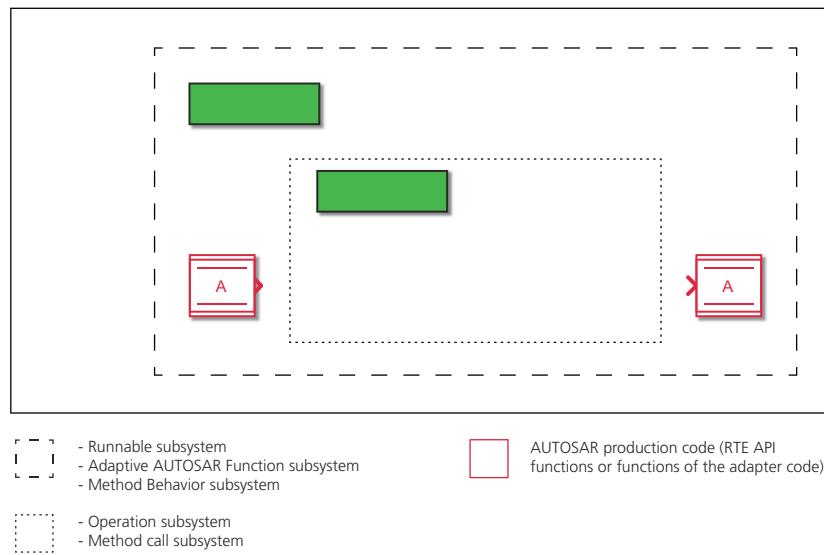
The code that is generated for the data stores depends on the location of the corresponding Data Store Read and Data Store Write blocks:

Placed outside of the following subsystem kinds, Data Store Read and Data Store Write blocks are used to generate code for simulation:

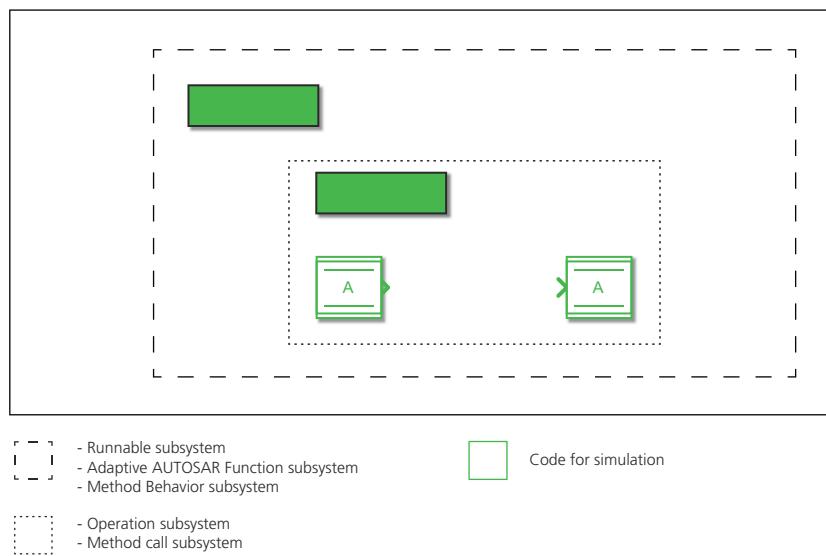
- [Runnable subsystem](#)
- [Adaptive AUTOSAR Function subsystem](#)
- [Method Behavior subsystem](#)



Placed inside of the subsystem kinds mentioned above but outside of [operation subsystem](#) or [method Call subsystem](#), Data Store Read and Data Store Write blocks are used to generate AUTOSAR production code (RTE API functions or functions of the [ARA adapter code](#)).



Placed inside of the subsystem kinds mentioned above but inside of [operation subsystem](#) or [method Call subsystem](#), Data Store Read and Data Store Write blocks are used to generate code to simulate the operation/method.



---

## Related topics

### Basics

Modeling Communication According to Classic AUTOSAR.....	89
Working with Data Stores in TargetLink ( TargetLink Preparation and Simulation Guide)	

## Dynamically Accessing AUTOSAR Data Stores via Custom Code (Type II) Blocks

---

### Introduction

You can dynamically access data stores that have AUTOSAR specification via Custom Code (type II) blocks.

You model these accesses via generic modeling patterns as shown in the AR\_ARRAY\_OF\_STRUCT\_DATA demo model:

Read Access Modeling Pattern	Write Access Modeling Pattern
This Custom Code (type II) block has an index as the input and the selected array element as the output:	This Custom Code (type II) block has an index and an array element as the input:
Its custom code file looks as follows:	Its custom code file looks as follows:
<pre>/* fpx_output_begin */ data_i = data_store_signal[i]; /* fpx_output_end */</pre>	<pre>/* fpx_output_begin */ data_store_signal[i] = data_i; /* fpx_output_end */</pre>
During code generation, TargetLink generates direct accesses to the RTE variables, or the correct RTE API functions and access points from the following information:	<ul style="list-style-type: none"> <li>The AUTOSAR specification at the associated Data Store Memory block</li> <li>The context of the Custom Code (type II) block</li> </ul>

The following tables show the code patterns that are generated from the generic modeling patterns. Each code pattern shows the position of the RTE API call relative to the custom code. If different RTE API functions share the same semantics, only one code pattern is shown. For example, reading from a local buffer for a scalar could be implemented as `Rte_Read(var)` or `var = Rte_IRead()` but only one is shown.

### Code pattern for read access

Read Actual	Code Pattern	Annotation
Via local buffer	<code>Rte_Read(&amp;AuxVar);</code> <code>CustomCodeBlockOperation(AuxVar);</code>	—
Via pointer	<code>pReadAuxVar = Rte_IRead();</code> <code>CustomCodeBlockOperation(*pReadAuxVar);</code>	—

### Code pattern for write access

Write Actual	Code Pattern	Annotation
Via local buffer	<code>CustomCodeBlockOperation(AuxVar);</code> <code>Rte_Write(&amp;AuxVar);</code>	The Guaranteed complete write user assertion must be selected.
Via pointer	<code>pWriteAuxVar = Rte_IWriteRef();</code> <code>CustomCodeBlockOperation(*pWriteAuxVar);</code>	—

### Code pattern for read-write access

Read Actual	Write Actual	Code Pattern	Annotation
Via local buffer	Via local buffer	<pre>Rte_Read(&amp;AuxVar); CustomCodeBlockOperation(AuxVar); Rte_Write(&amp;AuxVar);</pre>	If both the Guaranteed complete write and the Guaranteed write before read user assertions are selected, TargetLink generates the following simpler pattern: <pre>CustomCodeBlockOperation(AuxVar); Rte_Write(&amp;AuxVar);</pre>
Via local buffer	Via pointer	<pre>ReadAuxVar = Rte_IRead(); pWriteAuxVar = Rte_IWriteRef(); CustomCodeBlockOperation(ReadAuxVar); *pWriteAuxVar = ReadAuxVar;</pre>	TargetLink uses only the local buffer from the custom code and generates an additional assignment trailing the custom code to ensure for data consistency.
Via pointer	Via local buffer	<pre>pReadAuxVar = Rte_IRead(); WriteAuxVar = *pReadAuxVar; CustomCodeBlockOperation(WriteAuxVar); Rte_IWrite(WriteAuxVar);</pre>	TargetLink uses only the local buffer from the custom code and generates an additional assignment preceding the custom code to ensure for data consistency. If both the Guaranteed complete write and the Guaranteed write before read user assertions are selected, TargetLink generates the following simpler pattern: <pre>CustomCodeBlockOperation(WriteAuxVar); Rte_IWrite(WriteAuxVar);</pre>
Via pointer	Via pointer	<pre>pReadAuxVar = Rte_IRead(); pWriteAuxVar = Rte_IWriteRef(); CustomCodeBlockOperation(*pReadAuxVar,     *pWriteAuxVar);</pre>	Used for non-scalar signals. You must provide a tuple of separate <a href="#">custom code symbols</a> for read and write access via the Data Stores edit field, e.g., <code>&lt;pReadAuxVar,pWriteAuxVar&gt;</code> .

### Limitations

- The optimized, index-based write access for NvData communication via Data Store Write blocks is not supported for Data Store Write access via Custom Code (type II) block.
- For Custom Code (type II) blocks whose property **Guaranteed complete write** is set to **off**, TargetLink does not support *write-only* access to data stores that are specified as follows:
  - Data stores with explicit AUTOSAR communication
  - Data stores with implicit AUTOSAR specification (IWrite)

**Related topics**

**Basics**

AR_ARRAY_OF_STRUCT_DATA (	 TargetLink Demo Models)
Basics on Defining the Interface of Custom Code Blocks (	 TargetLink Preparation and Simulation Guide)
Basics on Using TargetLink Data Stores with Custom Code (Type II) Blocks (	 TargetLink Preparation and Simulation Guide)
Basics on Working With Array of Structs When Modeling Classic AUTOSAR in TargetLink.....	37

# Creating Interfaces, Ports, and Communication Subjects

## Where to go from here

## Information in this section

How To Create Interfaces.....	100
How to Create Ports.....	101
How to Create Communication Subjects for Classic AUTOSAR Communication.....	103

## How To Create Interfaces

### Interfaces

Interfaces must be created for the following kinds of AUTOSAR communication:

AUTOSAR Communication Kind	DD <Interface> Object	Communication Subject
Client-server communication	/Pool/Autosar/Interfaces/<ClientServerInterface>	Operation with optional OperationArgument
NvData communication	/Pool/Autosar/Interfaces/<NvDataInterface>	NvDataElement
Parameter communication	/Pool/Autosar/Interfaces/<CalPrmInterface>	CalPrmElement
Sender-receiver communication	/Pool/Autosar/Interfaces/<SenderReceiverInterface>	DataElement

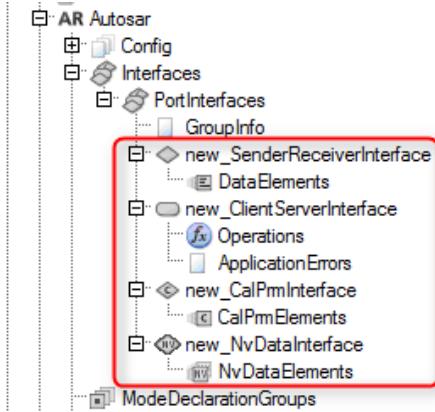
### Method

#### To create an interface

- 1 In the Data Dictionary Navigator, right-click the /Pool/Autosar/Interfaces/ subtree.
- 2 Choose Create InterfaceGroup from the context menu.
- 3 Enter a name for the interface group, e.g., **PortInterface**.
- 4 Right-click the interface group and choose Create GroupInfo from the context menu.
- 5 In the Property Value List, enter a package name, for example, **PortInterface**. This lets you export interfaces in the group to the specified package. Interfaces in AUTOSAR files located in the package can be imported to that location.
- 6 Right-click the interface group and choose Create <Interface> from the context menu.
- 7 Rename the <Interface> object created in step 6 as required.

**Result**

You created a DD <Interface> object that can be referenced by DD <Port> objects for communication. The following screenshot shows different DD <Interface> objects:

**Next steps**

AUTOSAR Communication Kind	Instructions
Client-server communication	<a href="#">How to Create Ports</a> on page 101
NvData communication	
Parameter communication	
Sender-receiver communication	
Interrunnable communication	<a href="#">How to Create Interrunnable Variables</a> on page 173

**Related topics****Basics**

[Overview of Communication According to Classic AUTOSAR.....](#) 90

## How to Create Ports

**Ports**

Software components communicate via connection points, which are called ports. Ports can be divided into provide ports, require ports, and provide-require ports:

<b>Mapping Between AUTOSAR Communication Kind, AUTOSAR Ports, and DD Port Objects</b>			
<b>AUTOSAR Communication Kind</b>	<b>Provide Port</b>	<b>Require Port</b>	<b>Provide-Require Port</b>
Sender-receiver communication	DD SenderPort object	DD ReceiverPort object	DD SenderReceiverPort object
Client-server communication	DD ServerPort object	DD ClientPort object	Not supported by TargetLink.
NvData communication	DD NvSenderPort object	DD NvReceiverPort object	DD NvSenderReceiverPort object
Parameter communication	DD ProvideCalPrmPort object <sup>1)</sup>	DD RequireCalPrmPort object	Not defined by AUTOSAR

<sup>1)</sup> Out of scope for TargetLink as a behavior modeling tool.

#### Preconditions

The following preconditions must be fulfilled:

- You created an SWC, such as MySWC. Refer to [How to Create Software Components](#) on page 66.
- You created an interface. Refer to [How To Create Interfaces](#) on page 100.

#### Method

##### To create a port

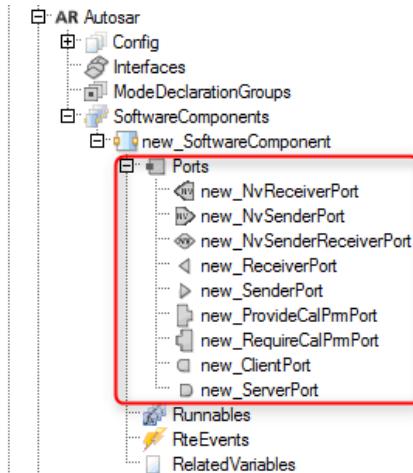
- 1 From the /Pool/Autosar/SoftwareComponents/<SoftwareComponent>/Ports/ object's context menu, select **Create <Port>**.
- 2 Rename the DD <Port> object created in step 1 as required.
- 3 In the DD <Port> object's Property Value List, locate the <InterfaceRef> property to reference a suitable <Interface> object:

<b>AUTOSAR Communication</b>	<b>&lt;InterfaceRef&gt; Property</b>	<b>&lt;Interface&gt; Object</b>
Sender-receiver communication	DD SenderReceiverInterfaceRef property	DD SenderReceiverInterface object
Client-server communication	DD ClientServerInterfaceRef property	DD ClientServerInterface object
NvData communication	DD NvDataInterfaceRef property	DD NvDataInterface object
Parameter communication	DD CalPrmInterfaceRef property	DD CalPrmInterface object

#### Result

You created a <Port> object and referenced a suitable <Interface> for communication.

The following screenshot shows the different <Port> objects:



## Next steps

You have to create the subject of communication. Refer to [How to Create Communication Subjects for Classic AUTOSAR Communication](#) on page 103.

## Related topics

### Basics

[Overview of Communication According to Classic AUTOSAR](#).....90

### HowTos

<a href="#">How To Create Interfaces</a> .....	100
<a href="#">How to Create Software Components</a> .....	66

## How to Create Communication Subjects for Classic AUTOSAR Communication

### Communication subjects

The following table shows the interfaces and communication subjects for several communication kinds as described by [Classic AUTOSAR](#) and where they are specified in the Data Dictionary:

AUTOSAR Communication Kind	Interface	Communication Subject <sup>1)</sup>
Client-server communication	DD ClientServerInterface object	<ClientServerInterface>/Operations/<Operation> <sup>2)</sup>
NvData communication	DD NvDataInterface object	<NvDataInterface>/NvDataElements/<NvDataElement>

AUTOSAR Communication Kind	Interface	Communication Subject <sup>1)</sup>
Parameter communication	DD CalPrmInterface object	<CalPrmInterface>/CalPrmElements/<CalPrmElement>
Sender-receiver communication	DD SenderReceiverInterface object	<SenderReceiverInterface>/DataElements/<DataElement>

<sup>1)</sup> The object path relative to /Pool/Autosar/Interfaces/.

<sup>2)</sup> Operations can have arguments. You specify the arguments of an operation as DD OperationArgument child objects.

The creation of communication subjects for interruptable communication is explained in [How to Create Interruptable Variables](#) on page 173.

#### Preconditions

The following preconditions must be fulfilled:

- You created a DD <Interface> object. Refer to [How To Create Interfaces](#) on page 100.
- You specified one or more types. Refer to:
  - [Creating Implementation Data Types from Scratch \(Classic AUTOSAR\)](#) on page 41
  - [Creating Application Data Types from Scratch \(Classic AUTOSAR\)](#) on page 49
  - [Using TargetLink's Wizards to Create Application Data Types or Implementation Data Types \(Classic AUTOSAR\)](#) on page 57
- You specified scalings and constrained range limits as required. Refer to [Defining Types, Scalings, and Constrained Range Limits \(Classic AUTOSAR\)](#) on page 31.

#### Method

##### To create a communication subject for Classic AUTOSAR communication

- 1 Select Create <CommunicationSubject> from one of the following objects' context menus:
  - .../<SenderReceiverInterface>/DataElements
  - .../<NvDataInterface>/NvDataElements/
  - .../<CalPrmInterface>/CalPrmElements/
  - .../<ClientServerInterface>/Operations/

##### Note

For client-server communication, the communication subject is comprised of an operation and its operation arguments. You have to create the following child objects for the .../<ClientServerInterface>/Operations/<MyOperation> object:

- .../<ClientServerInterface>/Operations/<MyOperation>/OperationArguments/
- .../<ClientServerInterface>/Operations/<MyOperation>/OperationArguments/<MyOperationArgument>

TargetLink creates a DD object for the communication subject.

- 2** Rename the DD object created in step 1 as required.
- 3** In the Property Value List, specify the following mandatory settings:

AUTOSAR Communication	Property	Value
Client-server communication	Kind <sup>1)</sup>	Select ARGIN for operation arguments that are input parameters of the operation or ARGOUT for operation arguments that are output parameters of the operation.
	Type <sup>1)</sup>	Select a user-defined data type. <sup>2)</sup>
	Width <sup>1)</sup>	If you select an array user type, you have to specify a consistent width. <sup>2)</sup>
NvData communication <sup>3)</sup>	Type	Select a user-defined data type. <sup>2)</sup>
	Width	If you select an array user type, you have to specify a consistent width.
Parameter communication	Type	Select a user-defined data type. <sup>2)</sup>
	Width	If you select an array user type, you have to specify a consistent width.
Sender-receiver communication	Type	Select a user-defined data type. <sup>2)</sup>
	ImplementationPolicy	For sender-receiver communication, you can control if received data elements are queued: Select <b>queued</b> if you want to model the queuing of received data elements.
	Width	If you select an array user type, you have to specify a consistent width.

<sup>1)</sup> At a DD .../<ClientServerInterface>/Operations/<MyOperation>/OperationArguments/<MyOperationArgument> object.

<sup>2)</sup> If you select a struct user type, you have to create components of the communication subject that are consistent with the selected struct user type. You can use the Adjust to Typedef context menu command to synchronize this communication subject with the referenced DD Typedef object.

<sup>3)</sup> TargetLink always sets the implementation policy of NvDataElements to **standard** during export.

## Result

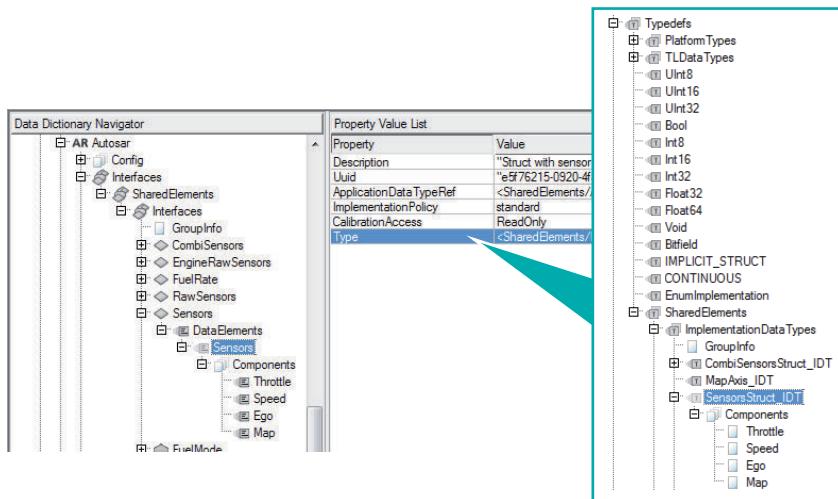
You created a communication subject for AUTOSAR communication.

## Next steps

AUTOSAR Communication	Instructions
Client-server communication	<a href="#">How to Specify Code Optimization Settings for Classic AUTOSAR Operations</a> on page 168
	<a href="#">Modeling Client-Server Communication</a> on page 133
NvData communication	<a href="#">Modeling NvData Communication</a> on page 178
Parameter communication	<a href="#">How to Model Parameter Communication for Calibration</a> on page 244
Sender-receiver communication	<a href="#">Modeling Sender-Receiver Communication</a> on page 108

### Example of a struct-based data element

The following illustration shows an example of a data element that is based on a struct type.



The data element has the Components object with data element components that are consistent with the struct type definition.

#### Note

- For the data element components, you have to select user types that are consistent with the struct user type definition.  
You can use the **Adjust to Typedef** context menu command to synchronize this communication subject with the referenced DD Typedef object.
- If you select an array user type for one of the components, you have to specify a consistent width at the component of the data type and the component of the data prototype.

Related topics

Basics

Basics on Client-Server Communication.....	133
Basics on Modeling Operation Calls on the Client Side of Client-Server Communication.....	136
Basics on NvData Communication.....	179
Basics on Preparing SWCs for Measurement and Calibration.....	235
Basics on Sender-Receiver Communication.....	108
Defining Types, Scalings, and Constrained Range Limits (Classic AUTOSAR).....	31

HowTos

How To Create Interfaces.....	100
-------------------------------	-----

References

[Adjust to Typedef](#) ( TargetLink Data Dictionary Manager Reference)

# Modeling Sender-Receiver Communication

## Where to go from here

## Information in this section

Basics on Sender-Receiver Communication.....	108
Example of Modeling Sender-Receiver Communication via Port Blocks.....	112
Example of Modeling Sender-Receiver Communication via Data Store Blocks.....	115
Basics on Communication Attributes and Acknowledgments.....	117
Basics on Signal Invalidation.....	119
How to Model Invalidation, Acknowledgment Notifications and Access to the RTE Status.....	122
How to Model Initialization of Data Elements.....	126
Basics on Checking the Update Flag of Data Elements in Explicit Sender-Receiver Communication.....	127
How to Model Checks of the Update Flag in Explicit Sender-Receiver Communication.....	130

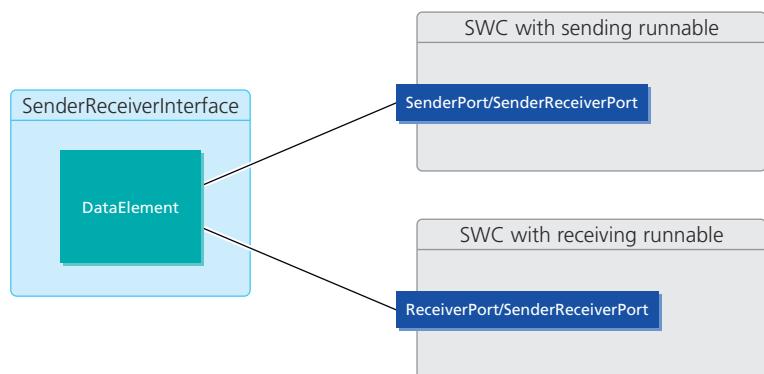
## Basics on Sender-Receiver Communication

### Sender-receiver communication

To model data exchange between SWCs, you can use sender-receiver communication.

### Sender-receiver communication according to Classic AUTOSAR

The illustration below shows sender-receiver communication between SWCs.



In sender-receiver communication, a sending runnable provides data and a receiving runnable requires it. The involved buffers depend on the kinds of ports that are used for communication:

- Separate buffer for reading and writing:
  - Require port
  - Provide port
- Combined buffer for reading and writing:
  - Provide-require port

## **Sender-receiver communication in TargetLink**

TargetLink supports sender-receiver communication as described above and lets you specify sender-receiver communication in the Data Dictionary.

**Communication mode** TargetLink supports [explicit](#) and [implicit](#) sender-receiver communication.

**Creating sending-receiver interfaces** You can create DD SenderReceiverInterface objects in the /Pool/Autosar/Interfaces/ subtree.

For instructions, refer to [How To Create Interfaces](#) on page 100.

**Creating sender-receiver ports** You can create ports for sender-receiver communication in the /Pool/Autosar/SoftwareComponents/<MySWC>/Ports subtree. The following DD <Port> objects are available:

Kind of Access	DD <Port> Object
Read	ReceiverPort
Write	SenderPort
Read-write	SenderReceiverPort

For instructions, refer to [How to Create Ports](#) on page 101.

**Creating data elements** You can create DD DataElement objects in the /Pool/Autosar/Interfaces/<SenderReceiverInterface>/DataElements/ subtree. TargetLink lets you create [data elements](#) with a simple structure or with a composite structure. [Data semantics](#), [event semantics](#), [implicit communication](#), and [explicit communication](#) are possible.

For instructions, refer to [How to Create Communication Subjects for Classic AUTOSAR Communication](#) on page 103.

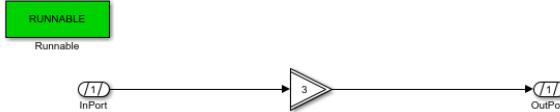
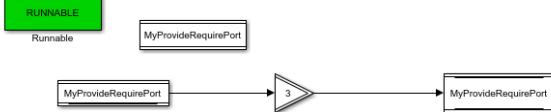
**Initializing data elements** TargetLink lets you reference variables that represent initialization constants at communication specifications for initializing data elements by the RTE. For instructions, refer to [How to Model Initialization of Data Elements](#) on page 126.

**Note****Avoid ambiguous initializations of data prototypes**

Specify an initialization value for each [data prototype](#) within the Data Dictionary whenever possible. This avoids ambiguities between [Classic AUTOSAR](#) initialization values and initialization values of model elements that are required by Simulink.

### Sender-receiver communication in the model

TargetLink offers various approaches for modeling sender-receiver communication. You can combine the following approaches as required:

Modeling Approaches	
Port blocks	Data Store Memory blocks
	
Supported Access Kinds	
<ul style="list-style-type: none"> <li>▪ Read</li> <li>▪ Write</li> <li>▪ Read-Write</li> </ul>	
Instructions	
Example of Modeling Sender-Receiver Communication via Port Blocks on page 112.	Example of Modeling Sender-Receiver Communication via Data Store Blocks on page 115.

**Tip**

You can model ports of software components with sender-receiver interfaces using SWC SenderPort and SWC ReceiverPort blocks. This is most useful for visualizing data flow between SWCs if your model contains more than one SWC.

### Simulation differences

Due to Simulink semantics, differences between MIL and SIL simulation modes might occur. This happens when read operations and write operations are performed on the *same* buffer during the *same* simulation step.

These situations can occur when you model your sender-receiver accesses by using port blocks and all of the following conditions apply:

- The same DD SenderReceiverPort is used at different blocks.
- The same DD DataElement is used at different blocks.
- A modeled write access precedes a read access modeled via a port block.

If you need consistent simulation behavior across the different simulation modes, model your sender-receiver accesses by using Data Store Memory blocks.

## Sender-receiver communication in production code

The production code generated for sender-receiver communication looks like this:

### Explicit communication

#### Read Access

With [data semantics](#):

```
Rte_Read_MyReceiverPort_MyDataElement(&MyDataElement);
```

With [event semantics](#):

```
Rte_Receive_MyReceiverPort_MyDataElement(&MyDataElement);
```

#### Write Access

With [data semantics](#):

```
Rte_Write_MySenderReceiverPort_MyDataElement(MyDataElement * 3);
```

With [event semantics](#):

```
Rte_Send_MySenderPort_MyDataElement(MyDataElement * 3);
```

### Implicit communication

#### Read Access

```
Sb1_Gain = Rte_IRead_MyRunnable_MyReceiverPort_MyDataElement() * 3;
```

#### Write Access

With [Rte\\_IWrite](#):

```
Rte_IWrite_MyRunnable_MySenderPort_MyDataElement(Sb1_Gain);
```

With [Rte\\_IWriteRef](#):

```
p_MyDataElement = Rte_IWriteRef_Runnable_MySenderPort_MyDataElement();
```

...

```
*p_MyDataElement = Sb1_Gain;
```

**Related topics****Basics**

Basics on Checking the Update Flag of Data Elements in Explicit Sender-Receiver Communication.....	127
Basics on Communication Attributes and Acknowledgments.....	117
Basics on Signal Invalidation.....	119

**HowTos**

How to Create Communication Subjects for Classic AUTOSAR Communication.....	103
How To Create Interfaces.....	100
How to Create Ports.....	101
How to Model Checks of the Update Flag in Explicit Sender-Receiver Communication.....	130
How to Model Initialization of Data Elements.....	126
How to Model Invalidation, Acknowledgment Notifications and Access to the RTE Status.....	122

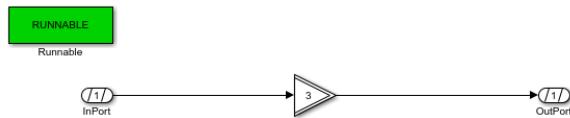
**Examples**

Example of Modeling Sender-Receiver Communication via Port Blocks.....	112
--	-----

## Example of Modeling Sender-Receiver Communication via Port Blocks

**Model overview**

This example uses the following simple model:

**Preconditions**

The following preconditions must be fulfilled:

- You created a DD SenderReceiverInterface object, such as `MySenderReceiverIF`.
- You created a DD DataElement object, such as `MyDataElement`.
- You created a DD SoftwareComponent object, such as `MySwc`.
- You created a DD ReceiverPort and a DD SenderPort object, such as `MyReceiverPort` and `MySenderPort`.
- You created a DD Runnable object, such as `MyRunnable`.

**Method****To model sender-receiver communication via a port block**

- 1 In the model, open the subsystem of MyRunnable.
- 2 Add the following blocks, depending on the data type of MyDataElement:
  - An InPort or Bus Import block
  - An OutPort or Bus Outport block
- 3 On the AUTOSAR page of the InPort/Bus Import block, make the following settings:

Control	Value
AUTOSAR mode	Classic
Kind	Sender-Receiver
Mode	As required
Port	MyReceiverPort
Data element	MyDataElement

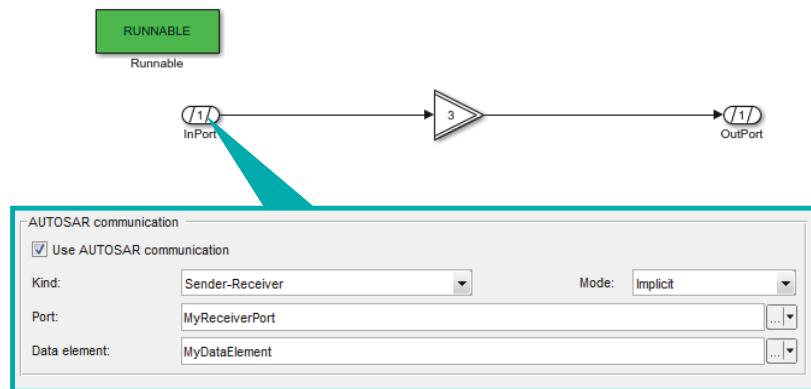
- 4 On the AUTOSAR page of the OutPort or Bus Outport block, make the following settings:

Control	Value
AUTOSAR mode	Classic
Kind	Sender-Receiver
Mode	As required
Port	MySenderPort
Data element	MyDataElement

- 5 Model the runnable's function as required.
- 6 Generate code.

**Result**

You modeled sender-receiver communication by using port blocks. The settings in the block dialog look like this:



The runnable's code looks like this:

## Explicit communication

### Read Access

With [? data semantics:](#)

```
Rte_Read_MyReceiverPort_MyDataElement(&MyDataElement);
```

With [? event semantics:](#)

```
Rte_Receive_MyReceiverPort_MyDataElement(&MyDataElement);
```

### Write Access

With [? data semantics:](#)

```
Rte_Write_MySenderPort_MyDataElement(MyDataElement * 3);
```

With [? event semantics:](#)

```
Rte_Send_MySender_MyDataElement(MyDataElement * 3);
```

## Implicit communication

### Read Access

```
Sb1_Gain = Rte_IRead_MyRunnable_MyReceiverPort_MyDataElement() * 3;
```

### Write Access

With [Rte\\_IWrite:](#)

```
MyDataElement = Sb1_Gain;
Rte_IWrite_MyRunnable_MySenderReceiverPort_MyDataElement(MyDataElement);
```

With [Rte\\_IWriteRef:](#)

```
p_MyDataElement = Rte_IWriteRef_Runnable_MySenderReceiverPort_MyDataElement();
...
*p_MyDataElement = Sb1_Gain;
```

## Related topics

### Basics

<a href="#">Basics on Sender-Receiver Communication</a> .....	108
---	-----

### HowTos

<a href="#">How to Create Communication Subjects for Classic AUTOSAR Communication</a> .....	103
<a href="#">How To Create Interfaces</a> .....	100
<a href="#">How to Create Ports</a> .....	101
<a href="#">How to Create Software Components</a> .....	66
<a href="#">How to Model Runnables</a> .....	71

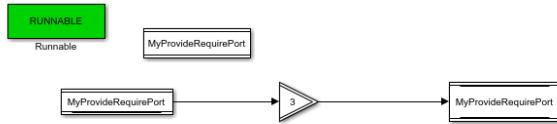
### References

<a href="#">Bus Import Block</a> (  <a href="#">TargetLink Model Element Reference</a> )
<a href="#">Bus Outport Block</a> (  <a href="#">TargetLink Model Element Reference</a> )
<a href="#">InPort Block</a> (  <a href="#">TargetLink Model Element Reference</a> )
<a href="#">OutPort Block</a> (  <a href="#">TargetLink Model Element Reference</a> )

## Example of Modeling Sender-Receiver Communication via Data Store Blocks

### Model overview

This example uses the following simple model:



### Preconditions

The following preconditions must be fulfilled:

- You created a DD SenderReceiverInterface object, such as `MySenderReceiverIF`.
- You created a DD DataElement object, such as `MyDataElement`.
- You created a DD SoftwareComponent object, such as `MySwc`.
- You created a DD SenderReceiverPort object, such as `MySenderReceiverPort`.
- You created a DD Runnable object, such as `MyRunnable`.

### Method

#### To model sender-receiver communication via data store blocks

1 In the model, open the subsystem of `MyRunnable`.

2 Add the following blocks:

- A Data Store Memory block

#### Tip

Data Store Memory blocks can also be created and specified in the Data Dictionary, refer to [How to Create a Global Data Store via Data Store Memory Blocks](#) ( [TargetLink Preparation and Simulation Guide](#)).

- A Data Store Read block
- A Data Store Write block

3 On the AUTOSAR page of the Data Store Memory block, make the following settings:

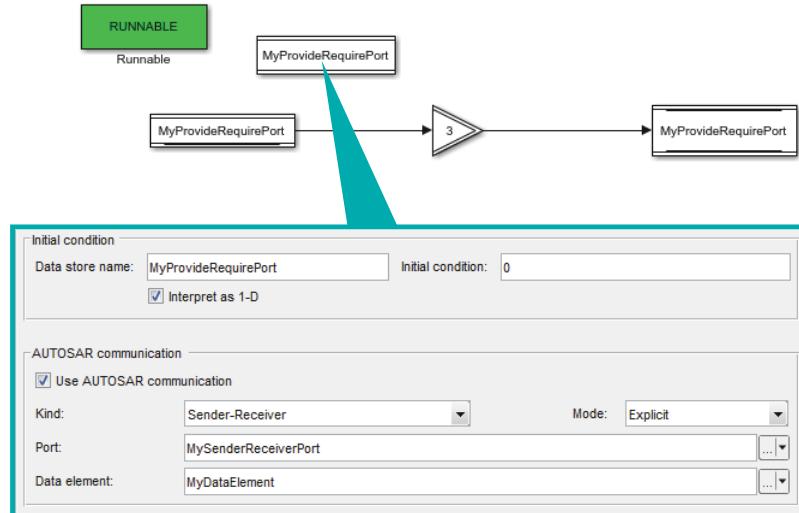
Control	Value
AUTOSAR mode	Classic
Kind	Sender-Receiver
Mode	As required
Port	<code>MySenderReceiverPort</code>
Data element	<code>MyDataElement</code>

Close the block dialog to confirm your settings.

- 4 Model the runnable's function as required.
- 5 Generate code.

**Result**

You modeled sender-receiver communication by using data store blocks. The settings in the block dialog look like this:



The runnable's code looks like this:

**Explicit communication****Read Access**

With [data semantics](#):

```
Sa1_Gain = MyDataElement * 3;
Rte_Read_MySenderReceiverPort_MyDataElement(&MyDataElement);
```

With [event semantics](#):

```
MyDataElement = Sa1_Gain;
Rte_Receive_MySenderReceiverPort_MyDataElement(&MyDataElement);
```

**Write Access**

```
Rte_Write_MySenderReceiverPort_MyDataElement(MyDataElement * 3);
```

**Implicit communication****Read**

```
MyDataElement = Rte_IRead_MyRunnable_MySenderReceiverPort_MyDataElement();
Sd1_Gain = MyDataElement * 3;
```

**Write**With `Rte_IWrite`:

```
MyDataElement = Sd1_Gain;
Rte_IWrite_MyRunnable_MySenderReceiverPort_MyDataElement(MyDataElement);
```

With `Rte_IWriteRef`:

```
p_MyDataElement = Rte_IWriteRef_MyRunnable_MySenderReceiverPort_MyDataElement();
...
*p_MyDataElement = Sd1_Gain;
```

**Related topics****Basics**

<a href="#">Basics on Sender-Receiver Communication</a> .....	108
---	-----

**HowTos**

<a href="#">How to Create Communication Subjects for Classic AUTOSAR Communication</a> .....	103
<a href="#">How To Create Interfaces</a> .....	100
<a href="#">How to Create Ports</a> .....	101
<a href="#">How to Create Software Components</a> .....	66
<a href="#">How to Model Runnables</a> .....	71

**References**

<a href="#">Data Store Memory Block</a> ( <a href="#">TargetLink Model Element Reference</a> )
<a href="#">Data Store Read Block</a> ( <a href="#">TargetLink Model Element Reference</a> )
<a href="#">Data Store Write Block</a> ( <a href="#">TargetLink Model Element Reference</a> )

## Basics on Communication Attributes and Acknowledgments

**Introduction**

Communication attributes and acknowledgments let you manage communication.

**Communication specifications**

The data flow of sender-receiver communication is characterized by the communication attributes of the participating ports. The communication attributes are grouped in communication specifications. The table below describes the communication attributes of the sender-receiver communication-related ports.

Communication Attribute	Description
Data receiver communication specification	
AliveTimeOut	Lets you specify a period after which the receiver is notified if a sent data element has not been received.
HandleDataStatus	Lets you specify whether the RTE shall provide via <code>Rte_IStatus</code> the status for the data element read in implicit communication.
InitValueRef	Lets you select a Variable object that specifies the initial receive value. The RTE initializes the software components during startup.
ResyncTime	Specifies the period in which a valid value has to arrive at the receiver if data elements get lost during communication.
HandleInvalid	Lets you select whether a received value that is marked by the RTE as invalid is replaced by the invalid value (REPLACE), or whether the old signal value is kept (KEEP).
Event receiver communication specification	
QueueLength	Lets you specify the number of event messages that can be present in the receiving <i>first-in-first-out</i> queue.
Data sender communication specification	
CanInvalidate	Lets you select whether the sender can mark a sent data element as invalid. <a href="#">Classic AUTOSAR</a> has defined the invalidate RTE API function for this purpose. You can configure how the data receiver handles invalid data elements.
InitValueRef	Lets you select a Variable object that specifies the initial send value.
Event sender communication specification	
No communication attribute defined	-

**Acknowledgments**

A sender can request an acknowledgment of the send operation. [Classic AUTOSAR](#) has defined the feedback RTE API function for this purpose. You can create acknowledgments as objects of every data and event sender communication specification. The table below lists the attributes that you can specify for acknowledgments.

Acknowledgment Attribute	Description
Timeout	The <code>Rte_Feedback</code> API function returns an error if the timeout expires without an acknowledgment of the send operation. Specifies the maximum time the sender runnable's execution is blocked if you have selected <b>BLOCKING</b> behavior.

**Related topics****Basics**

Basics on Signal Invalidation.....	119
Comprehensive Modeling of RTE Code Pattern.....	374

## Basics on Signal Invalidation

**Signal invalidation**

Signal invalidation can be used to communicate that a signal is invalid.

**Signal invalidation according to Classic AUTOSAR**

In [Classic AUTOSAR](#), signals sent via sender-receiver communication can be invalidated by a sender, according to an invalidation policy. If a signal is invalidated, an invalid value is sent instead of the signal's original value.

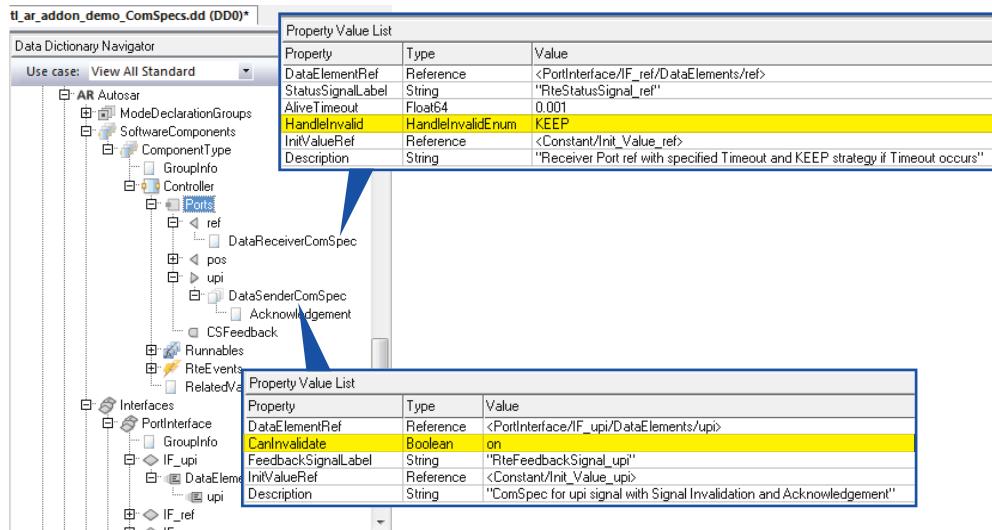
For details, refer to the *Software Component Template* document at <https://www.autosar.org/standards/classic-platform/>.

**Signal invalidation in TargetLink**

**Setting the invalidation policy** In TargetLink, you specify the invalidation policy at DD DataSenderComSpec/DataReceiverComSpec objects, referenced at SenderComSpec/ReceiverComSpec blocks.

At the DD DataSenderComSpec object, specify the CanInvalidate property.

At the DD DataReceiverComSpec object, specify the DD HandleInvalid property.

**Specifying invalid values**

In TargetLink, you specify invalid values at DD Typedef objects.

For [Classic AUTOSAR](#) releases older than AUTOSAR release 4.2.1 you specify invalidation values at the primitive data type.

As of [Classic AUTOSAR](#) release 4.2.1 you can specify invalidation values according to the following table:

Type	Invalid Value Specification	Description
Scalar	As the value of the InvalidValue property.	The value to be used as the invalid value.
	As the value of the InvalidValueToken <sup>1)</sup> property.	You can specify the invalid value as an element of the ConversionStrings property. This element is substituted by the corresponding element of the ConversionTable property that is used as the invalid value.
Vector	At vector type's InvalidValue property as vector	The vector to be used as the invalid value.
	At vector type's InvalidValue property as scalar	The invalid value used for each element of the vector.
	At scalar type as described above	
Matrix	At matrix type's InvalidValue property as matrix	The matrix to be used as the invalid value.
	At matrix type's InvalidValue property as scalar	The invalid value used for each element of the matrix.
	At scalar type as described above	
Struct	As the values of the components of a structured variable that is based on this struct type and that is referenced at the struct type's InvalidValueRef property.	The invalid value for each component of the structured data type is taken from the corresponding component of the referenced struct variable.
	Via a struct variable referenced at the InvalidValueRef property of a struct component's type that is itself struct-based.	
	At the InvalidValue or InvalidValueToken <sup>1)</sup> property of a struct component's type that is not struct-based.	As for scalars, vectors, and matrices.

<sup>1)</sup> Restricted to plain or Boolean data types whose scaling object's ConversionType property is set to TAB\_VERB.

## Specifying constraints

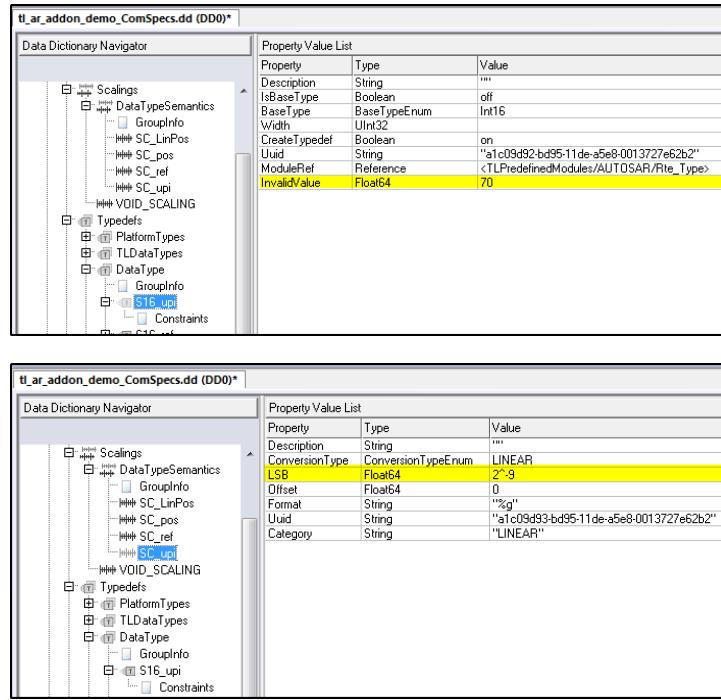
Provide the value of the InvalidValue property as a numerical value without scaling, as expected by the RTE.

When determining the invalid value for a particular data element, TargetLink scales this value according to the constraints specified at the type definition of the data element.

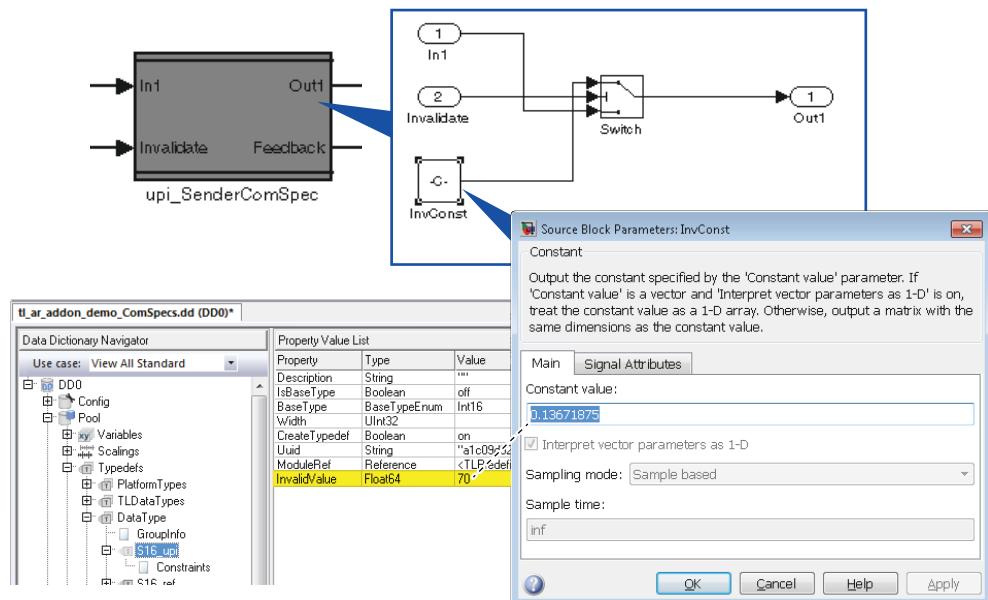
If you specify constraints at *both* the type definition *and* the data element, TargetLink chooses the constraints specified at the data element. If you do not specify the value of the InvalidValue property even though it is required, a message is displayed during the validation of the Data Dictionary.

**Example**

The following example shows the specification of an invalid value at a scalar data type:

**Modeling signal invalidation**

The following screenshot shows what is under the mask of the SenderComSpec block:



As you can see, the value of the InvConst block is the value provided at the DD InvalidValue property, scaled according to the constraints specified at the DD Typedef object, namely:  $70 \cdot 2^{-9} = 0.13671875$ .

For instructions on modeling signal invalidation, refer to [How to Model Invalidation, Acknowledgment Notifications and Access to the RTE Status](#) on page 122.

## Related topics

### Basics

Basics on Communication Attributes and Acknowledgments.....	117
Basics on Sender-Receiver Communication.....	108
Defining Types, Scalings, and Constrained Range Limits (Classic AUTOSAR).....	31

### HowTos

How to Model Invalidation, Acknowledgment Notifications and Access to the RTE Status.....	122
How to Simulate Signal Invalidation.....	303

## How to Model Invalidation, Acknowledgment Notifications and Access to the RTE Status

### Objective

To model access to RTE API functions for invalidation, acknowledgment notifications, and access to the status of sender-receiver communication.

#### Note

The instructions below show how to create a *data sender* communication specification. Data receiver, event receiver, and event sender communication specifications are created in a similar way.

### Preconditions

The following preconditions must be fulfilled:

- You have modeled sender-receiver communication. For instructions, refer to [Example of Modeling Sender-Receiver Communication via Port Blocks](#) on page 112.

### Method

#### To model invalidation, acknowledgment notifications and access to the RTE status

- In the Data Dictionary Navigator, navigate to the following subtree:

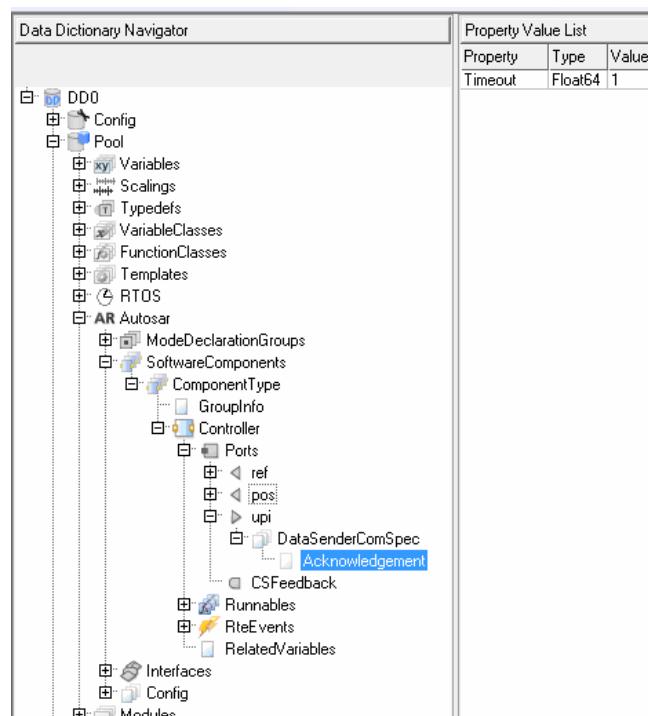
/Pool/Autosar/SoftwareComponents/<SoftwareComponent>  
 /Ports/<Port>

- 2 Right-click the Port object and select Create DataSenderComSpec as required.
- 3 In the Property Value List, use the DataElementRef property to reference a DD DataElement object.  
 Make sure that the DD DataElement object is contained in the SenderReceiverInterface object that is referenced at the DD <Port> object that parents the DD <ComSpec> object.
- 4 Specify the communication attributes for the data flow of the referenced data element.

#### 5 Note

If you want to create an acknowledgment, you have to perform the step below. Acknowledgments can only be created as objects of data or event sender communication specifications.

Right-click the DataSenderComSpec object, select Create Acknowledgment from the context menu and specify the acknowledgment's properties in the Property Value List.



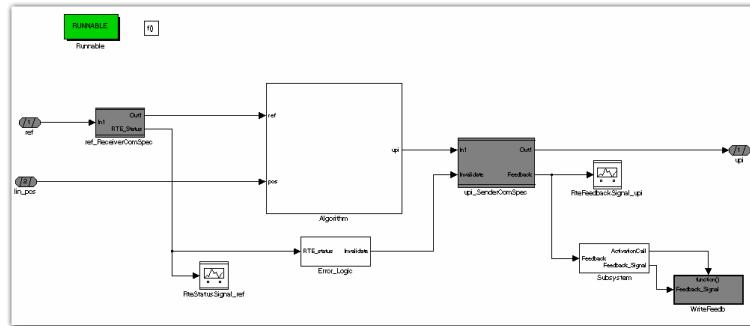
- 6 To the model, add a SenderComSpec block from the TargetLink Classic AUTOSAR Block Library directly before the OutPort/Bus Outport block that transfers the data element.

- 7 In the block dialog, select Add invalidate port, Add feedback port, and Add status port as required.

#### Note

- Select Add invalidate port only if you have specified the following DD properties:
- The CanInvalidate property of the DD DataSenderComSpec object as on
- The InvalidValue property of the DD Typedef object referenced at the data element.
- Select Add feedback port only if you have created an acknowledgment and you are modeling explicit sender-receiver communication.

- 8 Connect matching signals to the selected ports of the SenderComSpec block.



#### Result

You have created a communication specification with an acknowledgment for a data element. You have modeled access to the invalidate and feedback API.

TargetLink lets you generate code for sender-receiver communication such as the following to the specified module:

```
/* # InvalidateVariable denotes the signal of the invalidate inport of the SenderComSpec Block # */
if (InvalidateVariable != RTE_E_OK) {
    Rte_Invalidate_my_SenderPort_my_DataElement();
}
else {
    ...
    /* # combined # TargetLink output: Subsystem/Atomic Subsystem/OutPort */
    Rte_Write_my_SenderPort_my_DataElement(my_DataElement);
}
/* Feedback denotes the output of the SenderComSpec block's Feedback output port */
Feedback = Rte_Feedback_my_SenderPort_my_DataElement();
...
```

In the software component's code, the condition for signal invalidation is checked. Either the signal is invalidated or a standard explicit send operation is called and the status is stored in a variable for further processing. After sending or invalidating the signal, feedback is requested by the RTE.

You can find the function prototype of the `Rte_Invalidate` API function in the `RTE_<SWC>.c` file, which belongs to the stub RTE. Simulation frame files are located in the folder for simulation frame code files defined via the DD ProjectFolder and FolderStructure objects.

```
Std_ReturnType Rte_Invalidate_my_SenderPort_my_DataElement(void)
{
    Rte_Controller_my_SenderPort_my_DataElement = InvalidValue_my_datatype;
    return RTE_E_OK;
}
...
```

Invalidation takes place by assigning the invalid value to the output variable of the `SenderComSpec` block.

## Next steps

If you want to simulate a non-constant RTE status or feedback, you have to provide a source for the RTE status or acknowledgment notification. For instructions, refer to [How to Simulate the RTE Status](#) on page 299.

## Related topics

### Basics

Basics on Communication Attributes and Acknowledgments.....	117
Basics on Sender-Receiver Communication.....	108
Basics on Signal Invalidation.....	119
Defining Types, Scalings, and Constrained Range Limits (Classic AUTOSAR).....	31

### HowTos

<a href="#">How to Simulate the RTE Status</a> .....	299
--	-----

### Examples

<a href="#">Example of Modeling Sender-Receiver Communication via Port Blocks</a> .....	112
---	-----

## How to Model Initialization of Data Elements

### Objective

To initialize a data element by an initialization constant.

#### Note

##### Avoid ambiguous initializations of data prototypes

Specify an initialization value for each [data prototype](#) within the Data Dictionary whenever possible. This avoids ambiguities between [Classic AUTOSAR](#) initialization values and initialization values of model elements that are required by Simulink.

### Preconditions

You must have created a communication specification. For instructions, refer to [How to Model Invalidation, Acknowledgment Notifications and Access to the RTE Status](#) on page 122.

### Method

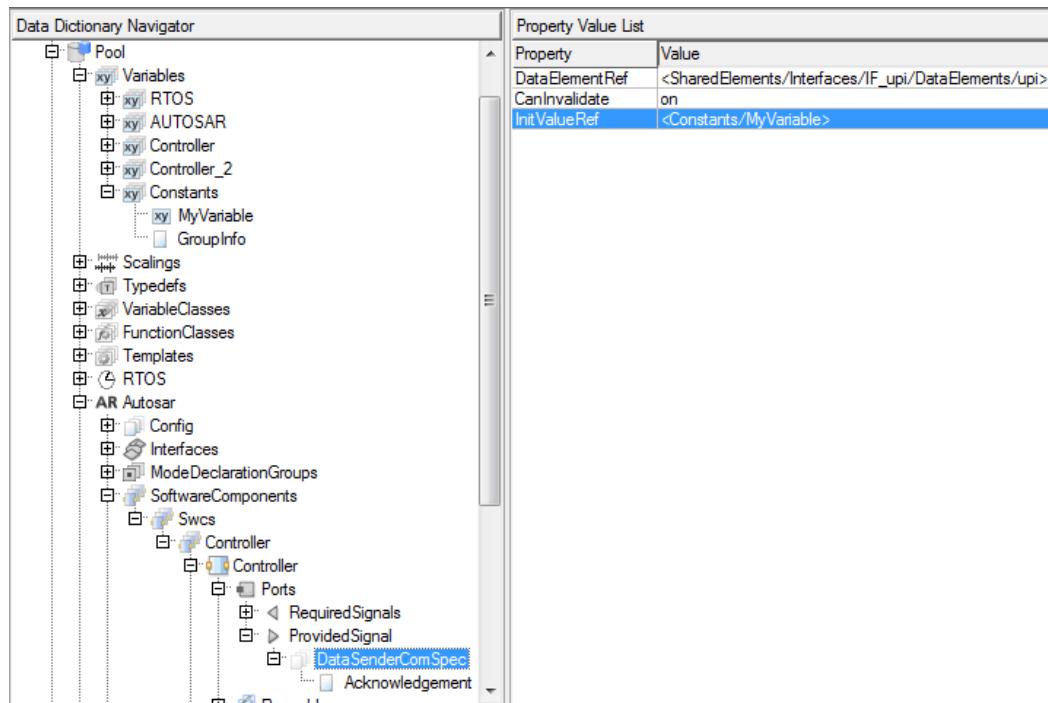
#### To model initialization of data elements

- 1 In the Data Dictionary Navigator, create a variable group for the initialization constants and name it **Constants**.
- 2 From the context menu, choose **Create GroupInfo**.
- 3 In the Property Value List, enter a package name, for example, **Constants**. This lets you export variables in the group to the specified package. Constants in AUTOSAR files located in the package can be imported to that location.
- 4 From the context menu, choose **Create Variable**.
- 5 Enter a name for the variable, for example, **my\_InitializationConstant**.
- 6 In the Property Value List, specify the following settings:

Property	Value
Type	Select the user-defined type that is consistent with the data element.
Value	Specify the initialization constant value, for example, 0.
VariableClass	Specify the variable class. The recommended class depends on the data type: <ul style="list-style-type: none"> <li>▪ Non structured AUTOSAR constants - <b>AUTOSAR/CONST_VALUE</b></li> <li>▪ Structured AUTOSAR constants - <b>AUTOSAR/CONST_STRUCT</b></li> <li>▪ Components of a struct - <b>AUTOSAR/CONST_STRUCT_COMPONENT</b></li> </ul>

- 7 Navigate to the communication specification of the data element that you want to initialize.

- 8** In the Property Value List of the data element, select the variable via the InitValueRef Browse button.

**Result**

You have specified an initialization constant for a data element. The initialization is performed by the RTE.

**Related topics****HowTos**

How to Model Invalidation, Acknowledgment Notifications and Access to the RTE Status..... 122

## Basics on Checking the Update Flag of Data Elements in Explicit Sender-Receiver Communication

**Introduction**

In [explicit sender-receiver communication](#) with [data semantics](#), runnables can check whether a data element for a specific [receiver port](#) or [sender-receiver port](#) changed since it was last read.

**Update flag according to  
Classic AUTOSAR**

In [Classic AUTOSAR](#), the RTE provides an update flag for combinations of [receiver ports](#) or [sender-receiver ports](#) and data elements with [data semantics](#). This flag allows for checking whether the data element was updated since it was last read. A runnable can check this flag if the ENABLE-UPDATE element of the nonqueued receiver comspec is set to true.

**Checking the update flag** A runnable can check the update flag via the `Rte_IsUpdated` RTE API function whose return value is a Boolean.

**Accessing the update flag in  
TargetLink**

TargetLink lets you generate code that contains calls to the `Rte_IsUpdated` API function.

**Specifying to use the update flag in the Data Dictionary** You enable the flag use by setting the EnableUpdate property of the following DD object to on:  
`/Pool/Autosar/SoftwareComponents/<MySwc>/Ports/<MyReceiverPort>/DataReceiverComSpec`

**Modeling checks of the  
update flag**

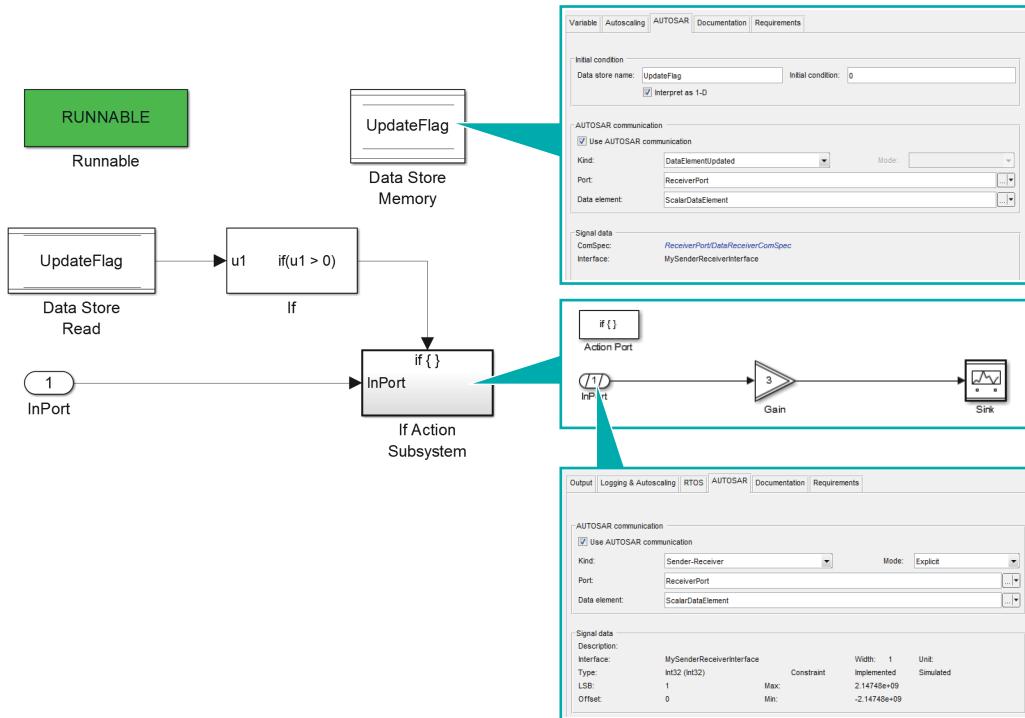
You model checks of the update flag via the following modeling elements:

Modeling Element	Location
Data Store Memory <sup>1)</sup>	Within the same <a href="#">code generation unit (CGU)</a> as the <a href="#">Runnable Subsystems</a> , depending on your use case
Data Store Read	<ul style="list-style-type: none"> <li>▪ Inside the runnable subsystem that models the runnable which checks the update flag.</li> <li>▪ Outside of the runnable subsystem.<sup>2)</sup></li> </ul>
Data Store Write	Outside the runnable subsystems <sup>2)</sup>
InPort or Data Store Read	Inside the runnable subsystem to model read access to the data element

<sup>1)</sup> Not required if a Data Store Memory block is specified in the Data Dictionary, refer to [How to Create a Global Data Store via Data Store Memory Blocks](#) ([TargetLink Preparation and Simulation Guide](#)).

<sup>2)</sup> Only for simulation purposes

**Modeling example** The following illustration shows an example for modeling the check of an update flag:



### Update flag in production code

Production code containing a call to the `Rte_IsUpdated` API function might look like this:

```
FUNC(void, MySwc_CODE) MyRunnable(void)
{
    if (Rte_IsUpdated_ReceiverPort_ScalarDataElement()) {
        sint32 ScalarDataElement;
        sint16 Sa2_Sink;
        Rte_Read_ReceiverPort_ScalarDataElement(&ScalarDataElement);
        Sa2_Sink = ((sint16) ScalarDataElement) * 3;
    }
}
```

### Related topics

#### Basics

[Basics on Sender-Receiver Communication.....](#) 108

#### HowTos

[How to Model Checks of the Update Flag in Explicit Sender-Receiver Communication.....](#) 130

## How to Model Checks of the Update Flag in Explicit Sender-Receiver Communication

**Restriction** Modeling checks of the update flag is restricted to [explicit sender-receiver communication](#) with [data semantics](#).

**Precondition** You modeled explicit sender-receiver communication. Refer to [Example of Modeling Sender-Receiver Communication via Port Blocks](#) on page 112 or [Example of Modeling Sender-Receiver Communication via Data Store Blocks](#) on page 115.

**Method** **To model checks of the update flag in explicit sender-receiver communication**

- 1 In the Data Dictionary, locate the DD ReceiverPort or SenderReceiverPort object used for the data element whose update flag you want to check.
- 2 Add a DD DataReceiverComSpec object to its subtree.
- 3 Set its EnableUpdate property to on.
- 4 To your model add a Data Store Memory block. Place it in the [code generation unit \(CGU\)](#) that contains the [Runnable subsystem](#) of the runnable you want to check the update flag.

The actual location depends on your use case.

**Tip**

Data Store Memory blocks can also be created and specified in the Data Dictionary, refer to [How to Create a Global Data Store via Data Store Memory Blocks](#) ([TargetLink Preparation and Simulation Guide](#)).

- 5 On the block's AUTOSAR page make the following settings:

Property	Value
Data store name	<UpdateFlag>
AUTOSAR mode	Classic
Kind	DataElementUpdated
Port <sup>1)</sup>	The DD ReceiverPort or SenderReceiverPort object that references the DD SenderReceiverInterface object which contains the DD DataElement object.
Data Element <sup>1)</sup>	Lets you select a DD DataElement object belonging to the DD SenderReceiverInterface object that is referenced at the selected DD ReceiverPort or SenderReceiverPort object.

<sup>1)</sup> Must match the modeling referred to in the precondition.

- 6 To your runnable subsystem, add a Data Store Read block, and specify it as follows:

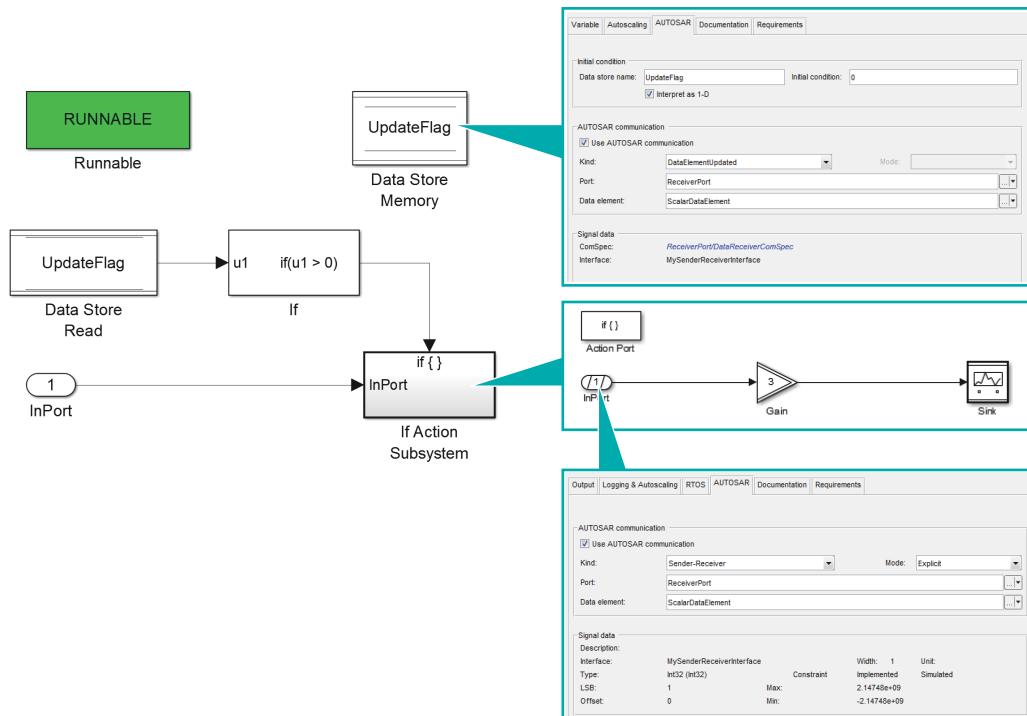
Model Element	Property	Value
Data Store Read	Data store name	<UpdateFlag>

- 7 Model the check of the update flag as required.

- 8 Generate code.

## Result

You modeled a check of an update flag for a combination of a receiver port or sender-receiver port and a data element with data semantics. Your runnable might look like this:



Production code containing a call to the `Rte_IsUpdated` API function might look like this:

```
FUNC(void, MySwc_CODE) MyRunnable(void)
{
    if (Rte_IsUpdated_ReceiverPort_ScalarDataElement()) {
        sint32 ScalarDataElement;
        sint16 Sa2_Sink;
        Rte_Read_ReceiverPort_ScalarDataElement(&ScalarDataElement);
        Sa2_Sink = ((sint16) ScalarDataElement) * 3;
    }
}
```

**Related topics**

**Basics**

Basics on Checking the Update Flag of Data Elements in Explicit Sender-Receiver Communication..... 127

**Examples**

Example of Modeling Sender-Receiver Communication via Port Blocks..... 112

**References**

Bus Import Block ( TargetLink Model Element Reference)

InPort Block ( TargetLink Model Element Reference)

# Modeling Client-Server Communication

## Where to go from here

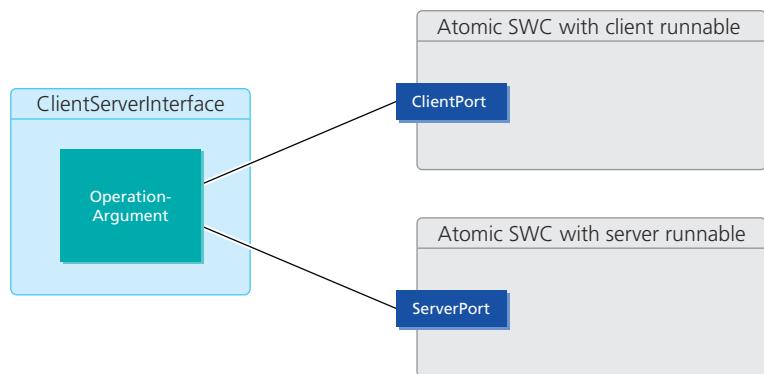
## Information in this section

Introduction.....	133
Modeling the Client Side of Client-Server Communication.....	135
Modeling the Server Side of Client-Server Communication.....	151
Specifying Code Optimization Settings for Classic AUTOSAR Operations.....	168

## Introduction

## Basics on Client-Server Communication

### Schematic of client-server communication



### Client-server communication according to Classic AUTOSAR

#### Operation arguments

Operation arguments have the following characteristics:

- Operation arguments can be derived from scalar, array and struct types.
- An operation of a client-server interface has a defined number of ARGIN and/or ARGOUT arguments. Parameters are passed to the server by the ARGIN arguments. The ARGOUT arguments are used to transfer the result of the server back to the client.

#### Application errors

Application errors of the server are transferred via the return value of the server runnable.

**Synchronous and asynchronous server calls** The client can call the server in the following two ways:

- In synchronous client-server communication, the client runnable is blocked and waits for the server after the server call was issued via `Rte_Call`. The server runnable that provides the operation in question is triggered by an RTE event of the `OPERATION_INVOKED_EVENT` type. The server returns the result to the client runnable via the RTE within the same `Rte_Call` API function call.
- In asynchronous client-server communication, the client runnable is not blocked after issuing the server call via `Rte_Call`. Instead, the result of the operation is returned via a call of the `Rte_Result` API function. This call can be blocking if a wait point is defined. Refer to [Classic AUTOSAR](#) to determine whether you have to specify an `ASYNCHRONOUS_SERVER_CALL RETURNS_EVENT`.

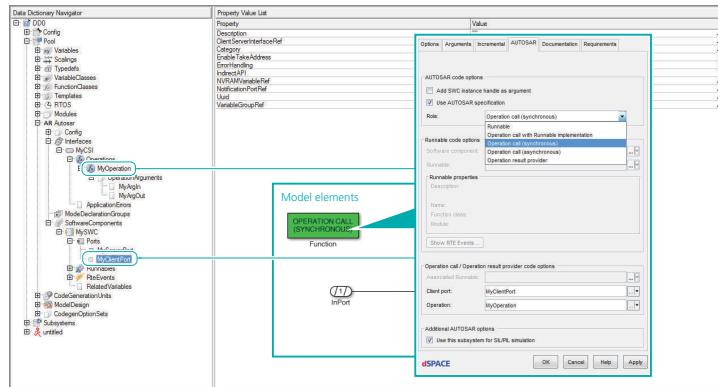
**Communication attributes** The data flow of client-server communication is characterized by the communication attributes of the participating ports. The communication attributes are grouped in communication specifications.

---

**Client-server communication in TargetLink** In TargetLink, different modeling styles exist, depending on whether you are modeling on the client or the server side of client-server communication:

Use Case	Modeling Side	Instructions
Modeling a synchronous server call	Client	<a href="#">Modeling Unidirectional Get or Set Operations (Synchronous Operation Calls via Port Blocks)</a> on page 140
	Client	<a href="#">Modeling Synchronous Client-Server Communication</a> on page 144
	Client/Server	<a href="#">Basics on Combining a Synchronous Operation Call with a Server Operation's Implementation</a> on page 156
Modeling an asynchronous server call	Client	<a href="#">Modeling Asynchronous Client-Server Communication</a> on page 146
Modeling transformer error logic	Client/Server	<a href="#">Modeling Transformer Error Logic</a> on page 205
Modeling application errors	Client/Server	<a href="#">Modeling Application Errors for Operations</a> on page 166
Modeling a server runnable	Server	<a href="#">Basics on Modeling the Server Side of Client-Server Communication</a> on page 152
Modeling port-defined argument values	Server	<a href="#">Modeling Port-Defined Argument Values</a> on page 160
Creating communication specifications	Server	<a href="#">Modeling Communication Specifications</a> on page 164

The following illustration shows the modeling of a synchronous operation call.



## Related topics

### Basics

[Introduction to Communication According to Classic AUTOSAR.....90](#)

### References

[Function Block \(TargetLink Model Element Reference\)](#)

# Modeling the Client Side of Client-Server Communication

## Where to go from here

## Information in this section

<a href="#">Introduction to Modeling the Client Side of Client-Server Communication.....</a>	136
<a href="#">Modeling Unidirectional Get or Set Operations (Synchronous Operation Calls via Port Blocks).....</a>	140
<a href="#">Modeling Synchronous Client-Server Communication.....</a>	144
<a href="#">Modeling Asynchronous Client-Server Communication.....</a>	146

# Introduction to Modeling the Client Side of Client-Server Communication

## Basics on Modeling Operation Calls on the Client Side of Client-Server Communication

**Operation calls on the client side** From TargetLink's perspective as a production code generator, an operation call is the call of an RTE API function from the production code of a client runnable. These operation calls can be synchronous or asynchronous.

**Synchronous operation calls** In synchronous client-server communication, a client runnable calls the `Rte_Call` RTE API function to initiate the communication with a [server runnable](#).

The input to the operation (operation arguments of the ARGIN kind) and the result of the operation (operation arguments of the ARGOUT kind) are transferred as arguments in the call of the `Rte_Call` RTE API function. This is why the client runnable has to wait for the server to complete or abort the operation.

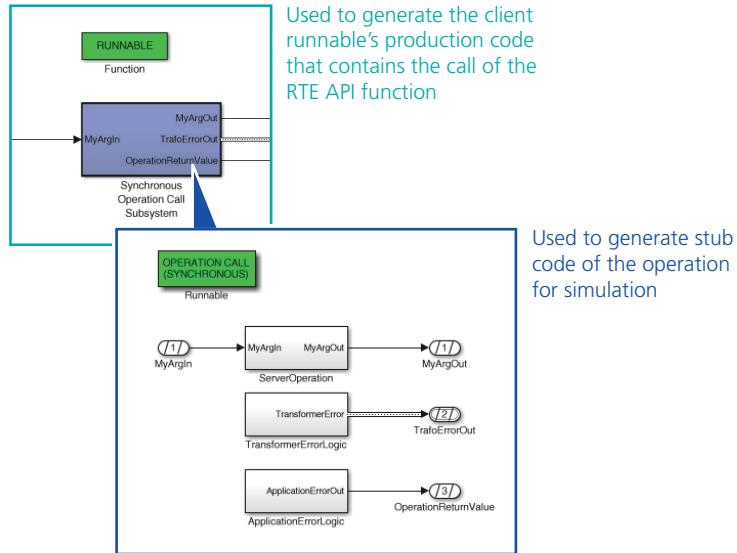
There are two modeling styles for modeling synchronous operation calls.

### Note

You must not mix these modeling styles in one [code generation unit \(CGU\)](#).

**Modeling via an operation call subsystem** The [synchronous operation call subsystem](#) provides the model elements that TargetLink uses to generate the call of the `Rte_Call` API function. Additionally, you can model the server

operation for simulation purposes. The following illustration shows a synchronous operation call subsystem:



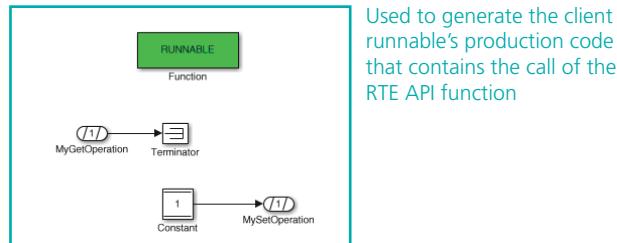
As you can see, the contents of the synchronous operation call subsystem are used to generate stub code for simulation. Refer to [How to Model Operation Calls via Synchronous Operation Call Subsystems](#) on page 144.

Within this modeling style, you can also model the following:

- Transformer errors (refer to [Modeling Transformer Error Logic](#) on page 205)
- Application errors (refer to [How to Model Application Errors of Operations](#) on page 166)

If you want to generate production code for the content of a synchronous operation call subsystem, TargetLink provides another modeling style. Refer to [Basics on Combining a Synchronous Operation Call with a Server Operation's Implementation](#) on page 156.

**Modeling via port blocks** Modeling via port blocks lets you model simple get or set operations.



Refer to [Modeling Unidirectional Get or Set Operations \(Synchronous Operation Calls via Port Blocks\)](#) on page 140.

### Asynchronous operation calls

In asynchronous client-server communication, a client runnable calls the **Rte\_Call** RTE API function to initiate the communication with a [server runnable](#).

The input to the operation (one or more operation arguments of the ARGIN kind) is transferred as an argument in the call of the `Rte_Call` RTE API function. The result of the operation (one or more operation arguments of the ARGOOUT kind) is transferred as an argument in the call of the `Rte_Result` RTE API function, which the same runnable or a different runnable of the same software component can call at a later time. This is why the client runnable that calls the `Rte_Call` RTE API does not have to wait for the server to complete or abort the operation.

The modeling style for modeling asynchronous client-server communication consists of two [operation subsystems](#) called [asynchronous operation call subsystem](#) and [operation result provider subsystem](#).

Within this modeling style, you can also model the following:

- Transformer errors (refer to [Modeling Transformer Error Logic](#) on page 205)
- Application errors (refer to [How to Model Application Errors of Operations](#) on page 166)

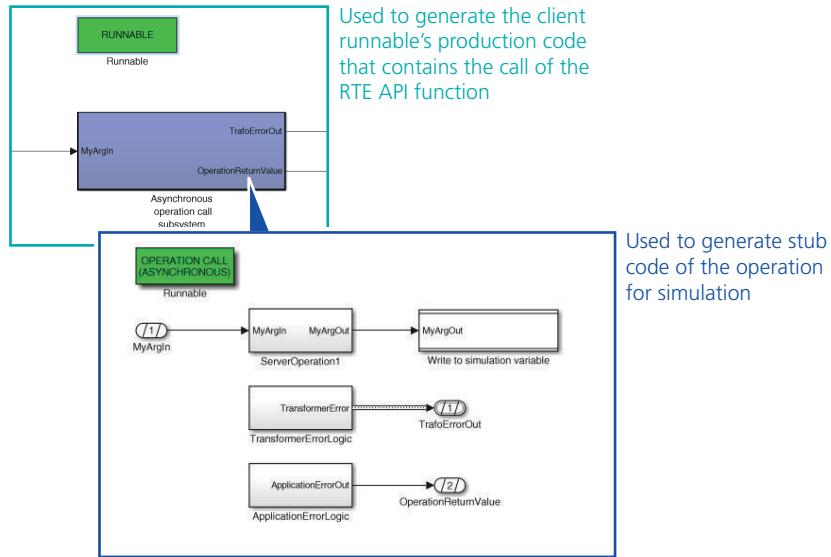
#### Note

If you model the behavior of the operation for simulation purposes, make sure to model it only once, either in the asynchronous operation call subsystem or in the operation result provider subsystem.

The following illustrations show the operation modeled for simulation purposes within the asynchronous operation call subsystem.

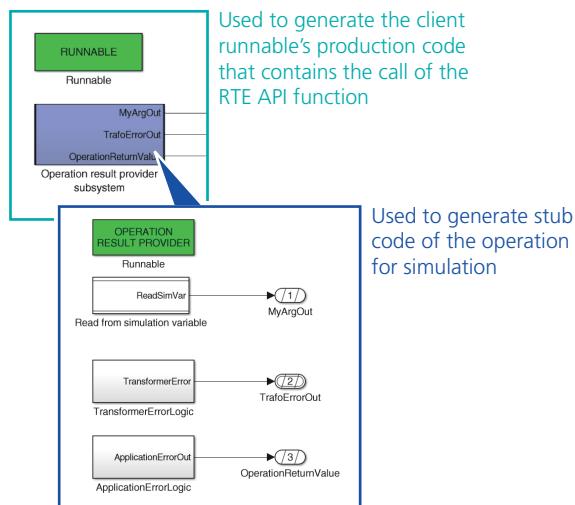
Further modeling is required if you want to simulate the communication between the two operation subsystems. Refer to [How to Simulate Operation Calls in Asynchronous Client-Server Communication](#) on page 309.

**Asynchronous operation call subsystem** The [asynchronous operation call subsystem](#) provides the model elements that TargetLink uses to generate the call of the `Rte_Call` API function. Additionally, you can model the server operation for simulation purposes. The following illustration shows an asynchronous operation call subsystem:



As you can see, the contents of the asynchronous operation call subsystem are used to generate stub code for simulation. Refer to [How to Model Operation Calls via Asynchronous Operation Call Subsystems](#) on page 146.

**Operation result provider subsystem** The [operation result provider subsystem](#) provides the model elements that TargetLink uses to generate the call of the `Rte_Result` API function. Additionally, you can model the server operation for simulation purposes. The following illustration shows an operation result provider subsystem:



As you can see, the contents of the operation result provider subsystem are used to generate stub code for simulation. Refer to [How to Model Operation Results via Operation Result Provider Subsystems](#) on page 149.

<b>Limitation</b>	<p><a href="#">? Operation result provider subsystems</a> must only contain port blocks whose Kind is set to one of the following values:</p> <ul style="list-style-type: none"><li>▪ Operation</li><li>▪ OperationReturnValue</li><li>▪ TransformerError</li></ul>
-------------------	---

## Modeling Unidirectional Get or Set Operations (Synchronous Operation Calls via Port Blocks)

---

Where to go from here	Information in this section
	<a href="#">Basics on Modeling Unidirectional Get or Set Operations.....</a> 140
	<a href="#">How to Model a Unidirectional Get or Set Operation in Synchronous Client-Server Communication via a Port Block.....</a> 141

## Basics on Modeling Unidirectional Get or Set Operations

---

<b>Unidirectional get or set operations</b>	TargetLink lets you model unidirectional get or set operations via port blocks.
---	---

### Note

This modeling style is restricted to synchronous operation calls without access to application errors, transformer errors or operation return values. If you have to model access to application errors or transformer errors, refer to [Modeling Synchronous Client-Server Communication](#) on page 144. If you want to model asynchronous operation calls, refer to [Modeling Asynchronous Client-Server Communication](#) on page 146.

<b>Direction of data flow</b>	In client-server communication as defined by <a href="#">? Classic AUTOSAR</a> , the direction of the data flow is described from the server's point of view. When using this modeling style, you can model only unidirectional communication. The following table shows which kind of argument you have to select:
-------------------------------	---

Client's Get Request		Client's Set Request	
Output from the Server	Input to the Client	Output from the Client	Input to the Server
The operation arguments are of the ARGOUT kind.		The operation arguments are of the ARGIN kind.	

**Note**

TargetLink does not support operation arguments of the ARGINOUT kind.

**Use cases**

You can model unidirectional get or set operations. Refer to [How to Model a Unidirectional Get or Set Operation in Synchronous Client-Server Communication via a Port Block](#) on page 141.

**Related topics****Basics**

[Basics on Client-Server Communication](#)..... 133

**References**

[Bus Import Block](#) ( [TargetLink Model Element Reference](#))

[Bus Outport Block](#) ( [TargetLink Model Element Reference](#))

[InPort Block](#) ( [TargetLink Model Element Reference](#))

[OutPort Block](#) ( [TargetLink Model Element Reference](#))

## How to Model a Unidirectional Get or Set Operation in Synchronous Client-Server Communication via a Port Block

**Synchronous communication**

In synchronous client-server communication, a client runnable calls the `Rte_Call` RTE API function to initiate the communication with a [server runnable](#).

The input to the operation (operation arguments of the ARGIN kind) and the result of the operation (operation arguments of the ARGOUT kind) are transferred as arguments in the call of the `Rte_Call` RTE API function. This is why the client runnable has to wait for the server to complete or abort the operation.

**Restrictions**

This modeling style is restricted to synchronous operation calls without access to application errors, transformer errors or operation return values.

If you have to model access to application errors or transformer errors, refer to [Modeling Synchronous Client-Server Communication](#) on page 144.

If you want to model asynchronous operation calls, refer to [Modeling Asynchronous Client-Server Communication](#) on page 146.

**Preconditions**

The following preconditions must be fulfilled:

- You created an operation with operation arguments. Refer to [How to Create Communication Subjects for Classic AUTOSAR Communication](#) on page 103 and [Basics on Modeling Unidirectional Get or Set Operations](#) on page 140.
- You created a client port. Refer to [How to Create Ports](#) on page 101.
- You created a runnable. Refer to [How to Model Runnables](#) on page 71.

**Method****To model a unidirectional get or set operation in synchronous client-server communication via a port block**

- 1 In the model, open the runnable subsystem for which you want to model a unidirectional get or set operation via a port block.
- 2 Add the following port blocks, depending on your use case:

Use Case	Port Block
Get operation with exactly one operation argument of the ARGOUT <sup>1)</sup> kind	InPort <sup>2)</sup>
Get operation with more than one operation argument of the ARGOUT <sup>1)</sup> kind	Bus Inport <sup>2)</sup>
Set operation with exactly one operation argument of the ARGIN <sup>1)</sup> kind	OutPort <sup>2)</sup>
Set operation with more than one operation argument of the ARGIN <sup>1)</sup> kind	Bus Outport <sup>2)</sup>

<sup>1)</sup> The direction of data flow in client-server communication as defined by [Classic AUTOSAR](#) is from the server's point of view.

<sup>2)</sup> In Simulink, you model the data flow from the client's point of view. If your operation argument is structured, use a bus port.

- 3 Open the port block's block dialog and make the following settings on its AUTOSAR page:

Property	Value
AUTOSAR mode	Classic
Kind	Client-Server
Client port	The DD ClientPort object that is used to pass the operation arguments.
Operation	A DD Operation object that matches your use case.

- 4 Close the block dialog.

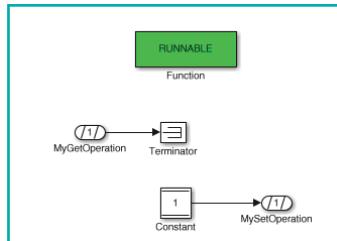
- 5 Connect the port blocks as required.

**Note**

To connect an operation with a bus signal, the sequence of bus signals has to match the sequence of operation arguments.

## Result

You modeled a unidirectional get or set operation via port blocks to generate the call of the Rte\_Call RTE API function in your production code.



Used to generate the client runnable's production code that contains the call of the RTE API function

The code for the get operation looks like this:

```
Rte_Call_ClientPortForPortBasedModeling_MyGetOperation(&MyArgOut);
```

The code for the set operation looks like this:

```
MyArgIn = 1;
Rte_Call_ClientPortForPortBasedModeling_MySetOperation(MyArgIn);
```

## Related topics

### Basics

Basics on Client-Server Communication.....	133
Basics on Modeling Operation Calls on the Client Side of Client-Server Communication.....	136
Basics on Modeling Unidirectional Get or Set Operations.....	140

### HowTos

How to Specify Code Optimization Settings for Classic AUTOSAR Operations.....	168
---	-----

### References

Bus Import Block (  TargetLink Model Element Reference)
Bus Outport Block (  TargetLink Model Element Reference)
InPort Block (  TargetLink Model Element Reference)
OutPort Block (  TargetLink Model Element Reference)

# Modeling Synchronous Client-Server Communication

## How to Model Operation Calls via Synchronous Operation Call Subsystems

### Preconditions

The following preconditions must be fulfilled:

- You created an operation with operation arguments. Refer to [How to Create Communication Subjects for Classic AUTOSAR Communication](#) on page 103.
- You created a client port. Refer to [How to Create Ports](#) on page 101.
- You created a runnable. Refer to [How to Model Runnables](#) on page 71.

### Method

#### To model an operation call via a synchronous operation call subsystem

- 1 In the model, open the runnable subsystem for which you want to model a synchronous operation call.
- 2 Add an atomic subsystem and open it.
- 3 Add a Function block and open its block dialog.
- 4 On the AUTOSAR page, make the following settings and close the dialog:

Property	Value
AUTOSAR mode	Classic
Role	Operation call (synchronous)
Client port	The DD ClientPort object that specifies the port used for the synchronous operation call.
Operation	The DD Operation object that specifies the operation and operation arguments used for the synchronous operation call.

- 5 To the [synchronous operation call subsystem](#), add the following model elements:

Model Element	Description
InPort <sup>1)</sup>	Add one block for each operation argument of the ARGIN kind to be delivered from the client to the server.
OutPort <sup>2)</sup>	Add one block for each operation argument of the ARGOUT kind to be delivered from the server to the client.

<sup>1)</sup> If the operation argument is structured, use a Bus Import block instead.

<sup>2)</sup> If the operation argument is structured, use a Bus Outport block instead.

These model elements define the interface of the subsystem that TargetLink uses to generate the call of the `Rte_Call` RTE API function.

- 6** In the block dialog of each port block, make the following settings:

Property	InPort <sup>1)</sup> Block	OutPort <sup>2)</sup> Block
AUTOSAR mode	Classic	
Kind	Operation	
Argument	The DD OperationArgument object that specifies the operation argument to be passed from the client to the server	The DD OperationArgument object that specifies the operation argument to be passed from the server to the client

<sup>1)</sup> If the operation argument is structured, use a Bus Import block instead.

<sup>2)</sup> If the operation argument is structured, use a Bus Outport block instead.

- 7** Optionally, you can add the following model element:

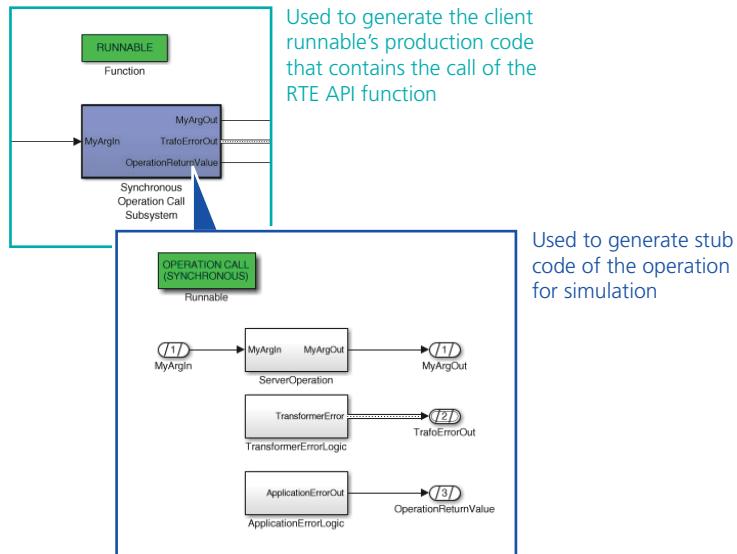
Model Element	Kind of AUTOSAR Communication	Instruction
Bus Outport	Transformer Error	<a href="#">Details on Data Transformation for Client-Server Communication on page 209</a>
OutPort	OperationReturnValue	<a href="#">How to Model Application Errors of Operations on page 166</a>

- 8** Optionally, you can model the operation within the subsystem for simulation purposes.

## Result

You modeled a synchronous operation call to generate the call of the `Rte_Call` RTE API function in your production code.

Optionally, you modeled the operation for simulation purposes.



The synchronous call of the `Rte_Call` API function looks like this:

```
Rte_Call_MyClientPort_MyOperation(MyInArg, &MyOutArg);
```

## Related topics

### Basics

Basics on Client-Server Communication.....	133
Basics on Modeling Operation Calls on the Client Side of Client-Server Communication.....	136
Basics on Modeling Unidirectional Get or Set Operations.....	140
Details on Data Transformation for Client-Server Communication.....	209

### HowTos

How to Specify Code Optimization Settings for Classic AUTOSAR Operations.....	168
---	-----

### References

Bus Import Block ( TargetLink Model Element Reference)	
Bus Outport Block ( TargetLink Model Element Reference)	
Function Block ( TargetLink Model Element Reference)	
InPort Block ( TargetLink Model Element Reference)	
OutPort Block ( TargetLink Model Element Reference)	

# Modeling Asynchronous Client-Server Communication

---

## Where to go from here

## Information in this section

How to Model Operation Calls via Asynchronous Operation Call Subsystems.....	146
How to Model Operation Results via Operation Result Provider Subsystems.....	149

## How to Model Operation Calls via Asynchronous Operation Call Subsystems

---

### Preconditions

The following preconditions must be fulfilled:

- You created an operation with operation arguments. Refer to [How to Create Communication Subjects for Classic AUTOSAR Communication](#) on page 103.
- You created a client port. Refer to [How to Create Ports](#) on page 101.
- You created a runnable. Refer to [How to Model Runnables](#) on page 71.

**Method****To model an operation call via an asynchronous operation call subsystem**

- 1** In the model, open the runnable subsystem for which you want to model an asynchronous operation call.
- 2** Add an atomic subsystem and open it.
- 3** Add a Function block and open its block dialog.
- 4** On the AUTOSAR page, make the following settings and close the dialog:

Property	Value
AUTOSAR mode	Classic
Role	Operation call (asynchronous)
Client port	The DD ClientPort object that specifies the port used for the asynchronous operation call
Operation	The DD Operation object that specifies the operation and operation arguments used for the asynchronous operation call

- 5** To the [asynchronous operation call subsystem](#), add the following model elements:

Model Element	Description
InPort <sup>1)</sup>	Add one block for each operation argument of the ARGIN kind to be delivered from the client to the server.

<sup>1)</sup> If the operation argument is structured, use a Bus Import block instead.

These model elements define the interface of the subsystem that TargetLink uses to generate the call of the `Rte_Call` RTE API function.

- 6** In the block dialog of each port block, make the following settings:

Property	InPort <sup>1)</sup> Block
AUTOSAR mode	Classic
Kind	Operation
Argument	The DD OperationArgument that specifies the operation argument to be passed from the client to the server

<sup>1)</sup> If the operation argument is structured, use a Bus Import block instead.

- 7** Optionally, you can add the following model element:

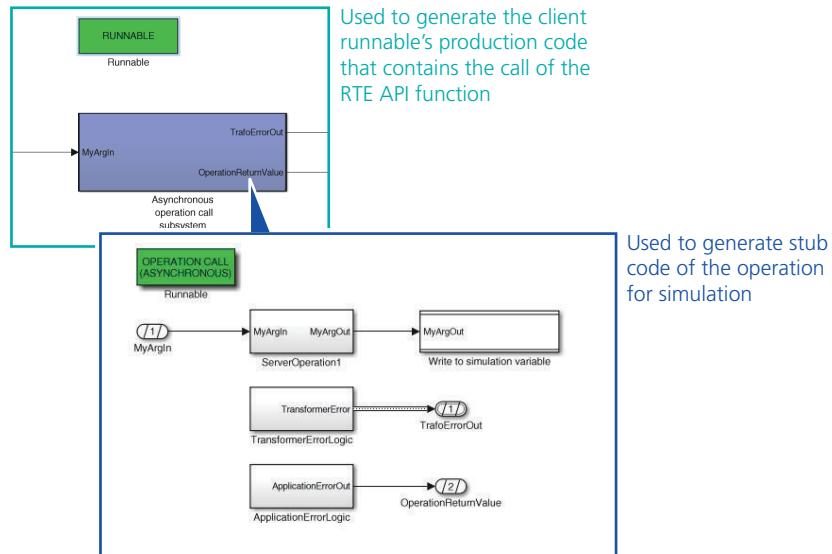
Model Element	Kind of AUTOSAR Communication	Instruction
Bus Outport	Transformer Error	<a href="#">Details on Data Transformation for Client-Server Communication</a> on page 209
OutPort	OperationReturnValue	<a href="#">How to Model Application Errors of Operations</a> on page 166

- 8** Optionally, you can model the operation within the subsystem for simulation purposes.

**Result**

You modeled an asynchronous operation call to generate the call of the `Rte_Call` RTE API function in your production code.

Optionally, you modeled the operation for simulation purposes.



The asynchronous call of the `Rte_Call` API function looks like this:

```
Rte_Call_MyClientPort_MyOperation(MyInArg);
```

## Next step

You can now model the call of the `Rte_Result` API function to be called by a runnable to get the operation's result. Refer to [How to Model Operation Results via Operation Result Provider Subsystems](#) on page 149.

## Related topics

### Basics

Basics on Client-Server Communication.....	133
Basics on Modeling Operation Calls on the Client Side of Client-Server Communication.....	136
Details on Data Transformation for Client-Server Communication.....	209

### HowTos

How to Model Operation Results via Operation Result Provider Subsystems.....	149
How to Specify Code Optimization Settings for Classic AUTOSAR Operations.....	168

### References

Bus Import Block ( TargetLink Model Element Reference)	
Bus Outport Block ( TargetLink Model Element Reference)	
Function Block ( TargetLink Model Element Reference)	
InPort Block ( TargetLink Model Element Reference)	
OutPort Block ( TargetLink Model Element Reference)	

## How to Model Operation Results via Operation Result Provider Subsystems

### Preconditions

You modeled the call of the corresponding `Rte_Call` API function. Refer to [How to Model Operation Calls via Asynchronous Operation Call Subsystems](#) on page 146.

### Method

#### To model an operation result via an operation result provider subsystem

- 1 In the model, open the runnable subsystem for which you want to model the call of the `Rte_Result` API function that corresponds to an asynchronous operation call.
- 2 Add an atomic subsystem and open it.
- 3 Add a Function block and open its block dialog.
- 4 On the AUTOSAR page, make the following settings and close the dialog:

Property	Value
AUTOSAR mode	Classic
Role list	Operation result provider
Associated Runnable	The DD Runnable object used in the runnable subsystem that contains the <a href="#">asynchronous operation call subsystem</a> used to generate the corresponding <code>Rte_Call</code> call. <sup>1)</sup>
Client port	The ClientPort object that specifies the port used for the asynchronous operation call.
Operation	The DD Operation object that specifies the operation and operation arguments used for the asynchronous operation call.

<sup>1)</sup> Specify this property only if the [code generation unit \(CGU\)](#) that contains the [operation result provider subsystem](#) does not contain the corresponding [asynchronous operation call subsystem](#).

- 5 To the [operation result provider subsystem](#), add the following model elements:

Model Element	Description
<code>OutPort</code> <sup>1)</sup>	Add one block for each operation argument of the ARGOUT kind to be delivered from the server to the client.

<sup>1)</sup> If the operation argument is structured, use a Bus Outport block instead.

These model elements define the interface of the subsystem that TargetLink uses to generate the call of the `Rte_Result` RTE API function.

- 6 In the block dialog of each port block, make the following settings:

Property	<code>OutPort</code> <sup>1)</sup> Block
AUTOSAR mode	Classic
Kind	Operation

Property	OutPort <sup>1)</sup> Block
Argument	The DD OperationArgument that specifies the operation argument to be passed from the server to the client. This object must belong to the DD Operation object you used in the <a href="#">asynchronous operation call subsystem</a> that TargetLink uses to generate the corresponding call of the Rte_Call API function.

<sup>1)</sup> If the operation argument is structured, use a Bus Outport block instead.

## 7 Optionally, you can add the following model element:

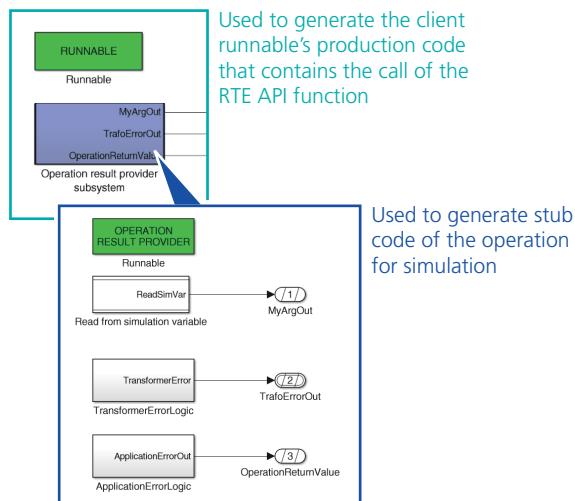
Model Element	Kind of AUTOSAR Communication	Instruction
Bus Outport	Transformer Error	<a href="#">Details on Data Transformation for Client-Server Communication</a> on page 209
OutPort	OperationReturnValue	<a href="#">How to Model Application Errors of Operations</a> on page 166

**8** Optionally, you can model the operation within the subsystem for simulation purposes.

## Result

You modeled an operation result provider subsystem to generate the call of the **Rte\_Result** API function in your production code.

Optionally, you modeled the operation for simulation purposes.



The call of the **Rte\_Result** API function looks like this:

```
Rte_Result_MyClientPort_MyOperation(&MyOutArg);
```

## Next step

Further modeling is required if you want to simulate the communication between the two operation subsystems. Refer to [How to Simulate Operation Calls in Asynchronous Client-Server Communication](#) on page 309.

**Related topics****Basics**

Basics on Client-Server Communication.....	133
Basics on Modeling Operation Calls on the Client Side of Client-Server Communication.....	136
Details on Data Transformation for Client-Server Communication.....	209

**HowTos**

How to Model Operation Calls via Asynchronous Operation Call Subsystems.....	146
--	-----

**References**

Bus Import Block ( TargetLink Model Element Reference)	
Bus Outport Block ( TargetLink Model Element Reference)	
Function Block ( TargetLink Model Element Reference)	
InPort Block ( TargetLink Model Element Reference)	
OutPort Block ( TargetLink Model Element Reference)	

## Modeling the Server Side of Client-Server Communication

**Where to go from here****Information in this section**

Introduction to Modeling the Server Side of Client-Server Communication.....	152
Modeling Port-Defined Argument Values.....	160
Modeling Communication Specifications.....	164
Modeling Application Errors for Operations.....	166

# Introduction to Modeling the Server Side of Client-Server Communication

## Where to go from here

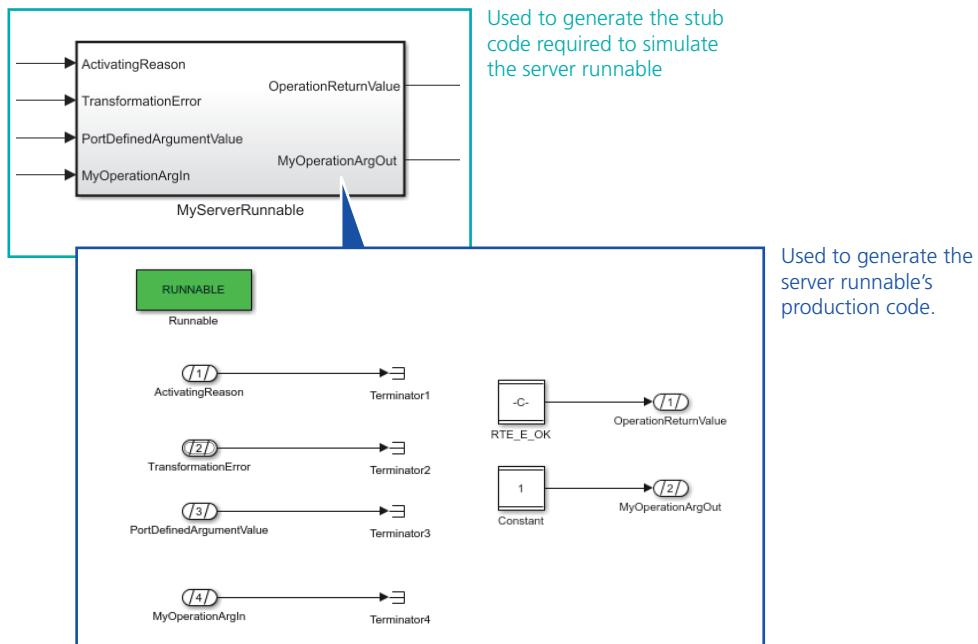
## Information in this section

Basics on Modeling the Server Side of Client-Server Communication.....	152
How to Model the Implementation of a Server Operation.....	153
Basics on Combining a Synchronous Operation Call with a Server Operation's Implementation.....	156
How to Combine a Synchronous Operation Call with a Server Operation's Implementation.....	157

## Basics on Modeling the Server Side of Client-Server Communication

### Introduction

TargetLink lets you model a [server runnable](#) via a [Runnable subsystem](#), which is shown in the following illustration:



## Workflow for modeling a server runnable

The workflow for creating a server runnable is shown in the following table:

Workflow Step	Instruction
Creating a <a href="#">software component (SWC)</a>	<a href="#">How to Create Software Components</a> on page 66
Creating a <a href="#">client-server interface</a>	<a href="#">How To Create Interfaces</a> on page 100
Creating a <a href="#">operation</a>	<a href="#">How to Create Communication Subjects for Classic AUTOSAR Communication</a> on page 103
Creating a <a href="#">server port</a>	<a href="#">How to Create Ports</a> on page 101
Creating a <a href="#">Runnable</a>	<a href="#">How to Model Runnables</a> on page 71
Creating an <a href="#">RTE event</a> of the <code>OPERATION_INVOKED_EVENT</code> event kind.	<a href="#">How to Specify an Event for a Runnable</a> on page 76
Modeling the implementation of a server operation.	<a href="#">How to Model the Implementation of a Server Operation</a> on page 153

## Combining operation call and sever runnable implementation

For synchronous client-server communication, TargetLink provides a convenient way to model the client's request together with the server runnable's implementation. Refer to [Basics on Combining a Synchronous Operation Call with a Server Operation's Implementation](#) on page 156.

## Related topics

### Basics

<a href="#">Basics on Activation Reasons</a> .....	80
<a href="#">Basics on Client-Server Communication</a> .....	133
<a href="#">Basics on Port-Defined Argument Values</a> .....	161
<a href="#">Modeling Transformer Error Logic</a> .....	205

### HowTos

<a href="#">How To Model a Runnable's Activation Reasons</a> .....	82
<a href="#">How to Model Application Errors of Operations</a> .....	166
<a href="#">How to Model Port-Defined Argument Values</a> .....	162

## How to Model the Implementation of a Server Operation

### Preconditions

The following preconditions must be fulfilled:

- You created a [software component \(SWC\)](#). Refer to [How to Create Software Components](#) on page 66.
- You created a [client-server interface](#). Refer to [How To Create Interfaces](#) on page 100.
- You created an [operation](#). Refer to [How to Create Communication Subjects for Classic AUTOSAR Communication](#) on page 103.

- You created a [server port](#). Refer to [How to Create Ports](#) on page 101.
- You created a [Runnable](#), such as `MyServerRunnable`. Refer to [How to Model Runnables](#) on page 71.
- You created an [RTE event](#) of the `OPERATION_INVOKED_EVENT` event kind that references the operation and the server port. Refer to [How to Specify an Event for a Runnable](#) on page 76.

Method	To model the implementation of a server operation
	<ol style="list-style-type: none"> <li>1 In the model, open the <a href="#">Runnable subsystem</a> that you modeled for <code>MyServerRunnable</code>.</li> <li>2 Open the Runnable block's dialog.</li> <li>3 On the AUTOSAR page, make the following settings and close the dialog:</li> </ol>

Property	Value
Software component	The DD SoftwareComponent object that contains the DD Runnable object named <code>MyServerRunnable</code> .
Runnable	The DD Runnable object named <code>MyServerRunnable</code> .
Client port	The ClientPort object that specifies the port used for the synchronous operation call.
Operation	The DD Operation object that specifies the operation and operation arguments.

- 4 Add the following model elements:

Model Element	Description
InPort <sup>1)</sup>	Add one block for each operation argument of the ARGIN kind to be delivered from the client to the server.
OutPort <sup>2)</sup>	Add one block for each operation argument of the ARGOUT kind to be delivered from the server to the client.

<sup>1)</sup> If the operation argument is structured, use a Bus Import block instead.

<sup>2)</sup> If the operation argument is structured, use a Bus Outport block instead.

These model elements define the interface of the subsystem that TargetLink uses to generate the call of the `Rte_Call` RTE API function.

- 5 In the block dialog of each port block, make the following settings:

Property	InPort <sup>1)</sup> Block	OutPort <sup>2)</sup> Block
AUTOSAR mode	Classic	
Kind	Operation	
Argument	The DD OperationArgument that specifies the operation argument to be passed from the client to the server.	The DD OperationArgument that specifies the operation argument to be passed from the server to the client.

<sup>1)</sup> If the operation argument is structured, use a Bus Import block instead.

<sup>2)</sup> If the operation argument is structured, use a Bus Outport block instead.

- 6 Model the server operation between the port blocks.

**7** Optionally, you can add the following model element:

Model Element	Kind of AUTOSAR Communication	Instruction
OutPort	OperationReturnValue	<a href="#">How to Model Application Errors of Operations on page 166</a>

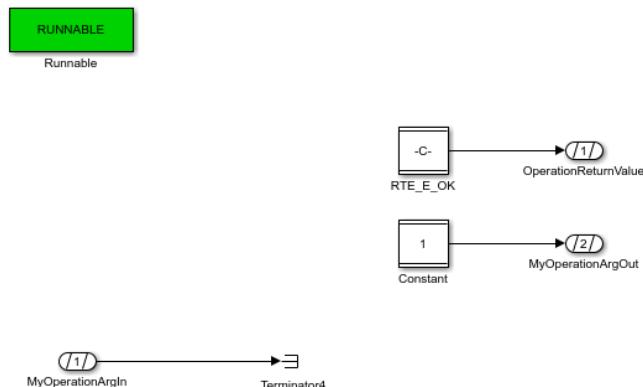
- 8** If the DD Operation object references a DD ApplicationError object via its PossibleErrorRef property, add an OutPort block and specify it as follows:

Property	Value
AUTOSAR mode	Classic
Kind	OperationReturnValue

Refer to [How to Model Application Errors of Operations on page 166](#).

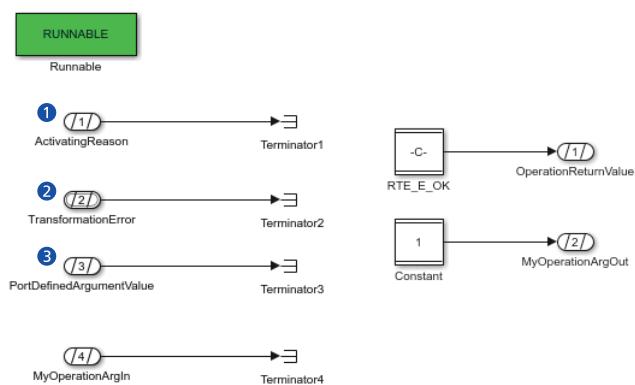
## Result

You modeled the implementation of a server operation.



## Next steps

You can now add further model elements as shown in the following illustration:



Number	Instruction
①	<a href="#">How To Model a Runnable's Activation Reasons</a> on page 82
②	<a href="#">Modeling Transformer Error Logic</a> on page 205
③	<a href="#">Modeling Port-Defined Argument Values</a> on page 160

**Related topics****Basics**

Basics on Client-Server Communication.....	133
Basics on Modeling Operation Calls on the Client Side of Client-Server Communication.....	136
Basics on Modeling Unidirectional Get or Set Operations.....	140

**HowTos**

How to Model Application Errors of Operations.....	166
How to Specify Code Optimization Settings for Classic AUTOSAR Operations.....	168

**References**

Bus Import Block ( <a href="#">TargetLink Model Element Reference</a> )	
Bus Outport Block ( <a href="#">TargetLink Model Element Reference</a> )	
Function Block ( <a href="#">TargetLink Model Element Reference</a> )	
InPort Block ( <a href="#">TargetLink Model Element Reference</a> )	
OutPort Block ( <a href="#">TargetLink Model Element Reference</a> )	

## Basics on Combining a Synchronous Operation Call with a Server Operation's Implementation

**Synchronous operation calls**

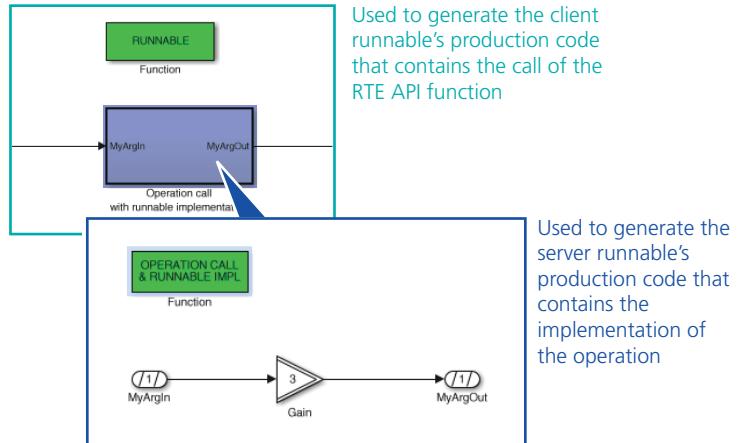
In synchronous client-server communication, a client runnable calls the `Rte_Call` RTE API function to initiate the communication with a [server runnable](#).

The input to the operation (operation arguments of the ARGIN kind) and the result of the operation (operation arguments of the ARGOUT kind) are transferred as arguments in the call of the `Rte_Call` RTE API function. This is why the client runnable has to wait for the server to complete or abort the operation.

**Operation call with runnable implementation**

From TargetLink's perspective as a production code generator, the production code generated for a [server runnable](#) is the implementation of a server's operation. To make simulating the data flow between client and server easier,

TargetLink offers the [operation call with runnable implementation subsystem](#), which is shown in the following illustration:



Using this modeling style instructs TargetLink to generate the synchronous operation call within the client runnable's production code via `Rte_Call` together with the implementation of the server runnable in production code.

Refer to [How to Combine a Synchronous Operation Call with a Server Operation's Implementation](#) on page 157.

#### Limitations

This modeling style is restricted to synchronous operation calls without `ActivationReason`, `PortDefinedArguments`, or `TransformerError`.

If you have to model access to application errors or transformer errors, refer to [Modeling Synchronous Client-Server Communication](#) on page 144.

If you want to model asynchronous operation calls, refer to [Modeling Asynchronous Client-Server Communication](#) on page 146.

## How to Combine a Synchronous Operation Call with a Server Operation's Implementation

#### Restrictions

This modeling style is restricted to synchronous operation calls without `ActivationReason`, `PortDefinedArguments`, or `TransformerError`.

If you have to model access to application errors or transformer errors, refer to [Modeling Synchronous Client-Server Communication](#) on page 144.

If you want to model asynchronous operation calls, refer to [Modeling Asynchronous Client-Server Communication](#) on page 146.

**Preconditions**

The following preconditions must be fulfilled:

- You created an operation with operation arguments. Refer to [How to Create Communication Subjects for Classic AUTOSAR Communication](#) on page 103.
- You created a DD ClientPort and a DD ServerPort object. Refer to [How to Create Ports](#) on page 101.
- You created two DD Runnable objects:
  - A DD Runnable object that represents the client runnable, such as `MyClientRunnable`. You also modeled this runnable via a runnable subsystem in the model.
  - A DD Runnable object that represents the [server runnable](#), such as `MyServerRunnable`, that is triggered by an `OPERATION_INVOKED_EVENT`. The DD RteEvent object specifying this event must reference the operation provided by `MyServerRunnable` and its server port.

Refer to [How to Model Runnables](#) on page 71 and [How to Specify an Event for a Runnable](#) on page 76.

**Method****To combine a synchronous operation call with a server operation's implementation**

- 1 In the model, open the runnable subsystem you modeled for `MyClientRunnable`.
- 2 Add an atomic subsystem and open it.
- 3 Add a Function block and open its block dialog.
- 4 On the AUTOSAR page, make the following settings and close the dialog:

Property	Value
AUTOSAR mode	Classic
Role	<code>Operation call with Runnable implementation</code>
Software component	The DD SoftwareComponent object that contains the DD Runnable object named <code>MyServerRunnable</code> .
Runnable	The DD Runnable object named <code>MyServerRunnable</code> .
Client port	The DD ClientPort object that specifies the port used for the synchronous operation call.
Operation	The DD Operation object that specifies the operation and operation arguments used for the synchronous operation call.

- 5 To the operation call with Runnable implementation subsystem add the following model elements:

Model Element	Description
InPort <sup>1)</sup>	Add one block for each operation argument of the ARGIN kind to be delivered from the client to the server.
OutPort <sup>2)</sup>	Add one block for each operation argument of the ARGOUT kind to be delivered from the server to the client.

<sup>1)</sup> If the operation argument is structured, use a Bus Import block instead.

<sup>2)</sup> If the operation argument is structured, use a Bus Outport block instead.

These model elements define the interface of the subsystem that TargetLink uses to generate the call of the `Rte_Call` RTE API function.

- In the block dialog of each port block, make the following settings:

Property	InPort <sup>1)</sup> Block	OutPort <sup>2)</sup> Block
AUTOSAR mode	Classic	
Kind	Operation	
Argument	The DD OperationArgument that specifies the operation argument to be passed from the client to the server.	The DD OperationArgument that specifies the operation argument to be passed from the server to the client.

<sup>1)</sup> If the operation argument is structured, use a Bus Import block instead.

<sup>2)</sup> If the operation argument is structured, use a Bus Outport block instead.

- Model the server operation between the port blocks. TargetLink generates production code for this operation together with the production code of the runnable subsystem that contains the operation call with Runnable implementation subsystem.
- If the DD Operation object references a DD ApplicationError object via its PossibleErrorRef property, add an OutPort block and specify it as follows:

Property	Value
AUTOSAR mode	Classic
Kind	OperationReturnValue

Refer to [How to Model Application Errors of Operations](#) on page 166.

## Result

You modeled a synchronous operation call to generate the call of the `Rte_Call` RTE API function in your production code.

Additionally, you modeled the server operation to generate the production code of the server runnable.

November 2020

TargetLink Classic AUTOSAR Modeling Guide

159

With a constant value of 3 fed into the operation call with runnable implementation subsystem, the synchronous call of the `Rte_Call` API function looks like this:

```
Rte_Call_MyClientPort_MyOperation(3, &MyOutArg);
```

Assuming that the DoSomething subsystem contains a Gain block whose gain parameter is set to 3, the server runnable's production code looks like this (without compiler abstraction):

```
void MyServerRunnable(sint16 MyArgIN, sint16 * MyArgOut)
{
    /* TargetLink outport: Subsystem/Operation call with runnable implementation subsystem/MyArgOut
     * combined # Gain: Subsystem/Operation call with runnable implementation subsystem/DoSomething
     */Gain */
    *MyArgOut = MyArgIN * 3;
}
```

## Related topics

### Basics

Basics on Client-Server Communication.....	133
Basics on Modeling Operation Calls on the Client Side of Client-Server	
Communication.....	136
Basics on Modeling Unidirectional Get or Set Operations.....	140

### HowTos

How to Model Application Errors of Operations.....	166
How to Specify Code Optimization Settings for Classic AUTOSAR Operations.....	168

### References

Bus Import Block ( TargetLink Model Element Reference)	
Bus Outport Block ( TargetLink Model Element Reference)	
Function Block ( TargetLink Model Element Reference)	
InPort Block ( TargetLink Model Element Reference)	
OutPort Block ( TargetLink Model Element Reference)	

# Modeling Port-Defined Argument Values

## Where to go from here

## Information in this section

Basics on Port-Defined Argument Values.....	161
How to Model Port-Defined Argument Values.....	162

## Basics on Port-Defined Argument Values

### Port-defined argument values

[?](#) Port-defined argument values are implicit values the RTE can pass to a server.

They are mainly intended for use with basic software services. For example, they can be used to pass the memory block ID to the NVRAM Manager.

### Port-defined argument values according to Classic AUTOSAR

According to [?](#) **Classic AUTOSAR**, the RTE can pass port-defined argument values as implicit values to a server. They are implicit because they do not appear as formal arguments in the `Rte_Call` API function's signature and are thus hidden from the client components.

### Port-defined argument values in TargetLink

TargetLink lets you specify DD PortDefinedArgument objects at DD ServerPort objects.

Each PortDefinedArgument object lets you specify the data type and the value of a port-defined argument value as defined by [?](#) **Classic AUTOSAR**.

**Changing the order of formal parameters** The order of the DD PortDefinedArgument objects defines the order of the port-defined argument values in the runnable's signature.

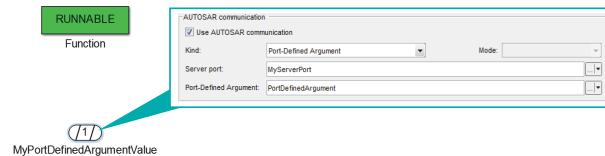
**Changing the port-defined argument value's name:** You can influence their name via the DD Runnable object's FormalArguments property.

#### Note

The FormalArguments property applies to operation arguments and to port-defined argument values. The number of strings in the property's string list must match the total number of formal arguments (operation arguments *plus* port-defined argument values) in the runnable function's signature.

### Port-defined argument values in the model

In the model, you reference a DD PortDefinedArgument object at the TargetLink InPort block that represents the port-defined argument value.



### Port-defined argument values in the production code

In production code, TargetLink generates port-defined argument values as formal arguments within the runnable function's signature:

```
FUNC(void, MySWC_CODE) MyRunnable(uint8 portDefArg1);
```

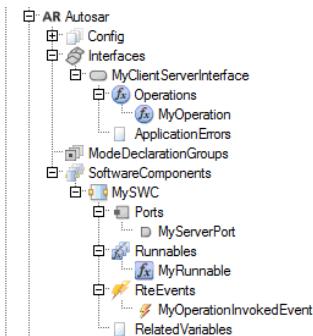
---

<b>Limitation</b>	TargetLink does not support port-defined argument values in an <a href="#">operation call with runnable implementation subsystem</a> .						
<b>Related topics</b>	<p><b>Basics</b></p> <table border="1"> <tr> <td>Basics on Client-Server Communication.....</td> <td>133</td> </tr> <tr> <td>Basics on Modeling the Server Side of Client-Server Communication.....</td> <td>152</td> </tr> </table> <p><b>HowTos</b></p> <table border="1"> <tr> <td>How to Model Port-Defined Argument Values.....</td> <td>162</td> </tr> </table>	Basics on Client-Server Communication.....	133	Basics on Modeling the Server Side of Client-Server Communication.....	152	How to Model Port-Defined Argument Values.....	162
Basics on Client-Server Communication.....	133						
Basics on Modeling the Server Side of Client-Server Communication.....	152						
How to Model Port-Defined Argument Values.....	162						

## How to Model Port-Defined Argument Values

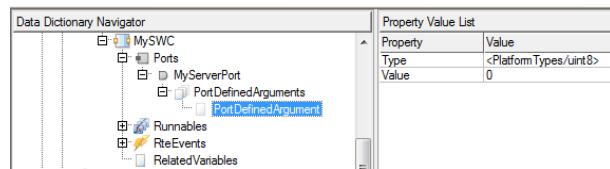
---

<b>Restrictions</b>	The following restrictions apply:
	<ul style="list-style-type: none"> <li>▪ Port-defined argument values can be modeled only for server runnables that are not part of other runnables themselves.</li> <li>▪ Port-defined argument values in operation call subsystems are not supported.</li> </ul>
<b>Precondition</b>	The following preconditions must be fulfilled:
	<ul style="list-style-type: none"> <li>▪ You created a DD SoftwareComponent together with the following objects: <ul style="list-style-type: none"> <li>▪ A DD Runnable object, such as <b>MyRunnable</b>.</li> <li>▪ A DD ServerPort object, such as <b>MyServerPort</b>.</li> <li>▪ An RteEvent object, such as <b>MyOperationInvokedEvent</b>.</li> </ul> </li> <li>▪ You created a DD ClientServerInterface object, such as <b>MyClientServerInterface</b>, together with an Operation object, such as <b>MyOperation</b>.</li> </ul>



**Method****To model a port-defined argument value**

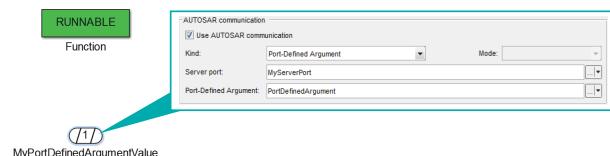
- 1 At MyServerPort, set the ClientServerInterfaceRef property to MyClientServerInterface.
- 2 From MyServerPort's context menu, select Create DefinedArguments to add the PortDefinedArguments object to MyServerPort's subtree.
- 3 Add a PortDefinedArgument object to PortDefinedArguments subtree and specify it as required.



- 4 At MyOperationInvokedEvent, set the following properties as shown in the following table:

Property	Description
RteEventKind	OPERATION_INVOKED_EVENT
OperationRef	MyClientServerInterface/Operations/MyOperation
RunnableRef	MyRunnable
ServerPortRef	MyServerPort

- 5 In the model, add a TargetLink InPort block to the root level of your runnable.
- 6 On the AUTOSAR page of the InPort block's block dialog, make the following settings:



- 7 Generate code.

**Result**

You modeled a port-defined argument value that appears as a formal parameter in the runnable function's signature:

```
FUNC(void, MySWC_CODE) MyRunnable(uint8 portDefArg1);
```

**Related topics****Basics**

Basics on Client-Server Communication.....	133
Basics on Modeling the Server Side of Client-Server Communication.....	152
Basics on Port-Defined Argument Values.....	161

**HowTos**

How to Create Communication Subjects for Classic AUTOSAR Communication.....	103
How To Create Interfaces.....	100
How to Create Ports.....	101
How to Create Software Components.....	66
How to Model Runnables.....	71
How to Specify an Event for a Runnable.....	76
How to Specify Code Optimization Settings for Classic AUTOSAR Operations.....	168

# Modeling Communication Specifications

## How to Create Communication Specifications for Operations

**Objective**

To specify communication attributes that manage the data flow of client-server communication.

**Basics on communication attributes**

The data flow of client-server communication is characterized by the communication attributes of the participating ports. The following table describes the communication attributes of server ports. According to AUTOSAR, no communication attributes are defined for client ports.

<b>Communication Attribute</b>	<b>Description</b>
Server communication specification	
QueueLength	Defines the number of client requests that can be stored in a <i>first-in-first-out</i> queue. This communication attribute cannot be simulated, but it is exported to the SWC description file.

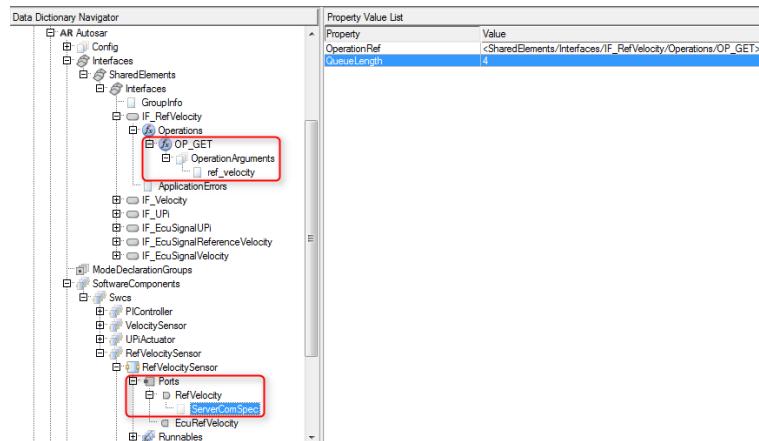
**Preconditions**

The following preconditions must be fulfilled:

- You created a server port. Refer to [How to Create Ports](#) on page 101.
- You created an operation. Refer to [How to Create Communication Subjects for Classic AUTOSAR Communication](#) on page 103.

**Method****To create communication specifications for operations**

- 1 In the Data Dictionary Navigator, navigate to the following subtree:  
`/Pool/Autosar/SoftwareComponents/<SoftwareComponent>/Ports/<Port>`
- 2 Right-click the Port object and select **Create ServerComSpec**.
- 3 In the Property Value List, browse the **OperationRef** property for the operation you want to create a communication specification for.
- 4 Specify the communication attributes for the data flow of the referenced operation.

**Result**

You created a communication specification for an operation.

**Related topics****Basics**

[Basics on Client-Server Communication](#)..... 133

**HowTos**

<a href="#">How to Create Communication Subjects for Classic AUTOSAR Communication</a> .....	103
<a href="#">How to Create Ports</a> .....	101
<a href="#">How to Specify Code Optimization Settings for Classic AUTOSAR Operations</a> .....	168

# Modeling Application Errors for Operations

## How to Model Application Errors of Operations

<b>Objective</b>	To specify application errors of a server operation and to model the client's access to the errors.
<b>Basics</b>	When a client calls a server for an operation or gets the result of an operation, the operation can return an error. TargetLink lets you specify error codes and categories for operation errors of client-server interfaces.
<b>Preconditions</b>	You created an operation. For instructions on creating operations, refer to <a href="#">How to Create Communication Subjects for Classic AUTOSAR Communication</a> on page 103.
<b>Restrictions</b>	<b>Note</b> TargetLink ignores application errors for operation calls of unidirectional ( <i>set</i> or <i>get</i> ) operations modeled via port blocks. Therefore, <b>PossibleErrorRef</b> properties specified at DD Operation objects do not exist as signals at the port blocks. When you model <a href="#">server runnables</a> , the following applies: If you specify the <b>PossibleErrorRef</b> property of a DD Operation object, an OutPort block using AUTOSAR communication of kind <b>OperationReturnValue</b> must exist in the model. When you model <a href="#">operation subsystems</a> that are not implemented as <a href="#">server runnables</a> , an OutPort block using AUTOSAR communication of kind <b>OperationReturnValue</b> may exist in the model regardless of the DD Operation object's <b>PossibleErrorRef</b> property.

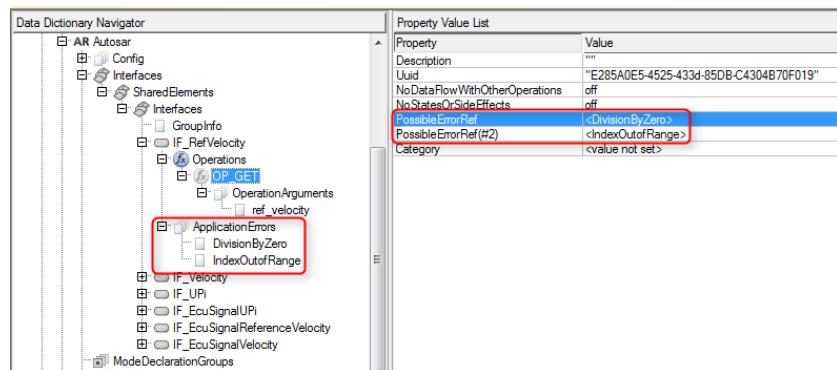
<b>Method</b>	<b>To model application errors of operations</b> <ol style="list-style-type: none"><li>In the Data Dictionary Navigator, right-click the following object: <code>/Pool/Autosar/Interfaces/&lt;ClientServerInterface&gt;/Application Errors</code></li><li>Choose <b>Create ApplicationError</b>.</li><li>Enter a name for the application error, for example, <code>my_ApplicationError</code>.</li></ol>
---------------	--

- 4 In the Property Value List, specify an error code via the ErrorCode Browse button. [Classic AUTOSAR](#) has reserved the values 1 to 63 of the 8-bit standard return value for this purpose.
- 5 In the Data Dictionary Navigator, select the operation you want to model application errors for.
- 6 In the Property Value List, click the PossibleErrorRef Browse button. A selection dialog opens.
- 7 Select the application error.

**Tip**

If you want to reference multiple application errors, you have to select Create Reference to PossibleError from the context menu of the following object:

```
/Pool/Autosar/Interfaces/<ClientServerInterface>/  
Operations/<Operation>  
Then, repeat steps 6 and 7.
```



- 8 In the Simulink/TargetLink model, add an OutPort block to the subsystem that implements the operation call to access application errors in the block diagram.
- 9 Double-click the block to open the AUTOSAR page.  
Set AUTOSAR mode to **Classic**.
- 10 From the Kind list, select **OperationReturnValue**.

**Result**

You modeled the access to application errors of a client-server operation.

**Related topics****Basics**

[Basics on Client-Server Communication.....](#) 133

# Specifying Code Optimization Settings for Classic AUTOSAR Operations

## How to Specify Code Optimization Settings for Classic AUTOSAR Operations

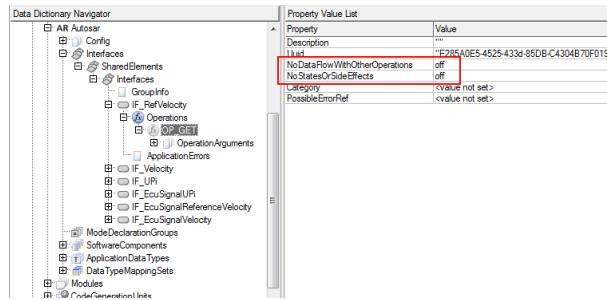
<b>Optimization of operations</b>	<p>Classic AUTOSAR operations are C functions. If you did not implement the server runnables that provide these operations with TargetLink, TargetLink treats these C functions as externally defined, which restricts optimization. However, you can instruct TargetLink to optimize the context in which these functions are called anyway.</p>
<b>Preconditions</b>	You created a DD Operation object. Refer to <a href="#">How to Create Communication Subjects for Classic AUTOSAR Communication</a> on page 103.
<b>Method</b>	<p><b>To specify code optimization settings for Classic AUTOSAR operations</b></p> <ol style="list-style-type: none"><li>1 Select the DD Operation object you want to set code optimization settings for.</li></ol>

- 2** In the Property Value List, adjust the values of the following properties, if required:

Property	Value
NoDataFlowWithOtherOperations	<p>When on, TargetLink treats this operation call as if the SIDE_EFFECT_FREE optimization property of the function class was set.</p> <p>TargetLink overrides this setting when the <a href="#">AssumeOperationCallsHaveNoUnknownDataFlow</a> on page 389 Code Generator option is enabled. Refer to <a href="#">AUTOSAR-Related Code Generator Options</a> on page 387.</p> <p><b>Note</b></p> <p>There is a semantic difference between synchronous and asynchronous Classic AUTOSAR:</p> <ul style="list-style-type: none"> <li>▪ Synchronous operation calls are essentially calls to external functions and often have no data flow with other functions or <a href="#">Classic AUTOSAR</a> operations via global variables or system states.</li> <li>▪ Asynchronous operation calls are only the first half of an <b>Rte_Call-Rte_Result</b> pair and there is always data flow between the operation call and operation result of the same <a href="#">Classic AUTOSAR</a> operation. Thus, RTE API calls to asynchronous <b>Rte_Call</b> and <b>Rte_Result</b> must never be moved past each other.</li> </ul> <p><b>NoDataFlowWithOtherOperations</b> can describe these RTE API calls' behavior only with respect to other <a href="#">Classic AUTOSAR</a> operations. If activated, RTE API calls to synchronous <b>Rte_Call</b> can be moved past RTE API calls to asynchronous <b>Rte_Call</b> or <b>Rte_Result</b> (or vice versa).</p>
NoStatesOrSideEffects	<p>When on, TargetLink treats this operation call as if the MOVABLE optimization property of the function class was set.</p> <p><b>Note</b></p> <p>For asynchronous operation calls and operation results, this property has no effect.</p>

## Result

You set the optimization settings for a DD Operation object:



## Next step

You can model client-server communication. Refer to [Modeling Client-Server Communication](#) on page 133.

## Related topics

### Basics

AUTOSAR-Related Code Generator Options.....	387
Basics on Client-Server Communication.....	133
Basics on Optimizing Functions via Property Values ( <a href="#">TargetLink Customization and Optimization Guide</a> )	

# Modeling Interrunnable Communication

## Where to go from here

## Information in this section

Basics on Interrunnable Communication.....	171
How to Create Interrunnable Variables.....	173
How to Model Interrunnable Communication.....	175

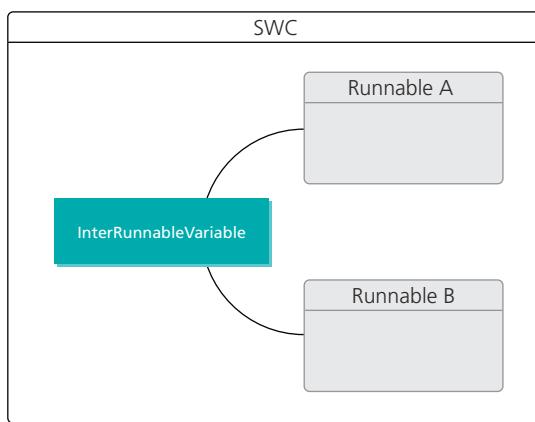
## Basics on Interrunnable Communication

### Interrunable communication

Interrunable communication is communication between the runnables of one atomic software component.

### Schematic of interruptable communication

The following illustration shows interruptable communication:



### Interrunable communication according to Classic AUTOSAR

**Data exchange between runnables of the same SWC** You can use interruptable communication if you want to model the data exchange between runnables of the same SWC.

Interruptable communication can be done implicitly (using a value copy for the run time of the runnable) or explicitly (using direct access to the variable). Interruptable communication is done via interruptable variables.

**Interruptable variables** Interruptable variables are variables that are used for interruptable communication. Each interruptable variable is referenced by the sender and the receiver runnable. According to [Classic AUTOSAR](#),

interrunnable variables are generated during RTE generation, while read/write access to interrunnable variables is part of the SWC.

#### Interrunnable communication in TargetLink

**Creating interrunnable variables in TargetLink** You can specify DD InterRunnableVariable objects in the following DD subtree:

```
/Pool/Autosar/SoftwareComponents/<SoftwareComponent>/  
InterRunnableVariables/
```

TargetLink lets you create interrunnable variables with a simple or composite structure. For instructions, refer to [How to Create Interrunnable Variables](#) on page 173.

**Modeling interrunnable communication** You can use the model elements shown in the following table to model interrunnable communication:

Model Element	Location
<b>Port blocks</b>	
InPort	Inside the receiving runnables
Bus Import	
OutPort	Inside the sending runnables
Bus Outport	
<b>Data store blocks</b>	
Data Store Memory <sup>1)</sup>	Outside the runnables
Data Store Read	Inside the receiving runnables <sup>2)</sup>
Data Store Write	Inside the sending runnables <sup>2)</sup>

<sup>1)</sup> Not required if a Data Store Memory block is specified in the Data Dictionary, refer to [How to Create a Global Data Store via Data Store Memory Blocks](#) ([TargetLink Preparation and Simulation Guide](#)).

<sup>2)</sup> You can place additional Data Store Read and Data Store Write blocks outside your runnables for simulation purposes.

For instructions, refer to [How to Model Interrunnable Communication](#) on page 175.

**RTE API functions for interrunnable communication** The following table shows you the calls to RTE API functions that TargetLink can generate for interrunnable communication:

Interrunnable Variable's Data Type	Access Kind	Generated RTE API Function
Scalar	Implicit read	Rte_IrvIRead
	Implicit write	Rte_IrvIW <sub>rite</sub>
	Explicit read	Rte_IrvRead
	Explicit write	Rte_IrvW <sub>rite</sub>
Non-scalar	Implicit read	Rte_IrvIRead
	Implicit write	Rte_IrvIW <sub>riteRef</sub> <sup>1)</sup> Rte_IrvIW <sub>rite</sub>
	Explicit read	Rte_IrvRead
	Explicit write	Rte_IrvW <sub>rite</sub>

<sup>1)</sup> Supported with [Classic AUTOSAR](#) Revision 4.2.2.

**Generating code without Rte\_IrvIWriteRef calls** By default, TargetLink generates calls to the `Rte_IrvIWriteRef` API function for implicit write access to non-scalar interruptible variables. You can force TargetLink to generate code with calls to `Rte_IrvIWrite` instead by setting the `GenerateIWriteForNonScalarIrvs` Code Generator option to `on`.

**Related topics****References**

GenerateIWriteForNonScalarIrvs.....	390
-------------------------------------	-----

## How to Create Interruptible Variables

**Objective**

To create an interruptible variable that you can use for interruptible communication.

**Preconditions**

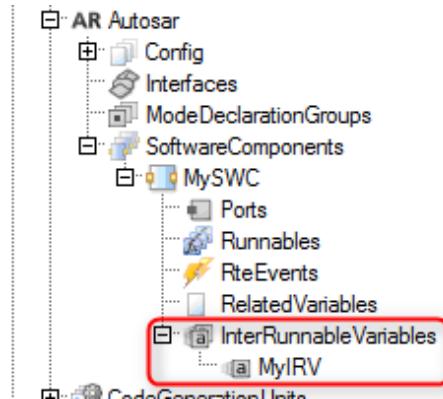
The following preconditions must be fulfilled:

- You created a software component `MySWC`.
- You specified one or more data types. Refer to:
  - [Creating Implementation Data Types from Scratch \(Classic AUTOSAR\)](#) on page 41
  - [Creating Application Data Types from Scratch \(Classic AUTOSAR\)](#) on page 49
  - [Using TargetLink's Wizards to Create Application Data Types or Implementation Data Types \(Classic AUTOSAR\)](#) on page 57
- You specified scalings and constrained range limits as required. Refer to [Defining Types, Scalings, and Constrained Range Limits \(Classic AUTOSAR\)](#) on page 31.

**Method****To create an interruptible variable**

- 1 Add the `InterRunnableVariables` subtree to `MySWC`.
  - 2 From the DD `InterRunnableVariables` subtree's context menu, select `Create InterRunnableVariable`.
- A new DD `InterRunnableVariable` object is added to the subtree.

- 3 Rename the new DD InterRunnableVariable object as required, for example: MyIRV.



- 4 Adjust the properties in MyIRV's Property Value List as required:

Property	Value
Type	Select a user-defined data type. To adjust the components of structured interruptable variables to the structured data type, use the Adjust to Typedef command from the context menu.
CommunicationMode	Specify whether this interruptable variable is used in explicit or implicit interruptable communication.
InitValueRef	Select a DD Variable that represents this interruptable variable's initialization constant.
<p><b>Note</b></p> <p><b>Avoid ambiguous initializations of data prototypes</b> Specify an initialization value for each <a href="#">data prototype</a> within the Data Dictionary whenever possible. This avoids ambiguities between <a href="#">Classic AUTOSAR</a> initialization values and initialization values of model elements that are required by Simulink.</p>	
Width	If you select an array user type, you have to specify a consistent width.

#### Result

You created an interruptable variable that you can use for interruptable communication.

#### Next steps

You can use the interruptable variable in interruptable communication. For instructions, refer to [How to Model Interruptable Communication](#) on page 175.

**Related topics****Basics**

Basics on Interrunnable Communication.....	171
Preparing SWCs for Measurement and Calibration.....	235

**HowTos**

How to Model Interrunnable Communication.....	175
---	-----

**References**

[Adjust to Typedef](#) (  TargetLink Data Dictionary Manager Reference)

## How to Model Interrunnable Communication

**Preconditions**

The following preconditions must be fulfilled:

- You created **Runnable1** and **Runnable2** that belong to the same SWC. For instructions, refer to [How to Model Runnables](#) on page 71.
- You created an interruptible variable. For instructions, refer to [How to Create Interrunnable Variables](#) on page 173.

**Possible Methods**

There are two ways of modeling interruptible communication:

- Using port blocks. Refer to Method 1.
- Using data store blocks. Refer to Method 2.

**Method 1****To model interruptible communication by using port blocks**

- 1 Add an OutPort or Bus Outport block to runnable **Runnable1** and open its block dialog.  
On the AUTOSAR page, set AUTOSAR mode to **Classic**.
- 2 Select **InterRunnable** from the Kind list.
- 3 Use the Interruptible variable Browse button to choose a DD **InterRunnableVariable** object.



- 4 Add an InPort or Bus Import block to runnable **Runnable2** and repeat the above steps in the same way to model the receiver side of the interruptible communication.
- 5 Connect the port blocks in the model with matching signals.

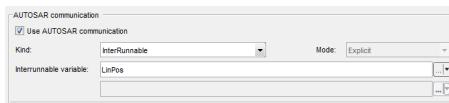
**Method 2****To model interruptable communication by using data store blocks**

- 1 Add a Data Store Write block to Runnable1.
- 2 Add a Data Store Read block to Runnable2.
- 3 Place a Data Store Memory block outside of the runnable subsystems but within the TargetLink subsystem that contains the runnable subsystems. Make sure that all three data store blocks use the same data store by specifying the same data store name.

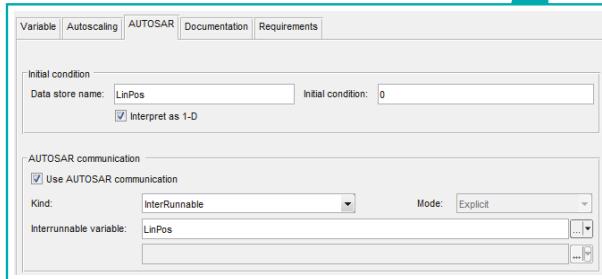
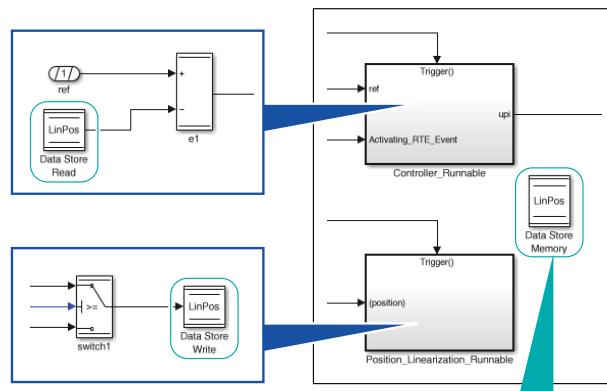
**Tip**

Data Store Memory blocks can also be created and specified in the Data Dictionary, refer to [How to Create a Global Data Store via Data Store Memory Blocks](#) ( [TargetLink Preparation and Simulation Guide](#)).

- 4 On the AUTOSAR page of the Data Store Memory block, set AUTOSAR mode to **Classic**.
- 5 Select **InterRunnable** from the Kind list.
- 6 Use the **Interruptable variable** Browse button to choose a DD **InterRunnableVariable** object.



- 7 Connect the Data Store Read/Data Store Write blocks with matching signals.



**Result**

You defined interruptible communication between Runnable1 and Runnable2.

**Related topics****Basics**

[Basics on Interrunnable Communication](#)..... 171

**HowTos**

[How to Create Interrunnable Variables](#)..... 173

[How to Model Runnables](#)..... 71

**References**

[Bus Import Block](#) ( TargetLink Model Element Reference)

[Bus Outport Block](#) ( TargetLink Model Element Reference)

[Data Store Memory Block](#) ( TargetLink Model Element Reference)

[InPort Block](#) ( TargetLink Model Element Reference)

[OutPort Block](#) ( TargetLink Model Element Reference)

# Modeling NvData Communication

## Where to go from here

## Information in this section

NvData Communication.....	178
Reducing Rewrites to NVRAM.....	190
Modeling Styles for Reducing Rewrites to NVRAM.....	194

# NvData Communication

## Where to go from here

## Information in this section

Basics on NvData Communication.....	179
Example of Modeling NvData Communication via Port Blocks.....	182
Example of Modeling NvData Communication via Data Store Memory Blocks.....	185
Example of Modeling NvData Communication via Block Parameters.....	188

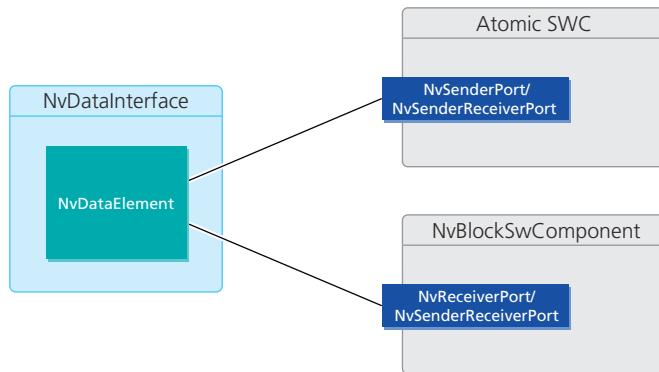
## Basics on NvData Communication

### NvData communication

To model data exchange between an atomic SWC and the NVRAM, you can use NvData communication.

### NvData according to Classic AUTOSAR

[Classic AUTOSAR](#) supports NvData communication that uses the same mechanisms as sender-receiver communication.



In NvData (nonvolatile data) communication, an atomic software component (`ApplicationSwComponentType`) can communicate via an NvData interface with another software component (`NvBlockSwComponent`) that manages the access to the NVRAM. The involved buffers depend on the kinds of ports that are used for communication:

- Separate buffer for reading and writing:
  - Require port
  - Provide port
- Combined buffer for reading and writing:
  - Provide-require port

### NvData in TargetLink

TargetLink supports NvData communication as described above and lets you specify NvData communication in the Data Dictionary.

**Communication mode** TargetLink supports [explicit](#) and [implicit](#) NvData communication.

For implicit NvData communication, TargetLink can generate the `Rte_IWrite` or `Rte_IWriteRef` RTE API functions. You can control which RTE API is used via the Mode property on the AUTOSAR page of the OutPort, Bus Outport, and Data Store Memory blocks.

**Creating NvData interfaces** You can create DD NvDataInterface objects in the `/Pool/Autosar/Interfaces/` subtree.

For instructions, refer to [How To Create Interfaces](#) on page 100.

**Creating NvData ports** You can create NvData ports in the /Pool/Autosar/SoftwareComponents/<MySWC>/Ports subtree. The following DD <Port> objects are available:

Kind of Access	DD <Port> Object
Read	NvReceiverPort
Write	NvSenderPort
Read-write	NvSenderReceiverPort

For instructions, refer to [How to Create Ports](#) on page 101.

**Creating NvData elements** You can create DD NvDataElement objects in the /Pool/Autosar/Interfaces/<NvDataInterface> subtree. TargetLink lets you create NvData elements with a simple structure or with a composite structure.

For instructions, refer to [How to Create Communication Subjects for Classic AUTOSAR Communication](#) on page 103.

**Initializing NvData elements** You can initialize your NvData elements.

#### Note

##### Avoid ambiguous initializations of data prototypes

Specify an initialization value for each [data prototype](#) within the Data Dictionary whenever possible. This avoids ambiguities between [Classic AUTOSAR](#) initialization values and initialization values of model elements that are required by Simulink.

Kind of Access	DD <Port> Object	DD <Comspec> Object	DD <Init> Property
Read	NvReceiverPort	NvReceiverComSpec	InitValueRef
Write	NvSenderPort	NvSenderComSpec	RamBlockInitValueRef <sup>1)</sup>
Read-write	NvSenderReceiverPort	NvReceiverComSpec NvSenderComSpec	InitValueRef <sup>2)</sup> RamBlockInitValueRef <sup>1), 2)</sup>

<sup>1)</sup> The RamBlockInitValueRef property is ignored by the Code Generator. Its value is relevant only in imports or exports.

<sup>2)</sup> If a single NvDataElement references both a NvReceiverComSpec and a NvSenderComSpec object, the InitValueRef and RamBlockInitValueRef properties must not conflict. Either, you reference the same variable at both properties or you only use one of the properties to specify an initialization value.

#### NvData in the model

TargetLink offers various approaches for modeling NvData communication. You can combine the following approaches as required:

Modeling Approaches		
Port blocks	Data Store Memory blocks	Block parameters

Modeling Approaches		
Port blocks	Data Store Memory blocks	Block parameters
<b>Supported Access Kinds</b>		
▪ Read ▪ Write ▪ Read-Write		▪ Read <sup>1), 2)</sup>
<b>Instructions</b>		
Example of Modeling NvData Communication via Port Blocks on page 182.	Example of Modeling NvData Communication via Data Store Memory Blocks on page 185.	Example of Modeling NvData Communication via Block Parameters on page 188.

<sup>1)</sup> Restricted to implicit communication.

<sup>2)</sup> Works for variables created from DD NvReceiverPort and NvSenderReceiverPort objects. You can create these variables via the Synchronize Interface Settings context menu command of <Port> objects.

#### Simulation differences

Due to Simulink semantics, differences between MIL and SIL simulation modes might occur. This happens when read operations and write operations are performed on the *same* buffer during the *same* simulation step.

These situations can occur when you model your NvData accesses by using port blocks or block parameters and all of the following conditions hold:

- The same DD NvSenderReceiverPort is used at different blocks.
- The same DD NvDataElement is used at different blocks.
- A modeled write access precedes a read access modeled via a port block or block parameter.

If you need consistent simulation behavior across the different simulation modes, model your NvData accesses by using Data Store Memory blocks.

#### NvData in production code

The production code generated for NvData communication looks like this:

##### Read

```
sint16 MyNvDataElement;
MyNvDataElement = Rte_IRead_MyRunnable_NvReceiverPort_MyNvDataElement();
```

##### Write

```
sint16 * p_MyNvDataElement;
p_MyNvDataElement = Rte_IWriteRef_MyRunnable_NvSenderPort_MyNvDataElement();
*p_MyNvDataElement = 0;
```

TargetLink lets you reduce unnecessary write accesses to the NVRAM. For details, refer to [Basics on Reducing Rewrites to NVRAM](#) on page 190.

**Related topics****Basics**

[Basics on Reducing Rewrites to NVRAM.....](#) 190

**HowTos**

<a href="#">How to Create Communication Subjects for Classic AUTOSAR Communication.....</a>	103
<a href="#">How To Create Interfaces.....</a>	100
<a href="#">How to Create Ports.....</a>	101
<a href="#">How to Reduce Write Accesses to the NVRAM.....</a>	192

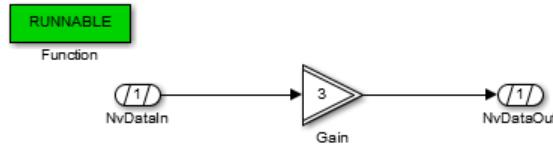
**Examples**

<a href="#">Example of Modeling NvData Communication via Block Parameters.....</a>	188
<a href="#">Example of Modeling NvData Communication via Data Store Memory Blocks.....</a>	185
<a href="#">Example of Modeling NvData Communication via Port Blocks.....</a>	182

## Example of Modeling NvData Communication via Port Blocks

**Model overview**

This example uses the following simple model:

**Tip**

This example shows [implicit NvData communication](#) and how to generate `Rte_IWriteRef` for write access. You can also specify [explicit NvData communication](#) or let TargetLink generate `Rte_IWrite` for write access. Refer to [Basics on NvData Communication](#) on page 179.

**Preconditions**

The following preconditions must be fulfilled:

- You created a DD NvDataInterface object, such as `MyNvDataInterface`.
- You created a DD NvDataElement object, such as `MyNvDataElement`.
- You created a DD SoftwareComponent object SWC, such as `MySwc`.

- You created a DD NvReceiverPort and a DD NvSenderPort object, such as `MyNvReceiverPort` and `MyNvSenderPort`.
- You created a DD Runnable object, such as `MyRunnable`.

**Method****To model NvData communication via a port block**

- 1** In the model, open the subsystem of `MyRunnable`.
- 2** Add the following blocks, depending on the data type of `MyNvDataElement`:
  - An InPort or Bus Import block
  - An OutPort or Bus Outport block
- 3** On the AUTOSAR page of the InPort/Bus Import block, make the following settings:

Control	Value
AUTOSAR mode	Classic
Kind	NvData
Mode	Implicit
NV port	<code>MyNvReceiverPort</code>
NV data element	<code>MyNvDataElement</code>

Close the block dialog to confirm your settings.

- 4** On the AUTOSAR page of the OutPort or Bus Outport block, make the following settings:

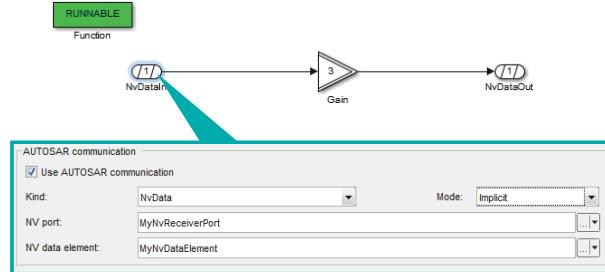
Control	Value
AUTOSAR mode	Classic
Kind	NvData
Mode	Implicit (IWriteRef)
NV port	<code>MyNvSenderPort</code>
NV data element	<code>MyNvDataElement</code>

Close the block dialog to confirm your settings.

- 5** Model the runnable's function as required.
- 6** Generate code.

**Result**

You modeled NvData communication by using port blocks. The settings in the block dialog look like this:



The runnable's code looks like this:

```
FUNC(void, MySWC_CODE) MyRunnable(void)
{
    /* SLLocal: Default storage class for Local variables | Width: 16 */
    sint16 MyNvDataElement;
    sint16 Sa1_Gain;

    /* SLLocal: Default storage class for Local variables | Width: 32 */
    sint16 * p_MyNvDataElement;

    /* TargetLink outport: NvDataViaPort/NvDataOut */
    p_MyNvDataElement = Rte_IWriteRef_MyRunnable_NvSenderPort_MyNvDataElement();

    /* TargetLink inport: NvDataViaPort/NvDataIn */
    MyNvDataElement = Rte_IRead_MyRunnable_NvReceiverPort_MyNvDataElement();

    /* Gain: NvDataViaPort/Gain */
    Sa1_Gain = MyNvDataElement * 3;

    /* TargetLink outport: NvDataViaPort/NvDataOut */
    *p_MyNvDataElement = Sa1_Gain;
}
```

**Related topics****Basics**

Basics on NvData Communication.....	179
Basics on Reducing Rewrites to NVRAM.....	190

**HowTos**

How to Create Communication Subjects for Classic AUTOSAR Communication.....	103
How To Create Interfaces.....	100
How to Create Ports.....	101
How to Create Software Components.....	66
How to Model Runnables.....	71
How to Reduce Write Accesses to the NVRAM.....	192

**Examples**

Example of Modeling NvData Communication via Block Parameters.....	188
Example of Modeling NvData Communication via Data Store Memory Blocks.....	185

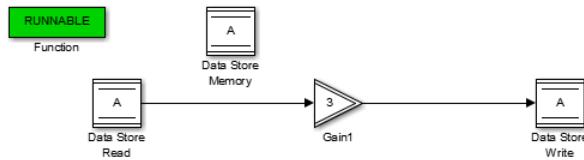
**References**

Bus Import Block ( TargetLink Model Element Reference)	
Bus Outport Block ( TargetLink Model Element Reference)	
InPort Block ( TargetLink Model Element Reference)	
OutPort Block ( TargetLink Model Element Reference)	

## Example of Modeling NvData Communication via Data Store Memory Blocks

**Model overview**

This example uses the following simple model:

**Tip**

This example shows [implicit NvData communication](#) and how to generate `Rte_IWriteRef` for write access. You can also specify [explicit NvData communication](#) or let TargetLink generate `Rte_IWrite` for write access. Refer to [Basics on NvData Communication](#) on page 179.

**Preconditions**

The following preconditions must be fulfilled:

- You created a DD NvDataInterface object, such as `MyNvDataInterface`.
- You created a DD NvDataElement object, such as `MyNvDataElement`.
- You created a DD SoftwareComponent object, such as `MySwc`.
- You created a DD NvSenderReceiverPort object, such as `MyNvSenderReceiverPort`.
- You created a DD Runnable object, such as `MyRunnable`.

**Method****To model NvData communication via a Data Store Memory block**

**1** In the model, open `MyRunnable`'s subsystem.

**2** Add the following blocks:

- A Data Store Memory block

**Tip**

Data Store Memory blocks can also be created and specified in the Data Dictionary, refer to [How to Create a Global Data Store via Data Store Memory Blocks](#) ( [TargetLink Preparation and Simulation Guide](#)).

- A Data Store Read block
- A Data Store Write block

**3** On the AUTOSAR page of the Data Store Memory block, make the following settings:

Control	Value
AUTOSAR mode	Classic
Kind	NvData
Mode	Implicit (IWriteRef)
NV port	<code>MyNvSenderReceiverPort</code>
NV data element	<code>MyNvDataElement</code>

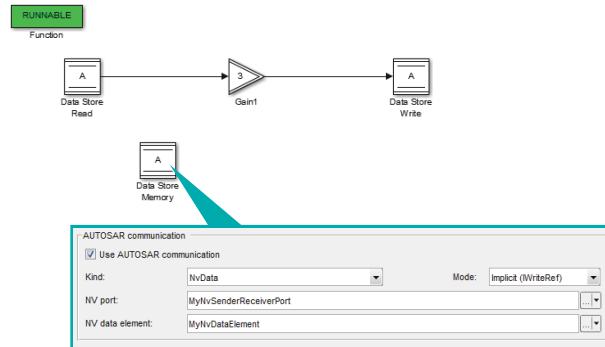
Close the block dialog to confirm your settings.

**4** Model the runnable's function as required.

**5** Generate code.

**Result**

You modeled NvData communication by using data store blocks. The settings in the block dialog look like this:



The runnable's code looks like this:

```

FUNC(void, MySWC_CODE) MyRunnable(void)
{
    /* $Local: Default storage class for Local variables | Width: 16 */
    sint16 MyNvDataElement;
    sint16 Sb1_Gain1;
    /* $Local: Default storage class for Local variables | Width: 32 */
    sint16 * p_MyNvDataElement;
    /* Data store write: NvDataViaDSM/Data Store Write */
    p_MyNvDataElement = Rte_IWriteRef_MyRunnable_NvSenderReceiverPort_MyNvDataElement();
    /* Data store read: NvDataViaDSM/Data Store Read */
    MyNvDataElement = Rte_IRead_MyRunnable_NvSenderReceiverPort_MyNvDataElement();
    /* Data store write: NvDataViaDSM/Data Store Write */
    *p_MyNvDataElement = Sb1_Gain1;
}
/* Gain: NvDataViaDSM/Gain1 */
Sb1_Gain1 = MyNvDataElement * 3;

```

**Related topics****Basics**

Basics on NvData Communication.....	179
Basics on Reducing Rewrites to NVRAM.....	190

**HowTos**

How to Reduce Write Accesses to the NVRAM.....	192
--	-----

**Examples**

Example of Modeling NvData Communication via Block Parameters.....	188
--	-----

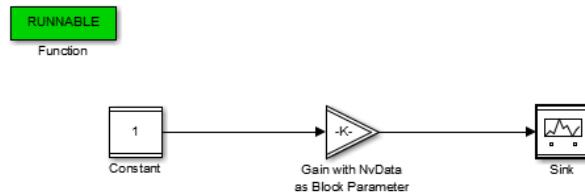
**References**

Data Store Memory Block ( TargetLink Model Element Reference)	
---	--

## Example of Modeling NvData Communication via Block Parameters

### Model overview

This example uses the following simple model:



### Restrictions

Modeling NvData communication via block parameters is restricted to read access in implicit communication.

You can model this read access by using variables created from DD NvReceiverPort and DD NvSenderReceiverPort objects.

### Preconditions

The following preconditions must be fulfilled:

- You created an NvData interface, such as MyNvDataInterface.
- You created an NvData element, such as MyNvDataElement.
- You created an SWC, such as MySwc.
- You created an NvData port, such as MyNvReceiverPort.
- You created a runnable, such as MyRunnable.

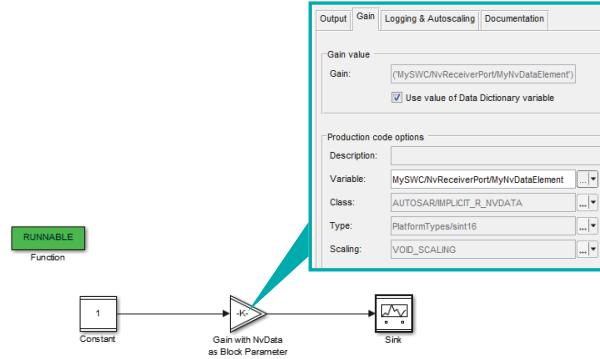
### Method

#### To model NvData communication via a block parameter

- 1 In the Data Dictionary Manager, open MyNvReceiverPort's context menu and select Synchronize Interface Settings.  
TargetLink creates suitable DD Variable objects for all the NvData elements communicated via the DD NvDataInterface object that is referenced at MyNvReceiverPort.
- 2 In the model, open MyRunnable's subsystem.
- 3 Locate the block whose block parameter you want to use for NvData communication and open its block dialog.
- 4 Click the Browse button of the appropriate Variable edit field and select the variable that TargetLink created in step 1:  
**/Pool/Variables/MySWC/NvReceiverPort/MyNvDataElement**  
For struct variables you can select from the struct variables' components.
- 5 Close the block dialog to confirm your settings.
- 6 Model the runnable's function as required.
- 7 Generate code.

**Result**

You modeled NvData communication by using block parameters. The settings in the block dialog look like this:



The runnable's code looks like this:

```
FUNC(void, MySWC_CODE) MyRunnable(void)
{
    /* $Local: Default storage class for Local variables | Width: 16 */
    sint16 Sc1_Gain_with__Block_Parameter;
    sint16 MyNvDataElement /* LSB: 2^0 OFF: 0 MIN/MAX: -32768 .. 32767 */;

    MyNvDataElement = Rte_IRead_MyRunnable_NvReceiverPort_MyNvDataElement();

    /* Gain: NvDataViaBlockParams/Gain with NvData as Block Parameter */
    Sc1_Gain_with__Block_Parameter = MyNvDataElement;
}
```

**Related topics****Basics**

Basics on NvData Communication.....	179
Basics on Reducing Rewrites to NVRAM.....	190

**HowTos**

How to Reduce Write Accesses to the NVRAM.....	192
--	-----

**Examples**

Example of Modeling NvData Communication via Data Store Memory Blocks.....	185
Example of Modeling NvData Communication via Port Blocks.....	182

**References**

Synchronize Interface Settings (🔗 TargetLink Data Dictionary Manager Reference)
---

# Reducing Rewrites to NVRAM

## Where to go from here

## Information in this section

Basics on Reducing Rewrites to NVRAM.....	190
How to Reduce Write Accesses to the NVRAM.....	192

## Basics on Reducing Rewrites to NVRAM

### Nonvolatile memory

Nonvolatile random access memory (NVRAM) can be used to persistently store ECU states.

However, writing to NVRAM has a few disadvantages:

- The write speed is comparably slow.
- The total amount of rewrites is limited.

### Write accesses

TargetLink can generate write accesses to the NVRAM as calls to `Rte_IWriteRef` RTE API functions.

After `Rte_IWriteRef` RTE API function is called, the RTE is responsible for managing write operations to the NVRAM.

When these write operations occur depends on the implementation details of the `NvBlockSwComponent` SWC that manages the access to the NVRAM. For example, access to the NVRAM can be implemented using the *dirty flag mechanism* as specified by [Classic AUTOSAR](#).

### Reducing write operations

TargetLink lets you generate code that reduces write operations to the NVRAM.

This is achieved by comparing the value to be written with its representation in the RTE buffer. The new value is written to the buffer via a call to the `Rte_IWriteRef` function only if the values differ. This comparison is not made for the whole `NvData` element but only for the elements or components that might have changed in the model.

If the value did not change, no call to the `Rte_IWriteRef` RTE API function is executed in the generated code. This results in fewer write operations to the NVRAM.

**Note**

To reduce rewrites to the NVRAM, you have to use a combined buffer for read accesses and write accesses. This is the case for provide-require ports as described by Classic AUTOSAR.

Additionally, you have to instruct TargetLink to generate the `Rte_IWriteRef` RTE API function for write accesses.

For details, refer to [Basics on NvData Communication](#) on page 179.

**Example code** The code looks like this:

```

1 p_DE_Vector = Rte_IRead_Run_0_PR_Port_NVI_DE_Vector();
2 if (p_DE_Vector[2] != Sa2_Assignment[2]) {
3     p_DE_Vector_a = Rte_IWriteRef_Run_0_PR_Port_NVI_DE_Vector();
4     p_DE_Vector_a[2] = Sa2_Assignment[2];
5 }
```

**Reducing rewrites to NVRAM**

You can reduce rewrites to the NVRAM by enabling the `ReduceWriteOperations` property of DD `NvDataElement` objects.

You have to model access to `NvDataElement` objects whose `ReduceWriteOperations` property is set to `on` by using special modeling styles that trigger code pattern.

**Note**

- Reducing rewrites to the NVRAM is not part of TargetLink's code optimization but depends on defined modeling styles for accessing NvData.  
Each modeling style determines a code pattern that might be optimized further.
- All the write accesses involving a DD `NvDataElement` object whose `ReduceWriteOperations` property is set to `on` are generated by using the code pattern for reducing rewrites to the NVRAM.  
If you use a modeling style that does not support reducing rewrites to the NVRAM, TargetLink displays an error message.

**Modeling styles**

There are various modeling styles, depending on the data type of the NvData element and how you want to access it:

Data Type	Use Case	Instructions
Matrix, vector	Writing elements of non-scalar (vector or matrix) variables.	<a href="#">Reducing Write Accesses to NVRAM when Accessing Elements of Vector or Matrix Variables</a> on page 194

Data Type	Use Case	Instructions
Struct	Writing components. Components can be scalar or non-scalar (vector or matrix) variables.	<a href="#">Reducing Write Accesses to NVRAM when Accessing Components of a Struct on page 197</a>
	Writing elements of non-scalar components (vector or matrix variables).	<a href="#">Reducing Write Accesses to NVRAM when Accessing Elements of Non-Scalar Struct Components (Vector or Matrix Variables) on page 199</a>
Any	Writing one or more signal elements. These can be elements of vectors, matrices and buses (i.e. struct components).	<a href="#">Reducing Write Accesses to NVRAM by Modeling With Data Store Blocks on page 203</a>

**Related topics****Basics**

<a href="#">Basics on NvData Communication.....</a>	179
<a href="#">Modeling Styles for Reducing Rewrites to NVRAM.....</a>	194

**HowTos**

<a href="#">How to Reduce Write Accesses to the NVRAM.....</a>	192
--	-----

## How to Reduce Write Accesses to the NVRAM

**Objective**

To generate code that contains reduced write accesses to the NVRAM.

**Write accesses**

TargetLink can generate write accesses to the NVRAM as calls to `Rte_IWriteRef` RTE API functions.

After `Rte_IWriteRef` RTE API function is called, the RTE is responsible for managing write operations to the NVRAM.

When these write operations occur depends on the implementation details of the `NvBlockSwComponent` SWC that manages the access to the NVRAM. For example, access to the NVRAM can be implemented using the *dirty flag mechanism* as specified by [Classic AUTOSAR](#).

**Restrictions**

Reducing write accesses to the NVRAM is restricted to clearly defined modeling styles. Refer to [Modeling Styles for Reducing Rewrites to NVRAM on page 194](#).

**Preconditions**

You modeled NvData communication as described in [Basics on NvData Communication on page 179](#).

**Method****To reduce write accesses to the NVRAM**

- 1 Select the DD NvDataElement object that you want to reduce write operations for.
- 2 Set its ReduceWriteOperations property to **on**.
- 3 Model the access to this NvData element as described in [Modeling Styles for Reducing Rewrites to NVRAM](#) on page 194.
- 4 Generate code.

**Result**

You generated code that contains reduced write accesses to the NVRAM.

The code looks like this:

```

1 p_DE_Vector = Rte_IRead_Run_0_PR_Port_NVI_DE_Vector();
2 if (p_DE_Vector[2] != Sa2_Assignment[2]) {
3     p_DE_Vector_a = Rte_IWriteRef_Run_0_PR_Port_NVI_DE_Vector();
4     p_DE_Vector_a[2] = Sa2_Assignment[2];
5 }
```

**Related topics****Basics**

Basics on NvData Communication.....	179
Basics on Reducing Rewrites to NVRAM.....	190
Reducing Write Accesses to NVRAM by Modeling With Data Store Blocks.....	203
Reducing Write Accesses to NVRAM when Accessing Components of a Struct.....	197
Reducing Write Accesses to NVRAM when Accessing Elements of Non-Scalar Struct Components (Vector or Matrix Variables).....	199
Reducing Write Accesses to NVRAM when Accessing Elements of Vector or Matrix Variables.....	194

**Examples**

Example of Modeling NvData Communication via Block Parameters.....	188
Example of Modeling NvData Communication via Data Store Memory Blocks.....	185

# Modeling Styles for Reducing Rewrites to NVRAM

## Where to go from here

## Information in this section

Reducing Write Accesses to NVRAM when Accessing Elements of Vector or Matrix Variables.....	194
Reducing Write Accesses to NVRAM when Accessing Components of a Struct.....	197
Reducing Write Accesses to NVRAM when Accessing Elements of Non-Scalar Struct Components (Vector or Matrix Variables).....	199
Reducing Write Accesses to NVRAM by Modeling With Data Store Blocks.....	203

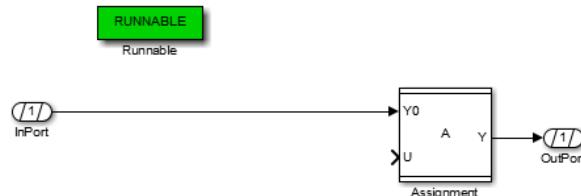
## Reducing Write Accesses to NVRAM when Accessing Elements of Vector or Matrix Variables

### Reducing rewrites to NVRAM

If you set a DD NvDataElement object's ReduceWriteOperations property to **on**, you instruct TargetLink to generate code that reduces write operations to the NVRAM.

### Modeling style

To access elements of NvData elements that are vector or matrix variables, you have to use the following modeling style:



### Note

This modeling style is also supported for data store blocks.

Ensure that your blocks are specified according to the following table.

<b>Block</b>	<b>Connection</b>	<b>Key Property</b>	<b>Value</b>	<b>Description</b>
InPort/Data Store Read <sup>1), 2)</sup>	Direct connection <sup>3)</sup> to Y0 of Assignment block	AUTOSAR mode	Classic	Enables NvData communication.
		Kind	NvData	
		NV port	Same DD NvSenderReceiverPort object as OutPort	Ensures that the same RTE buffer is used for read and write operations.
		NV data element	Same DD NvDataElement object as OutPort	Ensures that the same NvData element is read or written.
Assignment	Direct connection <sup>3)</sup> of InPort to Y0 and of Y to OutPort.	1st dimension/2nd dimension	Matching the NvData element referenced at Inport and Outport.	Ensures that the signal is not changed before or after the assignment is made.
OutPort/Data Store Write <sup>1), 4)</sup>	Direct connection <sup>3)</sup> from Y of Assignment block	AUTOSAR mode	Classic	Enables NvData communication and generates Rte_IWriteRef.
		Kind	NvData	
		Mode	Implicit (IWriteRef)	
		NV port	Same DD NvSenderReceiverPort object as InPort	Ensures that the same RTE buffer is used for read and write operations.
		NV data element	Same DD NvDataElement object as InPort	Ensures that the same NvData element is read or written.

<sup>1)</sup> The key settings are made at the Data Store Memory block.

<sup>2)</sup> Each InPort/Data Store Read block results in a call of the Rte\_IRead RTE API function.

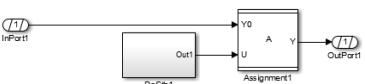
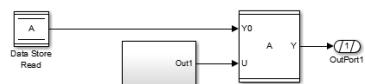
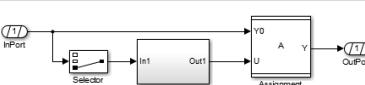
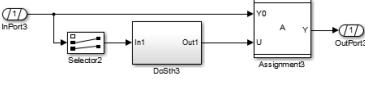
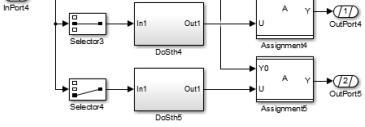
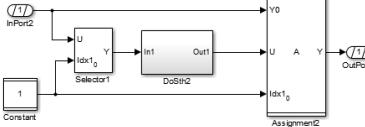
<sup>3)</sup> Simulink ports in virtual subsystems and Simulink From and Goto blocks can reside in between.

<sup>4)</sup> Each OutPort/Data Store Write block results in a call of the Rte\_IWriteRef RTE API function.

#### Extending the modeling style

Aside from the specifications listed in the table above, you can extend this modeling style as required.

The following table lists some examples:

Extension	Description
	The manipulation of the signal is contained in a subsystem. It can contain a selection of one or more elements.
	Same as first example, but a Data Store Read block is used instead of the InPort block.
	Selection of a single element via a Selector block. Only the index of the Assignment block determines which element is written.
	Selection of two elements that are manipulated in the same way.
	Selection of two elements, each of which is manipulated independently and written by a unique Rte_IWriteRef function.
	Use of a variable index for selection. Only the index of the Assignment block determines which element is written.

## Related topics

### Basics

Basics on NvData Communication.....	179
Basics on Reducing Rewrites to NVRAM.....	190
Reducing Write Accesses to NVRAM by Modeling With Data Store Blocks.....	203
Reducing Write Accesses to NVRAM when Accessing Components of a Struct.....	197
Reducing Write Accesses to NVRAM when Accessing Elements of Non-Scalar Struct Components (Vector or Matrix Variables).....	199

### HowTos

How to Reduce Write Accesses to the NVRAM.....	192
--	-----

### References

Data Store Memory Block (  TargetLink Model Element Reference)
---

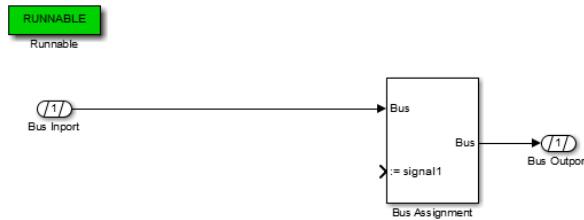
## Reducing Write Accesses to NVRAM when Accessing Components of a Struct

### Reducing rewrites to NVRAM

If you set a DD NvDataElement object's ReduceWriteOperations property to **on**, you instruct TargetLink to generate code that reduces write operations to the NVRAM.

### Modeling style

To access components of NvData elements that are struct variables, you have to use the following modeling style:



#### Note

This modeling style is also supported for data store blocks that do not have elements assigned in the Simulink block dialog.

If you want to model access to the NVRAM by using data store blocks that have elements assigned in the Simulink block dialog, refer to [Reducing Write Accesses to NVRAM by Modeling With Data Store Blocks](#) on page 203.

Ensure that your blocks are specified according to the following table.

Block	Connection	Key Property	Value	Description
Bus Import/Data Store Read <sup>1), 2)</sup>	Direct connection <sup>3)</sup> to Bus input of Bus Assignment block	AUTOSAR mode	Classic	Enables NvData communication.
		Kind	NvData	
		NV port	Same DD NvSenderReceiverPort object as Bus Outport	Ensures that the same RTE buffer is used for read and write operations.
		NV data element	Same DD NvDataElement object as Bus Outport	Ensures that the same NvData element is read or written.
Bus Assignment	Direct connection <sup>3)</sup> of Bus Import to Bus input and of Bus outport to Bus Outport.	Signals that are being assigned	The signal that represents the struct component to be manipulated. The signal must be a part of the NvData element referenced at Bus Import and Bus Outport.	Ensures that the signal is not changed before or after the assignment is made.

Block	Connection	Key Property	Value	Description
Bus Outport/Data Store Write <sup>1), 4)</sup>	Direct connection <sup>3)</sup> from Bus outport of Bus Assignment block	AUTOSAR mode	Classic	Enables NvData communication and generates Rte_IWriteRef.
		Kind	NvData	
		Mode	Implicit (IWriteRef)	
	NV port	Same DD NvSenderReceiverPort object as Bus Import		Ensures that the same RTE buffer is used for read and write operations.
		NV data element	Same DD NvDataElement object as Bus Import	Ensures that the same NvData element is read or written.

1) The key settings are made at the Data Store Memory block.

2) Each Bus Import/Data Store Read block results in a call of the Rte\_IRead RTE API function.

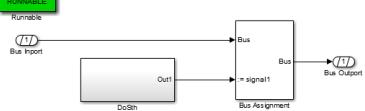
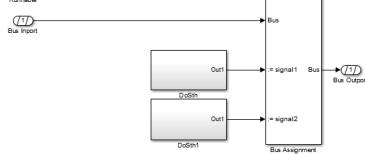
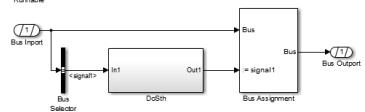
3) Simulink ports in virtual subsystems and Simulink From and Goto blocks can reside in between.

4) Each Bus Outport/Data Store Write block results in a call of the Rte\_IWriteRef RTE API function.

### Extending the modeling style

Aside from the specifications listed in the table above, you can extend this modeling style as required.

The following table lists some examples:

Extension	Description
	Direct manipulation of the signal.
	Manipulation of two components. Note that the Bus Assignment block allows assignments to more than one signal.
	Selection of a signal via a Bus Selector block before manipulation. Only the Bus Assignment block determines which signal of the bus is written.

**Related topics****Basics**

Basics on NvData Communication.....	179
Basics on Reducing Rewrites to NVRAM.....	190
Reducing Write Accesses to NVRAM by Modeling With Data Store Blocks.....	203
Reducing Write Accesses to NVRAM when Accessing Elements of Non-Scalar Struct Components (Vector or Matrix Variables).....	199
Reducing Write Accesses to NVRAM when Accessing Elements of Vector or Matrix Variables.....	194

**HowTos**

How to Reduce Write Accesses to the NVRAM.....	192
--	-----

**References**

[Data Store Memory Block](#) ( TargetLink Model Element Reference)

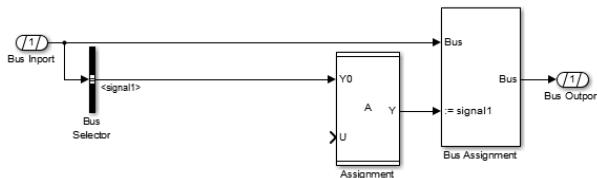
## Reducing Write Accesses to NVRAM when Accessing Elements of Non-Scalar Struct Components (Vector or Matrix Variables)

**Reducing rewrites to NVRAM**

If you set a DD NvDataElement object's ReduceWriteOperations property to **on**, you instruct TargetLink to generate code that reduces write operations to the NVRAM.

**Modeling style**

To access elements of non-scalar struct components, you have to use the following modeling style:

**Note**

This modeling style is also supported for data store blocks that do not have elements assigned in the Simulink block dialog.

If you want to model access to the NVRAM by using data store blocks that have elements assigned in the Simulink block dialog, refer to [Reducing Write Accesses to NVRAM by Modeling With Data Store Blocks](#) on page 203.

**Note**

Ensure that the following signals are the *same*:

- The signal selected by the Bus Selector block
- The signal at Y0 of the Assignment block
- The signal assigned by the Bus Assignment block

If the signals are not the same, TargetLink writes the entire component. This means that TargetLink switches to the code resulting from the modeling style described in [Reducing Write Accesses to NVRAM when Accessing Components of a Struct](#) on page 197.

Ensure that your blocks are specified according to the following table.

<b>Block</b>	<b>Connection</b>	<b>Key Property</b>	<b>Value</b>	<b>Description</b>
Bus Import/Data Store Read <sup>1), 2)</sup>	Direct connection <sup>3)</sup> to Bus import of Bus Assignment block and Bus Selector block.	AUTOSAR mode	Classic	Enables NvData communication.
		Kind	NvData	
		NV port	Same DD NvSenderReceiverPort object as Bus Outport	Ensures that the same RTE buffer is used for read and write operations.
		NV data element	Same DD NvDataElement object as Bus Outport	Ensures that the same NvData element is read or written.
Bus Selector	Direct connection <sup>3)</sup> from Bus Import and to Y0 of Assignment block.	Selected signals	The signal representing the non-scalar component whose element you want to manipulate. Must be the same signal as Y0 at the Assignment block and the signal assigned at the Bus Assignment block.	Selects the signal that represents the struct component whose elements you want to manipulate.
Assignment	Direct connection <sup>3)</sup> of the Bus Selector block to Y0 and of Y to the Bus Assignment block's := signal<n> import (assigned import).	1st dimension/2nd dimension	Matching the signal selected by the Bus Selector block. This signal represents the component of the NvData element whose elements you want to manipulate. Must be the same signal as the one selected by the Bus Selector block and the one assigned by the Bus Assignment block.	Ensures that the signal is not changed before or after the assignment is made.

Block	Connection	Key Property	Value	Description
Bus Assignment	Direct connection <sup>3)</sup> of Bus Import to Bus import, Y of the Assignment block to := signal<n> import and Bus to Bus Outport .	Signals that are being assigned	Matching the signal output by the Assignment block. This signal represents the struct component whose elements you want to manipulate. The signal must be a part of the NvData element referenced at Bus Import and Bus Outport.	Ensures that the signal is not changed before or after the assignment is made.
Bus Outport/Data Store Write <sup>1), 4)</sup>	Direct connection <sup>3)</sup> from Bus of Bus Assignment block	AUTOSAR mode	Classic	Enables NvData communication and generates Rte_IWriteRef.
		Kind	NvData	
		Mode	Implicit (IWriteRef)	
		NV port	Same DD NvSenderReceiverPort object as Bus Import	Ensures that the same RTE buffer is used for read and write operations.
		NV data element	Same DD NvDataElement object as Bus Import	Ensures that the same NvData element is read or written.

<sup>1)</sup> The key settings are made at the Data Store Memory block.

<sup>2)</sup> Each Bus Import/Data Store Read block results in a call of the Rte\_IRead RTE API function.

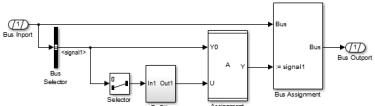
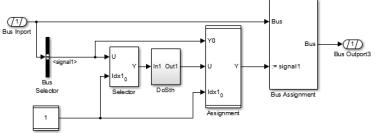
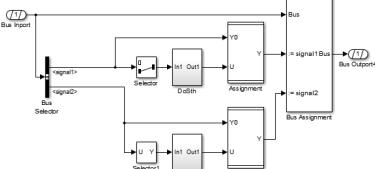
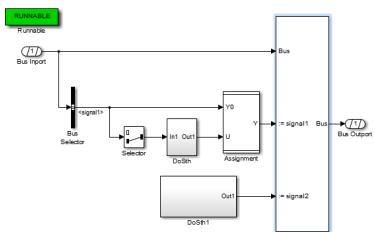
<sup>3)</sup> Simulink ports in virtual subsystems and Simulink From and Goto blocks can reside in between.

<sup>4)</sup> Each Bus Outport/Data Store Write block results in a call of the Rte\_IWriteRef RTE API function.

#### Extending the modeling style

Aside from the specifications listed in the table above, you can extend this modeling style as required.

The following table lists some examples:

Extension	Description
	Selection of a single element by using a Selector block.
	Selection of a single element by using a variable index modeled via a Constant block.
	Selection of two elements that belong to different components.
	Manipulation of a entire component and an element of another component. For instructions on how to model access to a struct component, refer to <a href="#">Reducing Write Accesses to NVRAM when Accessing Components of a Struct</a> on page 197.

## Related topics

### Basics

Basics on NvData Communication.....	179
Basics on Reducing Rewrites to NVRAM.....	190
Reducing Write Accesses to NVRAM by Modeling With Data Store Blocks.....	203
Reducing Write Accesses to NVRAM when Accessing Components of a Struct.....	197
Reducing Write Accesses to NVRAM when Accessing Elements of Vector or Matrix Variables.....	194

### HowTos

How to Reduce Write Accesses to the NVRAM.....	192
--	-----

### References

Data Store Memory Block (  TargetLink Model Element Reference)
---

## Reducing Write Accesses to NVRAM by Modeling With Data Store Blocks

### Reducing rewrites to NVRAM

If you set a DD NvDataElement object's ReduceWriteOperations property to **on**, you instruct TargetLink to generate code that reduces write operations to the NVRAM.

### Restriction

When you use this modeling style, you cannot use variable indices, because this is not supported by Simulink's data store blocks.

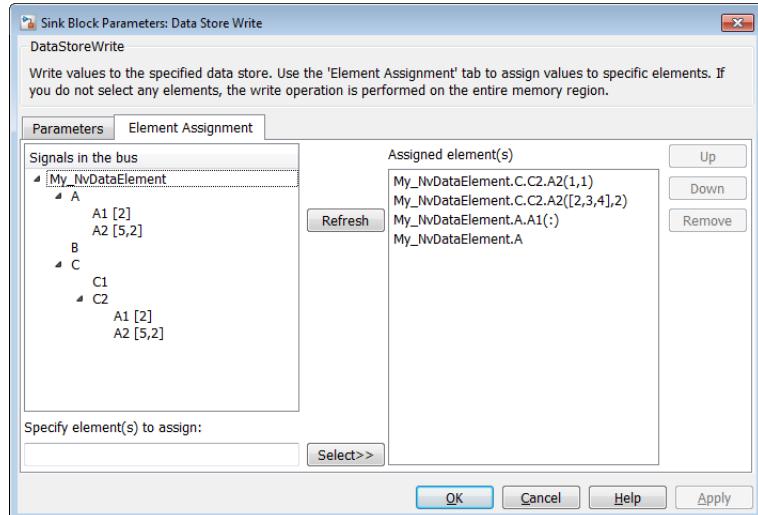
#### Tip

If you have to use variable indices, use one of the following modeling styles:

- [Reducing Write Accesses to NVRAM when Accessing Elements of Vector or Matrix Variables](#) on page 194
- [Reducing Write Accesses to NVRAM when Accessing Elements of Non-Scalar Struct Components \(Vector or Matrix Variables\)](#) on page 199

### Preconditions

You specified at least one element to assign in the block parameter dialog of the Data Store Write block, as shown in the following screenshot:

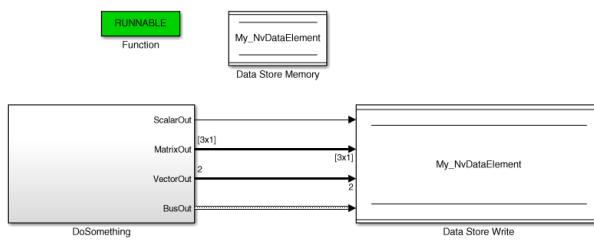


#### Note

When writing to components of structured NvDataElements, a Simulink.Bus object is required. You can create a Simulink.Bus object by using the **t1SimulinkBusObject** API function.

For bus signals, you must not assign the root bus signal.

## Modeling Style



## Related topics

### Basics

Basics on Modeling Buses via Data Store Blocks ( TargetLink Preparation and Simulation Guide)	
Basics on NvData Communication.....	179
Basics on Reducing Rewrites to NVRAM.....	190
Reducing Write Accesses to NVRAM when Accessing Components of a Struct.....	197
Reducing Write Accesses to NVRAM when Accessing Elements of Non-Scalar Struct Components (Vector or Matrix Variables).....	199
Reducing Write Accesses to NVRAM when Accessing Elements of Vector or Matrix Variables.....	194

### HowTos

How to Access the Simulink Block Properties ( TargetLink Preparation and Simulation Guide)	
How to Reduce Write Accesses to the NVRAM.....	192

### References

tISimulinkBusObject ( TargetLink API Reference)	
---	--

# Modeling Transformer Error Logic

## Where to go from here

## Information in this section

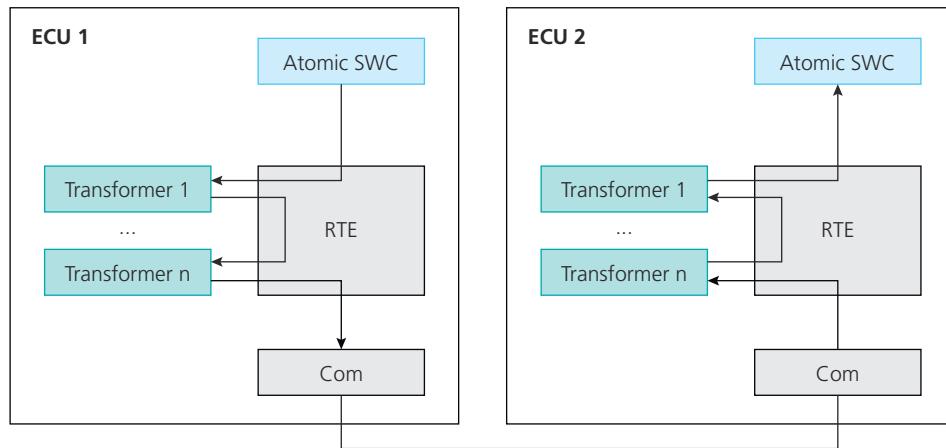
Basics on Data Transformation.....	205
Details on Data Transformation for Sender-Receiver Communication.....	208
Details on Data Transformation for Client-Server Communication.....	209

## Basics on Data Transformation

### Data transformation

 Classic AUTOSAR allows for the transformation of data in inter-ECU communication, such as end-to-end protection or serialization.

### Data transformation according to Classic AUTOSAR



In data transformation, the data of inter-ECU communication is transformed by one or more transformers. These transformers are managed by the  RTE and are not visible to the sending or receiving  SWCs.

**Transformer error** The RTE passes `Std_TransformerError` as a parameter to both SWCs if  `transformer_error` handling is enabled.

`Std_TransformerError` is a struct with the following components:

- The transformer class to which the error belongs
- The error code

**Note**

With AUTOSAR Classic R19-11 `Rte_TransformerError` is changed to `Std_TransformerError`. TargetLink lets you select the used AUTOSAR version. AUTOSAR versions prior R19-11 use `Rte_TransformerError`. In the production code depending on the AUTOSAR version `Std_TransformerError` or `Rte_TransformerError` is generated. The RTE always uses `Std_TransformerError` and maps `Rte_TransformerError` on `Std_TransformerError` to be compatible with both.

**Data transformation in TargetLink**

Because data transformation is managed by the RTE, it is out of the TargetLink scope. However, you can model transformer error logic and generate the production code with TargetLink.

**Controlling transformer error handling**

TargetLink provides the `ErrorHandling` property at DD `<Port>` objects. The property lets you control whether the generated code contains transformer errors.

**Transformer errors in the DD**

The DD template `dsdd_master_autosar4.dd` [System] provides several predefined DD objects for modeling transformer error logic:

DD Object	Description
<code>/Pool/Variables/AUTOSAR/TransformerClass</code>	<p>The predefined DD Variable objects in this subtree represent the transformer classes as described by Classic AUTOSAR:<sup>1)</sup></p> <ul style="list-style-type: none"> <li>▪ <code>TRANSFORMER_UNSPECIFIED</code> - Used for transformers of an unspecified class<sup>2)</sup></li> <li>▪ <code>TRANSFORMER_SERIALIZER</code> - Used for transformers of a serializer class</li> <li>▪ <code>TRANSFORMER_SAFETY</code> - Used for transformers of a safety class</li> <li>▪ <code>TRANSFORMER_SECURITY</code> - Used for transformers of a security class</li> <li>▪ <code>TRANSFORMER_CUSTOM</code> - Used for transformers of a custom class not described by Classic AUTOSAR</li> </ul> <p>The NameTemplate is <code>\$(TransformerErrorPrefix)_\${D}</code>. <code>\$(TransformerErrorPrefix)</code> is replaced by <code>Std</code> or <code>Rte</code> depending on the selected AUTOSAR version.</p>

DD Object	Description
/Pool/Variables/AUTOSAR/TransformerError	<p>The predefined DD Variable objects in this subtree are grouped by the Classic AUTOSAR transformer class to which they belong.<sup>1)</sup> Each transformer class is represented by a DD VariableGroup object that groups the error codes of the transformer class:</p> <ul style="list-style-type: none"> <li>▪ Serializer - Groups the error codes of a serializer class</li> <li>▪ Safety - Groups the error codes of a safety class</li> <li>▪ Security - Groups the error codes of a security class</li> <li>▪ Custom - Groups the error codes of a custom class not described by Classic AUTOSAR</li> </ul> <p><b>Note</b></p> <p>The value of each predefined DD Variable object is set according to the latest revision of Classic AUTOSAR. When migrating Data Dictionaries from projects related to Classic AUTOSAR revisions prior to 4.2.2, compare your project file with a workspace based on the dsdd_master_autosar4.dd [System] DD template to check for differences in this variable group.</p>
/Pool/Typedefs/TransformerError/TransformerError	<p>You can reference this DD Typedef object at Bus Import blocks or Bus Outport blocks in the model once you selected the checkbox on the block dialog's Output page. This ensures that the error code and the transformer class are generated as described by Classic AUTOSAR. Depending on the AUTOSAR version <b>TransformerError</b> is generated as <b>Std_TransformerError</b> or <b>Rte_TransformerError</b> in the production code.</p>

<sup>1)</sup> You can reference these Variable objects at blocks to model the error logic in your model, e.g., at Constant blocks in combination with Relational Operator blocks.

<sup>2)</sup> According to Classic AUTOSAR, this class is to be used when no error occurred.

## Related topics

### Basics

Basics on Simulating Classic-AUTOSAR-Compliant SWCs.....	296
Details on Data Transformation for Client-Server Communication.....	209
Details on Data Transformation for Sender-Receiver Communication.....	208

### References

Bus Import Block ( TargetLink Model Element Reference)
Bus Outport Block ( TargetLink Model Element Reference)

## Details on Data Transformation for Sender-Receiver Communication

### Transformer error handling in the model

In sender-receiver communication, you can model transformer error handling by using ReceiverComSpec or SenderComSpec blocks.

#### Note

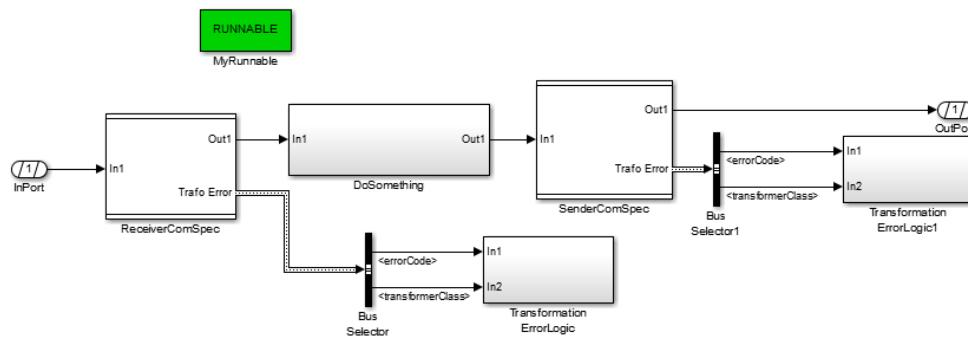
ReceiverComSpec and SenderComSpec blocks are supported only when modeling via port blocks.

You can add a port for transformation errors to these blocks by selecting its Add transformer error port checkbox.

The signal output by each ComSpec block is a bus that contains two bus elements:

- The transformer class to which the error belongs
- The error code

On the model side, this bus signal represents the `Std_TransformerError` struct in production code:



### Transformer errors in production code

The production code generated for sender-receiver communication with the `ErrorHandling` property set to `TransformerErrorHandlering` looks like this:

#### Explicit Communication

```

sint16 MyDataElement;
...
Std_TransformerError transformerError;
...
Rte_Read_MyReceiverPort_MyDataElement(&MyDataElement, &transformerError);
  
```

#### Implicit Communication

```

sint16 MyDataElement;
...
Std_TransformerError transformerError;
...
MyDataElement = Rte_IRead_MyRunnable_MyReceiverPort_MyDataElement();
Rte_IStatus_MyRunnable_MyReceiverPort_MyDataElement(&transformerError);
  
```

**Simulating transformer error logic**

To simulate transformer error logic in sender-receiver communication, you have to prepare your model.

TargetLink provides several API functions that assist you in preparing your model for simulation.

Refer to [How to Model and Simulate Transformer Error Logic in Sender-Receiver Communication](#) on page 305.

**Related topics****Basics**

Basics on Data Transformation.....	205
Modeling Sender-Receiver Communication.....	108

**HowTos**

How to Model and Simulate Transformer Error Logic in Sender-Receiver Communication.....	305
---	-----

**References**

ReceiverComSpec Block ( <a href="#">TargetLink Model Element Reference</a> )
SenderComSpec Block ( <a href="#">TargetLink Model Element Reference</a> )
<code>tlTransformerError</code> ( <a href="#">TargetLink API Reference</a> )
<code>tlTransformerError('CreateSimulinkBusObject', propertyName, PropertyValue, ...)</code> ( <a href="#">TargetLink API Reference</a> )
<code>tlTransformerError('CreateSimulinkSignalObjects', propertyName, PropertyValue, ...)</code> ( <a href="#">TargetLink API Reference</a> )
<code>tlTransformerError('CreateStimulusBlocks', propertyName, PropertyValue, ...)</code> ( <a href="#">TargetLink API Reference</a> )
<code>tlTransformerError('PrepareModelForSimulation', propertyName, PropertyValue, ...)</code> ( <a href="#">TargetLink API Reference</a> )

## Details on Data Transformation for Client-Server Communication

**Transformer error handling in the model**

In client-server communication, you can model transformer error handling by using Bus Import blocks for runnables or Bus Outport blocks for [operation subsystems](#).

You configure the block to route the transformer error by making the following settings on the block's AUTOSAR page:

Property	Value
AUTOSAR mode	Classic
Kind	Transformer Error

TargetLink automatically creates a Simulink.Bus object called `t1TransformerBus`. It contains two elements:

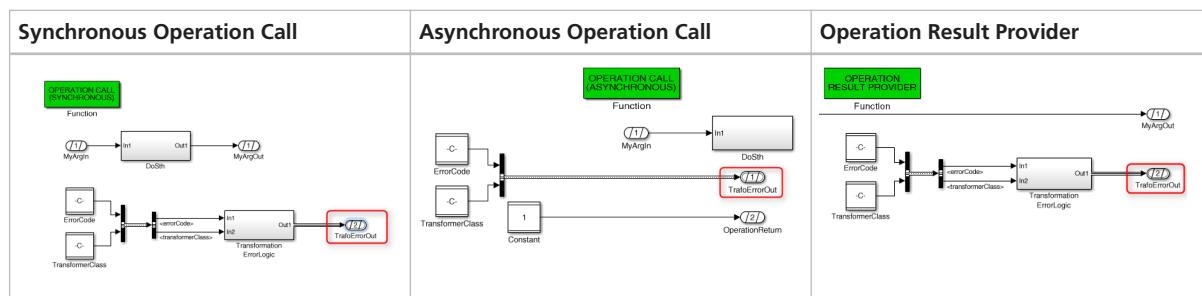
- The transformer class to which the error belongs
- The error code

### Modeling for operation subsystems

You model the transformer error for operation subsystems via a Bus Outport block that you add to the model and specify accordingly.

TargetLink uses this block to generate the call of the `Rte_Call` and `Rte_Result` RTE API functions that contain the transformer error as an `out` parameter.

**Examples** The following screenshots show some modeling examples for operation subsystems:

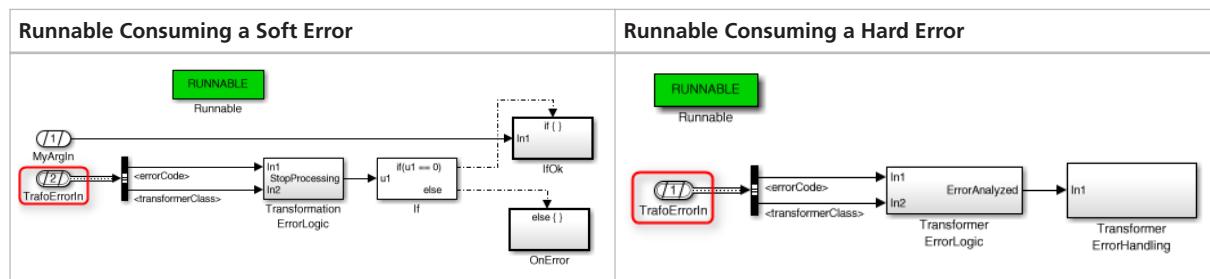


### Modeling for runnables

You model the transformer error for runnables via a Bus Import block that you add to the model and specify accordingly.

TargetLink uses this block to generate the code for the runnable function, whose interface contains the transformer error as an `in` parameter.

**Examples** The following screenshots show some modeling examples for runnables:



**Transformer errors in production code**

The production code generated for client-server communication with the DD ErrorHandling property set to **TransformerErrorHandler** looks like this:

**Operation subsystems**

```
...
Rte_Call_MyClientPort_MyOperation(MyInArg, &MyOutArg, &transformerError);
...
Rte_Result_MyClientPort_MyOtherOperation(&MyOtherOutArg, &transformerError);
```

**Runnables**

```
void MyRunnable(const Std_TransformerError * transformerError)
{
    ...
}
```

**Simulating transformer error logic**

You can simulate transformer error logic in client-server communication, because the signal representing the error is routed via port blocks in the model. When modeling transformer error logic in asynchronous communication, you can perform additional steps. Refer to [How to Simulate Operation Calls in Asynchronous Client-Server Communication](#) on page 309.

**Limitation**

You cannot model transformer error logic in [operation call subsystems](#) whose Role is set to **Operation call with Runnable implementation**.

**Related topics****Basics**

Basics on Data Transformation.....	205
Basics on Modeling the Server Side of Client-Server Communication.....	152
Modeling Client-Server Communication.....	133

**References**

- [Bus Import Block \(TargetLink Model Element Reference\)](#)
- [Bus Outport Block \(TargetLink Model Element Reference\)](#)



# Modeling Per Instance Memories

## Where to go from here

## Information in this section

Basics on Per Instance Memories.....	213
How to Model Per Instance Memories.....	214

## Basics on Per Instance Memories

### Per instance memories

A per instance memory is the definition of a data type that is instantiated for each instance of an atomic software component by the RTE.

### Per instance memories according to Classic AUTOSAR

**Variables for per instance memories** According to [Classic AUTOSAR](#), variables that can be accessed via the Rte\_Pim RTE API function, i.e., per instance memories, are generated during RTE generation, while read/write access to per instance memories is part of the SWC.

### Per instance memories in TargetLink

**Creating variables for per instance memories** You have to define DD Variable objects for the per instance memory variables to model access to per instance memories in TargetLink. You have to set the Class and the NameTemplate properties of the DD Variable objects that represent the per instance memory accordingly. For instructions, refer to [How to Model Per Instance Memories](#) on page 214.

**Using Data Store blocks for communicating via per instance memories** You can use per instance memories for communicating between runnables in a similar way to interruptable variables. However, communication via per instance memories is not covered by [Classic AUTOSAR](#).

For communication via per instance memories, you can reference variables for a per instance memory on the **Output** page of data store blocks.

**Related topics**

**Basics**

Preparing SWCs for Multiple Instantiation..... 227

## How to Model Per Instance Memories

**Objective**

To model per instance memories that you can access in an atomic software component.

**Preconditions**

The following preconditions must be fulfilled:

- You specified one or more types. Refer to:
  - [Creating Implementation Data Types from Scratch \(Classic AUTOSAR\)](#) on page 41
  - [Creating Application Data Types from Scratch \(Classic AUTOSAR\)](#) on page 49
  - [Using TargetLink's Wizards to Create Application Data Types or Implementation Data Types \(Classic AUTOSAR\)](#) on page 57
- You specified scalings and constrained range limits as required. Refer to [Defining Types, Scalings, and Constrained Range Limits \(Classic AUTOSAR\)](#) on page 31.

**Method**

**To model per instance memories**

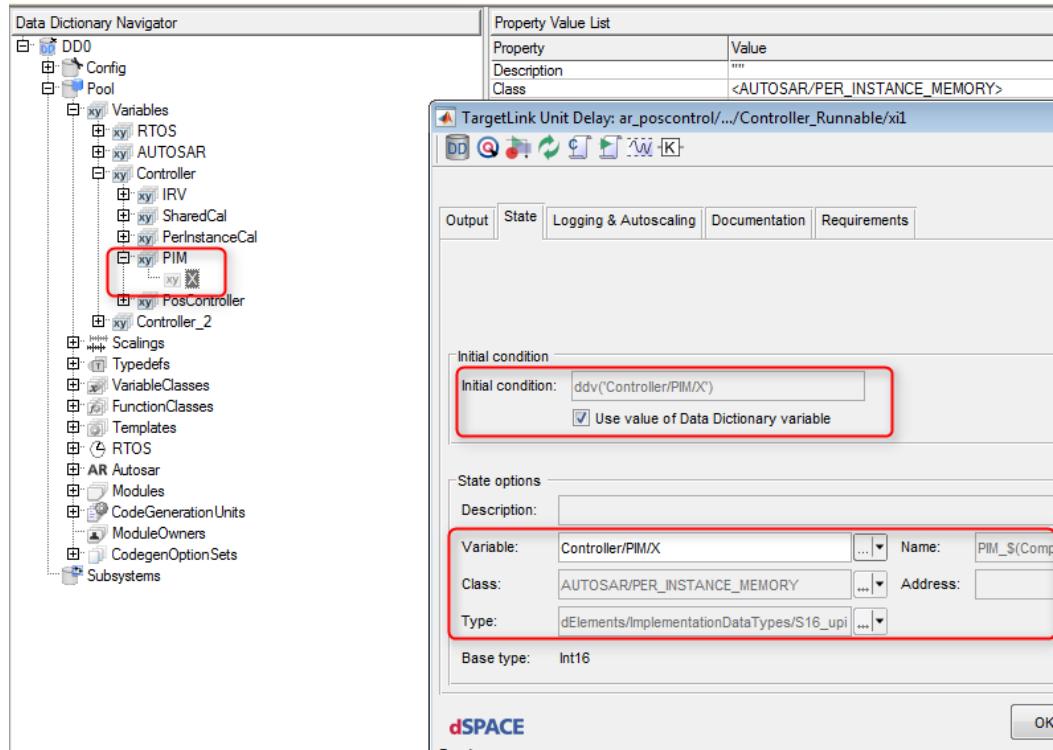
- 1 In the Data Dictionary Navigator, right-click the `/Pool/Variables` subtree or a `VariableGroup` object.
- 2 Create one variable group for the software component and name it `<SWC>`. Add another variable group below for the per instance memories (`<SWC>/PIM`).
- 3 In the Data Dictionary Navigator, select the `RelatedVariables` object of the software component you want to create a per instance memory for.
- 4 In the Property Value List, click the `PerInstanceMemoriesRef` Browse button and select the variable group.
- 5 In the Data Dictionary Navigator, right-click the `/Pool/Variables/<SWC>/PIM` subtree and choose **AUTOSAR - Create PIMVariable** from the context menu.

**6** In the Property Value List, specify the following settings:

Property	Value
Type	Select a user-defined data type. <sup>1)</sup>

<sup>1)</sup> If you select a struct user type, you have to create components of the communication subject that are consistent with the selected struct user type. You can use the Adjust to Typedef context menu command to synchronize this communication subject with the referenced DD Typedef object.

**7** In the model, reference the variable for the per instance memory at a block.



## Result

You have modeled a per instance memory.

TargetLink lets you generate code with the `Rte_Pim` RTE API function such as the following:

```

1  /* SLLocal: Default storage class for local variables | Width: 16 */
2  my_sint16 local_Variable;
3  ...
4  local_Variable = Rte_Pim_my_PIM();
5  ...

```

## Related topics

### References

[Adjust to Typedef](#) ( [TargetLink Data Dictionary Manager Reference](#))



# Modeling Modes

## Introduction

Modes are used to model software components that depend on ECU states. Runnables of an SWC can be triggered according to a mode, and runnable triggering can be prohibited in a mode.

## Where to go from here

### Information in this section

Basics on Modes.....	217
How to Create Mode Declarations.....	219
How to Create Interfaces and Ports for Mode Communication.....	220
How to Model Mode Communication.....	222
How to Model Mode Disabling Dependencies.....	224

## Basics on Modes

### Modes

Modes are operating states of an ECU, a single functional unit, etc.

### Modes according to Classic AUTOSAR

**Modes** An ECU can be in different modes, which is a synonym for states. Initialization and finalization are examples of ECU modes. Modes are changed by mode switches that are initiated by the mode managing part of the ECU software.

**Declaring modes** You can define states of the ECU and its functional units in mode declarations that you can collect in a mode declaration group. A mode declaration group provides a predefined set of modes that can be reused.

**Initial mode** ECUs are started in an initial mode. Therefore at least one mode has to be defined for an ECU. The initial mode can be used for SWC initialization.

**Mode user** A mode user is an SWC with runnables whose execution depends on modes.

- A runnable can be triggered by a *mode switch event* that triggers the runnable as a result of a mode switch. The runnable can be triggered on entry to or exit from the mode. For basic information on RTE events, refer to [Basics on RTE Events](#) on page 74.
- The RTE events of a runnable can have *mode disabling dependencies* that prohibit the triggering of the runnable in specified modes.

**Mode manager** Modes are managed by one of the following parts of the ECU software:

- An ECU state manager which is a part of the basic software.
- A mode manager which can be implemented as an SWC of the application layer.

For basic information on ECU software according to [Classic AUTOSAR](#), refer to [ECU Software Architecture \(Classic AUTOSAR\)](#) on page 16.

---

## Modes in TargetLink

**Creating mode declarations and mode declaration groups** You can define mode declarations that are part of mode declaration groups in the `/Pool/Autosar/ModeDeclarationGroups/<ModeDeclarationGroup>/Modes/` subtree. You can specify the initial mode of each mode declaration group. For instructions, refer to [How to Create Mode Declarations](#) on page 219.

### Specifying SWCs with runnables whose execution depends on modes

- TargetLink lets you specify runnables of an SWC that are triggered by RTE events of the `MODE_SWITCH_EVENT` type. For instructions on specifying RTE events of runnables, refer to [How to Specify an Event for a Runnable](#) on page 76.
- TargetLink lets you specify mode disabling dependencies for each RTE event. For instructions, refer to [How to Model Mode Disabling Dependencies](#) on page 224.

**Modeling mode communication** TargetLink lets you generate code to request a mode switch (`Rte_Switch`) and to get the currently active mode (`Rte_Mode`). You can also generate variables that represent modes in the TargetLink model and allow you to connect modes in the model. For instructions on generating code and an example for connecting modes in the model, refer to [How to Model Mode Communication](#) on page 222.

# How to Create Mode Declarations

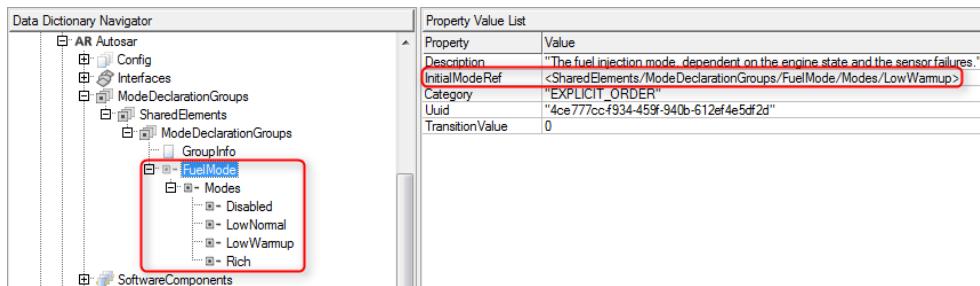
## Objective

To create mode declarations that you can use to define mode communication and mode disabling dependencies.

## Method

### To create mode declarations

- 1 In the Data Dictionary Navigator, right-click the /Pool/Autosar/ModeDeclarationGroups/ subtree.
- 2 From the context menu, choose Create ModeDeclarationGroupGroup.
- 3 Enter a name for the mode declaration group group, for example **Shared**.
- 4 Right-click the mode declaration group group and choose Create GroupInfo.
- 5 In the Property Value List, enter a package name, for example, **Shared**. This lets you export mode declaration groups in the group to the specified package. Mode declaration groups in AUTOSAR files located in the package can be imported to that location.
- 6 Right-click the mode declaration group group and choose Create ModeDeclarationGroup.
- 7 Enter a name for the mode declaration group, for example **my\_ModeDeclarationGroup**.
- 8 Right-click the **Modes** object of the mode declaration group and choose Create Mode.
- 9 Enter a name for the mode, for example **my\_InitialMode**.
- 10 In the Data Dictionary Navigator, left-click the mode declaration group.
- 11 In the Property Value List, specify an initial mode via the InitialModeRef Browse button.



## Result

You have modeled a mode declaration group with an initial mode.

## Next steps

- You can create and specify mode switch events for the software component's runnables. For instructions on specifying RTE events for runnables, refer to [How to Specify an Event for a Runnable](#) on page 76.

- You can create mode switch interfaces and mode receiver/sender ports for communicating modes. For instructions, refer to [How to Create Interfaces and Ports for Mode Communication](#) on page 220.
- You can specify mode disabling dependencies for runnables. For instructions, refer to [How to Model Mode Disabling Dependencies](#) on page 224.

<b>Related topics</b>	Basics
	<a href="#">Basics on Modes.....</a> ..... 217
	HowTos
	<a href="#">How to Specify an Event for a Runnable.....</a> ..... 76

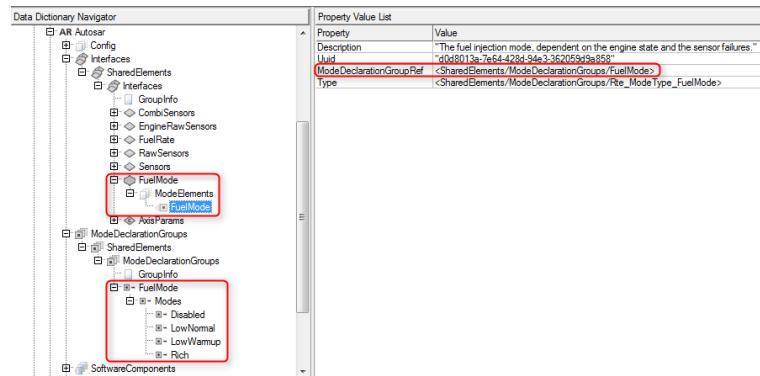
## How to Create Interfaces and Ports for Mode Communication

<b>Objective</b>	To create a mode switch interface for mode communication or specifying mode disabling dependencies.
<b>Preconditions</b>	You have created a mode declaration group with mode declarations. For instructions, refer to <a href="#">How to Create Mode Declarations</a> on page 219.
<b>Methods</b>	<ul style="list-style-type: none"> <li>▪ You can create a mode switch interface and create variables for the referenced mode declaration group. For instructions, refer to Method 1.</li> <li>▪ You can create a mode receiver/sender port and reference the mode switch interface. For instructions, refer to Method 2.</li> </ul> <p>Modeling a mode receiver is taken as an example. However, you can model a mode sender in a similar way.</p>

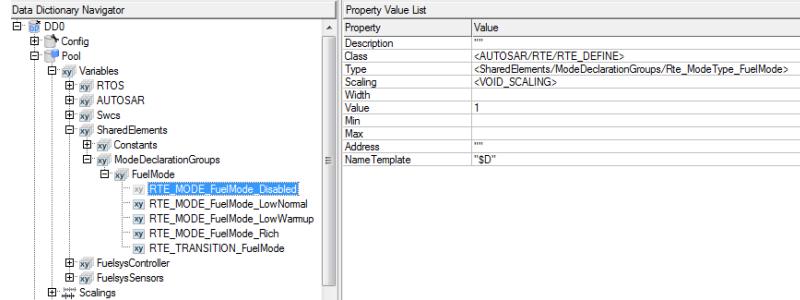
<b>Method 1</b>	<b>To create interfaces for mode communication</b>
	<ol style="list-style-type: none"> <li>1 In the Data Dictionary Navigator, right-click the /Pool1/Autosar/Interfaces/ subtree.</li> <li>2 Choose Create InterfaceGroup from the context menu.</li> <li>3 Enter a name for the interface group, for example, <b>Shared</b>.</li> <li>4 Right-click the interface group and choose Create GroupInfo from the context menu.</li> <li>5 In the Property Value List, enter a package name, for example, <b>Shared</b>.</li> </ol>

This lets you export interfaces in the group to the specified package. Interfaces in AUTOSAR files located in the package can be imported to the group.

- 6 Right-click the interface group and choose **Create ModeSwitchInterface** from the context menu.
- 7 Enter a name for the interface, for example, **my\_ModeSwitchInterface**.
- 8 Right-click the interface's **ModeElements** object and choose **Create ModeElement**.
- 9 Enter a name for the mode element, for example, **my\_ModeElement**.
- 10 In the **Property Value List**, select a mode declaration group via the **ModeDeclarationGroupRef** property.



- 11 From the context menu of the **ModeDeclarationGroup** object, select the **Synchronize Mode Settings** command. For each mode declaration in the selected mode declaration group, TargetLink creates a variable that represents the mode value.



## Interim result

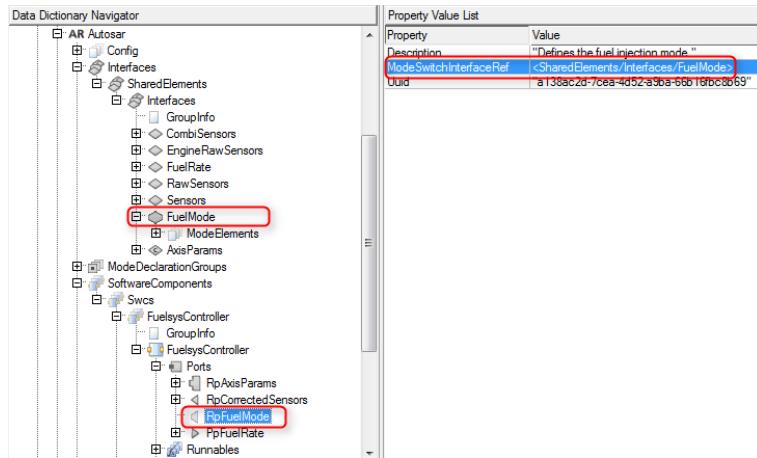
You have created a mode switch interface.

## Method 2

### To create ports for mode communication

- 1 Right-click the following object:  
`/Pool/Autosar/SoftwareComponents/<SoftwareComponent>/Ports/`
- 2 Choose **Create ModeReceiverPort** from the context menu.

- 3 Enter a name for the new port, for example, `my_ModeReceiverPort`.
- 4 In the Property Value List, select the mode switch interface via the ModeSwitchInterfaceRef Browse button.



## Result

You have created an interface and a port for mode communication.

## Next steps

- You can model communication to get the current mode or request a mode switch. For instructions, refer to [How to Model Mode Communication](#) on page 222.
- You can model mode disabling dependencies for an RTE event. For instructions, refer to [How to Model Mode Disabling Dependencies](#) on page 224.

## Related topics

### Basics

Basics on Modes.....	217
----------------------	-----

### HowTos

How to Create Mode Declarations.....	219
How to Model Mode Communication.....	222
How to Model Mode Disabling Dependencies.....	224

## How to Model Mode Communication

### Objective

To model the request for a mode switch or to get the currently active mode.

Modeling the request of a mode switch is taken as an example. However, you can model to get the currently active mode in a similar way.

---

<b>Preconditions</b>	You have created a mode switch interface and a mode receiver/sender port. For instructions, refer to <a href="#">How to Create Interfaces and Ports for Mode Communication</a> on page 220.
----------------------	---

---

<b>Method</b>	<b>To model mode communication</b>
	<ol style="list-style-type: none"> <li>1 Copy an OutPort block from the dSPACE TargetLink Classic AUTOSAR blockset to your model and double-click the block to open the AUTOSAR page. Set AUTOSAR mode to <b>Classic</b>.</li> <li>2 In the block dialog, select <b>Mode-Switch</b> from the Kind list.</li> <li>3 Select a port defined in the Data Dictionary via the Port Browse button. TargetLink lets you choose only from the mode sender ports of the software component where the OutPort block resides.</li> <li>4 Select a mode element via the Mode element Browse button.</li> </ol>

**Tip**

Mode element properties are displayed in the Signal data group box of the block dialog.

- 5 Connect the OutPort block in the model with a matching signal.

---

<b>Result</b>	You have modeled the request for a mode switch.  TargetLink lets you generate code for mode communication to the specified module, for example:
---------------	---

```
/* # combined # TargetLink outport: TL_FuelsysSensors/Run_DetectSensorFailures/FuelMode. */
Rte_Switch_PpFuelMode_FuelMode(fuel_mode);
```

---

<b>Example of connecting modes in a model</b>	To model mode communication, you have to connect a matching signal to an enhanced InPort/OutPort block that is configured for <a href="#">Classic AUTOSAR</a> mode communication. You need properly defined variables for this purpose.  You can use the Synchronize Mode Settings command from the context menu of DD ModeDeclarationGroup objects to generate the DD Variable objects that represent modes in the model. Reference a generated DD Variable object at a TargetLink block, such as a Constant block, to connect a matching signal.
---	--

**Related topics****Basics**

[Basics on Modes.....](#) 217

**HowTos**

[How to Create Interfaces and Ports for Mode Communication.....](#) 220

## How to Model Mode Disabling Dependencies

**Objective**

To model the mode-dependent execution of runnables.

**Preconditions**

The following preconditions must be fulfilled:

- You have created an RTE event for triggering a runnable. For instructions, refer to [How to Specify an Event for a Runnable](#) on page 76.
- You have created a mode switch interface and a mode receiver port. For instructions, refer to [How to Create Interfaces and Ports for Mode Communication](#) on page 220.

**Method****To model mode disabling dependencies**

- 1 In the Data Dictionary Navigator, right-click an RTE event.
- 2 From the context menu, choose **Create ModeDisablingDependencies**.
- 3 Right-click the **<RTE event>/ModeDisablingDependencies** subtree.
- 4 From the context menu, choose **Create ModeDisablingDependency**.
- 5 In the Property Value List, specify the following settings:

Property	Value
ModeDeclarationRef	Select a mode declaration. The triggered runnable will not be executed in the selected mode.
ModeElementRef	Select a mode element of a mode switch interface. The mode element must reference the mode declaration group where the selected mode declaration resides.
ModeReceiverPortRef	Select a mode receiver port.

- 6** Repeat the above steps to create more mode disabling dependencies as required.

Property	Value
ModeElementRef	<SharedElements/Interfaces/FuelMode/ModeElements/FuelMode>
ModeReceiverPortRef	<RpFuelMode>
ModeDeclarationRef	<SharedElements/ModeDeclarationGroups/FuelMode/Modes/LowWarmup>

**Result**

You have specified modes where the RTE event does not trigger an associated runnable.

**Related topics****Basics**

Basics on Modes.....	217
----------------------	-----

**HowTos**

How to Create Interfaces and Ports for Mode Communication.....	220
How to Specify an Event for a Runnable.....	76



# Preparing SWCs for Multiple Instantiation

## Where to go from here

## Information in this section

Basics on Preparing SWCs for Multiple Instantiation.....	227
How to Prepare SWCs for Multiple Instantiation.....	231

## Basics on Preparing SWCs for Multiple Instantiation

### Multiple instantiation

Multiple instantiation of an `ApplicationSwComponentType` on one ECU is beneficial for reducing [code size](#).

### Multiple instantiation according to Classic AUTOSAR

[Classic AUTOSAR](#) supports multiple instantiation by sharing code. This lets you reduce memory overhead and testing efforts. Take the following pseudo-code as example:

```
TASK(Task1){
    ...
    Runnable1(instance1);
    Runnable1(instance2);
    ...
}
```

The code of `Runnable1` is shared by two different instances. To distinguish which instance the code belongs to, an instance handle (`instance<n>`) is always passed as the first parameter of all the multiple instantiated entities.

According to [Classic AUTOSAR](#), it is best to use per instance memories (PIMs) to collect each instances' states in separate memory areas.

**Multiple instantiation in TargetLink**

TargetLink helps you prepare an `ApplicationSwComponentType` for multiple instantiation.

To instruct TargetLink to prepare an `ApplicationSwComponentType` for multiple instantiation, set the DD `SoftwareComponent` object's `SupportsMultipleInstantiation` property to `on`. During code generation, TargetLink performs all the further required steps.

**Instance-specific calibration parameters** TargetLink can create instance-specific calibration parameters during code generation. The import and export of instance-specific calibration parameters from/to AUTOSAR files is also supported.

You have to explicitly specify how to implement calibration parameters via the `Class` property of the DD `Variable` objects used to specify the calibration parameters:

Parameter Type	Variable Class
Shared	AUTOSAR/SHARED_CALPRM
Instance specific	AUTOSAR/PER_INSTANCE_CALPRM

Reference the DD `VariableGroup` object containing your parameters at the `CalibratablesRef/PerInstanceCalibratablesRef` property of the `RelatedVariables` object belonging to the DD `SoftwareComponent` object.

**Tip**

Preparing `ApplicationSwComponentType` for multiple instantiation is similar to function reuse in non-AUTOSAR contexts.

For details on function reuse, refer to [Reusing C Functions and Variables for Identical Model Parts \(Function Reuse\)](#) ( [TargetLink Customization and Optimization Guide](#)).

**Storing states in per instance memories**

The internal states of an `ApplicationSwComponentType` are usually realized via variables with static duration. Because each instance needs its own set of states, TargetLink follows  [Classic AUTOSAR](#) in creating a PIM for each `ApplicationSwComponentType` prepared for multiple instantiation.

Predicting the data type of a PIM collecting all the states of an `ApplicationSwComponentType` is next to impossible for complex models. Therefore, during code generation, TargetLink performs the following actions:

**Gathering states** TargetLink analyzes your model and DD based specification and collects all the variables with static duration, that belong to the `ApplicationSwComponentType` prepared for multiple instantiation (states).

**Creating the PIM** TargetLink creates a PIM for each `ApplicationSwComponentType` prepared for multiple instantiation. It automatically determines the correct data type of the PIM, based on the gathered states and moves them to the auto-created PIM.

**Reporting** TargetLink reports variables that could not be implemented as PIM in its Code Generation Report.

**Instance handle**

The instance handle is a pointer to the software component's data structure that contains all the instance-specific data of the SWC instance. As such, it serves as a unique identifier to distinguish data that belongs to different instances of an **ApplicationSwComponentType**.

TargetLink generates code such that all calls to instance specific entities carry the instance handle as their first parameter. In [compatibility mode](#), you cannot change the instance handle, because it is standardized.

**Propagation** TargetLink ensures that the instance handle is propagated to subfunctions accordingly. By design, TargetLink does not propagate the instance handle to externally defined functions.

When modeling calls to externally defined functions whose implementations contain calls of RTE API functions, you have to select the **Add SWC instance handle as argument** checkbox on the AUTOSAR page of the Function Block.

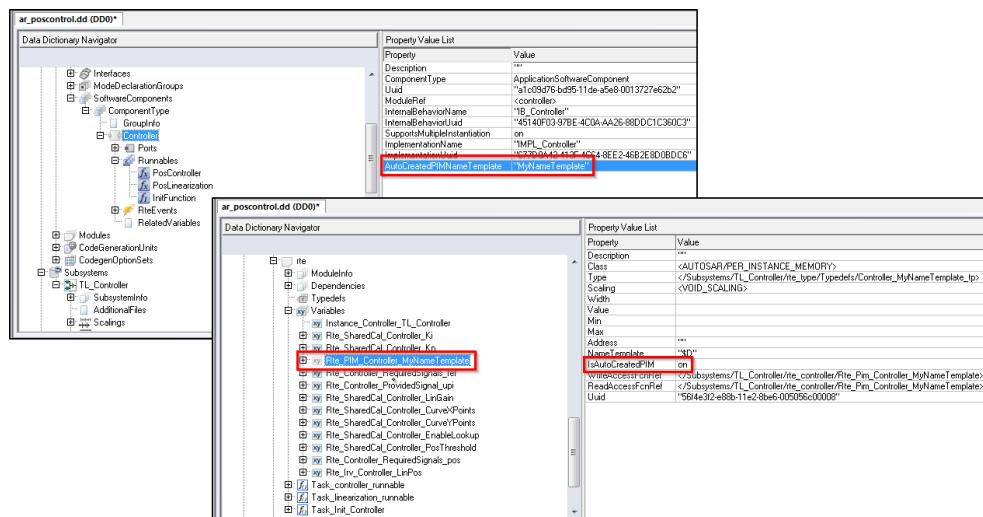
**Controlling which states to move to PIM**

It is best to let TargetLink determine which variables to implement as PIM.

However, you can configure TargetLink to not move variables to the auto-created PIM via the **DoNotImplementAsPIM** property of DD VariableClass objects. If this option is set to **on**, TargetLink will not move the variable to the PIM during code generation and will not report this variable in the **Variables excluded from PIM** section of the Code Generation Report.

**Controlling the PIM's name**

Optionally, you can control how TargetLink names the auto-created PIM. For each DD SoftwareComponent object, you can set the DD **AutoCreatedPIMNameTemplate**, which defaults to **ACP\_\$I\$R**.



**Specifying package and AUTOSAR file**

You can specify to which AUTOSAR file and package to export the artifacts related to the auto-created PIM via a DD ExportRule object, whose ElementKind property is set to PerInstanceMemoryArtifact.

For details on packages, refer to [Basics on Packages](#) ( [TargetLink Interoperation and Exchange Guide](#)).

---

**Simulating an SWC instance**

TargetLink is used to create and test SWCs of the `ApplicationSwComponentType`, which is why SIL or PIL simulations with *multiple instances* of the same `ApplicationSwComponentType` are out of TargetLink's scope. However, you can simulate a single instance of an `ApplicationSwComponentType` in all the three simulation modes. Simulations of multiple SWC instances are possible in the context of virtual validation. For details, refer to the [Virtual Validation Overview](#) document.

---

**Decomposing models**

You can partition models that contain software components prepared for multiple instantiation. For details, refer to [Partitioning Model and Code for Classic AUTOSAR](#) on page 271.

---

**Related topics**

**Basics**

<a href="#">Basics on Packages</a> ( <a href="#">TargetLink Interoperation and Exchange Guide</a> )	235
<a href="#">Basics on the Code Generation Report</a> ( <a href="#">TargetLink Preparation and Simulation Guide</a> )	241
<a href="#">Preparing SWCs for Measurement and Calibration</a> .....	235

**HowTos**

<a href="#">How to Model Per Instance Parameters for Calibration</a> .....	239
<a href="#">How to Model Shared Parameters for Calibration</a> .....	241
<a href="#">How to Prepare SWCs for Multiple Instantiation</a> .....	231

**References**

<a href="#">Function Block</a> ( <a href="#">TargetLink Model Element Reference</a> )
---

# How to Prepare SWCs for Multiple Instantiation

## Preconditions

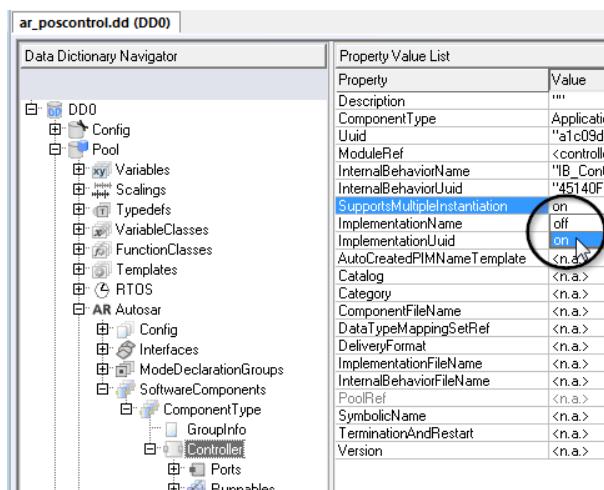
The following preconditions must be fulfilled:

- The TargetLink Data Dictionary is open.
- Your DDO workspace contains one or more SoftwareComponent object(s). For instructions on creating software components, refer to [How to Create Software Components](#) on page 66.
- The EnableReportGeneration and ReportVariablesExcludedFromPIM Code Generator options are enabled.

## Method

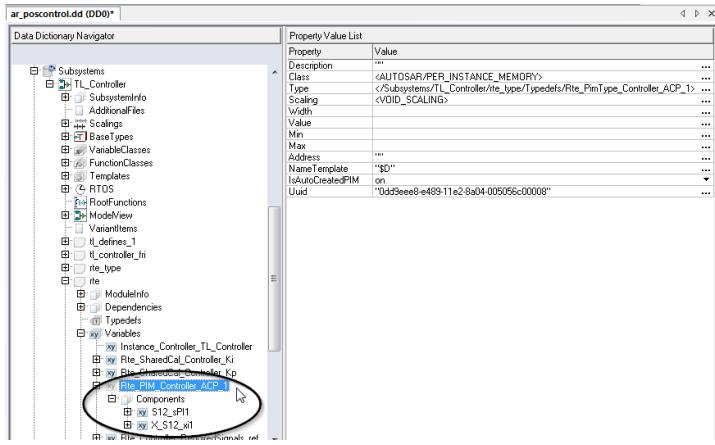
### To prepare a SWC for multiple instantiation

- 1 In the TargetLink Data Dictionary Manager locate the SoftwareComponent object you want to prepare for multiple instantiation.
- 2 In the SoftwareComponent object's Property Value List, locate the SupportsMultipleInstantiation property and set it to on.



- 3 Generate code.

TargetLink gathers the software component's relevant states in an auto-created PIM called ACP\_<CodeGenerationUnitID>. In the DD Subsystems subtree, it is represented by the Rte\_PIM\_\$(SoftwareComponent)\_ACP\_<SubSpec> variable, created by the Code Generator as an access function.



- 4 From the MATLAB Command Window, open the HTML Code Generation Report to check for variables that could not be implemented within the auto-created PIM:

```
Note    INVOKING the post code generation hooks: 'tl_post_codegen_hooks'
Note    FINISHED all code generation steps for system TL_Controller
Note    Open HTML Code Generation Report
Note    Show list of the generated files

Invoking the post code generation finished hooks: 'tl_post_codegen_finished'

PRODUCTION CODE GENERATION SUCCEEDED
>>
```

The HTML Code Generation Report is displayed in MATLAB's Web Browser.

- 5 Check each relevant entry and either solve the problem or set the associated variable class' DoNotImplementAsPIM property to on.

#### Note

Do not change the DoNotImplementAsPIM property of TargetLink's predefined variable classes.

Instead, derive a new DD VariableClass object from the predefined variable class, set its DoNotImplementAsPIM to on, and reference it at the corresponding DD Variable object(s)

## Result

You prepared a SWC for multiple instantiation and generated code such as the following:

```
FUNC(void, RTE_APPL_CODE) controllerRunnable(Rte_Instance instance)
{
    /* SLLocal: Default storage class for local variables | Width: 16 */
    sint16 S12_e1 /* LSB: 2^-9 OFF: 0 MIN/MAX: -64 .. 63.998046875 */;
    sint16 Aux_a /* LSB: 2^-9 OFF: 0 MIN/MAX: -64 .. 63.998046875 */;
    /* Sum: TL_Controller/Controller_Runnable/e1
       # combined # TargetLink import: TL_Controller/Controller_Runnable/(ref)
       # combined # TargetLink import: TL_Controller/Controller_Runnable/(pos) */
    S12_e1 = (sint16) (((sint32) Rte_IRead_PosController_RequiredSignals_ref(instance)) - ((sint32)
        Rte_IrvRead_PosController_LinPos(instance))) >> 1;
    LOG_VAR(1, _S12_e1, S12_e1);
```

```

/* Sum: TL_Controller/Controller_Runnable/si1
   # combined # Unit delay: TL_Controller/Controller_Runnable/xi1 */
Rte_Pim_ACP_1(instance)->X_S12_xi1 = (sint16) (S12_e1 + Rte_Pim_ACP_1(instance)->X_S12_xi1);
/* Sum: TL_Controller/Controller_Runnable/sPI1
   # combined # Gain: TL_Controller/Controller_Runnable/Ki1
   # combined # Gain: TL_Controller/Controller_Runnable/Kp1
   # combined # Unit delay: TL_Controller/Controller_Runnable/xi1 */
Rte_Pim_ACP_1(instance)->S12_sPI1 = (sint16) ((S12_e1 * ((sint16) Rte_CData_Kp(instance))) +
((sint16) (((sint32) Rte_Pim_ACP_1(instance)->X_S12_xi1) * ((sint32) Rte_CData_Ki(instance)))
>> 9));
Aux_a = Rte_Pim_ACP_1(instance)->S12_sPI1;
/* # combined # TargetLink outport: TL_Controller/Controller_Runnable/(upi) */
Rte_IWrite_PosController_ProvidedSignal_upi(instance, Aux_a);
}

```

**Related topics****Basics**

[Basics on Preparing SWCs for Multiple Instantiation](#)..... 227

[Basics on the Code Generation Report](#) ( [TargetLink Preparation and Simulation Guide](#))

**HowTos**

[How to Create Software Components](#)..... 66

**References**

[EnableReportGeneration](#) ( [TargetLink Model Element Reference](#))

[ReportVariablesExcludedFromPIM](#) ( [TargetLink Model Element Reference](#))



# Preparing SWCs for Measurement and Calibration

## Where to go from here

## Information in this section

Basics on Preparing SWCs for Measurement and Calibration.....	235
How to Model Per Instance Parameters for Calibration.....	239
How to Model Shared Parameters for Calibration.....	241
How to Model Parameter Communication for Calibration.....	244
Basics on Static and Constant Memories.....	247
How to Model Static Memories for Measurement.....	250
How to Model Constant Memories for Calibration.....	252
Basics on Preparing Look-Up Tables for Measurement and Calibration (Classic AUTOSAR).....	254
Example of Preparing a Look-Up Table for Measurement and Calibration.....	260

## Basics on Preparing SWCs for Measurement and Calibration

### Introduction

You can prepare software components for measurement and calibration.

### Basics on measurement and calibration

**Parameter calibration** After code generation, the parameter values of the generated ECU code are often fine-tuned to optimize the performance of the control algorithm. Calibrating the ECU's parameter values – often the last ECU development step – is usually carried out on a test bench or in a test vehicle.

A measurement and calibration (MC) system, such as dSPACE ControlDesk is used for parameter calibration. To access an ECU for calibration, the MC system requires information on the parameters and measurement variables in the ECU code, such as:

- Scaling factors and conversion methods
- ECU memory addresses
- Specification of the interface between the ECU and the MC system

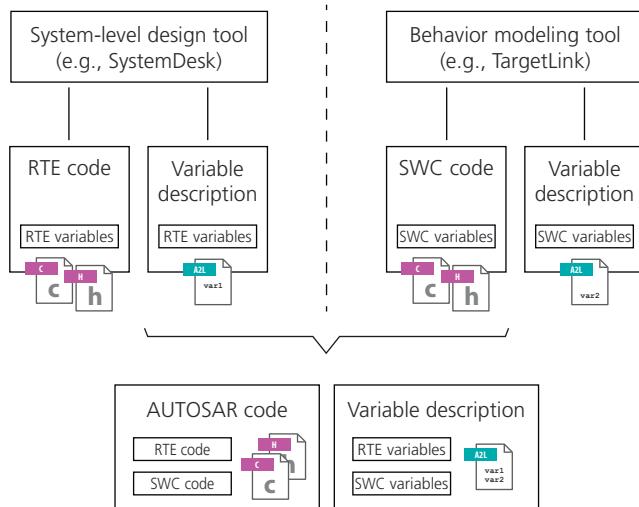
**ASAM MCD-2 MC** ASAM MCD-2 MC is a standard commonly used for describing ECU-internal data. The standard was developed by ASAM e.V. (Association for Standardization of Automation and Measuring Systems e.V.). A2L files – formerly known as ASAP2 files – contain the information required by an MC system to access an ECU for parameter calibration. A2L files are ASCII files.

You can download the specification of the ASAM MCD-2 MC standard from <http://www.asam.net>.

**Responsibility for A2L files** As a behavior modeling tool, TargetLink generates production code for software components (SWC code). Accordingly, the A2L files generated by TargetLink contain only descriptions of measurable or calibratable variables that were defined by TargetLink. By design, variables belonging to the [Classic AUTOSAR RTE](#) are *not* included, because they are defined by a system-level design tool, such as dSPACE SystemDesk.

If you need an A2L file that contains descriptions of [Classic AUTOSAR RTE](#) variables (RTE code), have it generated by a system-level design tool, such as dSPACE SystemDesk.

For more information, refer to [Basics on Tools for Developing ECU Software as Described by Classic AUTOSAR](#) on page 18, [Basics on Simulating Classic-AUTOSAR-Compliant SWCs](#) on page 296, and [Basics on Generated Code](#) on page 276.



## Measurement and calibration according to Classic AUTOSAR

[Classic AUTOSAR](#) distinguishes between measurement variables and calibration parameters. Both are described in AUTOSAR files (ARXML) and are characterized in their `<SW-DATA-DEF-PROPS>` element. This element contains the `<SW-CALIBRATION-ACCESS>` element, which can be set as shown in the following table:

Value	Description
<code>notAccessible</code>	The element is not accessible via MCD tools, i.e., it does not appear in the A2L file <sup>1)</sup> .
<code>readOnly</code>	The element only appears as read-only in an A2L file <sup>1)</sup> .
<code>readWrite</code>	The element appears in the A2L file <sup>1)</sup> with both read and write access.

<sup>1)</sup> Generated by a system-level design tool, such as SystemDesk, or an RTE generator.

### Note

For simplicity's sake, the interplay of the [Classic AUTOSAR](#) properties `swImplPolicy` and `swCalibrationAccess` is ignored. Refer to *Implementation of an interface element* in the *AUTOSAR\_SWS\_RTE* document.

**Measurement variables** The `<SW-CALIBRATION-ACCESS>` element of measurement variables is set to `readOnly`.

[Classic AUTOSAR](#) describes the following measurement variables:

- Arguments of client-server interface operations
- Elements of NvData interfaces
- Elements of sender-receiver interfaces
- Interrunnable variables
- Mode elements
- Per instance memories

**Calibration parameters** The `<SW-CALIBRATION-ACCESS>` element of calibration parameters is set to `readWrite`.

[Classic AUTOSAR](#) defines three different ways to specify calibration parameters. Which to choose depends on whether the calibration parameters are accessed by SWCs of the same `ApplicationSwComponentType`. This is shown in the following table:

Number of <code>ApplicationSwComponentTypes</code>	Possible Parameter Specification		
	<code>perInstanceParameter</code>	<code>sharedParameter</code>	<code>parameterInterface</code>
One or more instances of the same <code>ApplicationSwComponentType</code>	✓ <sup>1)</sup>	✓ <sup>2)</sup>	✓
Several SWCs of <i>different ApplicationSwComponentType</i>	–	–	✓

<sup>1)</sup> Different value for each SWC instance.

<sup>2)</sup> Same parameter value for all SWC instances.

## Measurement and calibration in TargetLink

TargetLink lets you export descriptions of measurement variables and calibration parameters to AUTOSAR files (ARXML).

**Measurement variables** The <SW-CALIBRATION-ACCESS> element of the following [Classic AUTOSAR](#) entities can be influenced during export:

AUTOSAR Element	Instructions
Arguments of client-server interface operations	Specify the DD OperationArgument object's CalibrationAccess property as required.
Elements of NvData interfaces	Specify the DD NvDataElement object's CalibrationAccess property as required.
Elements of sender-receiver interfaces	Specify the DD DataElement object's CalibrationAccess property as required.
Interrunnable variables	Specify the DD InterRunnableVariable object's CalibrationAccess property as required.
Elements of mode-switch interfaces	Specify the DD ModeElement object's CalibrationAccess property as required.
Per instance memories	Specify the DD Variable object's CalibrationAccess property as required.

**Calibration parameters** TargetLink lets you model per instance parameters, shared parameters, and parameter interfaces with parameter elements as described by [Classic AUTOSAR](#):

Access from SWCs of Different ApplicationSwComponentType	Parameter Sharing	Modeling Style	Instructions
No	Always	Shared parameter	<a href="#">How to Model Shared Parameters for Calibration</a> on page 241
	Possible <sup>1)</sup>	ParameterInterface	<a href="#">How to Model Parameter Communication for Calibration</a> on page 244
	Never	Per instance parameter	<a href="#">How to Model Per Instance Parameters for Calibration</a> on page 239
Yes	Possible <sup>1)</sup>	ParameterInterface	<a href="#">How to Model Parameter Communication for Calibration</a> on page 244

<sup>1)</sup> Depends on the connection of the parameter ports on the architecture level. Out of scope of TargetLink.

#### Specifics for look-up tables

To prepare a look-up table for measurement and calibration, its axes must be described at the level of [application data types](#). Additionally, the table input values must be specified at the access point of the runnable. Refer to the following topics:

- [Basics on Preparing Look-Up Tables for Measurement and Calibration \(Classic AUTOSAR\)](#) on page 254
- [Example of Preparing a Look-Up Table for Measurement and Calibration](#) on page 260

#### Measurement and calibration without involvement of the RTE

In [Classic AUTOSAR](#), static memories and constant memories let you perform measurement and calibration without involvement of the RTE. This can make implementations more efficient by avoiding the indirection caused by the RTE API.

Refer to:

- [Basics on Static and Constant Memories on page 247](#)
- [How to Model Static Memories for Measurement on page 250](#)
- [How to Model Constant Memories for Calibration on page 252](#)

## Related topics

### Basics

<a href="#">Basics on Preparing Look-Up Tables for Measurement and Calibration (Classic AUTOSAR)</a> .....	254
<a href="#">Basics on Static and Constant Memories</a> .....	247
<a href="#">Exchanging A2L Files (TargetLink Interoperation and Exchange Guide)</a>	

### HowTos

<a href="#">How to Model Constant Memories for Calibration</a> .....	252
<a href="#">How to Model Static Memories for Measurement</a> .....	250

## How to Model Per Instance Parameters for Calibration

### Per instance parameters

Per instance parameters are calibration parameters that are used by different instances of the same `ApplicationSwComponentType`. Each parameter has a different value for each SWC instance. Per instance parameters are accessed via the `Rte_CData` API function.

### Restriction

#### Note

According to [Classic AUTOSAR](#), measurement variables and calibration parameters are instantiated by the RTE. Accordingly, TargetLink does not generate definitions of these variables and does not generate A2L files that contain descriptions of these variables. Refer to [Basics on Preparing SWCs for Measurement and Calibration](#) on page 235.

### Preconditions

The following preconditions must be fulfilled:

- You specified one or more data types. Refer to:
  - [Creating Implementation Data Types from Scratch \(Classic AUTOSAR\)](#) on page 41
  - [Creating Application Data Types from Scratch \(Classic AUTOSAR\)](#) on page 49
  - [Using TargetLink's Wizards to Create Application Data Types or Implementation Data Types \(Classic AUTOSAR\)](#) on page 57

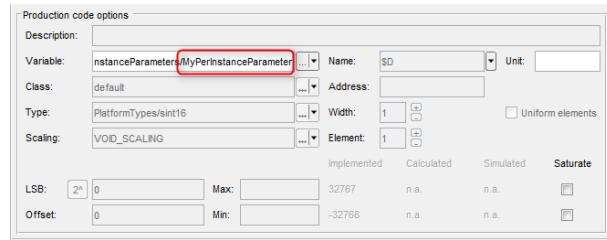
- You specified scalings and constrained range limits as required. Refer to [Defining Types, Scalings, and Constrained Range Limits \(Classic AUTOSAR\)](#) on page 31.

---

Method	To model a per instance parameter for calibration										
	<p><b>1</b> In the Data Dictionary Navigator, create a DD VariableGroup object and rename it as required, e.g., MyPerInstanceParams.</p> <p><b>2</b> On the MyPerInstanceParams object's context menu, open the AUTOSAR submenu and select Create PerInstanceCalPrm. Rename the created DD Variable object as required, e.g., MyPerInstanceParam.</p> <p><b>3</b> In the MyPerInstanceParam object's Property Value List, specify the following settings:</p> <table border="1" style="width: 100%; border-collapse: collapse;"> <thead> <tr> <th style="text-align: left; padding: 2px;">Property</th><th style="text-align: left; padding: 2px;">Value</th></tr> </thead> <tbody> <tr> <td style="padding: 2px;">Type</td><td style="padding: 2px;">Select a user-defined data type.<sup>1)</sup></td></tr> <tr> <td style="padding: 2px;">ApplicationDataTypeRef</td><td style="padding: 2px;">The DD ApplicationDataType object that is associated with the <a href="#">implementation data type (IDT)</a> that you referenced at the Type property.</td></tr> <tr> <td style="padding: 2px;">Width</td><td style="padding: 2px;">If you select an array user type, you have to specify a consistent width.</td></tr> <tr> <td style="padding: 2px;">Value</td><td style="padding: 2px;">Enter an initialization value, e.g., 0. The calibration parameter is initialized by the RTE.</td></tr> </tbody> </table> <div style="background-color: #f0f0f0; padding: 5px; margin-top: 10px;"> <p><b>Note</b></p> <p><b>Avoid ambiguous initializations of data prototypes</b></p> <p>Specify an initialization value for each <a href="#">data prototype</a> within the Data Dictionary whenever possible. This avoids ambiguities between <a href="#">Classic AUTOSAR</a> initialization values and initialization values of model elements that are required by Simulink.</p> </div>	Property	Value	Type	Select a user-defined data type. <sup>1)</sup>	ApplicationDataTypeRef	The DD ApplicationDataType object that is associated with the <a href="#">implementation data type (IDT)</a> that you referenced at the Type property.	Width	If you select an array user type, you have to specify a consistent width.	Value	Enter an initialization value, e.g., 0. The calibration parameter is initialized by the RTE.
Property	Value										
Type	Select a user-defined data type. <sup>1)</sup>										
ApplicationDataTypeRef	The DD ApplicationDataType object that is associated with the <a href="#">implementation data type (IDT)</a> that you referenced at the Type property.										
Width	If you select an array user type, you have to specify a consistent width.										
Value	Enter an initialization value, e.g., 0. The calibration parameter is initialized by the RTE.										

<sup>1)</sup> If you select a struct user type, you have to create components of the communication subject that are consistent with the selected struct user type. You can use the Adjust to Typedef context menu command to synchronize this communication subject with the referenced DD Typedef object.

- 4 Locate the DD SoftwareComponent object for which you want to use per instance parameters and select its RelatedVariables subtree.
- 5 From its context menu, select Create Reference to PerInstanceCalibratables.
- 6 In the Property Value List, set the PerInstanceCalibratablesRef property to MyPerInstanceParams.
- 7 In the model, reference the per instance parameter at a TargetLink block that is used to model the SWC's internal behavior. The following screenshot shows the reference on the Gain page of a Gain Block:



## Result

You modeled a per instance parameter and used it at a TargetLink block. During code generation, TargetLink will create a call of the `Rte_CData` API function in its production code that looks like this:

```
Rte_CData_MyPerInstanceParameter();
```

## Related topics

### Basics

Basics on Preparing SWCs for Measurement and Calibration.....	235
Creating Application Data Types from Scratch (Classic AUTOSAR).....	49
Creating Implementation Data Types from Scratch (Classic AUTOSAR).....	41
Defining Types, Scalings, and Constrained Range Limits (Classic AUTOSAR).....	31
Using TargetLink's Wizards to Create Application Data Types or Implementation Data Types (Classic AUTOSAR).....	57

### References

- [Adjust to Typedef \(TargetLink Data Dictionary Manager Reference\)](#)
- [Create Reference to <Object> \(TargetLink Data Dictionary Manager Reference\)](#)
- [Gain Block \(TargetLink Model Element Reference\)](#)

# How to Model Shared Parameters for Calibration

## Shared parameters

Shared parameters are calibration parameters that are used by different instances of the same `ApplicationSwComponentType`. Each parameter has the same value for all SWC instances. Shared parameters are accessed via the `Rte_CData` API function.

Restriction	Note
	<p>According to <a href="#">Classic AUTOSAR</a>, measurement variables and calibration parameters are instantiated by the RTE. Accordingly, TargetLink does not generate definitions of these variables and does not generate A2L files that contain descriptions of these variables. Refer to <a href="#">Basics on Preparing SWCs for Measurement and Calibration</a> on page 235.</p>

Preconditions	<p>The following preconditions must be fulfilled:</p> <ul style="list-style-type: none"> <li>▪ You specified one or more data types. Refer to:           <ul style="list-style-type: none"> <li>▪ <a href="#">Creating Implementation Data Types from Scratch (Classic AUTOSAR)</a> on page 41</li> <li>▪ <a href="#">Creating Application Data Types from Scratch (Classic AUTOSAR)</a> on page 49</li> <li>▪ <a href="#">Using TargetLink's Wizards to Create Application Data Types or Implementation Data Types (Classic AUTOSAR)</a> on page 57</li> </ul> </li> <li>▪ You specified scalings and constrained range limits as required. Refer to <a href="#">Defining Types, Scalings, and Constrained Range Limits (Classic AUTOSAR)</a> on page 31.</li> </ul>
---------------	---

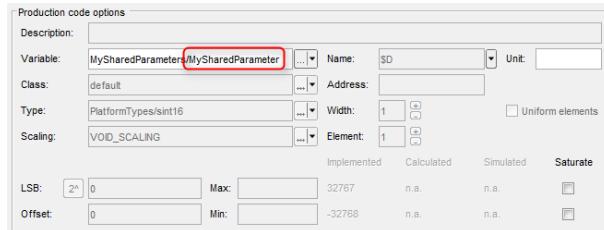
Method	<p><b>To model a shared parameter for calibration</b></p> <ol style="list-style-type: none"> <li>1 In the Data Dictionary Navigator, create a DD VariableGroup object and rename it as required, e.g., MySharedParams.</li> <li>2 On the MySharedParams object's context menu, open the AUTOSAR submenu and select Create SharedCalPrm. Rename the created DD Variable object as required, e.g., MySharedParam.</li> <li>3 In the MySharedParam object's Property Value List, specify the following settings:</li> </ol>
--------	--

<b>Property</b>	<b>Value</b>
Type	Select a user-defined data type. <sup>1)</sup>
ApplicationDataTypeRef	The DD ApplicationDataType object that is associated with the <a href="#">implementation data type (IDT)</a> that you referenced at the Type property.
Width	If you select an array user type, you have to specify a consistent width.
Value	Enter an initialization value, e.g., 0. The calibration parameter is initialized by the RTE.
<p><b>Note</b></p> <p><b>Avoid ambiguous initializations of data prototypes</b></p> <p>Specify an initialization value for each <a href="#">data prototype</a> within the Data Dictionary whenever possible. This avoids ambiguities between <a href="#">Classic AUTOSAR</a> initialization values and initialization values of model elements that are required by Simulink.</p>	

Property	Value
InitConstantName	Lets you specify a name for the initialization constant of the calibration parameter.
InitConstantPackage	Lets you specify the package for the initialization constant of the calibration parameter, e.g., <b>Constants</b> .

<sup>1)</sup> If you select a struct user type, you have to create components of the communication subject that are consistent with the selected struct user type. You can use the Adjust to Typedef context menu command to synchronize this communication subject with the referenced DD Typedef object.

- 4 Locate the DD SoftwareComponent object for which you want to use shared parameters and select its RelatedVariables subtree.
- 5 From its context menu, select Create Reference to Calibratables.
- 6 In the Property Value List, set the CalibratablesRef property to **MySharedParams**.
- 7 In the model, reference the shared parameter at a TargetLink block that is used to model the SWC's internal behavior. The following screenshot shows the reference on the Gain page of a Gain Block:



- 8 Generate code.

## Result

You modeled a shared parameter and used it at a TargetLink block. During code generation, TargetLink will create a call of the **Rte\_CData** API function in its production code that looks like this:

```
Rte_CData_MySharedParameter();
```

## Related topics

### Basics

Basics on Preparing SWCs for Measurement and Calibration.....	235
Creating Application Data Types from Scratch (Classic AUTOSAR).....	49
Creating Implementation Data Types from Scratch (Classic AUTOSAR).....	41
Defining Types, Scalings, and Constrained Range Limits (Classic AUTOSAR).....	31
Using TargetLink's Wizards to Create Application Data Types or Implementation Data Types (Classic AUTOSAR).....	57

### References

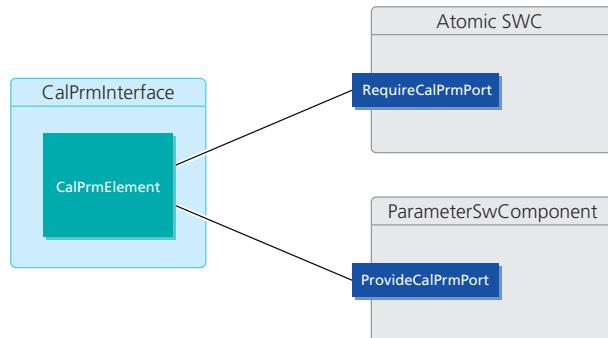
Adjust to Typedef (TargetLink Data Dictionary Manager Reference)
Create Reference to <Object> (TargetLink Data Dictionary Manager Reference)
Gain Block (TargetLink Model Element Reference)

## How to Model Parameter Communication for Calibration

### Parameter communication

In parameter communication, calibration parameters (CalPrmElements) are provided by a SWC of the `ParameterSwComponentType` via provide ports and communicated via a parameter interface. Several SWCs of different types can use these calibration parameters via require ports.

CalPrmElements are accessed via the `Rte_Prm` API function.



### Restrictions

You cannot model software components of the `ParameterSwComponentType` with provide ports for parameter communication in TargetLink, because they are outside the scope of behavior modeling tools.

#### Note

According to [Classic AUTOSAR](#), measurement variables and calibration parameters are instantiated by the RTE. Accordingly, TargetLink does not generate definitions of these variables and does not generate A2L files that contain descriptions of these variables. Refer to [Basics on Preparing SWCs for Measurement and Calibration](#) on page 235.

### Preconditions

The following preconditions must be fulfilled:

- You created an SWC, such as `MySWC`.  
Refer to [How to Create Software Components](#) on page 66.
- You created a parameter interface, such as `MyCalPrmInterface`.  
Refer to [How To Create Interfaces](#) on page 100.
- You created a require port for parameter communication, such as `MyRequireCalPrmPort`. Refer to [How to Create Ports](#) on page 101.
- You created a CalPrm element, such as `MyCalPrmElement`. Refer to [How to Create Communication Subjects for Classic AUTOSAR Communication](#) on page 103.

**Workflow**

This workflow consists of the following parts:

1. To specify an initialization constant for a CalPrmElement, refer to Part 1.
2. To model parameter communication for calibration, refer to Part 2.

**Part 1****To specify an initialization constant for a CalPrmElement**

- 1 In the Data Dictionary Navigator, create a variable group for the initialization constants and name it, for example, **Constants**.
- 2 From the context menu, choose **Create GroupInfo**.
- 3 In the Property Value List, enter a package name, for example, **Constants**. This lets you export variables in the group to the specified package. Constants in AUTOSAR files located in the package can be imported to that location.
- 4 From the context menu, choose **Create Variable**.
- 5 Enter a name for the variable, for example, **MyInitializationConstant**.
- 6 In the Property Value List, specify the following settings:

Property	Value
Type	Select the user-defined type that is consistent with the calibration parameter. <sup>1)</sup>
ApplicationDataTypeRef	The DD ApplicationDataType object that is associated with the <a href="#">implementation data type (IDT)</a> that you referenced at the Type property.
Value	Specify the initialization constant value, for example, 0.
VariableClass	Specify the variable class. The recommended class depends on the data type: <ul style="list-style-type: none"> <li>▪ Non structured AUTOSAR constants - AUTOSAR/CONST_VALUE</li> <li>▪ Structured AUTOSAR constants - AUTOSAR/CONST_STRUCT</li> <li>▪ Components of a struct - AUTOSAR/CONST_STRUCT_COMPONENT</li> </ul>

<sup>1)</sup> If you select a struct user type, you have to create components of the communication subject that are consistent with the selected struct user type. You can use the **Adjust to Typedef** context menu command to synchronize this communication subject with the referenced DD Typedef object.

- 7 In the Data Dictionary Navigator, select the DD MyRequireCalPrmPort object.
- 8 From its context menu, select **Create CalPrmComSpec**.
- 9 In the DD CalPrmComSpec object's Property Value List, make the following settings:

Property	Value
CalPrmElementRef	MyCalPrmElement
InitValueRef	MyInitializationConstant

**Interim result**

You specified an initialization constant for a CalPrmElement. The initialization is performed by the RTE.

Next you model parameter communication for calibration.

**Part 2****To model parameter communication for calibration**

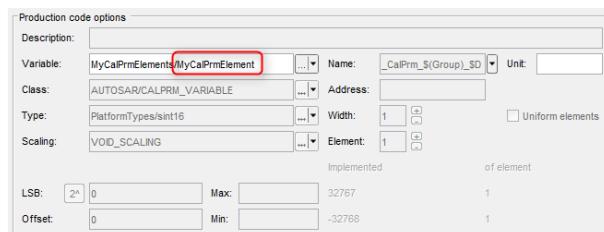
- 1 In the Data Dictionary Navigator, create a DD VariableGroup object and rename it as required, e.g., MyCalPrmElements.
- 2 At the MyRequireCalPrmPort object, set the VariableGroupRef property to MyCalPrmElements.
- 3 From the context menu of MyRequireCalPrmPort, select Synchronize Interface Settings.

TargetLink creates a properly specified DD Variable object for every DD CalPrmElement object contained in the DD CalPrmInterface object that is referenced via the require port's CalPrmInterfaceRef property.

**Note**

When importing parameter interfaces and CalPrm elements from an AUTOSAR file, TargetLink automatically synchronizes the interface settings. You can change this behavior via the **Autosar/Config/ImportExport/SynchronizeCalPrmInterfaceSettings** property. Remember to resynchronize the interface settings after changing, adding, or removing DD CalPrmElement objects from the interface. If you do not specify the VariableGroupRef property of the RequireCalPrmPort object, TargetLink sets this property to **Variables/<SWC>/<RequireCalPrmPort>/<CalPrmElement>** and creates the DD Variable objects accordingly. If the referenced DD VariableGroup object already exists, TargetLink deletes all the DD Variable objects contained in this group before creating new ones.

- 4 In the model, reference the calibration parameter at a TargetLink block that is used to model the SWC's internal behavior. The following screenshot shows the reference on the Gain page of a Gain Block:

**Result**

You modeled parameter communication and created a DD Variable object that represents a CalPrmElement that you used while modeling a SWC's internal behavior.

TargetLink will create a call of appropriate RTE API function:

```
Rte_Prm_MyRequireCalPrmPort_MyCalPrmElement();
```

**Related topics****Basics**

[Basics on Preparing SWCs for Measurement and Calibration](#)..... 235

**HowTos**

<a href="#">How to Create Communication Subjects for Classic AUTOSAR Communication</a> .....	103
<a href="#">How To Create Interfaces</a> .....	100
<a href="#">How to Create Ports</a> .....	101
<a href="#">How to Create Software Components</a> .....	66

**References**

- [Adjust to Typedef](#) ( [TargetLink Data Dictionary Manager Reference](#))
- [Gain Block](#) ( [TargetLink Model Element Reference](#))
- [Synchronize Interface Settings](#) ( [TargetLink Data Dictionary Manager Reference](#))

## Basics on Static and Constant Memories

**Introduction**

In [Classic AUTOSAR](#), static memories and constant memories let you perform measurement and calibration without involvement of the RTE. This can make implementations more efficient by avoiding the indirection caused by the RTE API.

**Static and constant memories according to Classic AUTOSAR**

According to [Classic AUTOSAR](#), **staticMemory** and **constantMemory** belong to a software component's internal behavior and are to be used in special use cases of measurement and calibration that do not require RTE-based memory allocation or variable management.

From the SWC's point of view, constant memories are constants that are read-only while static memories can also be written to. Functionally, constant memories are like shared calibration parameters the SWC defines itself.

In AUTOSAR (ARXML) files, they are described in **SWC-INTERNAL-BEHAVIOR** elements, as shown in the following table:

<b>Static Memory</b>	<b>Constant Memory</b>
VARIABLE-DATA-PROTOTYPE elements of STATIC-MEMORY elements	PARAMETER-DATA-PROTOTYPE elements of CONSTANT-MEMORY elements

**Note**

According to [Classic AUTOSAR](#), static memories cannot be used for software components that are prepared for multiple instantiation.

**Static and constant memories in TargetLink**

In TargetLink, you specify static and constant memories via DD Variable objects. From a C point of view, their scope must be global (or static global if your compiler supports it). This ensures that their addresses are known so they can be accessed during run time. The following table shows how static memories and constant memories are specified in the Data Dictionary:

Kind	DD Location
Static memory	In a DD VariableGroup object that is referenced via the StaticMemoriesRef property of a DD SoftwareComponent object's RelatedVariables object.
Constant memory	In a DD VariableGroup object that is referenced via the ConstantMemoriesRef property of a DD SoftwareComponent object's RelatedVariables object.

**Default variable classes** TargetLink always uses the same variable classes for importing constant memories and static memories to the Data Dictionary, respectively.

For constant memories, TargetLink uses `MERGEABLE_CAL`, for static memories, TargetLink uses `MERGEABLE_DISP`.

**Defining user-specified variable classes** You can also use your own user-specified DD VariableClass objects to specify constant memories and static memories. These objects must be specified as shown in the following table:

Property	Value	
	Constant Memory	Static Memory
Info	<code>readonly</code>	<code>readwrite</code>
Scope	<code>global</code>	

Additionally, if your compiler is able to get the addresses of static local variables, consider setting the VariableClass object's `Storage` property to `static`.

**Tip**

For tool chain automation, you can change the variable classes via the `tl_post_arimport_hook` hook script.

**Static and constant memories in the model**

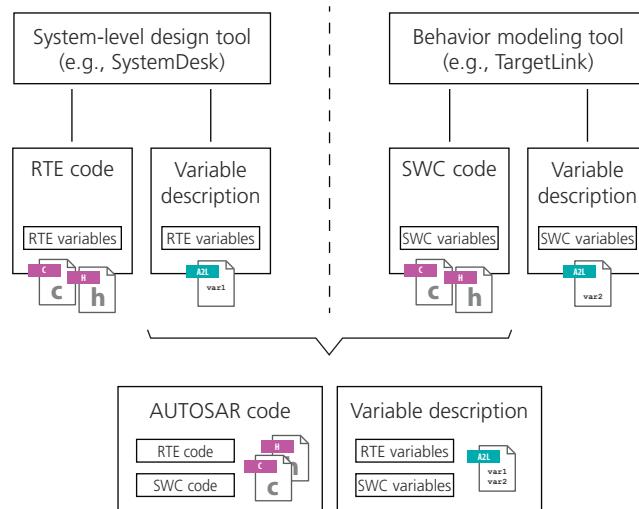
In the model, you can reference the DD Variable objects that represent your static and constant memories at blocks you use to model an SWC's internal behavior.

**Importing and exporting static and constant memories**

During AUTOSAR import, TargetLink creates DD Variable objects for static memories and constant memories defined in the ARXML in predefined DD VariableGroup objects. This is shown in the following table:

Kind	DD Location
Static memory	/Pool/Variables/<SoftwareComponent>/StaticMem/
Constant memory	/Pool/Variables/<SoftwareComponent>/ConstMem/

**Static and constant memories in A2L files** Because the variables for static memories and constant memories are not managed by the RTE but belong to the SWC code, they are defined by TargetLink. Accordingly, they are included in A2L files generated by TargetLink:



## Related topics

Basics

- |  |   |
|--|---|
| Exchanging A2L Files (                         |  TargetLink Interoperation and Exchange Guide) |
| Exchanging AUTOSAR Files (                     |  TargetLink Interoperation and Exchange Guide) |
| Preparing SVCs for Multiple Instantiation..... | 227   |

HowTos

- |   |     |
|---|-----|
| How to Model Constant Memories for Calibration..... | 252 |
| How to Model Static Memories for Measurement.....   | 250 |

## References

- tl post arimport hook ( TargetLink File Reference)

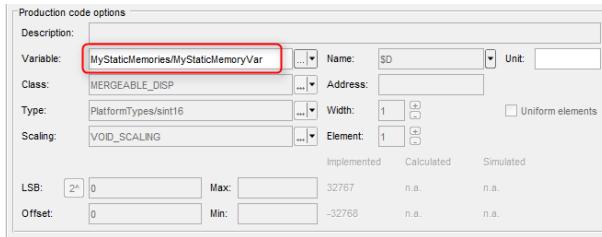
## How to Model Static Memories for Measurement

<b>Static memories</b>	Static memories are measurable variables that belong to a software component's production code. They have an <a href="#">Classic AUTOSAR</a> data type but are not instantiated by the RTE.												
<b>Restrictions</b>	Static memories are not supported for software components that are prepared for multiple instantiation.												
<b>Preconditions</b>	<p>The following preconditions must be fulfilled:</p> <ul style="list-style-type: none"> <li>▪ You specified one or more data types. Refer to:           <ul style="list-style-type: none"> <li>▪ <a href="#">Creating Implementation Data Types from Scratch (Classic AUTOSAR)</a> on page 41</li> <li>▪ <a href="#">Creating Application Data Types from Scratch (Classic AUTOSAR)</a> on page 49</li> <li>▪ <a href="#">Using TargetLink's Wizards to Create Application Data Types or Implementation Data Types (Classic AUTOSAR)</a> on page 57</li> </ul> </li> <li>▪ You specified scalings and constrained range limits as required. Refer to <a href="#">Defining Types, Scalings, and Constrained Range Limits (Classic AUTOSAR)</a> on page 31.</li> </ul>												
<b>Method</b>	<p><b>To model a static memory for measurement</b></p> <ol style="list-style-type: none"> <li>1 In the Data Dictionary Navigator, create a DD VariableGroup object and rename it as required, e.g., MyStaticMemories.</li> <li>2 To the MyStaticMemories variable group, add a DD Variable object and rename it as required, e.g., MyStaticMemoryVar.</li> <li>3 In the MyStaticMemoryVar object's Property Value List, specify the following settings:</li> </ol> <table border="1"> <thead> <tr> <th>Property</th><th>Value</th></tr> </thead> <tbody> <tr> <td>Class</td><td>MERGEABLE_DISP<sup>1)</sup></td></tr> <tr> <td>Type</td><td>Select a user-defined data type.<sup>2)</sup></td></tr> <tr> <td>ApplicationDataTypeRef</td><td>The DD ApplicationDataType object that is associated with the <a href="#">implementation data type (IDT)</a> that you referenced at the Type property.</td></tr> <tr> <td>Width</td><td>If you select an array user type, you have to specify a consistent width.</td></tr> <tr> <td>Value</td><td>Enter an initialization value, e.g., 0.</td></tr> </tbody> </table> <p><b>Note</b></p> <p><b>Avoid ambiguous initializations of data prototypes</b></p> <p>Specify an initialization value for each <a href="#">data prototype</a> within the Data Dictionary whenever possible. This avoids ambiguities between <a href="#">Classic AUTOSAR</a> initialization values and initialization values of model elements that are required by Simulink.</p>	Property	Value	Class	MERGEABLE_DISP <sup>1)</sup>	Type	Select a user-defined data type. <sup>2)</sup>	ApplicationDataTypeRef	The DD ApplicationDataType object that is associated with the <a href="#">implementation data type (IDT)</a> that you referenced at the Type property.	Width	If you select an array user type, you have to specify a consistent width.	Value	Enter an initialization value, e.g., 0.
Property	Value												
Class	MERGEABLE_DISP <sup>1)</sup>												
Type	Select a user-defined data type. <sup>2)</sup>												
ApplicationDataTypeRef	The DD ApplicationDataType object that is associated with the <a href="#">implementation data type (IDT)</a> that you referenced at the Type property.												
Width	If you select an array user type, you have to specify a consistent width.												
Value	Enter an initialization value, e.g., 0.												

Property	Value
InitConstantName	Lets you specify a name for the initialization constant of the measurement variable.
InitConstantPackage	Lets you specify the package for the initialization constant of the measurement variable, e.g., <b>Constants</b> .

- 1) Or a user-specified DD VariableClass object whose Info property is set to **readonly** and whose Scope property is set to **global**. If your compiler is able to get the addresses of static local variables, consider setting the VariableClass object's Storage property to **static**.
- 2) If you select a struct user type, you have to create components of the communication subject that are consistent with the selected struct user type. You can use the Adjust to Typedef context menu command to synchronize this communication subject with the referenced DD Typedef object.

- 4) Locate the DD SoftwareComponent object for which you want to use static memories and select its RelatedVariables subtree.
- 5) From its context menu, select Create Reference to StaticMemories.
- 6) In the Property Value List, set the StaticMemoriesRef property to **MyStaticMemories**.
- 7) In the model, reference the **MyStaticMemoryVar** at a TargetLink block that is used to model the SWC's internal behavior. The following screenshot shows the reference on the Variable page of a Data Store Memory Block:



- 8) Generate code and export an AUTOSAR file.

## Result

You modeled a static memory and used it at a TargetLink block. During code generation, TargetLink creates a variable definition of the static memory.

An entry for a static memory in the software component's ARXML file looks like this:

```

<STATIC-MEMORY>
  <VARIABLE-DATA-PROTOTYPE>
    <SHORT-NAME>MyStaticMemoryVar</SHORT-NAME>
    <SW-DATA-DEF-PROPS>
      <SW-DATA-DEF-PROPS-VARIANTS>
        <SW-DATA-DEF-PROPS-CONDITIONAL>
          <SW-IMPL-POLICY>MEASUREMENT-POINT</SW-IMPL-POLICY>
        </SW-DATA-DEF-PROPS-CONDITIONAL>
      </SW-DATA-DEF-PROPS-VARIANTS>
    </SW-DATA-DEF-PROPS>
    <TYPE-TREF DEST="APPLICATION-PRIMITIVE-DATA-TYPE">/ADTGroup_MySWC/ADT_sint16</TYPE-TREF>
    <INIT-VALUE>
      <CONSTANT-REFERENCE>
        <CONSTANT-REF DEST="CONSTANT-SPECIFICATION">/StaticMemories/MyInitConstant</CONSTANT-REF>
      </CONSTANT-REFERENCE>
    </INIT-VALUE>
  </VARIABLE-DATA-PROTOTYPE>
</STATIC-MEMORY>

```

**Related topics****Basics**

Basics on Static and Constant Memories.....	247
Creating Application Data Types from Scratch (Classic AUTOSAR).....	49
Creating Implementation Data Types from Scratch (Classic AUTOSAR).....	41
Defining Types, Scalings, and Constrained Range Limits (Classic AUTOSAR).....	31
Using TargetLink's Wizards to Create Application Data Types or Implementation Data Types (Classic AUTOSAR).....	57

**HowTos**

How to Model Constant Memories for Calibration.....	252
---	-----

**References**

- [Adjust to Typedef \(TargetLink Data Dictionary Manager Reference\)](#)
- [Create Reference to <Object> \(TargetLink Data Dictionary Manager Reference\)](#)
- [Data Store Memory Block \(TargetLink Model Element Reference\)](#)

## How to Model Constant Memories for Calibration

**Constant memories**

Constant memories are calibratable variables that belong to a software component's production code. They have an [Classic AUTOSAR](#) data type but are not instantiated by the RTE.

**Preconditions**

The following preconditions must be fulfilled:

- You specified one or more data types. Refer to:
  - [Creating Implementation Data Types from Scratch \(Classic AUTOSAR\)](#) on page 41
  - [Creating Application Data Types from Scratch \(Classic AUTOSAR\)](#) on page 49
  - [Using TargetLink's Wizards to Create Application Data Types or Implementation Data Types \(Classic AUTOSAR\)](#) on page 57
- You specified scalings and constrained range limits as required. Refer to [Defining Types, Scalings, and Constrained Range Limits \(Classic AUTOSAR\)](#) on page 31.

**Method****To model a constant memory for calibration**

- 1 In the Data Dictionary Navigator, create a DD VariableGroup object and rename it as required, e.g., MyConstantMemories.
- 2 To the MyConstantMemories variable group, add a DD Variable object and rename it as required, e.g., MyConstantMemoryVar.

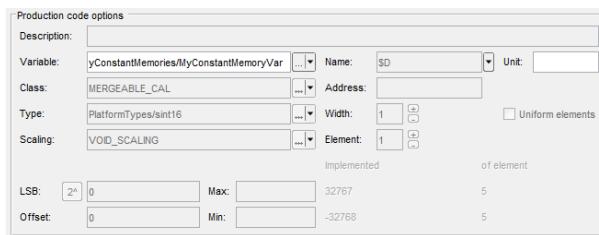
- 3** In the MyConstantMemoryVar object's Property Value List, specify the following settings:

Property	Value
Class	MERGEABLE_CAL <sup>1)</sup>
Type	Select a user-defined data type. <sup>2)</sup>
ApplicationDataTypeRef	The DD ApplicationDataType object that is associated with the <a href="#">implementation data type (IDT)</a> that you referenced at the Type property.
Width	If you select an array user type, you have to specify a consistent width.
Value	Enter an initialization value, e.g., 0.
<b>Note</b> <p><b>Avoid ambiguous initializations of data prototypes</b> Specify an initialization value for each <a href="#">data prototype</a> within the Data Dictionary whenever possible. This avoids ambiguities between <a href="#">Classic AUTOSAR</a> initialization values and initialization values of model elements that are required by Simulink.</p>	
InitConstantName	Lets you specify a name for the initialization constant of the calibration parameter.
InitConstantPackage	Lets you specify the package for the initialization constant of the calibration parameter, e.g., <b>Constants</b> .

<sup>1)</sup> Or a user-specified DD VariableClass object whose Info property is set to **readwrite** and whose Scope property is set to **global**. If your compiler is able to get the addresses of static local variables, consider setting the VariableClass object's Storage property to **static**.

<sup>2)</sup> If you select a struct user type, you have to create components of the communication subject that are consistent with the selected struct user type. You can use the Adjust to Typedef context menu command to synchronize this communication subject with the referenced DD Typedef object.

- 4** Locate the DD SoftwareComponent object for which you want to use constant memories and select its RelatedVariables subtree.
- 5** From its context menu, select Create Reference to ConstantMemories.
- 6** In the Property Value List, set the ConstantMemoriesRef property to **MyConstantMemories**.
- 7** In the model, reference the MyConstantMemoryVar at a TargetLink block that is used to model the SWC's internal behavior. The following screenshot shows the reference on the Gain page of the Gain Block:



- 8** Generate code and export an AUTOSAR file.

**Result**

You modeled a constant memory and used it at a TargetLink block. During code generation, TargetLink creates a variable definition of the constant memory.

An entry for a constant memory in the software component's ARXML file looks like this:

```
<CONSTANT-MEMORYS>
  <PARAMETER-DATA-PROTOTYPE>
    <SHORT-NAME>MyConstantMemoryVar</SHORT-NAME>
    <SW-DATA-DEF-PROPS>
      <SW-DATA-DEF-PROPS-VARIANTS>
        <SW-DATA-DEF-PROPS-CONDITIONAL>
          <SW-IMPL-POLICY>CONST</SW-IMPL-POLICY>
        </SW-DATA-DEF-PROPS-CONDITIONAL>
      </SW-DATA-DEF-PROPS-VARIANTS>
    </SW-DATA-DEF-PROPS>
    <TYPE-TREF DEST="APPLICATION-PRIMITIVE-DATA-TYPE">/ADTGroup_MySWC/ADT_sint16</TYPE-TREF>
    <INIT-VALUE>
      <CONSTANT-REFERENCE>
        <CONSTANT-REF DEST="CONSTANT-SPECIFICATION">/ConstantMemories/MyInitConstant</CONSTANT-REF>
      </CONSTANT-REFERENCE>
    </INIT-VALUE>
  </PARAMETER-DATA-PROTOTYPE>
</CONSTANT-MEMORYS>
```

**Related topics****Basics**

Basics on Static and Constant Memories.....	247
Creating Application Data Types from Scratch (Classic AUTOSAR).....	49
Creating Implementation Data Types from Scratch (Classic AUTOSAR).....	41
Defining Types, Scalings, and Constrained Range Limits (Classic AUTOSAR).....	31
Using TargetLink's Wizards to Create Application Data Types or Implementation Data Types (Classic AUTOSAR).....	57

**HowTos**

How to Model Static Memories for Measurement.....	250
---	-----

**References**

Adjust to Typedef (TargetLink Data Dictionary Manager Reference)
Create Reference to <Object> (TargetLink Data Dictionary Manager Reference)
Gain Block (TargetLink Model Element Reference)

## Basics on Preparing Look-Up Tables for Measurement and Calibration (Classic AUTOSAR)

### Look-up tables as described by Classic AUTOSAR

According to [Classic AUTOSAR](#), [characteristic tables](#) are described by [compound primitive data types](#).

**Axis description** The table axes are described via SW-CALPRM-AXIS-SET elements that belong to the [application data type's \(ADT\)](#) SW-DATA-DEF-PROPS element. Each SW-CALPRM-AXIS-SET element contains one or more SW-CALPRM-AXIS elements, each of which describes an axis. Each axis can be individual or grouped, as shown in the following table:

Element	Description
SW-AXIS-INDIVIDUAL	An axis that is specific to a characteristic table.
SW-AXIS-GROUPED	An axis that is shared across different characteristic tables.

**Input values** The table input values are described in the SW-VARIABLE-REFS element of each SW-AXIS-INDIVIDUAL element that describes an individual axis. Each SW-VARIABLE-REFS element contains further elements that depend on the communication kind.

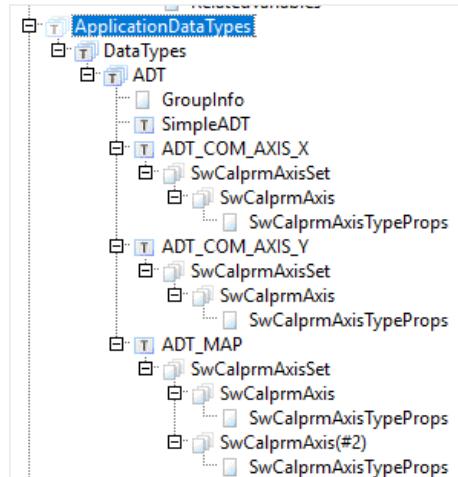
## Look-up tables in TargetLink

TargetLink lets you work with look-up tables as described by [Classic AUTOSAR](#) and prepare them for measurement and calibration. The following [compound primitive data types](#) are supported:

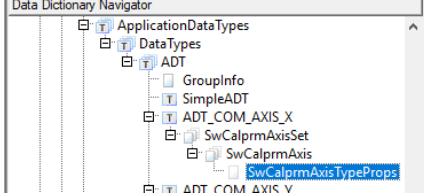
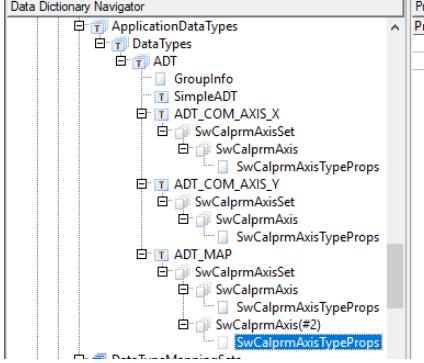
Supported Category	Description
COM_AXIS	An axis that can be used in multiple <a href="#">characteristic tables</a> .
CURVE	A 1-D look-up table.
MAP	A 2-D look-up table.

## Specifying axis descriptions

You specify axis descriptions at DD ApplicationDataType objects via their SwCalprmAxisSet child object:



Both, individual axes and grouped axes are possible:

DD Object	Axis Type	Example
SwCalprmAxisTypeProps	Individual	 <p>Data Dictionary Navigator</p> <ul style="list-style-type: none"> <li>ApplicationDataTypes</li> <li>DataTypes           <ul style="list-style-type: none"> <li>ADT               <ul style="list-style-type: none"> <li>GroupInfo</li> <li>SimpleADT</li> <li>ADT_COM_AXIS_X                   <ul style="list-style-type: none"> <li>SwCalprmAxisSet                       <ul style="list-style-type: none"> <li>SwCalprmAxis                           <ul style="list-style-type: none"> <li>SwCalprmAxisTypeProps</li> </ul> </li> </ul> </li> </ul> </li> <li>ADT_COM_AXIS_Y</li> <li>ADT_MAP                   <ul style="list-style-type: none"> <li>SwCalprmAxisSet                       <ul style="list-style-type: none"> <li>SwCalprmAxis                           <ul style="list-style-type: none"> <li>SwCalprmAxisTypeProps</li> </ul> </li> </ul> </li> </ul> </li> </ul> </li> </ul> </li> </ul>
	Grouped	 <p>Data Dictionary Navigator</p> <ul style="list-style-type: none"> <li>ApplicationDataTypes</li> <li>DataTypes           <ul style="list-style-type: none"> <li>ADT               <ul style="list-style-type: none"> <li>GroupInfo</li> <li>SimpleADT</li> <li>ADT_COM_AXIS_X                   <ul style="list-style-type: none"> <li>SwCalprmAxisSet                       <ul style="list-style-type: none"> <li>SwCalprmAxis                           <ul style="list-style-type: none"> <li>SwCalprmAxisTypeProps</li> </ul> </li> </ul> </li> </ul> </li> <li>ADT_COM_AXIS_Y                   <ul style="list-style-type: none"> <li>SwCalprmAxisSet                       <ul style="list-style-type: none"> <li>SwCalprmAxis                           <ul style="list-style-type: none"> <li>SwCalprmAxisTypeProps</li> </ul> </li> </ul> </li> </ul> </li> <li>ADT_MAP                   <ul style="list-style-type: none"> <li>SwCalprmAxisSet                       <ul style="list-style-type: none"> <li>SwCalprmAxis                           <ul style="list-style-type: none"> <li>SwCalprmAxisTypeProps</li> <li>SwCalprmAxis(#2)                               <ul style="list-style-type: none"> <li>SwCalprmAxisTypeProps</li> </ul> </li> </ul> </li> </ul> </li> </ul> </li> </ul> </li></ul></li></ul>

### Specifying the runnable's parameter accesses

To prepare the look-up table for calibration and measurement, you specify the runnable's parameter accesses to that table.

#### Note

TargetLink considers the information stored in DD SwCalprmAxisSet objects that are placed in the subtree of a parameter access only for exporting the ARXML file.

Parameter accesses are specified via the following child elements of DD Runnable objects:

DD Object	Instruction
SharedCalPrmVariableAccess	<p>Specifies the runnable's access to a shared parameter.</p> <ul style="list-style-type: none"> <li>Reference a suitable DD Variable object that specifies the shared parameter.</li> <li>Specify the axis descriptions via the SwCalprmAxisSet object:</li> </ul>
PerInstanceCalPrmVariableAccess	<p>Specifies the runnable's access to a per instance calibration parameter.</p> <ul style="list-style-type: none"> <li>Reference a suitable DD Variable object that specifies the per instance parameter.</li> <li>Specify the axis descriptions via the SwCalprmAxisSet object:</li> </ul>
CalPrmAccessPoint	<p>Specifies the runnable's access to calibration parameters in parameter communication.</p> <ul style="list-style-type: none"> <li>Specify the axis descriptions via the SwCalprmAxisSet object:</li> </ul> <ul style="list-style-type: none"> <li>At each DD CalPrmAccessPoint object, reference the correct DD CalPrmElement and RequireCalPrmPort object.</li> </ul>

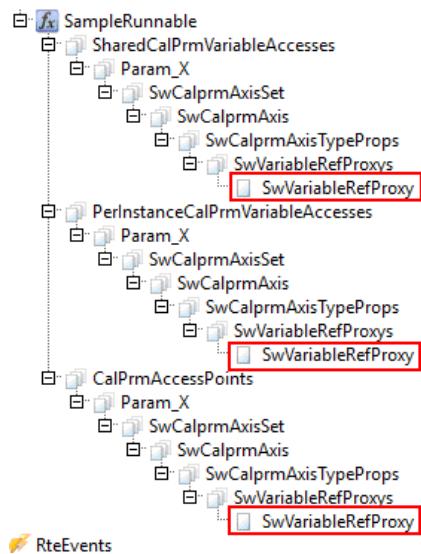
DD Object	Instruction
	<ul style="list-style-type: none"> <li>At the DD Variable objects that represent your calibration parameters, reference the correct DD ApplicationDataType object:</li> </ul> <pre>     graph TD       Variables[Variables] --&gt; RTOS[RTOS]       Variables --&gt; AUTOSAR[AUTOSAR]       Variables --&gt; CalibrationSWC[CalibrationSWC]       CalibrationSWC --&gt; MyInitializationConstants[MyInitializationConstants]       MyInitializationConstants --&gt; ConstantXy[ConstantXy]       ConstantXy --&gt; Map4x4[Map4x4]       MyInitializationConstants --&gt; MyCalPrm[MyCalPrm]       MyCalPrm --&gt; Axis_X[Axis_X]       MyCalPrm --&gt; Axis_Y[Axis_Y]       MyCalPrm --&gt; MAP[MAP]   </pre>

**Note**

The description of the axes made at the parameter accesses must be consistent with the description made at the ADTs.

### Specifying the data input of the table

The table input data is specified at the runnable's parameter accesses via DD SwVariableRefProxy objects that belong to an individual axis:



You specify its properties depending on the communication kind that is used to input the data into the table, e.g., sender-receiver communication:

Property	Value
DataElementRef	<InputData/DataElements/InputDataX>
ReceiverPortRef	<SomeInputData>
ClientPortRef	<value not set>
InterRunnableVariableRef	<value not set>
NvDataElementRef	<value not set>
NvReceiverPortRef	<value not set>
NvSenderPortRef	<value not set>
NvSenderReceiverPortRef	<value not set>
OperationArgumentRef	<value not set>
SenderPortRef	<value not set>
SenderReceiverPortRef	<value not set>
ServerPortRef	<value not set>
VariableRef	<value not set>

If you want to use per instance memory as table input data, you can use the DD `SwVariableRefProxy` object's `VariableRef` property to reference the DD `Variable` object that specifies the per instance memory.

## Modeling example

Usually, the required data is imported to the Data Dictionary from an ARXML file that is provided by an architecture tool such as dSPACE SystemDesk. If you have to specify the required data yourself, refer to [Example of Preparing a Look-Up Table for Measurement and Calibration](#) on page 260.

## Related topics

### Basics

Basics on Preparing SWCs for Measurement and Calibration.....	235
Introduction to Data Types in Classic AUTOSAR.....	32

### HowTos

How to Model Parameter Communication for Calibration.....	244
How to Model Per Instance Memories.....	214
How to Model Per Instance Parameters for Calibration.....	239
How to Model Shared Parameters for Calibration.....	241

### Examples

Example of Preparing a Look-Up Table for Measurement and Calibration.....	260
---	-----

## Example of Preparing a Look-Up Table for Measurement and Calibration

### Overview of the example

The variables that you want to visualize in a calibration and measurement tool must be described in an A2L file. If these variables belong to a [characteristic table \(Classic AUTOSAR\)](#), the variables must be described as table elements. This means, the A2L file must describe to which axis of the table each variable belongs.

#### Note

According to [Classic AUTOSAR](#), these A2L files are generated from the ARXML file by the RTE generator.

To ensure that the ARXML file that is exported from TargetLink contains the correct description of the table's variables, the Data Dictionary must contain the following:

- The specification of the characteristic table via [primitive application data types](#).
- The specification of the calibration parameters used to calibrate the table.
- The parameter accesses of the runnable.
- The table input data.

Usually, the required data is imported to the Data Dictionary from an ARXML file that is provided by an architecture tool such as dSPACE SystemDesk. However, this example assumes that you cannot import the required data but have to specify it yourself.

The example uses shared calibration parameters and sender-receiver communication. Using per instance calibration parameters, calibration parameters, or another communication kind works in a similar way.

### Preconditions

The following preconditions must be fulfilled:

- You modeled a runnable. Refer to [How to Model Runnables](#) on page 71.
- You modeled sender-receiver communication with a receiver port called `InputData` and two data elements called `InputDataX` and `InputDataY`. Refer to [Modeling Sender-Receiver Communication](#) on page 108.

### Workflow

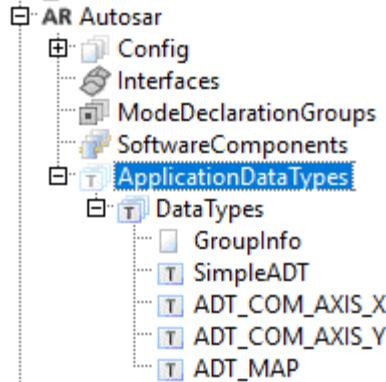
This workflow consists of the following parts:

- To specify a characteristic table via application data types, refer to [Part 1](#) on page 261.
- To specify variables for shared calibration parameters, refer to [Part 2](#) on page 263.
- To specify parameter accesses, refer to [Part 3](#) on page 264.
- To specify the table input data, refer to [Part 4](#) on page 266.

- To model the data input of a look-up table block via sender-receiver communication, refer to [Part 5](#) on page 266.
- To export an ARXML file, refer to [Part 6](#) on page 267.

**Part 1****To specify a characteristic table via application data types**

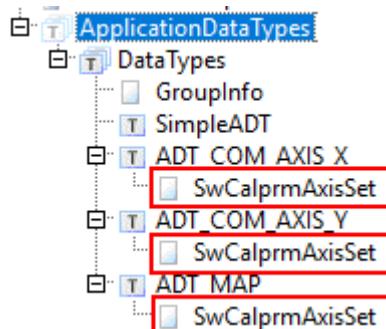
- 1 In /Pool/Autosar/ApplicationDataTypes, create suitable DD ApplicationDataType objects for the table axes and the table map:



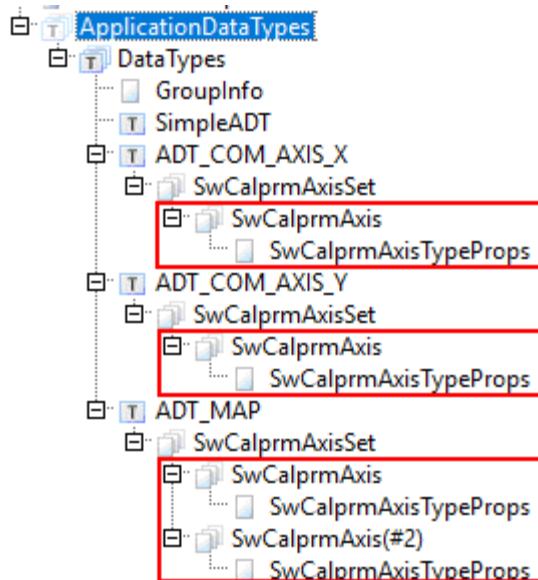
- 2 Specify their properties according to the following table:

DD ApplicationDataType Object	Property	Value
SimpleADT	ApplicationDataTypeKind	Primitive
	Category	VALUE
	CalibrationAccess	ReadWrite
ADT_COM_AXIS_X and ADT_COM_AXIS_Y	ApplicationDataTypeKind	Primitive
	Category	COM_AXIS
	CalibrationAccess	ReadWrite
ADT_MAP	ApplicationDataTypeKind	Primitive
	Category	MAP
	CalibrationAccess	ReadWrite

- 3 For ADT\_COM\_AXIS\_X, ADT\_COM\_AXIS\_Y, and ADT\_MAP, create the DD SwCalprmAxisSet child object via their context menus:



- 4 To each DD `SwCalprmAxisSet` object, add a suitable number of DD `SwCalprmAxis` child objects via the Create `SwCalprmAxis` context menu command:

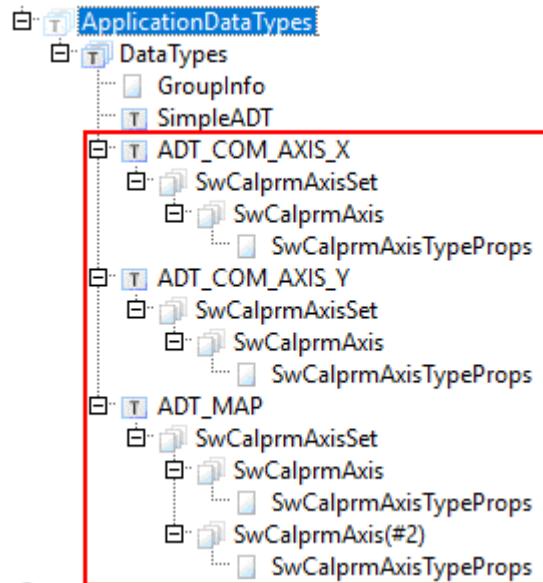


- 5 Specify each DD `SwCalprmAxis` object according to the following table:

DD ApplicationDataType Object	Child Object	Property	Value
<code>ADT_COM_AXIS_X</code>	<code>SwCalprmAxis</code>	<code>SwAxisIndex</code>	2
	<code>SwCalprmAxisTypeProps</code>	<code>AxisKind</code>	<code>SwAxisIndividual</code>
		<code>SwMinAxisPoints</code>	4
		<code>SwMaxAxisPoints</code>	4
		<code>InputVariableTypeRef</code>	<code>DataTypes/ADT/SimpleADT</code>
<code>ADT_COM_AXIS_Y</code>	<code>SwCalprmAxis</code>	<code>SwAxisIndex</code>	1
	<code>SwCalprmAxisTypeProps</code>	<code>AxisKind</code>	<code>SwAxisIndividual</code>
		<code>SwMinAxisPoints</code>	4
		<code>SwMaxAxisPoints</code>	4
		<code>InputVariableTypeRef</code>	<code>DataTypes/ADT/SimpleADT</code>
<code>ADT_COM_AXIS_MAP</code>	<code>SwCalprmAxis</code>	<code>SwAxisIndex</code>	1
		<code>CalprmAxisCategory</code>	<code>COM_AXIS</code>
	<code>SwCalprmAxisTypeProps</code>	<code>AxisKind</code>	<code>SwAxisGrouped</code>
		<code>SharedAxisTypeRef</code>	<code>DataTypes/ADT/ADT_COM_AXIS_X</code>
	<code>SwCalprmAxis(#2)</code>	<code>CalprmAxisCategory</code>	<code>COM_AXIS</code>
		<code>SwAxisIndex</code>	2
	<code>SwCalprmAxisTypeProps</code>	<code>AxisKind</code>	<code>SwAxisGrouped</code>
		<code>SharedAxisTypeRef</code>	<code>DataTypes/ADT/ADT_COM_AXIS_Y</code>

**Interim result**

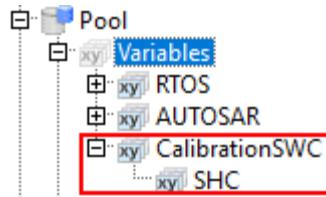
You created the description of a characteristic table as described by [Classic AUTOSAR](#) via DD ApplicationDataType objects:



Next, you specify the calibration parameters.

**Part 2****To specify variables for shared calibration parameters**

- 1 To the /Pool/Variables subtree, add a suitable DD VariableGroup object to collect the DD Variable objects that represent the shared calibration parameters:



- 2 Add three DD Variable objects to the group via its AUTOSAR/CreateSharedCalPrmVariable context menu command and specify them according to the following table:

Name of DD Variable Object	Property	Value
SharedParameter_Map	ApplicationDataTypeRef	DataTypes/ADT/ADT_MAP
	Value	[1 2 3 4;5 6 7 8;9 10 11 12;13 14 15 16]
SharedParameter_Axis_X	ApplicationDataTypeRef	DataTypes/ADT/ADT_COM_AXIS_X
	Value	[1 2 3 4]
SharedParameter_Axis_Y	ApplicationDataTypeRef	DataTypes/ADT/ADT_COM_AXIS_Y
	Value	[1 2 3 4]

- 3 Locate the DD SoftwareComponent object and select its RelatedVariables child object.
- 4 At the CalibratablesRef property, reference the VariableGroup object that contains the calibration parameters.
- 5 From the context menu of the SoftwareComponent, select AUTOSAR Tools/ImplementationDataType Creation Wizard.
- 6 Use the ImplementationDataType Creation Wizard to create and map suitable [implementation data types](#).

**Interim result**

You created shared calibration parameters and established a mapping between ADTs and IDTs as described by [Classic AUTOSAR](#).

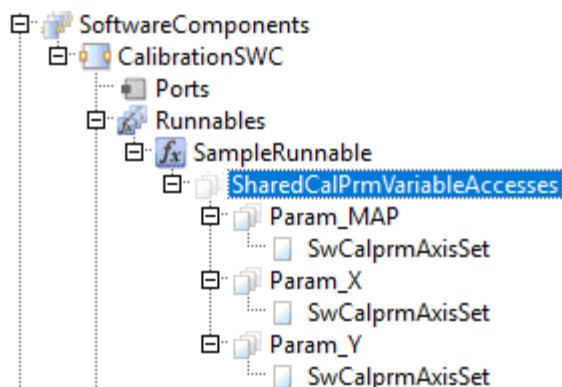
Next, you specify the runnable's parameter accesses.

**Part 3****To specify parameter accesses**

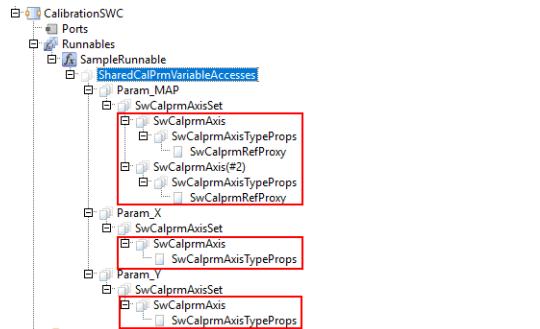
- 1 Select the DD Runnable object.
- 2 From its context menu, select Create SharedCalprmVariableAccesses.
- 3 To the SharedCalprmVariableAccess object, add three SharedCalprmVariableAccess objects and specify them according to the following table:

Name of DD SharedCalprmVariableAccess Object	Property	Value
Param_X	VariableRef	CalibrationSWC/SHC/SharedParameter_Axis_X
Param_Y		CalibrationSWC/SHC/SharedParameter_Axis_Y
Param_MAP		CalibrationSWC/SHC/SharedParameter_Map

- 4 To Param\_MAP, Param\_X, and Param\_Y, add the SwCalprmAxisSet child object via their context menus:



- 5 Create suitable SwCalprmAxis, SwCalprmAxisTypeProps, and SwCalprmRefProxy objects:



- 6 Specify them according to the following table.

**Note**

The description of the axes made at the parameter accesses must be consistent with the description made at the ADTs in step 5 of [Part 1](#) on page 261.

DD SharedCalPrmVariableAccess Object	Child Object	Property	Value
Param_X	SwCalprmAxisTypeProps	AxisKind	SwAxisIndividual
		SwMinAxisPoints	4
		SwMaxAxisPoints	4
Param_Y	SwCalprmAxisTypeProps	AxisKind	SwAxisIndividual
		SwMinAxisPoints	4
		SwMaxAxisPoints	4
Param_MAP	SwCalprmAxis	SwAxisIndex	1
	SwCalprmAxisTypeProps	AxisKind	SwAxisGrouped
	SwCalprmRefProxy	VariableRef	CalibrationSWC/SHC/SharedParameter_Axis_X
	SwCalprmAxis(#2)	SwAxisIndex	2
	SwCalprmAxisTypeProps	AxisKind	SwAxisGrouped
	SwCalprmRefProxy	VariableRef	CalibrationSWC/SHC/SharedParameter_Axis_Y

**Interim result**

You created the parameter accesses of the runnable.

Next, you specify the table input data.

**Part 4****To specify the table input data**

- 1 To the SwCalprmAxisTypeProps subtree of PARAM\_X and PARAM\_Y, add the SwVariableRefProxy object and specify it as shown in the following table:

DD SharedCalPrmVariableAccess Object	Child Object	Property	Value
Param_X	SwVariableRefProxy	ReceiverPortRef	InputData
		DataElementRef	InputDataX
Param_Y	SwVariableRefProxy	ReceiverPortRef	InputData
		DataElementRef	InputDataX

**Interim result**

You specified the table input data.

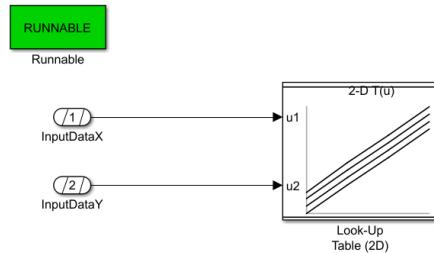
Next, you use the shared calibration parameters at model elements.

**Part 5****To model the data input of a look-up table block via sender-receiver communication**

- 1 Open the [runnable subsystem](#) and add a Look-Up Table (2D) block.
- 2 Open its block dialog and make the following settings:

Dialog Page	Property	Value
General	Generate map struct	Cleared
Table	Variable	CalibrationSWC/SHC/SharedParameter_Map
	Use value of Data Dictionary variable	Selected
Axis#1 (row)	Variable	CalibrationSWC/SHC/SharedParameter_Axis_X
	Use value of Data Dictionary variable	Selected
Axis#2 (column)	Variable	CalibrationSWC/SHC/SharedParameter_Axis_Y
	Use value of Data Dictionary variable	Selected

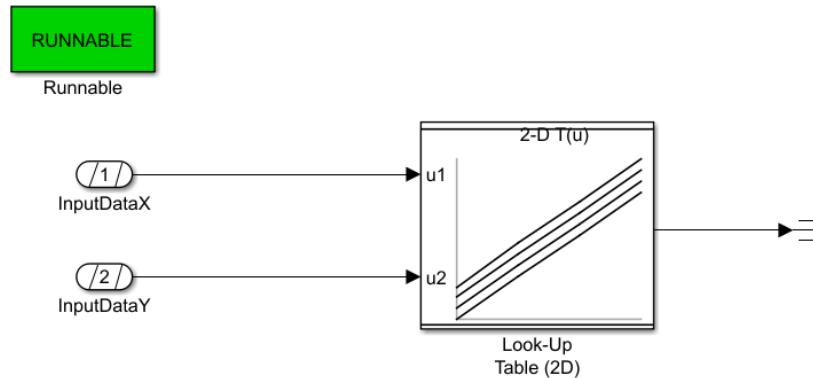
- 3 Add two InPort blocks called InputDataX and InputDataY to model the table input data:



- 4 Specify each block for sender-receiver communication and reference the correct DD ReceiverPort and DataElement objects.
- 5 Add further model elements as required.

**Interim result**

You used a look-up table block for modeling. The [Runnable subsystem](#) looks like this:



Next, you export an ARXML file that contains the representation of the characteristic table as described by [Classic AUTOSAR](#).

**Part 6****To export an ARXML file**

- 1** Generate code.
- 2** Export the ARXML file.

**Result**

You prepared a look-up table for measurement and calibration by creating the following specifications in the Data Dictionary:

- DD ApplicationDataType objects that represent the characteristic table as described by [Classic AUTOSAR](#).
- DD Variable objects that represent the shared calibration parameters.
- DD SharedCalPrmVariableAccess objects that represent the runnable's access to the shared calibration parameters.

You modeled a runnable involving a look-up table block whose variables are calibratable and whose data is input via sender-receiver communication.

You generated code and exported an ARXML file.

The ARXML file contains all the necessary information for the RTE Generator to generate an A2L file that contains a description of the characteristic table.

**Tip**

You can also use the DD API. Consider the following example for creating and specifying DD ApplicationDataType objects:

```
%Create a DD ApplicationDataType object called SimpleADT
dsdd('AddApplicationDataTypeObject', ...
      '/Pool/Autosar/ApplicationDataTypes', 'SimpleADT');

%Set its Kind property to Primitive
dsdd('SetApplicationDataTypeKind', ...
      '/Pool/Autosar/ApplicationDataTypes/SimpleADT', 'Primitive');

%Set its Category property to Value
dsdd('SetCategory', ...
      '/Pool/Autosar/ApplicationDataTypes/SimpleADT', 'VALUE');

%Set its CalibrationAccess property to ReadWrite
dsdd('SetCalibrationAccess', ...
      '/Pool/Autosar/ApplicationDataTypes/SimpleADT', 'ReadWrite')
```

**Related topics****Basics**

Basics on Preparing Look-Up Tables for Measurement and Calibration (Classic AUTOSAR).....	254
Defining Types, Scalings, and Constrained Range Limits (Classic AUTOSAR).....	31
Exchanging AUTOSAR Files (TargetLink Interoperation and Exchange Guide)	
Preparing SWCs for Measurement and Calibration.....	235

**HowTos**

How to Model Runnables.....	71
-----------------------------	----

**References**

ImplementationDataType Creation Wizard Command (TargetLink Data Dictionary Manager Reference)	
Look-Up Table (2D) Block (TargetLink Model Element Reference)	
Look-Up Table Block (TargetLink Model Element Reference)	

# Using TargetLink's Modeling Features

## Where to go from here

## Information in this section

Basics on Modeling Classic AUTOSAR and non-AUTOSAR Controllers in one Model.....	269
Partitioning Model and Code for Classic AUTOSAR.....	271

## Basics on Modeling Classic AUTOSAR and non-AUTOSAR Controllers in one Model

### Introduction

To generate [Classic AUTOSAR](#) compliant code and non-AUTOSAR code from one model.

### One model for generating Classic AUTOSAR compliant code non-AUTOSAR code

**Classic AUTOSAR and Standard code generation modes** TargetLink's [Classic AUTOSAR](#) and Standard code generation modes allow you to generate code from one model for AUTOSAR and non-AUTOSAR use cases. For this purpose TargetLink provides blocks for specifying AUTOSAR and non-AUTOSAR data on different dialog pages at the same block (refer to [AUTOSAR and non-AUTOSAR block data](#) on page 270).

**Specifying AUTOSAR and non-AUTOSAR block data** TargetLink lets you view and edit block data for the selected code generation mode only. However, you can click  in blocks that can hold AUTOSAR and non-AUTOSAR data to switch to the dual edit mode, which allows you to view and edit block data for [Classic AUTOSAR](#) and standard code generation.

**Switching between code generation modes** TargetLink lets you switch a model's code generation mode in the TargetLink Main Dialog. Block

specifications are kept per code generation mode, i.e., switching from one code generation mode to another and back again does not change any block data.

**Tip**

Press **Ctrl + D** to update your model after switching code generation modes.

---

**AUTOSAR and non-AUTOSAR block data**

The following TargetLink blocks can hold AUTOSAR and non-AUTOSAR data at the same time. They have different behaviors in [Classic AUTOSAR](#) and in standard code generation modes.

- Function
- InPort/Bus Import
- OutPort/Bus Outport
- Data Store Memory

The following blocks hold only AUTOSAR data. They are ignored/fed through in general code generation mode.

- ReceiverComSpec
- SenderComSpec

All the other Simulink/TargetLink blocks that are supported in TargetLink models have identical behaviors in [Classic AUTOSAR](#) and in standard code generation modes.

---

**Variable classes for Classic AUTOSAR and non-AUTOSAR use cases**

Variable classes for per instance memories and calibration parameters should be switched with the code generation mode, i.e., they can be different for the [Classic AUTOSAR](#) and standard code generation modes.

You can switch variable classes by performing the following steps:

1. Copy the `/Pool/VariableClasses/AUTOSAR` subtree, for example, to `/Pool/VariableClasses/Standard`.
2. Adapt the variable class properties of the copy as required for standard code generation.
3. Write and run a script that switches *AUTOSAR* with *Standard* variable classes for all the variables. If required, you have to additionally switch the name templates of the variables.

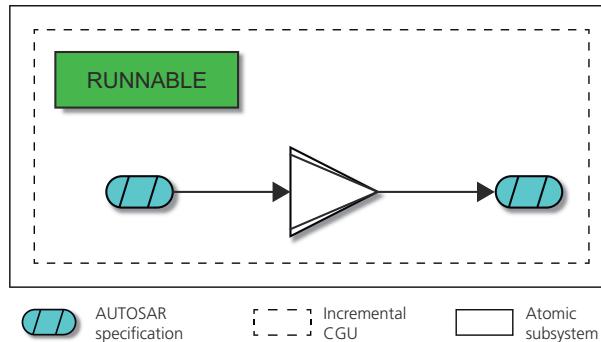
# Partitioning Model and Code for Classic AUTOSAR

## Introduction

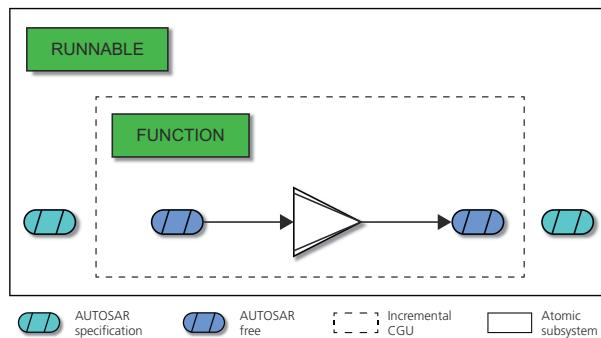
When modeling a [Runnable subsystem](#), you can partition it by using [incremental CGUs](#).

## Port blocks carrying AUTOSAR specification

You have to place the port blocks you use to model the runnable's interface and that carry AUTOSAR specifications in the same [code generation unit \(CGU\)](#) as the Runnable block:



Accordingly, you *must not* place the port blocks in an [incremental code generation unit \(CGU\)](#) that is located below the system that contains the Runnable block. Instead you have to model the interface of the incremental CGU via port blocks without AUTOSAR specification:



**Contents of incremental CGU**

**Model elements carrying AUTOSAR specification** The CGUs that you nest in a runnable subsystem can contain the following elements:

Model Element Carrying AUTOSAR Specification	Classic AUTOSAR Context
Block variables <sup>1)</sup>	Measurement and calibration <ul style="list-style-type: none"> <li>▪ CalPrmElement</li> <li>▪ PerInstanceCalPrm</li> <li>▪ SharedCalPrm</li> </ul> NvData communication PerInstanceMemory
Data store blocks <sup>2)</sup>	InterrunnableVariable DataElement is updated NvData communication Sender-receiver communication

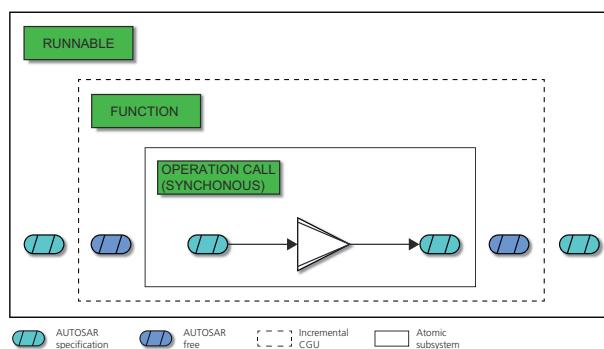
<sup>1)</sup> Cascades of incremental CGUs are possible.

<sup>2)</sup> To avoid differences between MIL and SIL simulation modes when modeling nested CGUs, use a DD Block object with its BlockType property set to TL\_DataStoreMemory instead.

**Special subsystems with AUTOSAR specification** The following special subsystems must not be modeled as incremental CGUs:

- [Operation call subsystem](#)
- [Operation result provider subsystem](#)
- [Operation call with runnable implementation subsystem](#)

You must not select the Enable incremental code generation checkbox on the Incremental page of the subsystems' Function blocks. Instead, model each of the subsystems as an atomic subsystem that is placed on the root level of the incremental CGU:

**Limitations**

You cannot directly model the following subsystems as [incremental CGUs](#):

- [Operation call subsystem](#)
- [Operation result provider subsystem](#)
- [Operation call with runnable implementation subsystem](#)

**Related topics****Basics**

Basics on Preparing SWCs for Measurement and Calibration.....	235
Comprehensive Modeling of RTE Code Pattern.....	374
Decomposing Models into Model Parts (Model Referencing) (TargetLink Customization and Optimization Guide)	
Decomposing Models Logically (Incremental Code Generation) (TargetLink Customization and Optimization Guide)	
Principles of Partitioning Models and Code (TargetLink Customization and Optimization Guide)	

**HowTos**

How to Model Checks of the Update Flag in Explicit Sender-Receiver Communication.....	130
How to Model Interrunnable Communication.....	175
How to Model Parameter Communication for Calibration.....	244
How to Model Per Instance Memories.....	214
How to Model Per Instance Parameters for Calibration.....	239
How to Model Shared Parameters for Calibration.....	241

**Examples**

Example of Modeling NvData Communication via Block Parameters.....	188
Example of Modeling NvData Communication via Data Store Memory Blocks.....	185
Example of Modeling Sender-Receiver Communication via Data Store Blocks.....	115

**References**

Bus Import Block (TargetLink Model Element Reference)	
Bus Outport Block (TargetLink Model Element Reference)	
Function Block (TargetLink Model Element Reference)	
InPort Block (TargetLink Model Element Reference)	
OutPort Block (TargetLink Model Element Reference)	



# Generating Classic AUTOSAR-Compliant Code

---

## Where to go from here

## Information in this section

Introduction.....	276
Compiler Abstraction.....	280
Mapping Variables and Functions to Memory Sections.....	282

# Introduction

## Where to go from here

## Information in this section

Basics on Classic AUTOSAR Code Generation.....	276
Basics on Generated Code.....	276
How to Generate Classic AUTOSAR-Compliant Code via the TargetLink Main Dialog.....	278

## Basics on Classic AUTOSAR Code Generation

### Introduction

You have to select the [Classic AUTOSAR](#) code generation mode for generating [Classic AUTOSAR](#) code.

### Code generation modes

TargetLink lets you generate code in the following code generation modes:

- Standard
- RTOS
- Classic AUTOSAR
- Adaptive AUTOSAR

Select AUTOSAR for generating AUTOSAR-compliant code. For basic information on AUTOSAR code, refer to [Basics on Generated Code](#) on page 276.

### Related topics

### References

[Files Related to Code Generation](#) ( [TargetLink File Reference](#))

## Basics on Generated Code

### Software component code

The code of a software component consists of runnables, i.e., functions. According to [Classic AUTOSAR](#), runnables have to communicate via [Classic AUTOSAR](#) interfaces (such as sender-receiver or client-server) or via interRunnable variables. You can specify the interface of a runnable via the AUTOSAR pages of

the InPort/Bus Import, OutPort/Bus Outport, and Data Store Memory blocks inside of a runnable subsystem.

You can enable/disable TargetLink to check runnable interfaces during code generation via the StrictRunnableInterfaceChecks Code Generator option. Disabling this option allows you to implement [Classic AUTOSAR](#) software components with non-AUTOSAR interfaces. This is useful if you want to implement a complex device driver component that communicates both via [Classic AUTOSAR](#) and non-AUTOSAR interfaces.

**Code optimization** TargetLink allows you to increase code efficiency by performing optimizations during production code generation. For details on AUTOSAR specific code optimizations, refer to [AUTOSAR-Related Code Generator Options](#) on page 387.

---

**RTE generator operation mode**

TargetLink generates the `Rte_<SWC>.h` RTE application header file of a software component in [compatibility mode](#).

You can use an RTE generator from any vendor (provided that the RTE generator supports the compatibility mode).

---

**RTE code**

Type definitions and variable declarations used for [Classic AUTOSAR](#) communication are subjects of the RTE code together with OS task bodies that organize software component function calls.

**Note**

The `Rte_<SWC>.c` file is generated by TargetLink and belongs to the stub RTE code. It is intended for simulation purposes only. To integrate the generated software component code on an ECU, you have to generate RTE code with an architecture tool such as dSPACE's SystemDesk. The `Rte_<SWC>.h` file is used for simulation purposes *and* for delivering SWCs as object code (OBJ). The application header file is generated in compatibility mode.

---

**Array passing scheme of RTE API functions**

TargetLink supports array passing only by pointing to the array element types.

**Note**

Make sure that your RTE generator uses the same array passing scheme as TargetLink.

---

**RTE API**

TargetLink supports RTE API functions as defined by [Classic AUTOSAR](#). For a list of RTE API functions and information on modeling their generation, refer to [Comprehensive Modeling of RTE Code Pattern](#) on page 374.

For details on the [Classic AUTOSAR](#) definition of RTE API functions, refer to the *Specification of RTE* document at <https://www.autosar.org/standards/classic-platform/>.

### Note

According to [Classic AUTOSAR](#), multiple instantiation means that there are several instances of the same software component on one ECU. If a software component supports multiple instantiation, the software component instance handle is passed as the first parameter of its API calls, otherwise no instance handles are passed.

### Related topics

#### Basics

[AUTOSAR-Related Code Generator Options](#)..... 387

#### References

[StrictRunnableInterfaceChecks](#)..... 392

## How to Generate Classic AUTOSAR-Compliant Code via the TargetLink Main Dialog

### Objective

To generate code that you can integrate into [Classic AUTOSAR](#) projects.

### Preconditions

- You specified the [Classic AUTOSAR](#) version in the TargetLink Data Dictionary at /Pool/Autosar/Config/AutosarVersion.
- If you want to generate code with compiler abstraction as defined by [Classic AUTOSAR](#), configure your project accordingly.

### Method

#### To generate Classic-AUTOSAR-compliant code via the TargetLink Main Dialog

- 1 Open the TargetLink Main Dialog.
- 2 From the Code generation mode list, select **Classic AUTOSAR**.
- 3 Select Generate Code/Build SIL/Build PIL as required.

### Result

You have generated [Classic AUTOSAR](#)-compliant code.

---

**Next steps**

- You can simulate the system.
  - You can export the generated code.
  - You can export AUTOSAR files that describe the software components for which you have generated code.
- 

**Related topics****Basics**

Basics on Classic AUTOSAR Code Generation.....	276
Basics on Compiler Abstraction According to Classic AUTOSAR.....	280

**References**

Files Related to Code Generation (	 TargetLink File Reference)
------------------------------------	--

# Compiler Abstraction

## Basics on Compiler Abstraction According to Classic AUTOSAR

### Compiler abstraction according to Classic AUTOSAR

To make source code platform-independent, [Classic AUTOSAR](#) has defined a set of macro definitions for source code elements such as functions, variables, and pointers. TargetLink can use these macros to define runnables, thus generating software component code that complies with the compiler abstraction definition of [Classic AUTOSAR](#).

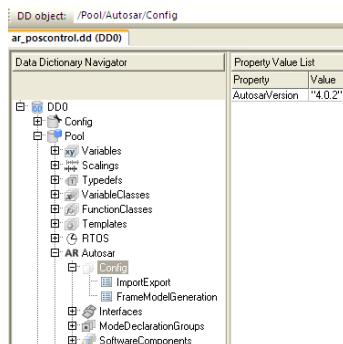
### Selecting compiler abstraction

#### Note

TargetLink generates compiler abstractions only for the declaration and definition of runnable functions in the generated production code.

If you want to use compiler abstractions as defined by [Classic AUTOSAR](#) for code generation, you have to do the following:

- Specify the [Classic AUTOSAR](#) version in the TargetLink Data Dictionary at /Pool/Autosar/Config/AutosarVersion.



- Select one of the following [function classes](#) at the runnables of the software component for which you generate code.

Function Class	Generated Code ...
<n.a.>	... includes compiler abstractions according to AUTOSAR Release 4.x.
RUNNABLE4	... does not include compiler abstractions.
other	... does not include compiler abstractions.

For instructions on modeling runnables, refer to [How to Model Runnables](#) on page 71.

**Runnable code with compiler abstraction macros** The following extract shows an example of runnable code with [Classic AUTOSAR](#) compiler abstraction macros:

```
FUNC(void, MySoftwareComponent_CODE) Run(sint16 ScalarIn,
    P2CONST(sint16, AUTOMATIC, RTE_APPL_DATA) ArrayIn[4],
    P2CONST(StructType, AUTOMATIC, RTE_APPL_DATA) StructIn,
    P2VAR(sint16, AUTOMATIC, RTE_APPL_DATA) ScalarOut,
    P2VAR(sint16, AUTOMATIC, RTE_APPL_DATA) ArrayOut[4],
    P2VAR(StructType, AUTOMATIC, RTE_APPL_DATA) StructOut)
```

The `Rte_Type.h` header file provides access to the compiler abstraction macro definitions. It is included in the software component header file (`<SWC/Runnable>.h`) if you generate code that supports compiler abstractions. Otherwise, the `Rte_Type.h` header file is included in the software component C file (`<SWC/Runnable>.c`).

**Memory class symbol** TargetLink automatically generates the memory class symbol of the FUNC macro of a runnable as specified by [Classic AUTOSAR](#).

You can specify a custom value of the `<MEM-CLASS-SYMBOL>` element of the `<MEMORY-SECTION>` element by specifying the `Settings.MemClassSymbol` property of a DD CodeDecoration object.

AUTOSAR 4.x example:

`MemClassSymbol = MyOwnMemClassSymbol`

- Extract of the `FuelsysController` ARXML file:

```
<MEMORY-SECTION>
  <SHORT-NAME>Code</SHORT-NAME>
  <DESC>
    <L-2 L="FOR-ALL">To be used for mapping code to application block, boot block, external flash etc.</L-2>
  </DESC>
  <EXECUTABLE-ENTITY-REFS>
    <EXECUTABLE-ENTITY-REF DEST="RUNNABLE">
      ENTITY"/>/Swcs/FuelsysController/FuelsysController/FuelsysControllerBehavior/FuelsysControllerInit</EXECUTABLE-ENTITY-REF>
    </EXECUTABLE-ENTITY-REFS>
    <MEM-CLASS-SYMBOL>MyOwnMemClassSymbol</MEM-CLASS-SYMBOL>
    <SW-ADDRMETHOD-REF DEST="SW-ADDR-METHOD">/FuelsysController/CODE</SW-ADDRMETHOD-REF>
    <SYMBOL>CODE</SYMBOL>
  </MEMORY-SECTION>
```

## Related topics

### Basics

[Basics on Memory Mapping in Classic AUTOSAR..... 282](#)

# Mapping Variables and Functions to Memory Sections

## Where to go from here

## Information in this section

Basics on Memory Mapping in Classic AUTOSAR..... 282

Example of Adapting the AR\_POSCONTROL Demo Model to use  
Memory Sections (Classic AUTOSAR)..... 287

## Basics on Memory Mapping in Classic AUTOSAR

### Memory mapping according to Classic AUTOSAR

According to [Classic AUTOSAR](#), [memory mapping](#) works as follows:

**High-level description via addressing method** A high-level description is provided in the ARXML file via `<SW-ADDR-METHOD>` elements that can be shared between software components. Because the description is high-level, it abstracts from implementation-specific characteristics such as data type, constness, or initialization.

For example, [Classic AUTOSAR](#) describes the address method named `VAR` to collect all the variables with global scope or static storage duration. Other examples of address methods for data are: `VAR_FAST`, `VAR_SLOW`, `VAR_SAVED_ZONE`, `CONST`, `CALIB` and `CONFIG_DATA`.

Each address method can be referenced by several different memory sections.

**Low-level description via memory sections** Memory sections are defined in the ARXML file within the `<MEMORY-SECTIONS>` element that belongs to the resource consumption of the software component implementation. They provide a low-level description, because they describe implementation-specific characteristics such as data type, constness, or initialization.

Examples of memory sections corresponding with the address method `VAR` are `VAR_INIT_BOOLEAN`, `VAR_INIT_8`, `VAR_INIT_16`, `VAR_INIT_32`, `VAR_INIT_UNSPECIFIED` and `VAR_NO_INIT_BOOLEAN`.

### Memory mapping in TargetLink

#### Note

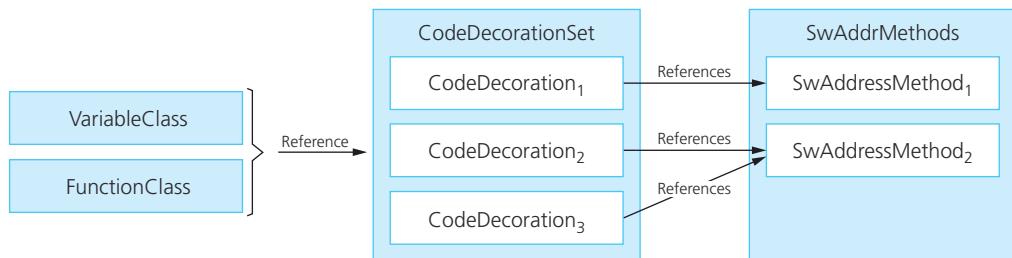
According to [Classic AUTOSAR](#), it is best practice not to use variables with local scope and static storage duration. TargetLink follows this advice and supports the use of memory sections only for variables of global scope. You can instruct TargetLink to suppress the generation of variables with local scope and static memory duration via the `AvoidStaticLocalScope` Code Generator option.

The following table shows the mapping between [Classic AUTOSAR](#) and TargetLink entities:

Description	AUTOSAR	TargetLink
High-Level	<SW-ADDR-METHOD>	DD SwAddrMethod object
Low-Level	<MEMORY-SECTION>	DD CodeDecoration object

**Providing the high-level description** Usually, the high-level descriptions are specified in a system-level design tool such as SystemDesk. During container exchange or ARXML import, they are imported to the Data Dictionary as SwAddrMethod objects in the /Pool/Autosar/SwAddrMethods/ subtree. Each SwAddrMethod object can be referenced at one or more DD CodeDecoration objects via the SwAddrMethodRef property that belongs to the AutosarExportInfo child object.

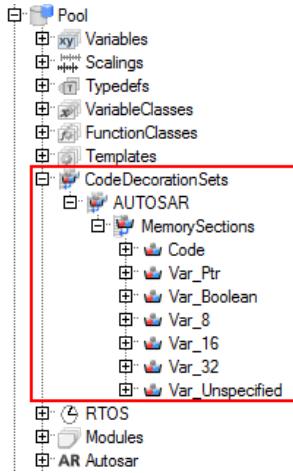
**Providing the low-level description** You provide the low-level descriptions via DD CodeDecoration objects that belong to a DD CodeDecorationSet object. This object is referenced at the DD FunctionClass/VariableClass object, which is associated with the functions/variables belonging to the memory section.



**Mapping SWCs to CodeDecorationSet Objects** To instruct TargetLink to export the <Memory-Section> element into the resource consumption of the correct software component implementation, you must reference the CodeDecorationSet object at the MemorySectionsRef property of the SoftwareComponent object.

#### Specifying the code decoration

**Predefined CodeDecorationSet for Classic AUTOSAR** Data Dictionaries based on the dsdd\_master\_autosar4.dd [System] template contain a predefined CodeDecorationSet called **MemorySections** that you can use to specify memory sections as defined by [Classic AUTOSAR](#). This object contains several CodeDecoration objects to generate the data section VAR and the code section CODE:



**Specifying the Filter object of a CodeDecoration object** Via the properties contained in the Filter object of a CodeDecoration object, you can specify to which code elements the decoration is to be applied.

The following screenshot shows the filter of a decoration for the VAR\_16 memory section as example:

Property	Value
ExtendedFor	Variables
ScopeSpec	APPLY_TO_EXTERN_GLOBAL_APPLY_TO_GLOBAL_APPLY_TO_STATIC_GLOBAL
WidthSpec	APPLY_TO_16BIT

**Specifying the Settings object of a CodeDecoration object** Via the properties contained in the Settings object of a CodeDecoration object, you can specify how the decoration appears in the production code.

The following screenshot shows the decoration settings for the VAR\_16 memory section as an example:

Property	Value
SectionName	"VAR_16"
DeclarationStatements	<value not set>
ImplementedIn	<value not set>
MemClassSymbol	<value not set>
Prefix	<value not set>

The most important property is the **SectionName** property that lets you specify the <NAME> of the memory allocation keyword as defined by [Classic AUTOSAR](#).

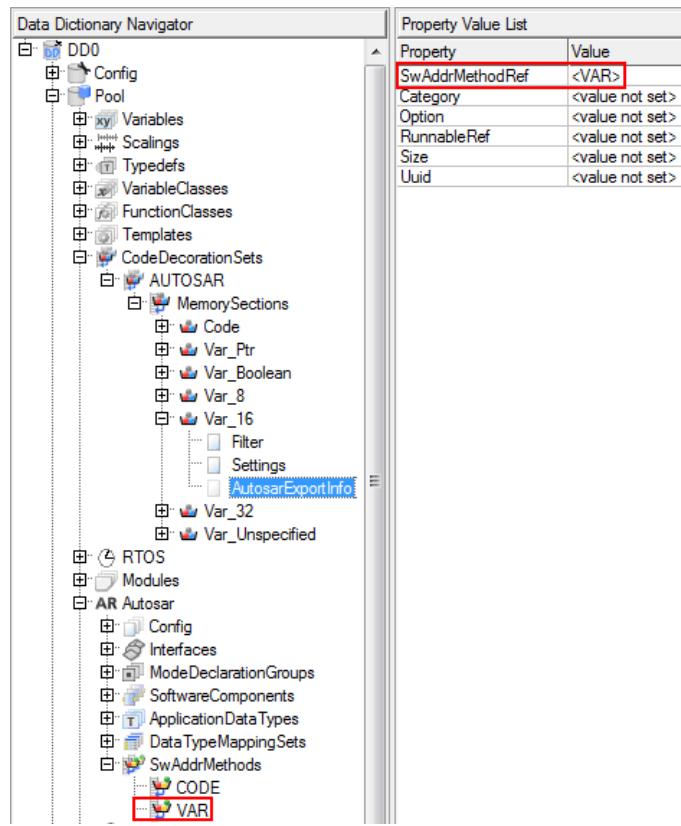
#### Note

TargetLink creates memory allocation keywords for code elements that fulfill the following conditions:

- The **SectionName** property of the applied DD **CodeDecoration** object is specified.
- The code elements belong to the SWC's production code.
- The code elements are used only by one SWC.

**Specifying the AutosarExportInfo object of a code decoration** Via the properties contained in the **AutosarExportInfo** child object of a **CodeDecoration** object, you can specify how the code decoration is used in AUTOSAR export.

The following screenshot shows the decoration settings for the **VAR\_16** memory section as an example:



For full reference information on all the object's properties, refer to [CodeDecoration](#).

**Memory mapping in production code**

In production code, the allocation of code elements to different memory sections is controlled by memory allocation keywords.

**Naming schema of the memory allocation keyword** The syntax of a typical memory allocation keyword for a data section looks like this:

```
<PREFIX>_START_SEC_<NAME>
#include "<MEM_MAP_FILE>.h"
...
<PREFIX>_STOP_SEC_<NAME>
#include "<MEM_MAP_FILE>.h"
```

TargetLink replaces the placeholders as shown in the following table:

Placeholder	Replacement
<PREFIX>	<SWCShortName>
<NAME>	The value of the CodeDecoration.Settings.SectionName property.
<MEM_MAP_FILE>	<SWCShortName>_MemMap

For code other than runnable functions, you can also set the desired values via the SettingsImplementedIn or SettingsPrefix properties of CodeDecoration objects.

**Example of a memory allocation keyword** A typical memory allocation keyword for [Classic AUTOSAR](#) looks like this in production code:

```
/* start of memory section 'VAR_32' */
#define FuelsysCombi_START_SEC_VAR_32
#include "FuelsysCombi_MemMap.h"
...
/* end of memory section 'VAR_32' */
#define FuelsysCombi_STOP_SEC_VAR_32
#include "FuelsysCombi_MemMap.h"
```

**Simulating production code**

TargetLink generates an empty <MemMap\_file>.h file as stub code for simulation purposes.

**Importing and exporting**

TargetLink imports and exports the following ARXML elements:

Import	Export
<SW-ADDR-METHOD>	<SW-ADDR-METHOD>
	<MEMORY-SECTION>

**Related topics****Basics**

AR_MEMORY_MAPPING (TargetLink Demo Models)	280
Basics on Compiler Abstraction According to Classic AUTOSAR.....	280
Decorating Production Code (TargetLink Customization and Optimization Guide)	
Exchanging AUTOSAR Files (TargetLink Interoperation and Exchange Guide)	
Interoperating with SystemDesk via SWC Containers (TargetLink Interoperation and Exchange Guide)	

**Examples**

Example of Adapting the AR_POSCONTROL Demo Model to use Memory Sections (Classic AUTOSAR).....	287
--	-----

**References**

AvoidStaticLocalScope (TargetLink Model Element Reference)	
--	--

## Example of Adapting the AR\_POSCONTROL Demo Model to use Memory Sections (Classic AUTOSAR)

**Introduction**

This example shows how to adapt the AR\_POSCONTROL demo model to use the [Classic AUTOSAR](#) memory sections `Code`, `Var_<Width>` and `Var_Fast_<Width>`.

**Restrictions**

The AR\_POSCONTROL demonstrates a software component that is prepared for multiple instantiation. In this context, per instance memories can conflict with memory mapping. To make this example work, you must set the `SupportsMultipleInstantiation` property of the DD `SoftwareComponent` object called `Controller` to `off`.

For details, refer to [Basics on Preparing SWCs for Multiple Instantiation](#) on page 227

**Precondition**

The following preconditions must be fulfilled:

- You opened the AR\_POSCONTROL demo model.

**Workflow**

This example consists of the following parts:

1. Preparing the Data Dictionary for specifying memory sections as defined by Classic AUTOSAR, refer to Part 1.

2. Specifying the memory sections for global variables and runnable code, refer to Part 2.
3. Specifying the **VAR\_FAST** memory section, refer to Part 3.

**Part 1****Preparing the Data Dictionary for specifying memory sections as defined by Classic AUTOSAR**

- 1 In the Data Dictionary, duplicate the `/Pool/CodeDecorationSets/AUTOSAR/MemorySections` object (**CTRL + D**).  
Rename the objects as **Default** and **VAR\_FAST**, respectively.
- 2 Locate the `/Pool/Autosar` object and add its `SwAddrMethods` child object.

**Tip**

Usually, the Classic AUTOSAR `SwAddressMethods` objects are specified in a system-level design tool such as SystemDesk and added to the Data Dictionary via import of AUTOSAR files or container exchange.

- 3 To the DD `SwAddrMethods` object, add three DD `SwAddrMethod` objects and name them **CODE**, **VAR** and **VAR\_FAST**.
- 4 Set their properties as shown in the following table:

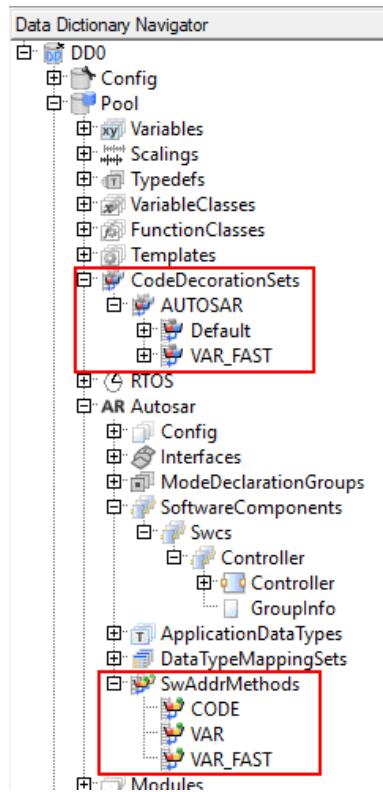
Property	<b>CODE</b>	<b>VAR</b>	<b>VAR_FAST</b>
SectionType	Code	Var	VarFast
MemoryAllocationKeywordPolicy	AddrMethodShortName	AddrMethodShortNameAndAlignment	

- 5 Locate the DD `SoftwareComponent` object named **Controller** and add two DD `MemorySectionsRef` properties to its property value list via the **Create Reference to MemorySections** context menu command.
- 6 Specify these properties as shown in the following table:

Property	Value
MemorySectionsRef	AUTOSAR/Default
MemorySectionsRef#2	AUTOSAR/VAR_FAST

**Interim result**

You prepared the Data Dictionary of the AR\_POSCONTROL demo model for specifying memory sections for the address methods **VAR**, **CODE**, and **VAR\_FAST** as defined by Classic AUTOSAR. Your Data Dictionary looks like this:



Next you specify the memory sections for all the global variables (`Var_<Width>`) and for the code of the runnables (`Code`) belonging to the software component named `Controller`.

## Part 2

### Specifying the memory sections for global variables and runnable code

- Locate the DD /Pool/Autosar/Config object and set the following properties:

Property	Value
DefaultDecorationSetRef	Default
AlwaysUseDefaultDecorationSet	on

- Expand the /Pool/CodeDecorationSets/AUTOSAR/Default object. From the context menu of one of its CodeDecoration child objects, select Create ExportInfo to create a DD AutosarExportInfo object as child object.
- Set its SwAddrMethodRef property to VAR.
- Copy the DD AutosarExportInfo object to all the sibling DD CodeDecoration objects.
- Select the DD CodeDecorationSets/AUTOSAR/Default/Code/AutosarExportInfo object and use its Create Reference to Runnable context menu command to add three RunnableRef options to its Property Value List.

- 6 Reference all the DD Runnable objects that belong to the DD SoftwareComponent object named **Controller** via the RunnableRef properties.
- 7 Change the value of the SwAddrMethodRef property to **CODE**.

The screenshot shows the Data Dictionary Navigator interface. On the left, the tree view under 'CodeDecorationSets' shows 'AUTOSAR' expanded, with 'Code' and 'Default' further down. 'Code' has 'Filter' and 'Settings' as children. 'Default' has 'AutosarExportInfo' as a child. Under 'AutosarExportInfo', there are three entries: 'Var\_Ptr', 'Var\_Boolean', and 'Var\_FAST'. On the right, a 'Property Value List' table is displayed with two rows:

Property	Value
SwAddrMethodRef	<CODE>
RunnableRef	<Swcs/Controller/Controller/Runnables/PosController>
RunnableRef(#2)	<Swcs/Controller/Controller/Runnables/PosLinearization>
RunnableRef(#3)	<Swcs/Controller/Controller/Runnables/InitFunction>

### Interim result

By setting the DD CodeDecorationSet object named **Default** as default code decoration set and setting the **AlwaysUseDefaultDecorationSet** property to **on**, you instructed TargetLink to use this DD CodeDecorationSet object for all the [explicit objects](#) and [implicit objects](#) (variables and functions) generated by TargetLink.

By referencing the **VAR** and **CODE** address methods at the DD CodeDecoration objects that belong to the DD CodeDecorationSet named **Default**, you established the connection between the high-level description of the address method and the low-level description of the memory section.

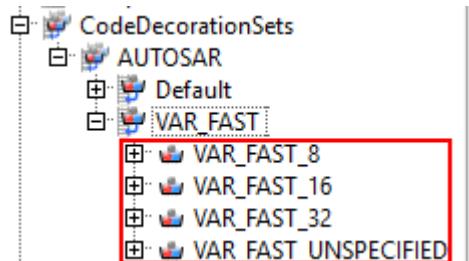
By referencing the runnables that belong to the software component named **Controller** via the three RunnableRef properties of the DD CodeDecoration object named **Code**, you instructed TargetLink to generate **<EXECUTABLE-ENTITY-REFS>** in the **<MEMORY-SECTION>** element of the ARXML file.

Next you specify a memory section **VAR\_FAST\_<Width>** to collect all the global variables that belong to the software component named **Controller** that are accessed frequently.

### Part 3

#### Specifying the VAR\_FAST memory section

- 1 Expand the DD CodeDecorationSet object named **VAR\_FAST** and delete all but the following DD CodeDecoration objects:



- 2 From the context menu of one of the remaining DD CodeDecoration objects, select **Create ExportInfo** to create a DD AutosarExportInfo object as a child object.
- 3 Set its **SwAddrMethodRef** property to **VAR\_FAST**.

- 4 Copy the DD AutosarExportInfo object to all the sibling CodeDecoration objects.
- 5 Rename the DD CodeDecoration objects and adjust the value of the SectionName property contained in the Settings object of each CodeDecoration object as shown in the following table:

CodeDecoration Object	New Name	Property Value of Settings.SectionName	Description
Var_8	VAR_FAST_8	VAR_FAST_8	Used for frequently accessed global variables which have to be aligned to 8 bit.
Var_16	VAR_FAST_16	VAR_FAST_16	Used for frequently accessed global variables which have to be aligned to 16 bit.
Var_32	VAR_FAST_32	VAR_FAST_32	Used for frequently accessed global variables which have to be aligned to 32 bit.
Var_Unspecified	VAR_FAST_Unspecified	VAR_FAST_UNSPECIFIED	Used for frequently accessed global variables when size (alignment) does not fit the criteria of 8, 16, or 32 bit.

- 6 Create a DD VariableClass object, rename it as VAR\_FAST, and set the following properties:

Property	Value
CodeDecorationSetRef	AUTOSAR/VAR_FAST
Optimization	MOVABLE

- 7 In the model, open the [Runnable subsystem](#) called Controller\_Runnable and reference the DD VariableClass object called VAR\_FAST at the Product blocks called Multiply\_Kp and Multiply\_Ki.
- 8 Generate code and export the ARXML files of the software component named Controller.

## Result

You adjusted the AR\_POSCONTROL demo model to use Classic AUTOSAR memory sections.

The production code now contains memory allocation keywords:

```
/* start of memory section 'VAR_FAST_16' */
#define Controller_START_SEC_VAR_FAST_16
#include "Controller_MemMap.h"
/*********************************************************************
 * VAR_FAST: Variable class | Width: 16
 *********************************************************************/
sint16 S12_Multiply_Ki; /* LSB: 2^-9 OFF: 0 MIN/MAX: -64 .. 63.998046875 */
sint16 S12_Multiply_Kp; /* LSB: 2^-9 OFF: 0 MIN/MAX: -64 .. 63.998046875 */

/* end of memory section 'VAR_FAST_16' */
#define Controller_STOP_SEC_VAR_FAST_16
#include "Controller_MemMap.h"

...
```

```

/* start of memory section 'CODE' */
#define Controller_START_SEC_CODE
#include "Controller_MemMap.h"

FUNC(void, Controller_CODE) controllerRunnable(CONSTP2CONST(Rte_CDS_Controller,
    AUTOMATIC, RTE_CONST) instance,
    Rte_ActivatingEvent_controllerRunnable activation)
{
    ...
}

/* end of memory section 'CODE' */
#define Controller_STOP_SEC_CODE
#include "Controller_MemMap.h"

```

The AUTOSAR files contain <MEMORY-SECTION> elements such as the following:

```

<MEMORY-SECTIONS>
<MEMORY-SECTION UUID="b25de62c-9f35-11ea-9823-94e6f79e88be">
    <SHORT-NAME>VAR_FAST_16</SHORT-NAME>
    <DESC>
        <L-2 L="FOR-ALL">Used for frequently accessed global variables which have to be aligned to 16 bit.</L-2>
    </DESC>
    <ALIGNMENT>16</ALIGNMENT>
    <SW-ADDRMETHOD-REF DEST="SW-ADDR-METHOD">/Controller/VAR_FAST</SW-ADDRMETHOD-REF>
    <SYMBOL>VAR_FAST_16</SYMBOL>
</MEMORY-SECTION>
<MEMORY-SECTION UUID="b25de62b-9f35-11ea-9823-94e6f79e88be">
    <SHORT-NAME>Code</SHORT-NAME>
    <DESC>
        <L-2 L="FOR-ALL">To be used for mapping code to application block, boot block, external flash etc.</L-2>
    </DESC>
    <EXECUTABLE-ENTITY-REFS>
        <EXECUTABLE-ENTITY-REF DEST="RUNNABLE-ENTITY">
            /Swcs/Controller/Controller/IB_Controller/PosController
        </EXECUTABLE-ENTITY-REF>
        <EXECUTABLE-ENTITY-REF DEST="RUNNABLE-ENTITY">
            /Swcs/Controller/Controller/IB_Controller/PosLinearization
        </EXECUTABLE-ENTITY-REF>
        <EXECUTABLE-ENTITY-REF DEST="RUNNABLE-ENTITY">
            /Swcs/Controller/Controller/IB_Controller/InitFunction
        </EXECUTABLE-ENTITY-REF>
    </EXECUTABLE-ENTITY-REFS>
    <SW-ADDRMETHOD-REF DEST="SW-ADDR-METHOD">/Controller/CODE</SW-ADDRMETHOD-REF>
    <SYMBOL>CODE</SYMBOL>
</MEMORY-SECTION>
<MEMORY-SECTION UUID="b25de629-9f35-11ea-9823-94e6f79e88be">
    <SHORT-NAME>Var_8</SHORT-NAME>
    <DESC>
        <L-2 L="FOR-ALL">Used for variables which have to be aligned to 8 bit.</L-2>
    </DESC>
    <ALIGNMENT>8</ALIGNMENT>
    <SW-ADDRMETHOD-REF DEST="SW-ADDR-METHOD">/Controller/VAR</SW-ADDRMETHOD-REF>
    <SYMBOL>VAR_8</SYMBOL>
</MEMORY-SECTION>
</MEMORY-SECTIONS>

```

---

## Related topics

### Basics

[AR\\_POSCONTROL](#) ( TargetLink Demo Models)  
[Basics on Code Decorations](#) ( TargetLink Customization and Optimization Guide)  
[Basics on Memory Mapping in Classic AUTOSAR](#)..... 282

### HowTos

[How to Specify Code Decorations in the Data Dictionary](#) ( TargetLink Customization and Optimization Guide)



# Simulating SWCs

## Introduction

You can use TargetLink to simulate and analyze [Classic AUTOSAR-compliant](#) models in a similar way to standard models. There are some points to note when you simulate [Classic AUTOSAR-compliant](#) models.

## Where to go from here

### Information in this section

Basics on Simulating Classic-AUTOSAR-Compliant SWCs.....	296
How to Simulate the RTE Status.....	299
How to Simulate Acknowledgment Notifications.....	301
How to Simulate Signal Invalidation.....	303
How to Model and Simulate Transformer Error Logic in Sender-Receiver Communication.....	305
How to Simulate Operation Calls in Asynchronous Client-Server Communication.....	309

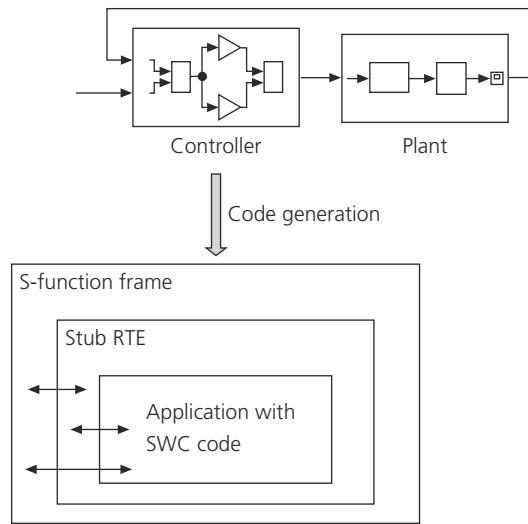
## Basics on Simulating Classic-AUTOSAR-Compliant SWCs

### Simulating models

You can simulate an [Classic AUTOSAR](#)-compliant model to verify its behavior in the MIL/SIL/PIL simulation modes.

### Simulation concept

The following illustration shows details of the simulation concept of an [Classic AUTOSAR](#)-compliant application with SWC code:



In general, the simulation concept is the same as with standard models. For basic information on simulating standard models, refer to [Simulating Models and Analyzing Simulation Results](#) ([TargetLink Preparation and Simulation Guide](#)).

One exception to this is the additional *stub RTE*, which is generated to simulate the behavior of the RTE. The *stub RTE* maps the RTE macro function calls in the *application with SWC code* to global variables and also generates access functions to global variables. The *stub RTE* is replaced by RTE code when you integrate the generated SWC code on an ECU. You can generate RTE code with AUTOSAR-compliant system-level design tools such as dSPACE SystemDesk. For basic information on developing ECU software according to AUTOSAR, refer to [Basics on Tools for Developing ECU Software as Described by Classic AUTOSAR](#) on page 18.

### Simulation modes

The TargetLink MIL, SIL, and PIL simulation modes are supported for simulating the behavior of an [Classic AUTOSAR](#)-compliant model. Refer to [Verifying Model and Code in Different Simulation Modes](#) ([TargetLink Preparation and Simulation Guide](#)).

**Data logging**

TargetLink's data logging and analysis features are supported for AUTOSAR-compliant models. Refer to [Basics on Logging](#) ( [TargetLink Preparation and Simulation Guide](#)).

You can log data at TargetLink InPort, OutPort, Bus Import, and Bus Outport blocks. TargetLink supports logging of data elements, interruptable variables, operation arguments, and application errors. You can log data that represents an array element-wise and data that represents a structure component-wise.

**Runnable triggering during simulation**

In simulations in the MIL/SIL/PIL simulation modes, runnable subsystems are executed according to the data flow or, if specified, as function-call-triggered subsystems. For basic information on the control flow in TargetLink models, refer to [Controlling Subsystem Execution](#) ( [TargetLink Customization and Optimization Guide](#)).

If you want to model the triggering of runnables by RTE events during the simulation, you can use function calls. The [AR\\_POSCONTROL](#) ( [TargetLink Demo Models](#)) demo model shows an example of triggering runnables by function calls from a Stateflow diagram.

**Simulating the impact of the RTE on software components**

**Simulating RTE statuses** RTE API functions of sender-receiver communication can return the RTE status of a communication operation. However, the RTE is not available in the Simulink/TargetLink environment. You therefore have to provide values for the RTE status, e.g., via a signal generator for simulation purposes. To simulate the RTE status, you have to use the ReceiverComSpec and SenderComSpec blocks of the TargetLink Classic AUTOSAR Block Library. For instructions on simulating the RTE status, refer to [How to Simulate the RTE Status](#) on page 299.

**Simulating acknowledgment notifications and signal invalidation**

[Classic AUTOSAR](#) lets you specify how communication is handled by the RTE in communication specifications. These elements can be defined for each data element or operation and contain specifications such as initial values, queue lengths, acknowledgment notifications, and signal invalidation settings.

In TargetLink, you can model acknowledgment notifications and signal invalidation. However, no RTE is present during simulation. If you want to simulate the effect of acknowledgment notifications and signal invalidation, you have to use the SenderComSpec block of the TargetLink Classic AUTOSAR Block Library. For instructions on simulating acknowledgment notifications and signal invalidation, refer to [How to Simulate Acknowledgment Notifications](#) on page 301.

**Simulating transformer errors in sender-receiver communication** In TargetLink, RTE API functions of sender-receiver communication can return transformer errors. However, the implementation of the transformer (chain) is not available in the Simulink model. You therefore have to provide values for the transformer errors in the model that are to be simulated.

To simulate transformer errors, you have to use the ReceiverComSpec and SenderComSpec blocks of the TargetLink Classic AUTOSAR Block Library.

You also have to prepare your model for simulation by using the `t1TransformerError('PrepareModelForSimulation', propertyName, PropertyValue, ...)` API function.

Refer to [How to Model and Simulate Transformer Error Logic in Sender-Receiver Communication](#) on page 305.

**Simulating transformer errors in client-server communication** You can model transformer error logic in client-server communication when using the following model elements:

- [Operation subsystem](#)
- [Runnable subsystem](#) used to model runnables that are triggered by one of the following [RTE events](#):
  - [Operation invoked event](#)
  - [TransformerHardErrorEvent](#)

Here, the transformer error is modeled via a port block and can be simulated. Refer to [Details on Data Transformation for Client-Server Communication](#) on page 209.

## Related topics

### Basics

Basics on Logging ( <a href="#">TargetLink Preparation and Simulation Guide</a> )	
Basics on Tools for Developing ECU Software as Described by Classic AUTOSAR.....	18
Controlling Subsystem Execution ( <a href="#">TargetLink Customization and Optimization Guide</a> )	
Simulating Models and Analyzing Simulation Results ( <a href="#">TargetLink Preparation and Simulation Guide</a> )	
Verifying Model and Code in Different Simulation Modes ( <a href="#">TargetLink Preparation and Simulation Guide</a> )	

### HowTos

How to Model and Simulate Transformer Error Logic in Sender-Receiver Communication.....	305
How to Simulate Acknowledgment Notifications.....	301
How to Simulate the RTE Status.....	299

### References

<code>t1TransformerError('PrepareModelForSimulation', propertyName, PropertyValue, ...)</code>	
( <a href="#">TargetLink API Reference</a> )	

# How to Simulate the RTE Status

## Objective

To simulate the status of sender-receiver communication.

## Basics on the RTE status

 [Classic AUTOSAR](#) has defined a standard return type for API functions of the RTE. Communication operations can return the value `0`, i.e., `RTE_E_OK` if the operation was completed, or predefined error codes.

## Restriction

Modeling via data store blocks is not supported.

## Preconditions

The following preconditions must be fulfilled:

- You modeled sender-receiver communication using ComSpec blocks and created a communication specification for the data element. For instructions, refer to [How to Model Invalidation, Acknowledgment Notifications and Access to the RTE Status](#) on page 122.

### Note

You can simulate the RTE status of sender-receiver communication for the sender and the receiver. The following instructions show how you can simulate the RTE status of the sender. You can simulate the RTE status of the receiver in a similar way.

## Method

### To simulate the RTE status

- 1 In the MATLAB Command Window, create a `Simulink.Signal` object that acts as a global variable for the RTE status. The following listing shows you how to create an appropriate `Simulink.Signal` object:

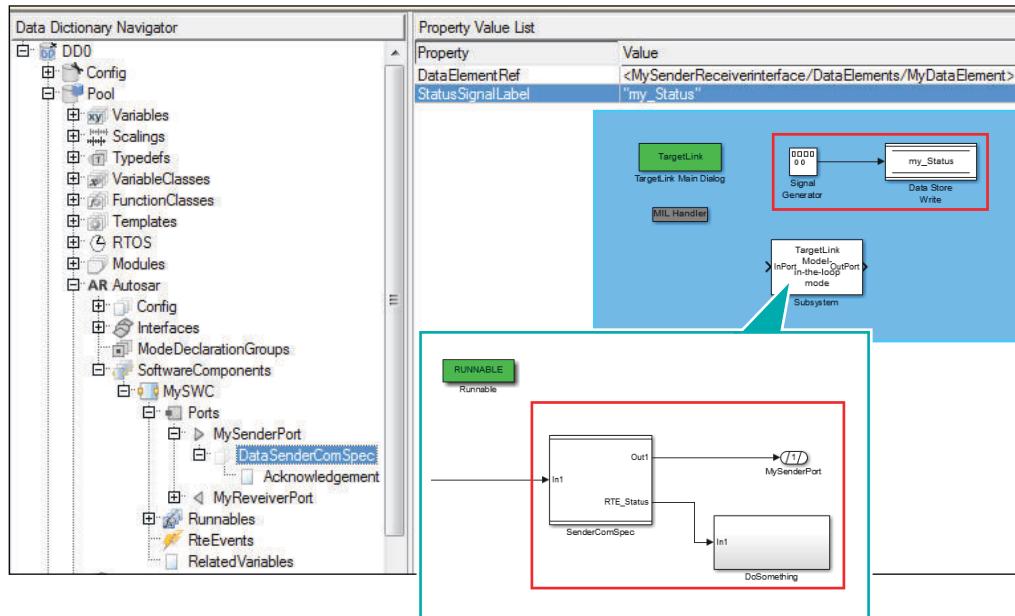
```
my_Status = Simulink.Signal;
my_Status.Dimensions = 1;
my_Status.DataType = 'double';
my_Status.SamplingMode = 'Sample based';
my_Status.Complexity = 'real';
```

### Tip

You should include the listing above in a script for opening the model in order to create the `Simulink.Signal` object whenever the model is used.

- 2 In the Simulink/TargetLink model, open the `SenderComSpec` block you want to simulate the RTE status for.
- 3 On the `ComSpec` page, select the `Add RTE status port` checkbox to connect the RTE status in the model.

- 4 Click the ComSpec edit field to access the assigned ComSpec object in the TargetLink Data Dictionary Manager.
- 5 In the Data Dictionary Navigator, enter the name of the Simulink.Signal object for the StatusSignalLabel property of the ComSpec object. TargetLink replaces the Constant block that is connected with the RTE\_Status port under the SenderComSpec block mask with a Data Store Read block. The Data Store Read block is configured to access the Simulink.Signal object that you created in the first step. To view the Data Store Read block, press Ctrl+D to update your model and select Look Under Mask from the context menu of the SenderComSpec block.



- 6 Outside the TargetLink subsystem add a Data Store Write block from the Simulink Block Library via Simulink – Signal Routing – Data Store Write.
- 7 Double-click the Data Store Write block to open its block dialog and enter the Simulink.Signal object's name in the Data store name edit field.

**Result**

You connected the RTE status signal of the SenderComSpec to a Simulink signal to simulate the RTE status.

**Next steps**

Connect the RTE\_Status port of the SenderComSpec block to process the status and the Data Store Write block with a signal source, and run simulations.

**Related topics****Basics**

Basics on Simulating Classic-AUTOSAR-Compliant SWCs..... 296

**HowTos**

How to Model Invalidation, Acknowledgment Notifications and Access to the RTE Status..... 122

**References**

ReceiverComSpec Block ( TargetLink Model Element Reference)

SenderComSpec Block ( TargetLink Model Element Reference)

## How to Simulate Acknowledgment Notifications

**Objective**

To simulate acknowledgment notifications of the sender in sender-receiver communication.

**Basics on acknowledgment notifications**

You can configure explicit sender-receiver communication to provide an acknowledgment notification of the send process to the sender. [Classic AUTOSAR](#) defined the `Rte_Feedback` API function for this purpose.

**Restrictions**

Simulating acknowledgment notifications of the sender in sender-receiver communication is restricted to [explicit communication](#).

Modeling via data store blocks is not supported.

**Preconditions**

The following prerequisites have to be fulfilled:

- You have modeled sender-receiver communication using ComSpec blocks and created a communication specification for the data element. For instructions, refer to [How to Model Invalidation, Acknowledgment Notifications and Access to the RTE Status](#) on page 122.

Method	To simulate acknowledgment notifications
<p><b>1</b> In the MATLAB Command Window, create a Simulink.Signal object. The following listing shows you how to create an appropriate Simulink.Signal object:</p> <pre data-bbox="600 428 1152 572"><code>my_Notification = Simulink.Signal; my_Notification.Dimensions = 1; my_Notification.DataType = 'double'; my_Notification.SamplingMode = 'Sample based'; my_Notification.Complexity = 'real';</code></pre> <p><b>Tip</b></p> <p>You should include the listing above in a script for opening the model to create the Simulink.Signal object whenever the model is used.</p> <p><b>2</b> In the Simulink/TargetLink model, open the SenderComSpec block you want to simulate acknowledgment notifications for.</p> <p><b>3</b> On the ComSpec page, select the Add feedback port checkbox to connect the acknowledgment notification in the model.</p> <p><b>4</b> Click the ComSpec edit field to access the assigned ComSpec object in the TargetLink Data Dictionary Manager.</p> <p><b>Note</b></p> <p>You must have created an Acknowledgment object for the DataSenderComSpec object in the Data Dictionary.</p> <p><b>5</b> In the Data Dictionary, enter the name of the Simulink.Signal object that you created in the first step for the FeedbackSignalLabel property of the DataSenderComSpec object.</p> <p>TargetLink replaces the Constant block that is connected with the Feedback port under the SenderComSpec block mask with a Data Store Read block. The Data Store Read block is configured to access the Simulink.Signal object that you created in the first step. To view the Data Store Read block, press Ctrl+D to update your model and select Look Under Mask from the context menu of the SenderComSpec block.</p> <p><b>6</b> Outside the TargetLink subsystem, i.e., where the TargetLink Main Dialog block resides, and add a Data Store Write block from the Simulink Block Library via Simulink – Signal Routing – Data Store Write.</p> <p><b>7</b> Double-click the Data Store Write block to open its block dialog and enter the Simulink.Signal object name in the Data store name edit field.</p>	

**Result** You connected the RTE feedback signal with a `Simulink.Signal` object to simulate acknowledgment notifications.

**Tip**

[How to Simulate the RTE Status](#) on page 299 also illustrates the simulation of acknowledgment notifications.

**Next steps** Connect the Feedback port of the `SenderComSpec` block to process the feedback, connect the `Data Store Write` block with a signal source, and run simulations.

**Related topics****Basics**

[Basics on Simulating Classic-AUTOSAR-Compliant SWCs](#)..... 296

**HowTos**

[How to Model Invalidation, Acknowledgment Notifications and Access to the RTE Status](#)..... 122

**References**

[ReceiverComSpec Block](#) ( TargetLink Model Element Reference)

[SenderComSpec Block](#) ( TargetLink Model Element Reference)

## How to Simulate Signal Invalidation

**Objective**

To simulate signal invalidation of the sender in sender-receiver communication with [data semantics](#).

**Preconditions**

The following prerequisites have to be fulfilled:

- You modeled unqueued sender-receiver communication using ComSpec blocks and created a communication specification for the data element. For instructions, refer to [How to Model Invalidation, Acknowledgment Notifications and Access to the RTE Status](#) on page 122.

---

Method	To simulate signal invalidation
	<ol style="list-style-type: none"><li>1 In the Simulink/TargetLink model, open the SenderComSpec block you want to simulate signal invalidation for.</li><li>2 On the ComSpec page, select the Add invalidate port checkbox to connect the invalidate condition to the block.</li><li>3 Click the ComSpec group box to access the assigned DD DataSenderComSpec object in the TargetLink Data Dictionary Manager.</li></ol>

**Note**

The CanInvalidate property of the DD DataSenderComSpec object in the Data Dictionary must be set to on. You also have to provide an invalid value at the DD Typedef object, that is referenced at the corresponding DD DataElement object.

- 4 In the model, connect a condition signal to the Invalidate port of the SenderComSpec block.

During simulation, the output signal of the SenderComSpec block is invalidated depending on the condition signal. If the signal is invalidated, the output of the block is the initial value of the assigned data sender communication specification. If you added an RTE status port to the SenderComSpec block, [Classic AUTOSAR](#) error code 129: 'Signallnvalidation' is sent for invalidation via the RTE status.

**Note**

TargetLink does not support simulating signal invalidation of struct-based data elements in MIL simulation. The data element is always sent.

**Result**

You added an Invalidate Port to a SenderComSpec block to simulate signal invalidation.

**Tip**

[How to Simulate the RTE Status](#) on page 299 also illustrates the simulation of signal invalidation.

**Related topics****Basics**

Basics on Signal Invalidation.....	119
Basics on Simulating Classic-AUTOSAR-Compliant SWCs.....	296

**References**

ReceiverComSpec Block ( TargetLink Model Element Reference)
SenderComSpec Block ( TargetLink Model Element Reference)

## How to Model and Simulate Transformer Error Logic in Sender-Receiver Communication

**Simulating transformer errors**

In TargetLink, RTE API functions of sender-receiver communication can return transformer errors. However, the implementation of the transformer (chain) is not available in the Simulink model. You therefore have to provide values for the transformer errors in the model that are to be simulated.

To simulate transformer errors, you have to use the `ReceiverComSpec` and `SenderComSpec` blocks of the TargetLink Classic AUTOSAR Block Library.

You also have to prepare your model for simulation by using the `t1TransformerError('PrepareModelForSimulation', propertyName, PropertyValue, ...)` API function.

**Restriction****Note**

Modeling the logic of transformer errors is restricted to modeling via port blocks.

Generating code for the logic of transformer errors is restricted to sender-receiver communication.

Importing and exporting the `ErrorHandling` property is also possible for client-server communication.

**Preconditions**

The following preconditions must be fulfilled:

- You modeled an SWC, such as `MySwC`.
- You modeled a Runnable, such as `MyRunnable`.
- You modeled sender-receiver communication.

Method	To model and simulate transformer error logic in sender-receiver communication	
Property	Value	Description
DataElementRef	Reference to a DD DataElement object.	The data element you want to model a communication specification for. Make sure that the data elements belongs to DD SenderReceiverInterface object that is referenced at the <Port> object. <sup>1)</sup>
TransformerErrorSignalLabel	The name of a Simulink.Signal object.	The Simulink.Signal object whose current value is propagated to <b>errorCode</b> and <b>transformerClass</b> during simulation. Its current value is modified by means of the Data Store Memory block created via the <b>t1TransformerError</b> API function in step 6.

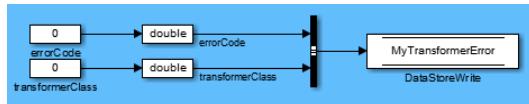
<sup>1)</sup> For <ComSpec> objects with event semantics, you have to reference a DD DataElement object whose **ImplementationPolicy** property is set to **queued**.

- 4 Add a <ComSpec> block to MyRunnable's subsystem and connect it to its corresponding <Port> block.
- 5 In the <ComSpec> block's block dialog, select the Add transformer error port checkbox and close the dialog.
- 6 In the MATLAB Command Window, type  

```
t1TransformerError('PrepareModelForSimulation',...
    'FileName','<MyBusAndSignalObjects>',...
    'SoftwareComponents','MySWC').
```

TargetLink performs the following actions:

- Creates a Simulink.Bus object and a Simulink Signal object. The code for creating the Simulink.Bus object and the Simulink.Signal object is saved to the <MyBusAndSignalObjects> M file. You can call this M file to easily create the objects after opening your model.
- Adds some model elements to the root level of your model to inject a bus signal into your MyRunnable's subsystem:



This bus signal represents the transformer error struct and contains the error code and the transformer class as leaf bus elements. The bus signal is injected into MyRunnable's subsystem via the ReceiverComSpec block's Trafo Error port.

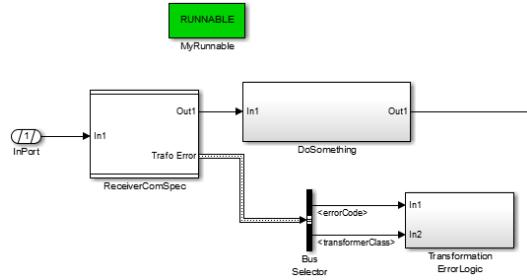
The Data store name is set according to the value of the DD TransformerErrorSignalLabel property at the <Port> object.

- 7 In MyRunnable's subsystem, connect the Trafo Error port of the <ComSpec> block to a Bus Selector block to split the bus' signals.
- 8 Model your error logic as required.
- 9 On the root level of your model, replace the Constant blocks created by the **t1TransformerError** API function in step 6 by stimuli as required.
- 10 Perform your simulation.

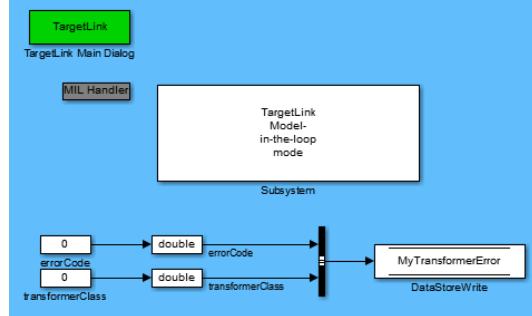
## Result

You prepared your model to simulate transformer errors.

For an example of MyRunnable, refer to the following illustration:

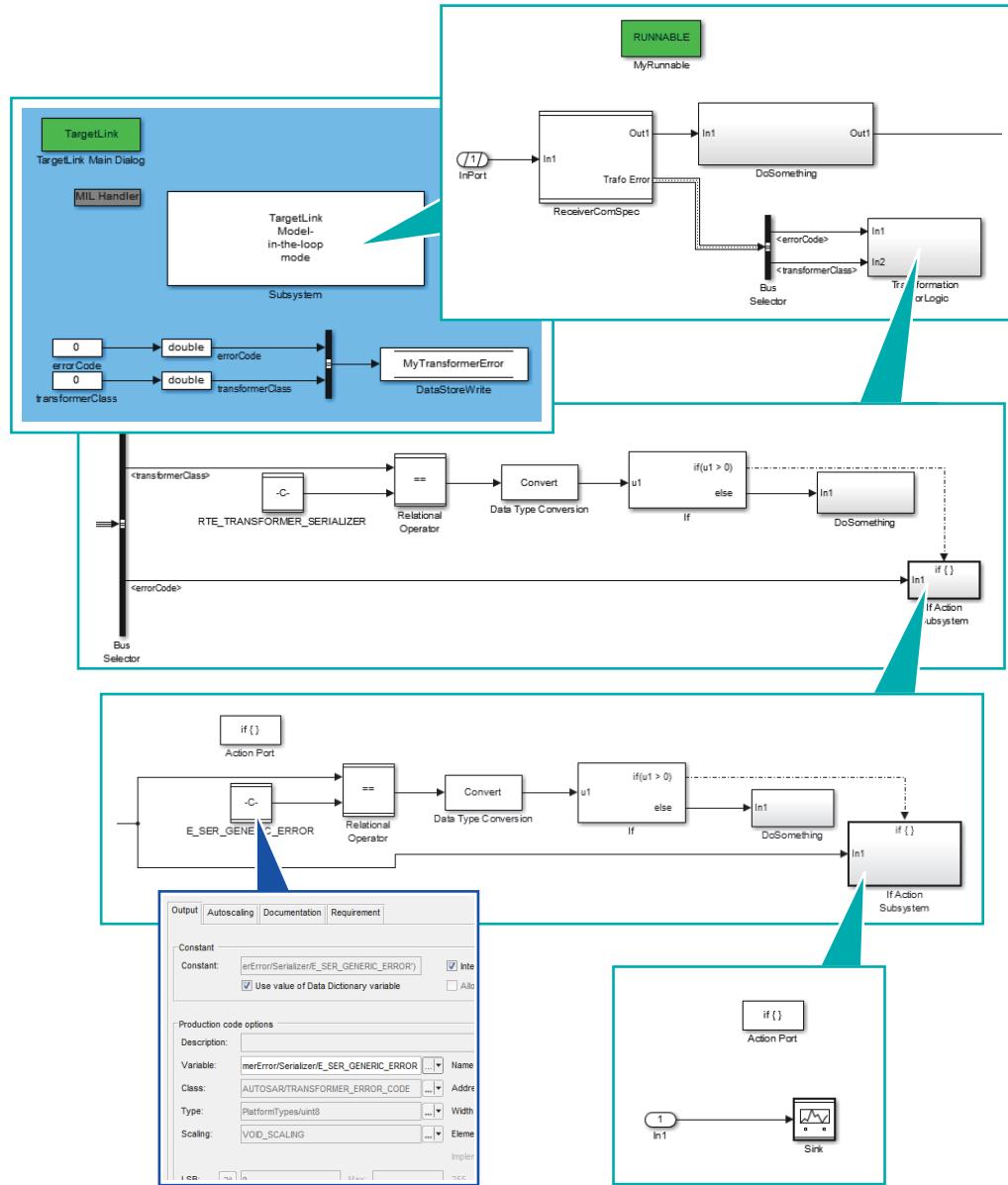


For an example of the root level of the model, refer to the following illustration:



### Error logic example

An example of an error logic looks like this:



**Related topics****Basics**

Basics on Data Transformation.....	205
Basics on Simulating Classic-AUTOSAR-Compliant SWCs.....	296

**References**

ReceiverComSpec Block ( TargetLink Model Element Reference)
SenderComSpec Block ( TargetLink Model Element Reference)
tlTransformerError ( TargetLink API Reference)
tlTransformerError('CreateSimulinkBusObject', propertyName, PropertyValue, ...) ( TargetLink API Reference)
tlTransformerError('CreateSimulinkSignalObjects', propertyName, PropertyValue, ...) ( TargetLink API Reference)
tlTransformerError('CreateStimulusBlocks', propertyName, PropertyValue, ...) ( TargetLink API Reference)
tlTransformerError('PrepareModelForSimulation', propertyName, PropertyValue, ...) ( TargetLink API Reference)

# How to Simulate Operation Calls in Asynchronous Client-Server Communication

**Simulating data flow in asynchronous client-server communication**

Rte\_Call and Rte\_Result are RTE functions. In TargetLink, you can model them and their data flow via [operation subsystems](#).

**Possible methods**

The following methods are possible:

- If your operation subsystems reside in the same [code generation unit \(CGU\)](#), refer to Method 1.
- If your operation subsystems reside in the different CGUs, refer to Method 2.

**Precondition**

From the View menu of the Data Dictionary Manager, the Show Unset Properties option is selected.

**Method 1**
**To model data flow for simulation if the operation subsystems reside in the same CGU**

- 1 Create a DD Module object called <SimulationVariablesModule>. This lets you generate the variable's definition in a separate code module to separate it from the production code of the runnable.
- 2 Create a DD Variable object and name it as required.

- 3** In its Property Value List specify the following properties:

Property	Value
Class	MERGEABLE_GLOBAL
ModuleRef	<SimulationVariablesModule>

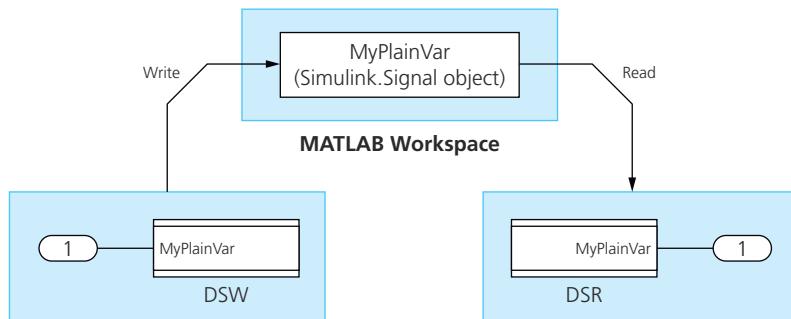
- 4** Specify additional properties, if required.
- 5** In the model, place a Data Store Memory block outside your operation subsystems.
- 6** Reference the DD Variable object created in step 2 at the block.
- 7** To your operation subsystems, add Data Store Read and Data Store Write blocks as required to model the data flow for simulation.  
Make sure that they access the Data Store Memory block.

## Method 2

### To model data flow for simulation if the operation subsystems reside in different CGUs

- 1** To the MATLAB workspace, add a Simulink.Signal object that meets your requirements.

In MIL simulation mode, this Simulink.Signal object is used to propagate the simulation data from one subsystem to another:



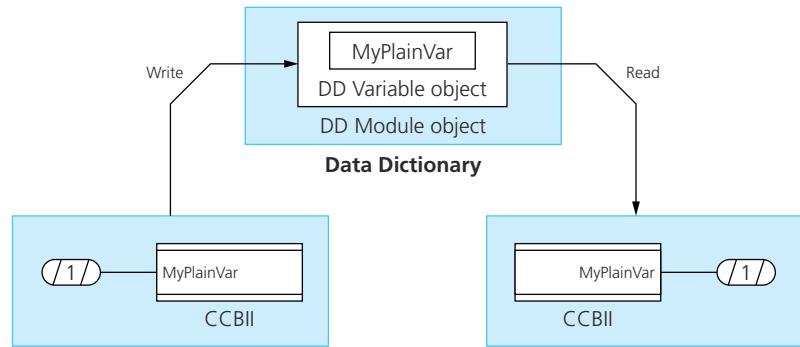
- 2** Create a DD Module object called <SimulationVariablesModule>.
- 3** Select its ModuleInfo subtree and set its CodeGenerationBasis property to DDBased.
- 4** Create a DD Variable object and name it as required.
- 5** In its Property Value List specify the following properties:

Property	Value
Class	MERGEABLE_GLOBAL
ModuleRef	<SimulationVariablesModule>

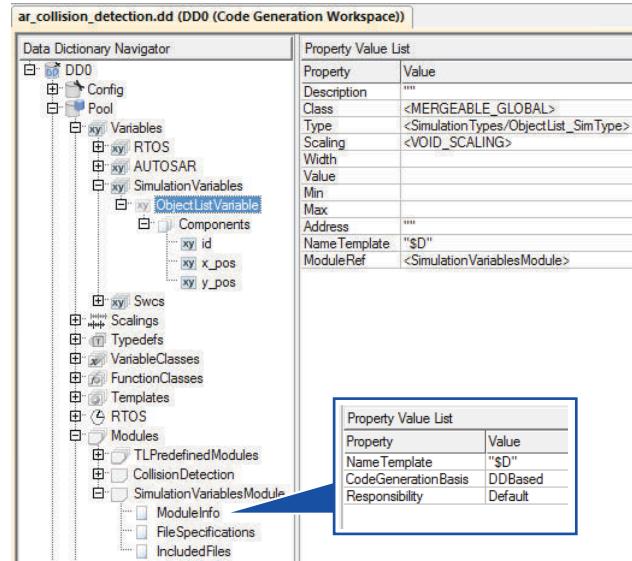
- 6** Specify additional properties, if required.
- 7** Create a DD CodeGenerationUnit object and name it <SimulationCodeGenerationUnit>.
- 8** Add its ModuleRefs subtree and reference the module created in step 2 at its ModuleRef property.

This lets you generate the variable's definition in a separate code module (generated independently of the model) to separate it from the production code of the runnable.

In SIL and PIL simulation modes, this variable is used to propagate the simulation data from one subsystem to another.



The following screenshot shows an example of the module specification in the Data Dictionary:



**9** Select Create ModelDesign from the /Pool context menu.

A /Pool/ModelDesign subtree is created as follows:



**10** Select Create DataStoreMemoryBlock from the /Pool/ModelDesign/Blocks context menu.

A block named new\_DataStoreMemoryBlock is created.

- 11** Rename the block to the name of the Simulink.Signal object described in step 1 on page 310 (e.g., MyPlainVar).

The screenshot shows the Data Dictionary Navigator interface. On the left, the 'Data Dictionary Navigator' pane displays a tree structure under 'ModelDesign'. The 'Blocks' node is expanded, showing 'MyPlainVar' as a child. Under 'MyPlainVar', there are three sub-nodes: 'variable', 'DataStore', and 'Documentation'. On the right, the 'Property Value List' pane shows the properties of 'MyPlainVar'. The properties listed are:

Property	Value
BlockType	TL_DataStoreMemory
Description	<value not set>
ModuleRef	<value not set>
SampleTime	<value not set>
SampleTimeOffset	<value not set>

- 12** Select the block's variable subtree.

The VariableRef property is not set but shown, refer to [Precondition](#) on page 309.

- 13** Copy and paste the DD path of the DD Variable object created in step 4 on page 310 in the Value field of the VariableRef property.

- 14** To your operation subsystems, add Data Store Read and Data Store Write blocks as required to model the data flow for simulation.

Make sure that they access the Simulink.Signal object for MIL simulation mode.

## Result

You modeled the data flow between two operation subsystems for simulation.

## Related topics

### Basics

Details on Data Transformation for Client-Server Communication.....	209
Modeling Client-Server Communication.....	133

### References

Adjust to Typedef (  TargetLink Data Dictionary Manager Reference)	
tlSimulinkBusObject (  TargetLink API Reference)	

# Using a Software Architecture Tool Together with TargetLink

---

## Where to go from here

## Information in this section

Generating/Updating a Frame Model from Classic AUTOSAR Data.....	314
Mode Management with SystemDesk and TargetLink.....	322

# Generating/Updating a Frame Model from Classic AUTOSAR Data

## Where to go from here

## Information in this section

Basics on Generating/Updating a Frame Model from Classic AUTOSAR Data.....	314
How to Generate a Frame Model from Classic AUTOSAR Data.....	316
How to Update an Existing Frame Model from Classic AUTOSAR Data.....	318
Example of Generating a Frame Model from an AUTOSAR File.....	319

## Basics on Generating/Updating a Frame Model from Classic AUTOSAR Data

### Modeling approach

Generating a frame model with imports/exports and a predefined substructure but without functionality is the recommended approach if you already have [Classic AUTOSAR](#) data for the model you want to develop.

### Update functionality

To facilitate round trips between TargetLink and software architecture tools such as SystemDesk, TargetLink's frame model feature allows you to

- Generate a new frame model from AUTOSAR files imported to the Data Dictionary.
- Update an existing frame model according to AUTOSAR files imported to the Data Dictionary.

This enables you to quickly adapt to changes in the AUTOSAR files and to implement the software component in TargetLink.

### Generating or updating a frame model

**Model generation** TargetLink generates a hierarchical subsystem according to the AUTOSAR data contained in DDO of the Data Dictionary.  
For instructions, refer to [How to Generate a Frame Model from Classic AUTOSAR Data](#) on page 316.

**Model updating** TargetLink updates an existing frame model by comparing it to the [Classic AUTOSAR](#) data contained in DDO of the Data Dictionary. An update report is generated, allowing quick access to changed model parts via hyperlinks.

For details on the updating process, refer to [How to Update an Existing Frame Model from Classic AUTOSAR Data](#) on page 318.

**MATLAB API** You can generate and update frame models via the MATLAB API. For reference information refer to [t1\\_generate\\_swc\\_model](#) ( [TargetLink API Reference](#)).

**Allowed properties** For a complete list of allowed parameters, refer to [t1\\_generate\\_swc\\_model](#) ( [TargetLink API Reference](#)).

### Generating Classic-AUTOSAR-compliant code immediately

TargetLink lets you generate subsystems with stimuli signals when generating a frame model from [Classic AUTOSAR](#) data. This allows you to immediately generate code compliant with [Classic AUTOSAR](#).

TargetLink generates and connects subsystems with stimuli signals (constant '0') to the following ports:

- TargetLink subsystem imports.
- Runnable outports, i.e., TargetLink OutPort blocks configured for [Classic AUTOSAR](#) communication.
- Imports and outports of subsystems that implement operation calls.

#### Note

TargetLink provides stimuli signals to generate code immediately for testing purposes. To simulate a TargetLink subsystem replace the generated stimuli signals (constant '0') for the TargetLink subsystem imports as required and model the subsystem's functional behavior.

### Customizing frame model generation/update

TargetLink provides hook script templates allowing you to add your own commands at specific points in frame model generation. This allows you to adapt frame model generation/update to your company's development tool chain.

- For basic information on using hook scripts, refer to [Basics on Using Hook Scripts for Customization \(AUTOSAR\)](#) ( [TargetLink Interoperation and Exchange Guide](#)).
- For details on frame model generation/update-specific hook script templates, refer to [Frame Model Generation Hook Scripts \(AUTOSAR\)](#) ( [TargetLink File Reference](#)).

Use TargetLink's `t1CustomizationFiles('Create', ...)` API function or the Create Customization Files dialog to derive customization files from their templates. You can open the dialog by executing the `t1CustomizationFiles` command without arguments. Refer to [Deriving Customization Files From Their Templates](#) ( [TargetLink Customization and Optimization Guide](#)).

**Related topics****Basics**

Basics on Using Hook Scripts for Customization (AUTOSAR) ( TargetLink Interoperation and Exchange Guide)

Deriving Customization Files From Their Templates ( TargetLink Customization and Optimization Guide)

**HowTos**

How to Generate a Frame Model from Classic AUTOSAR Data..... 316  
How to Update an Existing Frame Model from Classic AUTOSAR Data..... 318

**References**

Create Customization Files Dialog Description ( TargetLink Tool and Utility Reference)

Frame Model Generation Hook Scripts (AUTOSAR) ( TargetLink File Reference)

Generate Frame Model ( TargetLink Data Dictionary Manager Reference)

tl\_generate\_swc\_model ( TargetLink API Reference)

tlCustomizationFiles ( TargetLink API Reference)

Update AUTOSAR Model ( TargetLink Data Dictionary Manager Reference)

## How to Generate a Frame Model from Classic AUTOSAR Data

**Objective**

To generate a frame model from Classic AUTOSAR data.

**Target model**

You can specify the target model for TargetLink's Generate Frame Model command. To do so, specify a name in the **Model** property stored at `/Pool/Autosar/Config/FrameModelGeneration`.

If you do not specify a name, TargetLink generates the SWCs into the currently open model. If no model is open, TargetLink creates a model named *AutosarFrameModel* and generates the SWCs into it.

**Preconditions**

The [Classic AUTOSAR](#) data to create a frame model for is present in DDO. For instructions on importing AUTOSAR files, refer to [Basics on Importing AUTOSAR Files](#) ( TargetLink Interoperation and Exchange Guide).

**Method****To generate a frame model**

- 1 In the Data Dictionary Navigator select the following DD object:  
`/Pool/Autosar/Config/FrameModelGeneration`

- 2 Add the property `Model` to the Property Value List and set its value to `AutosarFrameModel`.  
Make further settings as required. For details on allowed property value pairs refer to `tl_generate_swc_model`.
- 3 Select the DD SoftwareComponent object(s) you want to generate a frame model for.
- 4 On the context menu of a DD SoftwareComponent object, open the AUTOSAR Tools menu and click Generate Frame Model.  
TargetLink generates a frame model for the selected DD SoftwareComponent object(s) and opens the frame model.

**Tip**

Start the Generate Frame Model command from the context menu of a DD SoftwareComponentGroup object to generate a frame model for all DD SoftwareComponent objects contained in that group.

- 5 In the model window, select File - Save As to save the generated model.

---

<b>Result</b>	You have generated a frame model.
---------------	-----------------------------------

---

<b>Next steps</b>	You can model the functionality of the frame model using TargetLink blocks.
-------------------	---

---

<b>Related topics</b>	<b>Basics</b>  <a href="#">Basics on Generating/Updating a Frame Model from Classic AUTOSAR Data</a> ..... 314 <a href="#">Development Approaches with the TargetLink Classic AUTOSAR Module</a> ..... 27
	<b>References</b>  <a href="#">Generate Frame Model</a> ( <a href="#">TargetLink Data Dictionary Manager Reference</a> ) <a href="#">tl_generate_swc_model</a> ( <a href="#">TargetLink API Reference</a> )

## How to Update an Existing Frame Model from Classic AUTOSAR Data

### Objective

To update an existing frame model from [Classic AUTOSAR](#) data.

### Preconditions

The following preconditions must be fulfilled:

- The [Classic AUTOSAR](#) data to update an existing frame model from is present in DD0. For instructions on importing AUTOSAR files, refer to [Basics on Importing AUTOSAR Files](#) ([TargetLink Interoperation and Exchange Guide](#)).
- The existing frame model is open.

### Method

#### To update an existing frame model

- 1 At the Property Value List stored in `/Pool/Autosar/Config/FrameModelGeneration`, configure the frame model update as required. For details on allowed property value pairs refer to `tl_generate_swc_model`.
- 2 Save the DD workspace.
- 3 Select the DD SoftwareComponent object(s) for that you want to update an existing frame model.
- 4 On the context menu of a DD SoftwareComponent object, open the AUTOSAR Tools menu and click Update AUTOSAR Model.

TargetLink updates the existing frame model for the selected DD SoftwareComponent object(s) and opens the SwcModel Update Report in the MATLAB Web browser.

#### Tip

Start the Update AUTOSAR Model command from the context menu of a DD SoftwareComponentGroup object to update a frame model for all DD SoftwareComponent objects contained in that group.

- 5 In the MATLAB Web browser examine the SwcModel Update Report.

SwcModel Update Report  
Software component model update date: 08-Sep-2015  
Model name: ar\_poscontrol  
Data Dictionary: ar\_poscontrol.dd

AUTOSAR Element	DD AUTOSAR Object	TargetLink Block	Annotation
SoftwareComponent	Controller	TL_Controller	No SWC ReceiverPort block added, since TargetLink has detected that for the current software component receiver ports are not modeled by SWC ReceiverPort blocks.
ReceiverPort	RequiredSignals		No SWC SenderPort block added, since TargetLink has detected that for the current software component sender ports are not modeled by SWC SenderPort blocks.
SenderPort	ProvidedSignal		
Runnable	PosController	Controller_Runnable	Block unchanged.
DataReadAccess		ref	For the block ar_poscontrol/TL_Controller/Subsystem/TL_Controller/Controller_Runnable/ref corresponding access point object was not found. This block and its connection must be deleted.
InterrunnableReadVariable		pos	For the block ar_poscontrol/TL_Controller/Subsystem/TL_Controller/Controller_Runnable/pos corresponding access point object was not found. This block and its connection must be deleted.
ActivationReason	ActivationReasons	Activating RTE Event	Block unchanged.
DataWriteAccess		upl	For the block ar_poscontrol/TL_Controller/Subsystem/TL_Controller/Controller_Runnable/upl corresponding access point object was not found. This block and its connection must be deleted.
Runnable	PosLinearization	Position_Linearization_Runnable	Block unchanged.
DataReceivePoint		(position)	For the block ar_poscontrol/TL_Controller/Subsystem/TL_Controller/Position_Linearization_Runnable/(position) corresponding access point object was not found. This block and its connection must be deleted.
InterrunnableWriteVariable		(lin.pos)	For the block ar_poscontrol/TL_Controller/Subsystem/TL_Controller/Position_Linearization_Runnable/(lin.pos) corresponding access point object was not found. This block and its connection must be deleted.

TargetLink's SwcModel Update Report informs you about updated blocks and whether user action is required.

- 6** Follow the hyperlinks to DD objects or model components and make changes as required.

For reference information on the update process, refer to [Update AUTOSAR Model](#) ( [TargetLink Data Dictionary Manager Reference](#)).

<b>Result</b>	You have updated a frame model.
---------------	---------------------------------

<b>Next steps</b>	You can add further functionality to the frame model using TargetLink blocks.
-------------------	---

<b>Related topics</b>	<p><b>Basics</b></p> <table> <tr> <td><a href="#">Basics on Generating/Updating a Frame Model from Classic AUTOSAR Data</a>.....</td><td>314</td></tr> <tr> <td><a href="#">Development Approaches with the TargetLink Classic AUTOSAR Module</a>.....</td><td>27</td></tr> </table> <p><b>HowTos</b></p> <table> <tr> <td><a href="#">How to Generate a Frame Model from Classic AUTOSAR Data</a>.....</td><td>316</td></tr> </table> <p><b>References</b></p> <table> <tr> <td><a href="#">tl_generate_swc_model</a> ( <a href="#">TargetLink API Reference</a>)</td></tr> <tr> <td><a href="#">Update AUTOSAR Model</a> ( <a href="#">TargetLink Data Dictionary Manager Reference</a>)</td></tr> </table>	<a href="#">Basics on Generating/Updating a Frame Model from Classic AUTOSAR Data</a> .....	314	<a href="#">Development Approaches with the TargetLink Classic AUTOSAR Module</a> .....	27	<a href="#">How to Generate a Frame Model from Classic AUTOSAR Data</a> .....	316	<a href="#">tl_generate_swc_model</a> ( <a href="#">TargetLink API Reference</a> )	<a href="#">Update AUTOSAR Model</a> ( <a href="#">TargetLink Data Dictionary Manager Reference</a> )
<a href="#">Basics on Generating/Updating a Frame Model from Classic AUTOSAR Data</a> .....	314								
<a href="#">Development Approaches with the TargetLink Classic AUTOSAR Module</a> .....	27								
<a href="#">How to Generate a Frame Model from Classic AUTOSAR Data</a> .....	316								
<a href="#">tl_generate_swc_model</a> ( <a href="#">TargetLink API Reference</a> )									
<a href="#">Update AUTOSAR Model</a> ( <a href="#">TargetLink Data Dictionary Manager Reference</a> )									

## Example of Generating a Frame Model from an AUTOSAR File

<b>Introduction</b>	Demonstrates how to generate a frame model from the AUTOSAR file of the <i>AUTOSAR poscontrol</i> demo.
---------------------	---

<b>Precondition</b>	You have exported one or more AUTOSAR files from the <i>AUTOSAR poscontrol</i> demo model.
---------------------	--

For instructions on how to export an AUTOSAR file, refer to [Basics on Exporting AUTOSAR Files](#) ( [TargetLink Interoperation and Exchange Guide](#)).

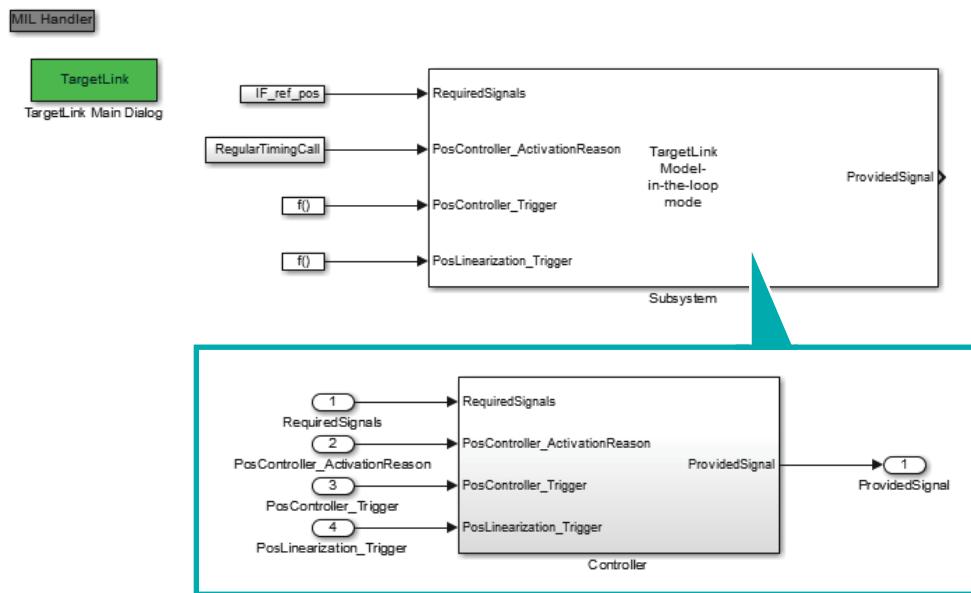
<b>Method</b>	<b>To generate a frame model from the AUTOSAR file of the AUTOSAR poscontrol demo</b>
---------------	---

- 1 In the Data Dictionary Manager, create a new workspace that is based on the dsdd\_master\_autosar4.dd [System] template.
- 2 Select File - Import - from AUTOSAR File....

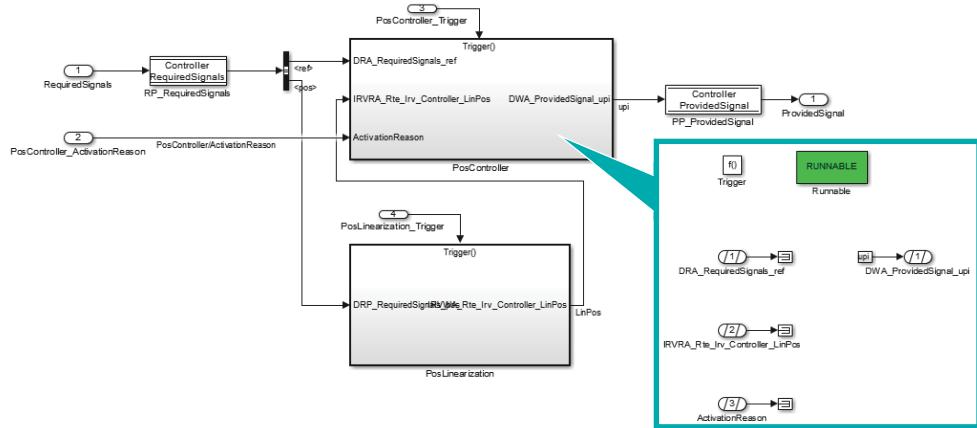
- 3 In the Import from AUTOSAR File dialog, select the AUTOSAR files to import and click OK.  
TargetLink imports the AUTOSAR data to the data dictionary.
- 4 In the Data Dictionary Navigator select the following DD object:  
`/Pool/Autosar/Config/FrameModelGeneration`
- 5 Review its settings and adapt them, if required. For details on allowed property value pairs, refer to the object's embedded help.
- 6 On the context menu of the  
`DD /Pool/Autosar/SoftwareComponents/ComponentType/Controller` object, open the AUTOSAR Tools menu and click Generate Frame Model.  
TargetLink generates a frame model and opens it.
- 7 In the model window, select File - Save As... to save the generated model.

**Result**

You have generated a frame model for the DD SoftwareComponent object named Controller. The following illustrations show the generated frame model.



TargetLink generates a frame model named **Controller** for the DD SoftwareComponent object named **Controller**.



TargetLink generates a subsystem with a preconfigured Function block for each runnable of an atomic software component. Port blocks that are preconfigured for [Classic AUTOSAR](#) communication are copied to the subsystems as required.

## Related topics

### Basics

[Basics on Generating/Updating a Frame Model from Classic AUTOSAR Data.....](#) 314

### References

[Generate Frame Model \(TargetLink Data Dictionary Manager Reference\)](#)

# Mode Management with SystemDesk and TargetLink

## Where to go from here

## Information in this section

Introduction to Mode Management.....	322
Modeling and Implementing Mode Management.....	327

## Introduction to Mode Management

## Where to go from here

## Information in this section

Basics on Mode Management.....	322
Mode Management in the AR_Fuelsys Demo.....	324

## Basics on Mode Management

### Introduction

Mode management involves switching an ECU or a functional unit within an ECU between its possible operating states.

### Managing operating states

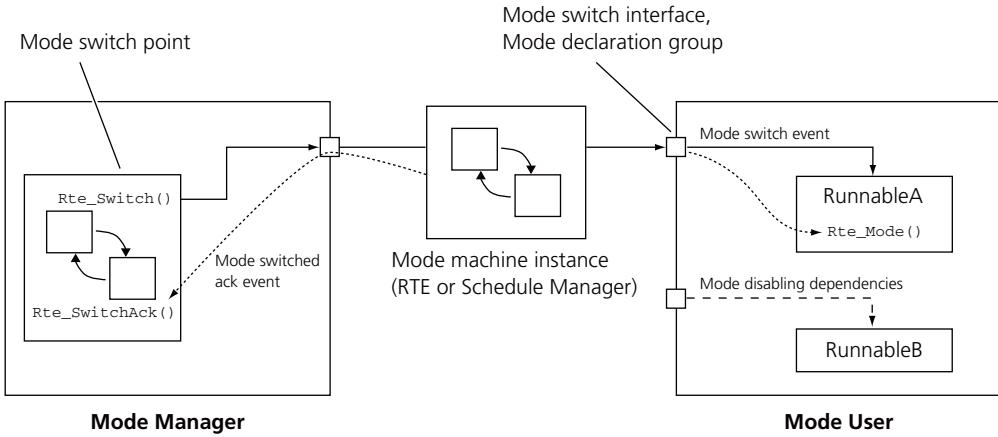
An ECU can be in different modes. These are the operating states of the ECU or its functional units, for example:

- Sleep mode
- Startup mode
- Network synchronization mode
- Normal operating mode
- Error condition mode
- Shutdown mode

The management of the modes is basically carried out by a state machine. It controls and manages the modes which can activate or deactivate different control algorithms during the different operating states. The mode management can also observe the transition between the modes which can take a certain time.

### Mode management according to Classic AUTOSAR

The purpose of modes is to start runnables on the transition between modes or to disable specified triggers of runnables in certain modes.



**Mode manager** Entering and leaving modes is initiated by a *mode manager*. It contains the master state machine to initiate the mode switches. The mode manager can be either a basic software module that provides a service including mode switches, for example, the ECU State Manager, or an application software component (SWC). In the latter case, the mode manager is called an *application mode manager*.

The mode manager communicates the mode switches to the *mode user(s)* via a port with a provided *mode switch interface*. The mode manager itself does not perform the mode switches, but the *mode machine instance* of the RTE or a schedule manager.

A mode switch interface can contain several *mode declaration groups*. A mode declaration group contains a list of the required modes which represent the possible operating states.

A *mode switch point* represents the position within the mode manager where the mode switch is initiated. The *mode switch point* defines the runnable that uses the *Rte\_Switch* API.

**Mode machine instance** The RTE performs all the necessary actions to switch between the modes. The RTE has its own state machine, the *mode machine instance*, to handle the modes and transitions initiated by the mode manager. The mode machine instance is not visible in the [Classic AUTOSAR](#) composition diagrams.

The RTE can be configured to send an acknowledgment to the mode manager when the requested transition is completed. This results in a *mode switched ack event* at the mode manager.

**Mode user** A mode user is an SWC or an atomic SWC with runnables whose execution depends on modes.

There are two mechanisms:

- For all RTE events, for example, a timing event, *mode disabling dependencies* can prevent a runnable from being triggered as long as one or more modes are active.

- If the triggering RTE event of a runnable is a *mode switch* event, the runnable is executed as a result of a mode switch. If the runnable is not mapped to a task, it is executed *synchronously*, i.e. in the same task that the `Rte_Switch` function call occurs in. If a runnable is mapped to a task, it is executed *asynchronously*, i.e. not until the corresponding task is running. The runnable can be triggered on entry to or exit from the mode.

The mode user can use the `Rte_Mode` API to get the currently active mode.

## Related topics

### Basics

[Basics on RTE Events \(SystemDesk Manual\)](#)

## Mode Management in the AR\_Fuelsys Demo

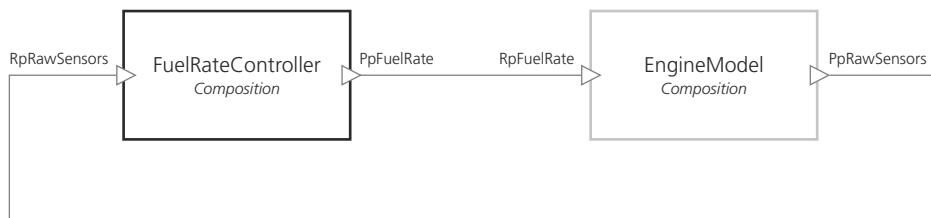
### Introduction

SystemDesk and the TargetLink AUTOSAR Module provide the *AR\_Fuelsys* demo, which models a fault-tolerant fuel injection control system. The demo features different mode declarations for controlling the fuel injection in compliance with [Classic AUTOSAR](#).

### Overview of the demo model

The *AR\_Fuelsys* demo model consists of two compositions:

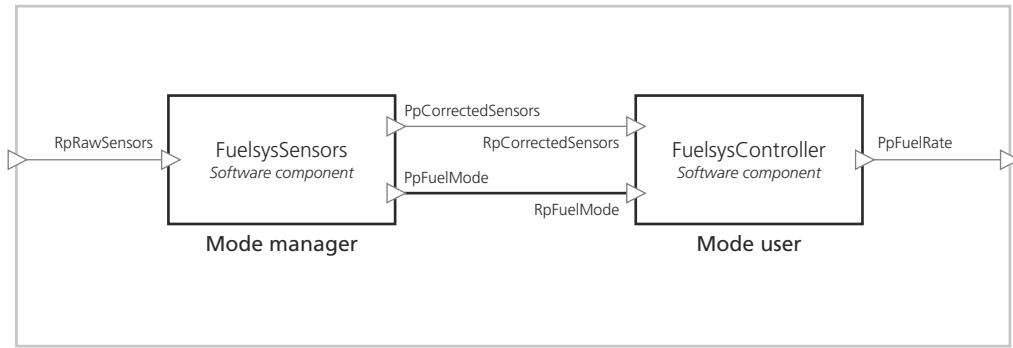
- A *fuel rate controller* (`ControllerComposition`)
- An *engine model* (plant model; `EngineModelComposition`)



Each composition consists of software components (SWCs) with different runnables.

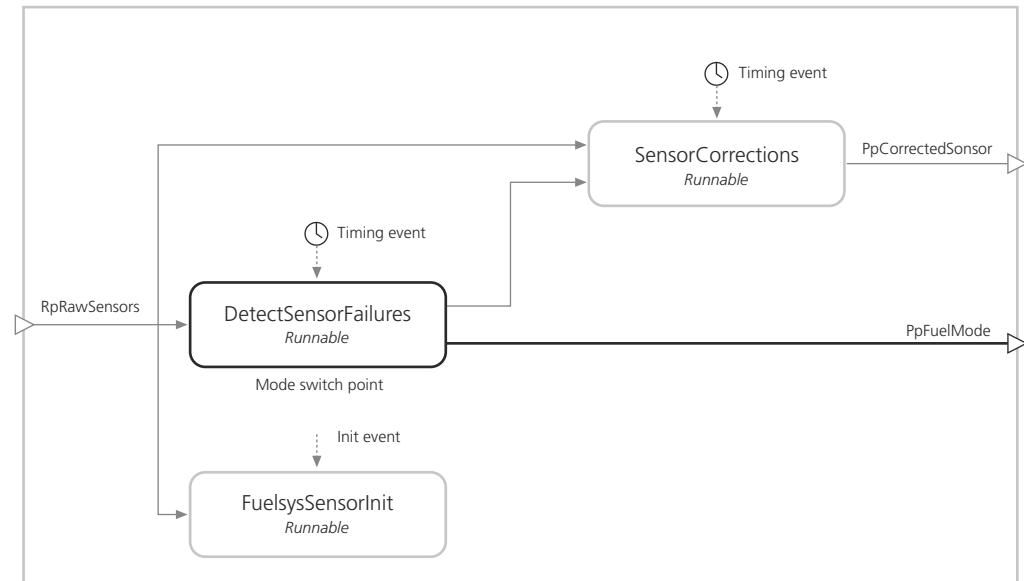
**FuelRateController composition** The software components for mode management of the *FuelRateController* are:

- The *FuelsysSensors* SWC which represents the *mode manager*
- The *FuelsysController* SWC which represents the *mode user*



**EngineModel composition** This composition contains the plant model and stimuli. The composition represents the system to be controlled, which allows you to simulate it in a closed loop. It is not important to mode management and will not be discussed in this section.

**Mode manager** In this demo, the *FuelsysSensors* SWC constitutes an *application mode manager* to handle the four modes for the fuel injection. It also corrects the sensor signals. The mode manager is realized by the *DetectSensorFailures* runnable. It detects sensor signal failures and specifies the fuel injection mode and contains the state machine for the modes and represents the mode switch point.

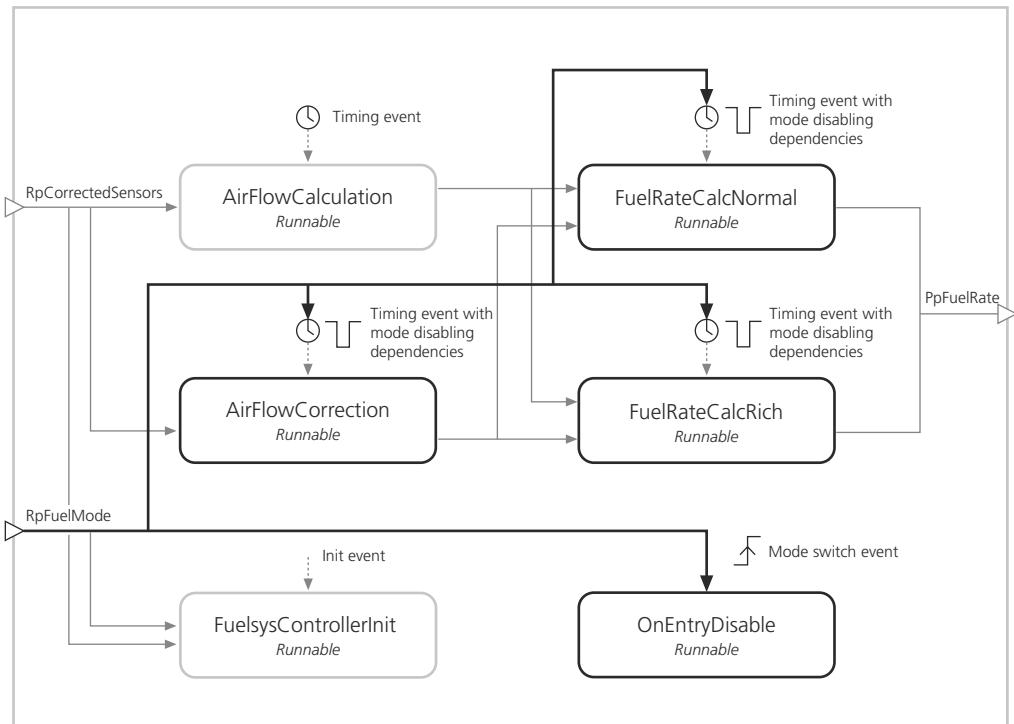


The *SensorCorrection* runnable performs sensor correction and manages fault redundancy. The *FuelsysSensorInit* runnable entity initializes the global variables of *FuelsysSensors* SWC. Both runnables are irrelevant to the demo's mode management.

**Mode declarations** The demo features four mode declarations for the fuel injection mode depending on the engine state and the sensor failures:

- The *LowWarmup* mode, to represent low emissions or a stoichiometric fuel mixture when the lambda sensor is in the warm-up state after a cold start. This is the initial mode. It is also the active mode when a simulation is started.
- The *LowNormal* mode, to represent low emissions or a stoichiometric fuel mixture when the lambda sensor is in the normal operating state.
- The *Rich* mode, to represent an operating state using a rich fuel mixture.
- The *Disabled* mode, to represent a faulty operating state if more than one sensor failure occurs.

**Mode user** The *FuelsysController* SWC is the demo's mode user. It controls the fuel injection rate depending on the currently activated mode and the corrected sensor signals. The mode user consists of six runnables.



Runnable	Description	RTE Event	Mode Disabling Dependencies
AirflowCalculation	To calculate the intake airflow regardless of the currently active fuel mode	Timing event	None
FuelRateCalcNormal	To calculate the fuel rate for a normal low mixture	Timing event	<i>LowWarmup</i> , <i>Rich</i> , or <i>Disabled</i>
AirflowCorrection	To calculate the intake airflow correction for a normal low mixture	Timing event	<i>LowWarmup</i> , <i>Rich</i> , or <i>Disabled</i>

Runnable	Description	RTE Event	Mode Disabling Dependencies
<i>FuelRateCalcRich</i>	To calculate the fuel rate for a rich mixture	Timing event	<i>LowNormal</i> or <i>Disabled</i>
<i>OnEntryDisable</i>	To set a failure counter if there is a faulty operating state and the system is running in the <i>Disabled</i> mode	Mode switch event	None
<i>FuelsysControllerInit</i>	To initialize the global variables of SWC FuelsysController	Init event	None

**Related topics****Basics**

[AR\\_Fuelsys Demo \(SystemDesk Manual\)](#)

## Modeling and Implementing Mode Management

**Where to go from here****Information in this section**

Workflow for Modeling Mode Management.....	327
Modeling the Software Architecture (SystemDesk).....	329
Defining the SWC Internal Behaviors (SystemDesk).....	332
Exporting a Container for each Controller SWC (SystemDesk).....	334
Importing the Controller SWCs (TargetLink).....	336
Implementing the Controller SWCs (TargetLink).....	338

## Workflow for Modeling Mode Management

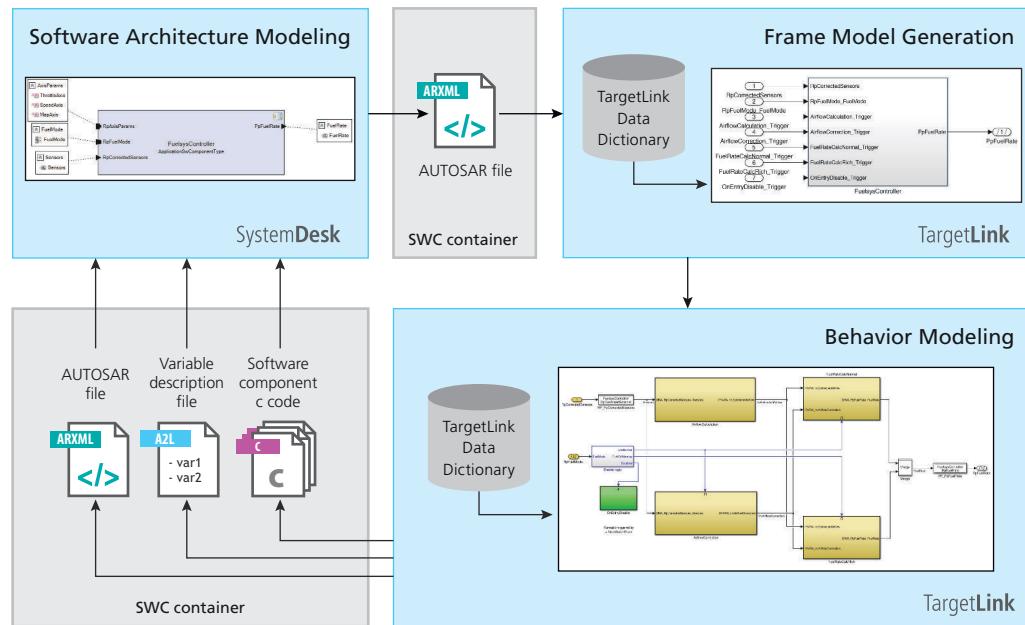
**Introduction**

Modeling mode management is based on a top-down software development approach using the interoperability of SystemDesk and TargetLink.

**Interoperability of SystemDesk and TargetLink**

SystemDesk and TargetLink can interact in different phases in the development of  Classic AUTOSAR-compliant application software components.

In a top-down development approach, you can use SystemDesk first to specify the software architecture on software component level and higher. After this, you can use TargetLink as a behavior modeling tool that provides the implementations of the individual software components. Data exchange between the tools is performed by means of containers, comprising AUTOSAR files (ARXML), A2L files and finally of C code/object files. Container management eases data exchange in providing a best-practice workflow for file exchange synchronization. In further steps, you can use SystemDesk again for generating the RTE and building and simulating the system.



The workflow for modeling and implementing mode management in the *AR\_Fuelsys* demo is described according to the top-down approach.

#### Steps for modeling mode management

The workflow according to the top-down development approach consists of the following steps:

Step	Description	Tool used	Refer to
1	In the first process step, you create the software architecture in SystemDesk.	SystemDesk	<a href="#">Modeling the Software Architecture (SystemDesk) on page 329</a>
2	In the second process step, you create and define the SWC internal behaviors for the two controller SWCs (FuelsysSensors and FuelsysController) in SystemDesk.	SystemDesk	<a href="#">Defining the SWC Internal Behaviors (SystemDesk) on page 332</a>
3	After creating and defining the software architecture, you define containers for the controller SWCs for export.	SystemDesk	<a href="#">Exporting a Container for each Controller SWC (SystemDesk) on page 334</a>

Step	Description	Tool used	Refer to
4	You import the container for each of the controller SWCs in TargetLink and generate a frame model for each SWC.	TargetLink	<a href="#">Importing the Controller SWCs (TargetLink) on page 336</a>
5	You implement the controller SWCs with TargetLink and generate production code. Exchanging the container of each SWC with SystemDesk allows you to use the production code when simulating with in SystemDesk.	TargetLink	<a href="#">Implementing the Controller SWCs (TargetLink) on page 338</a>

## Demo variants

There are different versions of the AR Fuelsys demo for SystemDesk and TargetLink:

- For SystemDesk, you can find the demo variants for one or more ECUs in the .\Demos folder of your SystemDesk installation. This folder contains ZIP archives with the demo files. Before using them, you have to copy them to a user folder, e.g., your Documents folder.
  - For TargetLink, you can find the demo model in the following folder:  
`<DocumentsFolder>\dSPACE\TargetLink\<Version>\Demos\ar_fuelsys` folder.

## Modeling the Software Architecture (SystemDesk)

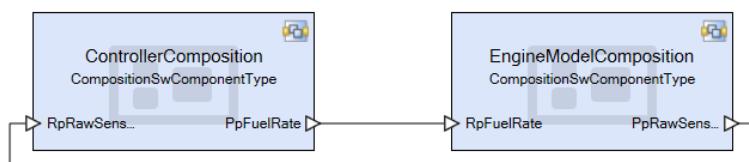
## Introduction

In the first process step, you create the software architecture in SystemDesk.

# Modeling the software architecture

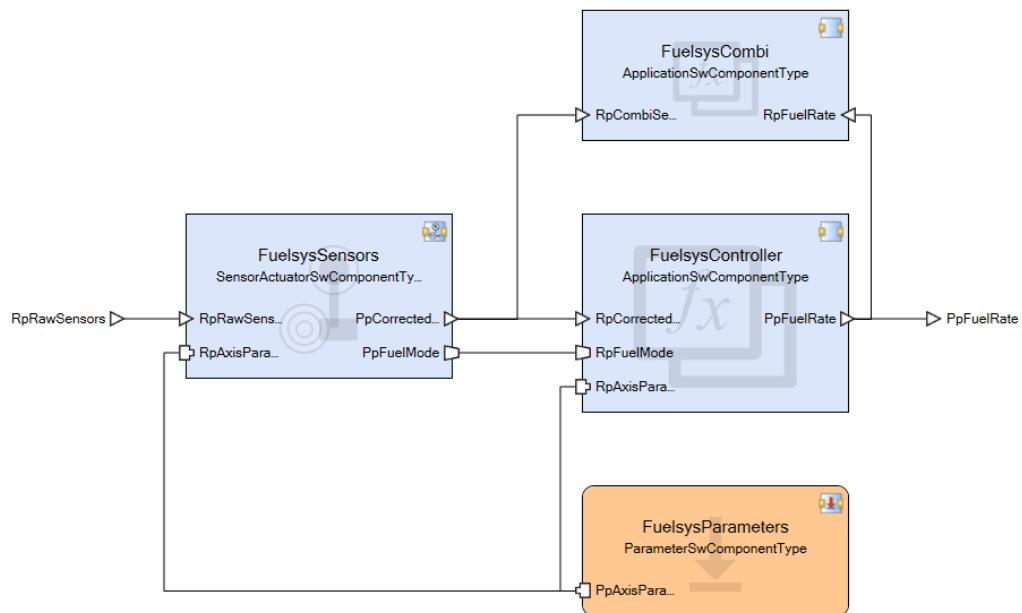
You start the development process by modeling the software architecture in SystemDesk. The software architecture includes the two SWCs for the fuel rate controller. First, you add ports, interfaces and connections to the SWCs.

**AR\_Fuelsys root composition** The AR\_Fuelsys root composition contains the ControllerComposition and the EngineModelComposition.



**ControllerComposition** The ControllerComposition contains the FuelsysSensors atomic SWC and FuelsysController atomic SWC.

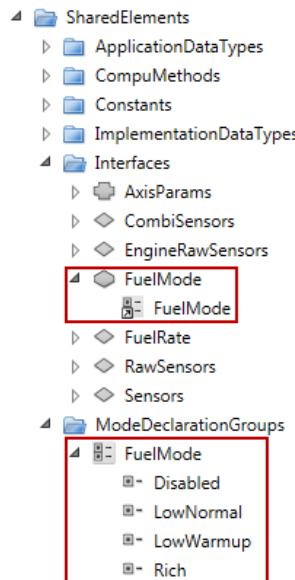
The PpFuelMode port and the RpFuelMode port have mode switch interfaces.



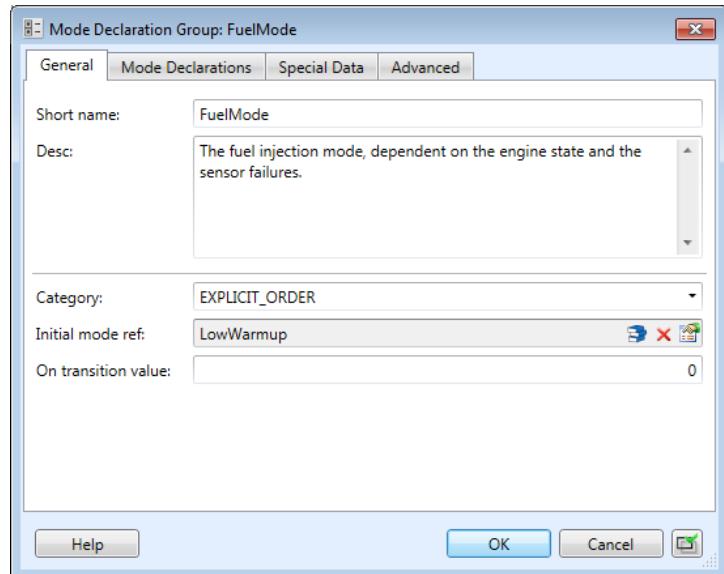
#### Note

The FuelsysCombi application SWC and the FuelsysParameters parameter SWC are irrelevant to mode management. The FuelsysCombi SWC is used to display the engine speed and the average fuel consumption. The FuelsysParameters SWC is used for calibration in the AR\_Fuelsys demo. This SWCs and their connected interfaces will not be discussed in this section.

**Modes** The IF\_FuelMode mode switch interface and the referenced FuelMode mode declaration group are used for mode management, refer to the illustration below.



You select the initial mode for the simulation start on the General page of the Properties dialog of the FuelMode mode declaration group, refer to the illustration below.



**Related topics****Basics**

[Modeling Software Components \(SystemDesk Manual\)](#)

**References**

[Mode Declaration \(SystemDesk Manual\)](#)  
[Mode Declaration Group \(SystemDesk Manual\)](#)  
[Mode Declaration Group Prototype \(SystemDesk Manual\)](#)  
[Mode Switch Interface \(SystemDesk Manual\)](#)

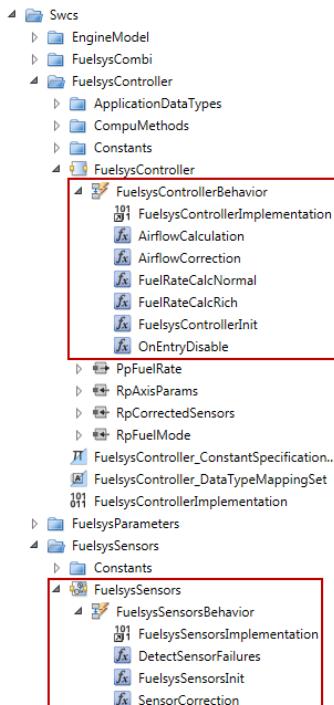
## Defining the SWC Internal Behaviors (SystemDesk)

**Introduction**

In the second process step, you create and define the SWC internal behaviors for the two controller SWCs (FuelsysSensors and FuelsysController) in SystemDesk.

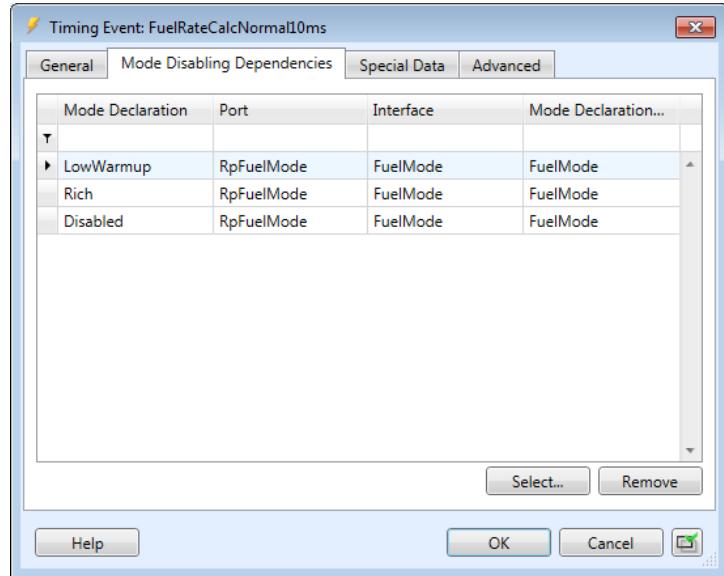
**Specifying Runnables**

The illustration below shows the runnable entities of the controller SWCs' internal behaviors.

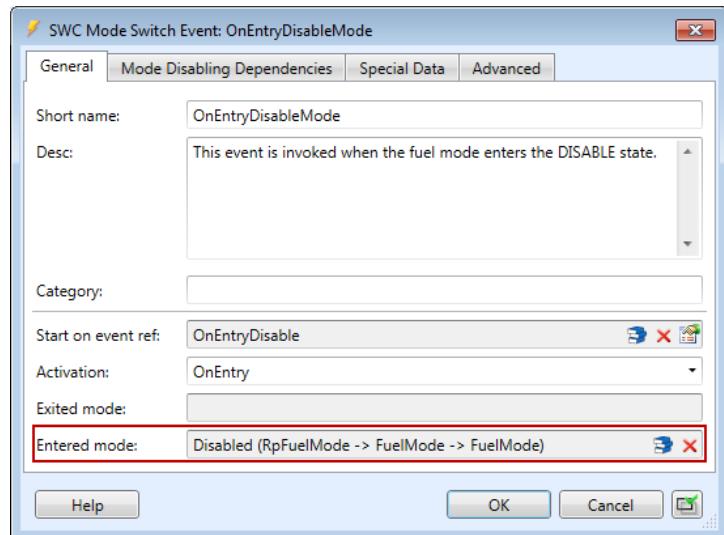


**Defining mode disabling dependencies** The FuelsysController runnable entities FuelRateCalcNormal, AirflowCorrection, and FuelRateCalcRich are

triggered by timing events with mode disabling dependencies. You can define the mode disabling dependencies on the Mode Disabling Dependencies page of an RTE event's Properties dialog, see example below.



**Defining a mode switch event** The FuelsysController runnable entity OnEntryDisable is triggered by the mode switch event OnEntryDisableMode. You can select the corresponding mode on the General page of the event's Properties dialog, see illustration below.



**Related topics**

**Basics**

[Basics on RTE Events \(SystemDesk Manual\)](#)

**References**

[RTE Event \(SystemDesk Manual\)](#)

[SWC Internal Behavior \(SystemDesk Manual\)](#)

## Exporting a Container for each Controller SWC (SystemDesk)

---

**Introduction**

After creating and defining the software architecture, you define containers for the controller SWCs for export.

**Exporting containers**

Structuring the project elements of the controller SWCs in container files is the basis for exporting SWCs.

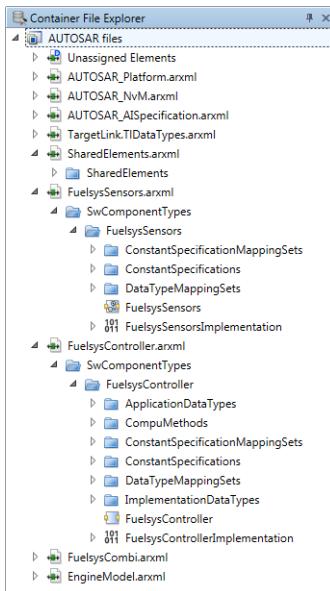
For container export, AUTOSAR elements belonging to one SWC must be kept separate from AUTOSAR elements shared by several SWCs. It is best practice to create one container file per atomic SWC and one container file for the shared elements.

After preparing the container export by separating each atomic SWC and their (shared) elements, you can export one container for each SWC from the **Project Manager**.

**Example**

The demo offers three package groups to categorize the SWCs and their (shared) elements:

- The **SharedElements** container file containing all the elements which are referenced by different SWCs, for example, data types, scalings, interfaces, and the mode declaration group.
- The **FuelsysSensors** container file containing all the elements which are referenced only by the FuelsysSensors SWC.
- The **FuelsysController** container file containing all the elements which are referenced only by the FuelsysController SWC.



Exporting containers for the controller SWCs results in the export of the following AUTOSAR files:

#### FuelsysSensors

```
_ComponentFiles\FuelsysController\FuelsysSensors.arxml
_SharedFiles\SharedElements.arxml
_StandardFiles\AUTOSAR_AISpecification.arxml
_StandardFiles\AUTOSAR_Platform.arxml
_StandardFiles\AUTOSAR_Platform.arxml
```

#### FuelsysController

```
_ComponentFiles\FuelsysSensors\FuelsysController.arxml
_SharedFiles\SharedElements.arxml
_StandardFiles\
AUTOSAR_AISpecification.arxml
_StandardFiles\AUTOSAR_Platform.arxml
_StandardFiles\AUTOSAR_Platform.arxml
```

In the next process step, you import each container in TargetLink.

---

#### Related topics

##### Basics

[Preparing Container Export in SystemDesk \(SystemDesk Manual\)](#)

## Importing the Controller SWCs (TargetLink)

**Introduction** You import the container for each of the controller SWCs in TargetLink and generate a frame model for each SWC.

**Importing the containers** Importing a SystemDesk container results in the creation of DD objects according to the ARXML files belonging to the container:

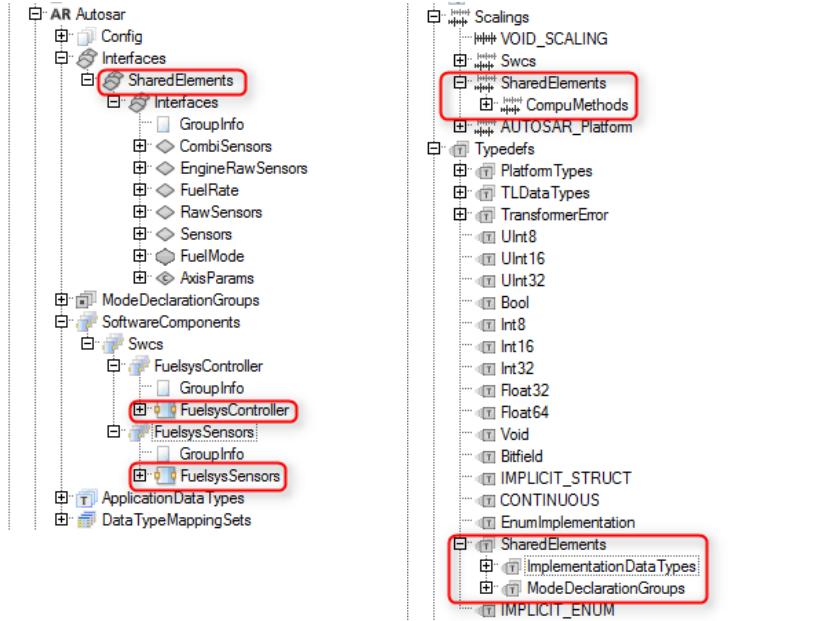
- Each software component described in the ARXML is represented by a DD SoftwareComponent object.
- Software components are grouped together in DD SoftwareComponentGroup objects.
- Shared elements are grouped in
  - DD InterfaceGroup objects for interfaces
  - DD TypedefGroup objects for typedefs
  - DD ScalingGroup objects for scalings

**Generating a frame model** Start the frame model generation for the controller SWCs imported from SystemDesk. TargetLink generates a frame model with imports/outputs and a predefined substructure but without implemented controller functionality.  
Save the frame model, to continue working with it in later process steps.

**Example** In this demo, importing the SystemDesk containers results in the creation of the following DD objects:

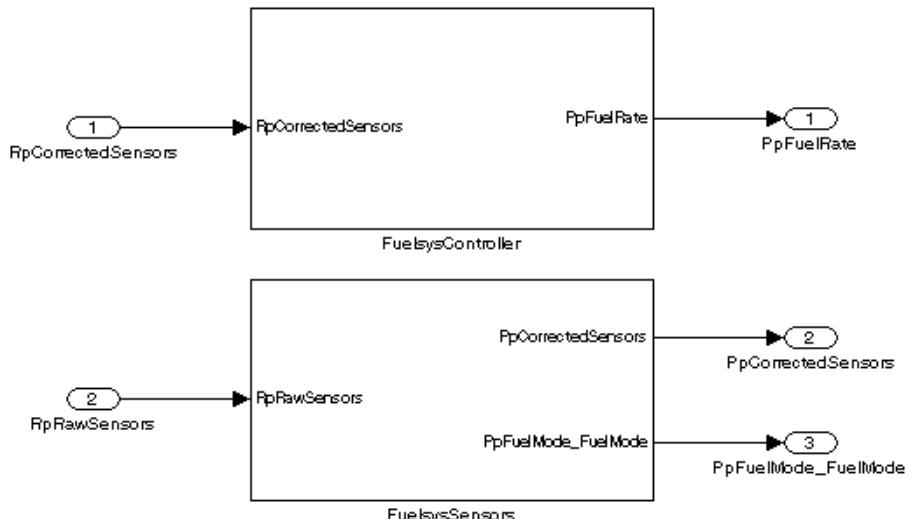
- A DD SoftwareComponent object named FuelsysController.
- A DD SoftwareComponent object named FuelsysSensors.
- A DD InterfaceGroup object named Shared.
- A DD TypedefGroup object named Shared.

- A DD ScalingGroup object named Shared.



If you have created application data types (ADTs) as introduced in AUTOSAR 4.x, your ADTs will be grouped in a DD ApplicationDataTypeGroup object.

Frame model generation results in a model as shown in the following illustration:



In the next process step, you implement the controller SWCs with TargetLink to get C code for the implementations in SystemDesk.

**Related topics****Basics**

Generating/Updating a Frame Model from Classic AUTOSAR Data..... 314

## Implementing the Controller SWCs (TargetLink)

**Introduction**

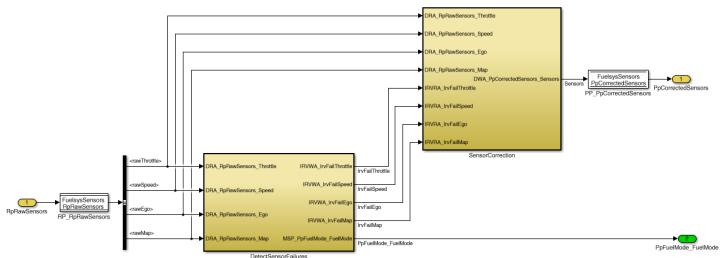
You implement the controller SWCs with TargetLink and generate production code. Exchanging the container of each SWC with SystemDesk allows you to use the production code when simulating with in SystemDesk.

**Basics**

This topic gives an overview of implementing the controller SWCs in TargetLink. The focus is on modeling properties for implementing mode management that is compliant with [Classic AUTOSAR](#). The implementation of the controller algorithm itself is not described here.

**Mode manager**

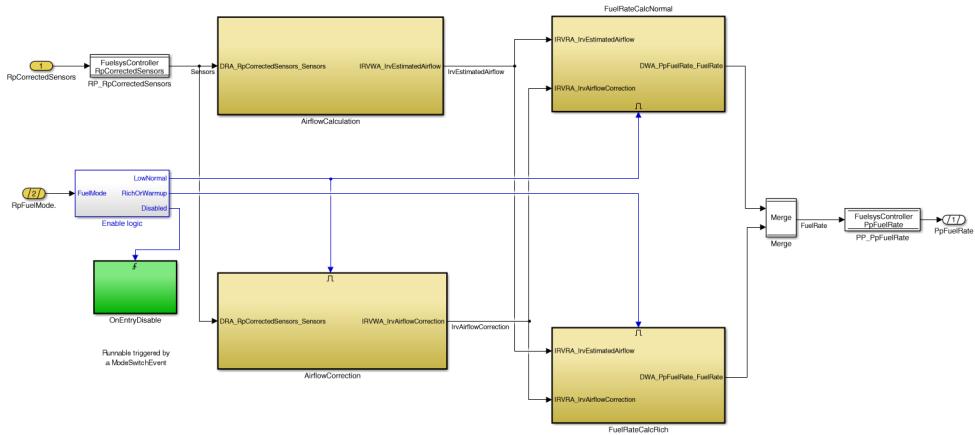
The TL\_FuelsysSensors TargetLink subsystem represents the mode manager. It consists of two subsystems representing the two mode manager runnables.



The Run\_DetectSensorFailures runnable contains a Stateflow chart to implement the master state machine. The PPFuelMode port provides the request to switch the fuel mode.

**Mode user**

The TL\_FuelsysController TargetLink subsystem represents the mode user. It consists of five subsystems representing the five mode user runnables and additional blocks. The additional blocks are included for simulation purposes in TargetLink only.



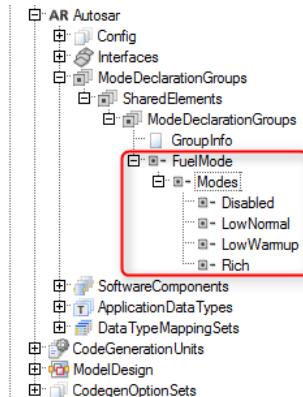
The RpFuelMode port receives the request to switch the fuel mode. The Enable logic subsystem is used to implement the AUTOSAR mode disabling dependencies during simulation with TargetLink. The Enable logic subsystem triggers the three runnables that are subject to mode disabling dependencies.

For simulation purposes, the three runnables that have mode disabling dependencies are realized as *enabled* subsystems in TargetLink.

No production code is generated for the additional blocks. The functionality of the Enable logic subsystem is implemented later by the RTE generated with SystemDesk.

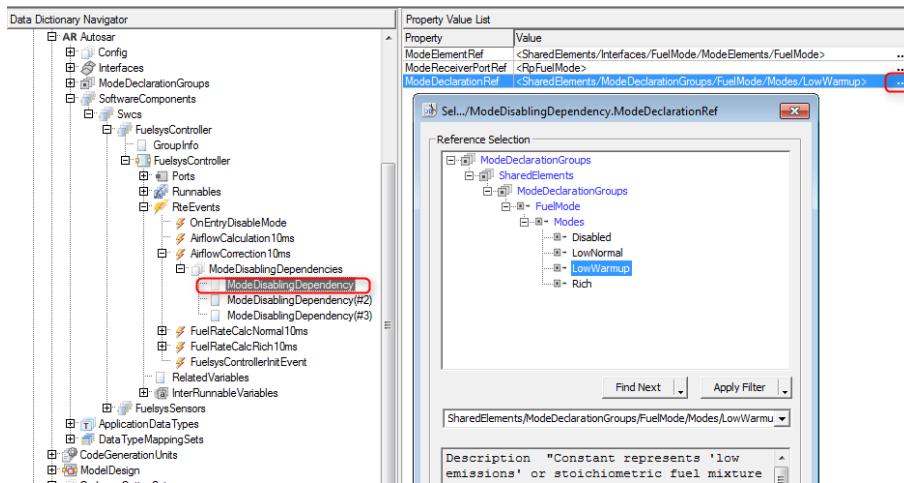
## Mode declarations in the TargetLink Data Dictionary

The illustration below shows where to find the mode declarations of the imported ARXML files in the TargetLink Data Dictionary.



## Defining mode disabling dependencies

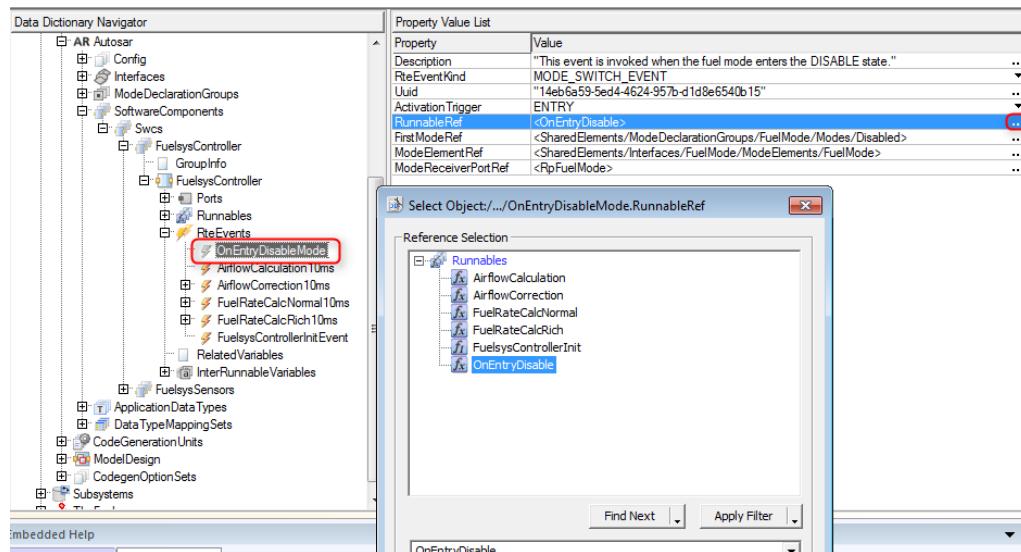
You define mode disabling dependencies in the TargetLink Data Dictionary. The illustration below shows where to reference modes for mode disabling dependencies to RTE events in the TargetLink Data Dictionary.



### Using mode switch events

The mode user's Run\_OnEntryDisable runnable is triggered by the *Ev\_OnEntryDisable* mode switch event. This event is called when the fuel mode enters the *Disabled* state.

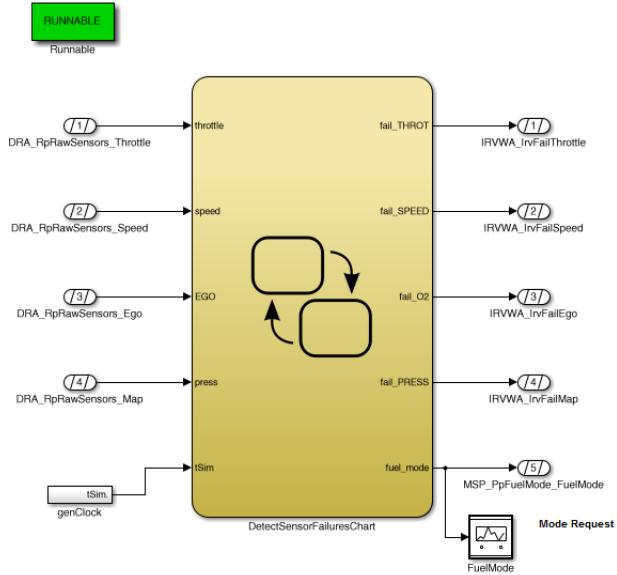
The illustration below shows how to reference a runnable to a mode switch event in the TargetLink Data Dictionary.



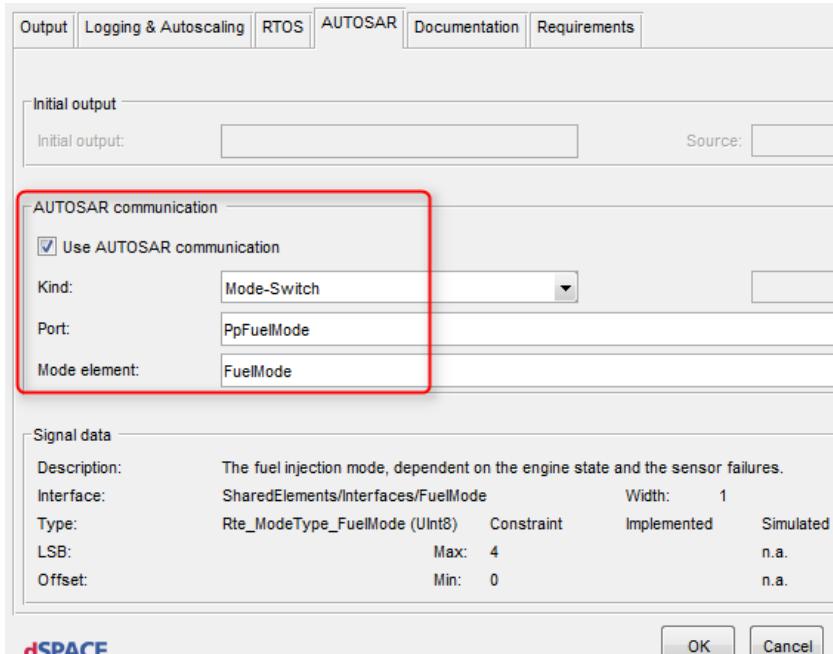
### Modeling mode switches

The FuelModeOutPort block of the mode manager's Run\_DetectSensorFailures runnable is used to implement a *mode switch point*.

The illustration below shows the mode manager's Run\_DetectSensorFailures runnable with the FuelModeOutPort block.



The FuelModeOutPort block is configured to implement a mode switch point. The illustration below shows that a mode switch is requested via the PpFuelMode port for the FuelMode mode.



As a result of the FuelModeOutPort, TargetLink generates the Rte\_Switch\_PpFuelMode\_FuelMode RTE API function.

## Exporting the containers

To use the implementations in SystemDesk you have to

- Generate production code for each SWC.
  - Export the container of each SWC.
- 

## Related topics

### Basics

Basics on Mode Management.....	322
Generating Classic AUTOSAR-Compliant Code.....	275
Mode Management in the AR_Fuelsys Demo.....	324

# Glossary

---

## Introduction

The glossary briefly explains the most important expressions and naming conventions used in the TargetLink documentation.

Where to go from here

Information in this section

Numerics.....	344
A.....	345
B.....	348
C.....	349
D.....	352
E.....	354
F.....	355
G.....	356
I.....	356
L.....	358
M.....	359
N.....	361
O.....	362
P.....	363
R.....	364
S.....	366
T.....	368
U.....	370
V.....	370
W.....	371

## Numerics

**1-D look-up table** A look-up table that maps one input value (x) to one output value (y).

**2-D look-up table** A look-up table that maps two input values (x,y) to one output value (z).

## A

**Abstract interface** An interface that allows you to map a project-specific, physical specification of an interface (made in the TargetLink Data Dictionary) to a logical interface of a [modular unit](#). If the physical interface changes, you do not have to change the Simulink subsystem or the [partial DD file](#) and therefore neither the generated code of the modular unit.

**Access function (AF)** A C function or function-like preprocessor macro that encapsulates the access to an interface variable.

See also [read/write access function](#) and [variable access function](#).

**Acknowledgment** Notification from the [RTE](#) that a [data element](#) or an [event message](#) have been transmitted.

**Activating RTE event** An RTE event that can trigger one or more runnables.

See also [activation reason](#).

**Activation reason** The [activating RTE event](#) that actually triggered the runnable.

Activation reasons can group several RTE events.

**Active page pointer** A pointer to a [data page](#). The page referenced by the pointer is the active page whose values can be changed with a calibration tool.

**Adaptive AUTOSAR** Short name for the AUTOSAR *Adaptive Platform* standard. It is based on a service-oriented architecture that aims at on-demand software updates and high-end functionalities. It complements [Classic AUTOSAR](#).

**Adaptive AUTOSAR behavior code** Code that is generated for model elements in [Adaptive AUTOSAR Function subsystems](#) or [Method Behavior subsystems](#). This code represents the behavior of the model and is part of an adaptive application. Must be integrated in conjunction with [ARA adapter code](#).

**Adaptive AUTOSAR Function** A TargetLink term that describes a C++ function representing a partial functionality of an adaptive application. This function can be called in the C++ code of an adaptive application. From a higher-level perspective, [Adaptive AUTOSAR](#) functions are analogous to runnables in [Classic AUTOSAR](#).

**Adaptive AUTOSAR Function subsystem** An atomic subsystem used to generate code for an [Adaptive AUTOSAR Function](#). It contains a Function block whose AUTOSAR mode is set to **Adaptive** and whose Role is set to **Adaptive AUTOSAR Function**.

**ANSI C** Refers to C89, the C language standard ANSI X3.159-1989.

**Application area** An optional DD object that is a child object of the DD root object. Each Application object defines how an [ECU](#) program is built from the generated subsystems. It also contains some experiment data, for example, a list of variables to be logged during simulations and results of code coverage tests.

**Build** Objects are children of Application objects. They contain all the information about the binary programs built for a certain target platform, for example, the symbol table for address determination.

**Application data type** Abstract type for defining types from the application point of view. It allows you to specify physical data such as measurement data. Application data types do not consider implementation details such as bit-size or endianness.

**Application data type (ADT)** According to AUTOSAR, application data types are used to define types at the application level of abstraction. From the application point of view, this affects physical data and its numerical representation. Accordingly, application data types have physical semantics but do not consider implementation details such as bit width or endianness.

Application data types can be constrained to change the resolution of the physical data's representation or define a range that is to be considered.

See also [implementation data type \(IDT\)](#).

**Application layer** The topmost layer of the [ECU software](#). The application layer holds the functionality of the [ECU software](#) and consists of [atomic software components \(atomic SWCs\)](#).

**ARA adapter code** Adapter code that connects [Adaptive AUTOSAR behavior code](#) with the Adaptive AUTOSAR API or other parts of an adaptive application.

**Array-of-struct variable** An array-of-struct variable is a structure that either is non-scalar itself or that contains at least one non-scalar substructure at any nesting depth. The use of array-of-struct variables is linked to arrays of buses in the model.

**Artifact** A file generated by TargetLink:

- Code coverage report files
- Code generation report files
- [Metadata files](#)
- Model-linked code view files
- [Production code](#) files
- Simulation application object files
- Simulation frame code files
- [Stub code](#) files

**Artifact location** A folder in the file system that contains an [artifact](#). This location is specified relatively to a [project folder](#).

**ASAP2 File Generator** A TargetLink tool that generates ASAP2 files for the parameters and signals of a Simulink model as specified by the corresponding TargetLink settings and generated in the [production code](#).

**ASCII** In production code, strings are usually encoded according to the ASCII standard. The ASCII standard is limited to a set of 127 characters implemented by a single byte. This is not sufficient to display special characters of different languages. Therefore, use another character encoding, such as UTF-8, if required.

**Asynchronous operation call subsystem** A subsystem used when modeling asynchronous client-server communication. It is used to generate the call of the `Rte_Call` API function and for simulation purposes.

See also [operation result provider subsystem](#).

**Asynchronous server call returns event** An [RTE event](#) that specifies whether to start or continue the execution of a [Runnable](#) after the execution of a [server Runnable](#) is finished.

**Atomic software component (atomic SWC)** The smallest element that can be defined in the [application layer](#). An atomic SWC describes a single functionality and contains the corresponding algorithm. An atomic SWC communicates with the outside only via the [interfaces](#) at the SWC's [ports](#). An atomic SWC is defined by an [internal behavior](#) and an [implementation](#).

**Atomic software component instance** An [atomic software component \(atomic SWC\)](#) that is actually used in a controller model.

**AUTOSAR** Abbreviation of *AUTomotive Open System ARchitecture*. The AUTOSAR partnership is an alliance in which the majority of OEMs, suppliers, tool providers, and semiconductor companies work together to develop and establish a de-facto open industry-standard for automotive electric/electronics (E/E) architecture and to manage the growing E/E complexity.

**AUTOSAR import/export** Exchanging standardized [software component descriptions](#) between [AUTOSAR tools](#).

**AUTOSAR subsystem** An atomic subsystem that contains a Function block whose AUTOSAR mode property is set to **Classic**. See also [operation subsystem](#), [operation call with runnable implementation subsystem](#), and [runnable subsystem](#).

**AUTOSAR tool** Generic term for the following tools that are involved in the ECU network software development process according to AUTOSAR:

- Behavior modeling tool
- System-level tool
- ECU-centric tool

TargetLink acts as a behavior modeling tool in the ECU network software development process according to AUTOSAR.

**Autoscaling** Scaling is performed by the Autoscaling tool, which calculates worst-case ranges and scaling parameters for the output, state and parameter variables of TargetLink blocks. The Autoscaling tool uses either worst-case ranges or simulated ranges as the basis for scaling. The upper and lower worst-case range limits can be calculated by the tool itself. The Autoscaling tool always focuses on a subsystem, and optionally on its underlying subsystems.

## B

**Basic software** The generic term for the following software modules:

- System services (including the operating system (OS) and the [ECU State Manager](#))
- Memory services (including the [NVRAM manager](#))
- Communication services
- I/O hardware abstraction
- Complex device drivers

Together with the [RTE](#), the basic software is the platform for the [application layer](#).

**Batch mode** The mode for batch processing. If this mode is activated, TargetLink does not open any dialogs. Refer to [How to Set TargetLink to Batch Mode](#) ([TargetLink Orientation and Overview Guide](#)).

**Behavior model** A model that contains the control algorithm for a controller (function prototyping system) or the algorithm of the controlled system (hardware-in-the-loop system). Can be connected in [ConfigurationDesk](#) via [model ports](#) to build a real-time application (RTA). The RTA can be executed on real-time hardware that is supported by [ConfigurationDesk](#).

**Block properties** Properties belonging to a TargetLink block. Depending on the kind of the property, you can specify them at the block and/or in the Data Dictionary. Examples of block properties are:

- Simulink properties (at a masked Simulink block)
- Logging options or saturation flags (at a TargetLink block)
- Data types or variable classes (referenced from the DD)
- Variable values (specified at the block or referenced from the DD)

**Bus** A bus consists of subordinate [bus elements](#). A bus element can be a bus itself.

**Bus element** A bus element is a part of a [bus](#) and can be a bus itself.

**Bus port block** Bus Import, Bus Outport are bus port blocks. They are similar to the TargetLink Input and Output blocks. They are virtual, and they let you configure the input and output signals at the boundaries of a TargetLink subsystem and at the boundaries of subsystems that you want to generate a function for.

**Bus signal** Buses combine multiple signals, possibly of different types. Buses can also contain other buses. They are then called [nested buses](#).

**Bus-capable block** A block that can process [bus signals](#). Like [bus port blocks](#), they allow you to assign a type definition and, therefore, a [variable class](#) to all the [bus elements](#) at once. The following blocks are bus-capable:

- Constant
- Custom Code (type II) block
- Data Store Memory, Data Store Read, and Data Store Write

- Delay
- Function Caller
- ArgIn, ArgOut
- Merge
- Multiport Switch (Data Input port)
- Probe
- Sink
- Signal Conversion
- Switch (Data Input port)
- Unit Delay
- Stateflow Data
- MATLAB Function Data

## C

**Calibratable variable** Variable whose value can be changed with a calibration tool during run time.

**Calibration** Changing the [calibration parameter](#) values of [ECUs](#).

**Calibration parameter** Any [ECU](#) variable type that can be calibrated. The term *calibration parameter* is independent of the variable type's dimension.

**Calprm** Defined in a [calprm interface](#). Calprms represent [calibration parameters](#) that are accessible via a [measurement and calibration system](#).

**Calprm interface** An [interface](#) that is provided or required by a [software component \(SWC\)](#) via a [port \(AUTOSAR\)](#).

**Calprm software component** A special [software component \(SWC\)](#) that provides [calprms](#). Calprm software components have no [internal behavior](#).

**Canonical** In the DD, [array-of-struct variables](#) are specified canonically. Canonical means that you specify one array element as a representative for all array elements.

**Catalog file (CTLG)** A description of the content of an SWC container. It contains file references and file category information, such as source code files (C and H), object code files (such as O or OBJ), variable description files (A2L), or AUTOSAR files (ARXML).

**Characteristic table (Classic AUTOSAR)** A look-up table as described by [Classic AUTOSAR](#) whose values are measurable or calibratable. See also [compound primitive data type](#)

**Classic AUTOSAR** Short name for the AUTOSAR *Classic Platform* standard that complements [Adaptive AUTOSAR](#).

**Classic initialization mode** The initialization mode used when the Simulink diagnostics parameter Underspecified initialization detection is set to **Classic**.

See also [? simplified initialization mode](#).

**Client port** A require port in client-server communication as described by [? Classic AUTOSAR](#). In the Data Dictionary, client ports are represented as DD ClientPort objects.

**Client-server interface** An [? interface](#) that describes the [? operations](#) that are provided or required by a [? software component \(SWC\)](#) via a [? port \(AUTOSAR\)](#).

**Code generation mode** One of three mutually exclusive options for generating TargetLink standard [? production code](#), AUTOSAR-compliant production code or RTOS-compliant (multirate RTOS/OSEK) production code.

**Code generation unit (CGU)** The smallest unit for which you can generate code. These are:

- TargetLink subsystems
- Subsystems configured for incremental code generation
- Referenced models
- DD CodeGenerationUnit objects

**Code output style definition file** To customize code formatting, you can modify a code output style definition file (XML file). By modifying this file, you can change the representation of comments and statements in the code output.

**Code output style sheets** To customize code formatting, you can modify code output style sheets (XSL files).

**Code section** A section of generated code that defines and executes a specific task.

**Code size** Amount of memory that an application requires specified in RAM and ROM after compilation with the target cross-compiler. This value helps to determine whether the application generated from the code files fits in the ECU memory.

**Code variant** Code variants lead to source code that is generated differently depending on which variant is selected (i.e., variantized at code generation time). For example, if the Type property of a variable has the two variants Int16 and Float32, you can generate either source code for a fixed-point ECU with one variant, or floating-point code with the other.

**Compatibility mode** The default operation mode of RTE generators. The object code of an SWC that was compiled against an application header generated in compatibility mode can be linked against an RTE generated in compatibility mode (possibly by a different RTE generator). This is due to using standardized data structures in the generated RTE code.

See also [? vendor mode](#).

**Compiler inlining** The process of replacing a function call with the code of the function body during compilation by the C compiler via [? inline expansion](#).

This reduces the function call overhead and enables further optimizations at the potential cost of larger [code size](#).

**Composition** A structuring element in the [application layer](#). A composition consists of [software components](#) and their interconnections via [ports](#).

**Compound primitive data type** A primitive [application data type \(ADT\)](#) as defined by [Classic AUTOSAR](#) whose category is one of the following:

- COM\_AXIS
- CUBOID
- CUBE\_4
- CUBE\_5
- CURVE
- MAP
- RES\_AXIS
- VAL\_BLK
- STRING

**Compute-through-overflow (CTO)** Calculation method for additions and subtraction where overflows are allowed in intermediate results without falsifying the final result.

**Concern** A concept in component-based development. It describes the idea that components separate their concerns. Accordingly, they must be developed in such a way that they provide the required functionality, are flexible and easy to maintain, and can be assembled, reused, or replaced by newer, functionally equivalent components in a software project without problems.

**Config area** A DD object that is a child object of the DD root object. The Config object contains configuration data for the tools working with the TargetLink Data Dictionary and configuration data for the TargetLink Data Dictionary itself. There is only one Config object in each DD workspace. The configuration data for the TargetLink Data Dictionary is a list of included DD files, user-defined views, data for variant configurations, etc. The data in the Config area is typically maintained by a Data Dictionary administrator.

**ConfigurationDesk** A dSPACE software tool for implementing and building real-time applications (RTA).

**Constant value expression** An expression for which the Code Generator can determine the variable values during code generation.

**Constrained range limits** User-defined minimum (Min) or maximum (Max) values that the user ensures will never be exceeded. The Code Generator relies on these ranges to make the generated [production code](#) more efficient. If no

Min/Max values are entered, the [implemented range](#) limits are used during production code generation.

**Constrained type** A DD Typedef object whose Constraints subtree is specified.

**Container** A bundle of files. The files are described in a catalog file that is part of the container. The files of a container can be spread over your file system.

**Container Manager** A tool for handling [containers](#).

**Container set file (CTS)** A file that lists a set of containers. If you export containers, one container set file is created for every TargetLink Data Dictionary.

**Conversion method** A method that describes the conversion of a variable's integer values in the ECU memory into their physical representations displayed in the Measurement and Calibration (MC) system.

**Custom code** Custom code consists of C code snippets that can be included in production code by using custom code files that are associated with custom code blocks. TargetLink treats this code as a black box. Accordingly, if this code contains custom code variables you must specify them via [custom code symbols](#). See also [external code](#).

**Custom code symbol** A variable that is used in a custom code file. It must be specified on the Interface page of custom code blocks.

**Customer-specific C function** An external function that is called from a Stateflow diagram and whose interface is made known to TargetLink via a scripting mechanism.

## D

---

**Data element** Defined in a [sender-receiver interface](#). Data elements are information units that are exchanged between [sender ports](#), [receiver ports](#) and [sender-receiver ports](#). They represent the data flow.

**Data page** A structure containing all of the [calibratable variables](#) that are generated during code generation.

**Data prototype** The generic term for one of the following:

- [Data element](#)
- [Operation argument](#)
- [Calprm](#)
- [Interrunnable variable \(IRV\)](#)
- Shared or PerInstance [Calprm](#)
- [Per instance memory](#)

**Data receive error event** An [RTE event](#) that specifies to start or continue the execution of a [Runnable](#) related to receiver errors.

**Data received event** An [RTE event](#) that specifies whether to start or continue the execution of a [Runnable](#) after a [data element](#) is received by a [receiver port](#) or [sender-receiver port](#).

**Data semantics** The communication of [data elements](#) with last-is-best semantics. Newly received data elements overwrite older ones regardless of whether they have been processed or not.

**Data send completed event** An [RTE event](#) that specifies whether to start or continue the execution of a [Runnable](#) related to a sender [acknowledgment](#).

**Data transformation** A transformation of the data of inter-ECU communication, such as end-to-end protection or serialization, that is managed by the [RTE](#) via [transformers](#).

**Data type map** Defines a mapping between [implementation data types](#) (represented in TargetLink by DD Typedef objects) and [application data types](#).

**Data type mapping set** Summarizes all the [data type maps](#) and [mode request type maps](#) of a [software component \(SWC\)](#).

**Data variant** One of two or more differing data values that are generated into the same C code and can be switched during ECU run time using a calibratable variant ID variable. For example, the Value property of a gain parameter can have the variants 2, 3, and 4.

**DataItemMapping (DIM)** A DataItemMapping object is a DD object that references a [ReplaceableDataItem \(RDI\)](#) and a DD variable. It is used to define the DD variable object to map an RDI object to, and therefore also the [implementation variable](#) in the generated code.

**DD child object** The [DD object](#) below another DD object in the [DD object tree](#).

**DD data model** The DD data model describes the object kinds, their properties and constraints as well as the dependencies between them.

**DD file** A DD file (\*.dd) can be a [DD project file](#) or a [partial DD file](#).

**DD object** Data item in the Data Dictionary that can contain [DD child objects](#) and DD properties.

**DD object tree** The tree that arranges all [DD objects](#) according to the [DD data model](#).

**DD project file** A file containing the [DD objects](#) of a [DD workspace](#).

**DD root object** The topmost [DD object](#) of the [DD workspace](#).

**DD subtree** A part of the [DD object tree](#) containing a [DD object](#) and all its descendants.

**DD workspace** An independent organizational unit (central data container) and the largest entity that can be saved to file or loaded from a [DD project file](#). Any number of DD workspaces is supported, but only the first (DD0) can be used for code generation.

**Default enumeration constant** Represents the default constant, i.e., the name of an [enumerated value](#) that is used for initialization if an initial value is required, but not explicitly specified.

**Direct reuse** The Code Generator adds the [instance-specific variables](#) to the reuse structure as leaf struct components.

## E

---

**ECU** Abbreviation of *electronic control unit*.

**ECU software** The ECU software consists of all the software that runs on an [ECU](#). It can be divided into the [basic software](#), [run-time environment \(RTE\)](#), and the [application layer](#).

**ECU State Manager** A piece of software that manages [modes](#). An ECU state manager is part of the [basic software](#).

**Enhanceable Simulink block** A Simulink® block that corresponds to a TargetLink simulation block, for example, the Gain block.

**Enumerated value** An enumerated value consists of an [enumeration constant](#) and a corresponding underlying integer value ([enumeration value](#)).

**Enumeration constant** An enumeration constant defines the name for an [enumerated value](#).

**Enumeration data type** A data type with a specific name, a set of named [enumerated values](#) and a [default enumeration constant](#).

**Enumeration value** An enumeration value defines the integer value for an [enumerated value](#).

**Event message** Event messages are information units that are defined in a [sender-receiver interface](#) and exchanged between [sender ports](#) or [receiver ports](#). They represent the control flow. On the receiver side, each event message is related to a buffer that queues the received messages.

**Event semantics** Communication of [data elements](#) with first-in-first-out semantics. Data elements are received in the same order they were sent. In simulations, TargetLink behaves as if [data semantics](#) was specified, even if you specified event semantics. However, TargetLink generates calls to the correct RTE API functions for data and event semantics.

**ExchangeableWidth** A DD object that defines [code variants](#) or improves code readability by using macros for signal widths.

**Exclusive area** Allows for specifying critical sections in the code that cannot preempt/interrupt each other. An exclusive area can be used to specify the mutual exclusion of [runnables](#).

**Executable application** The generic term for [offline simulation applications](#) and [real-time applications](#).

**Explicit communication** A communication mode in [Classic AUTOSAR](#). The data is exchanged whenever data is required or provided.

**Explicit object** An explicit object is an object in [production code](#) that the Code Generator created from a direct specification made at a [DD object](#) or at a [model element](#). For comparison, see [implicit object](#).

**Extern C Stateflow symbol** A C symbol (function or variable) that is used in a Stateflow chart but that is defined in an external code module.

**External code** Existing C code files/modules from external sources (e.g., legacy code) that can be included by preprocessor directives and called by the C code generated by TargetLink. Unlike [Custom code](#), external code is used as it is.

**External container** A container that is owned by the tool with that you are exchanging a software component but that is not the tool that triggers the container exchange. This container is used when you import files of a software component which were created or changed by the other tool.

## F

---

**Filter** An algorithm that is applied to received [data elements](#).

**Fixed-Point Library** A library that contains functions and macros for use in the generated [production code](#).

**Function AF** The short form for an [access function \(AF\)](#) that is implemented as a C function.

**Function algorithm object** Generic term for either a MATLAB local function, the interface of a MATLAB local function or a [local MATLAB variable](#).

**Function class** A class that represents group properties of functions that determine the function definition, function prototypes and function calls of a function in the generated [production code](#). There are two types of function classes: predefined function class objects defined in the `/Pool/FunctionClasses` group in the DD and implicit function classes (default function classes) that can be influenced by templates in the DD.

**Function code** Code that is generated for a [modular unit](#) that represents functionality and can have [abstract interfaces](#) to be reused without changes in different contexts, e.g. in different [integration models](#).

**Function inlining** The process of replacing a function call with the code of the function body during code generation by TargetLink via [inline expansion](#). This reduces the function call overhead and enables further optimizations at the potential cost of larger [code size](#).

**Function interface** An interface that describes how to pass the inputs and outputs of a function to the generated [production code](#). It is described by the function signature.

**Function subsystem** A subsystem that is atomic and contains a Function block. When generating code, TargetLink generates it as a C function.

**Functional Mock-up Unit (FMU)** An archive file that describes and implements the functionality of a model based on the Functional Mock-up Interface (FMI) standard.

## G

---

**Global data store** The specification of a DD DataStoreMemoryBlock object that references a variable and is associated with either a Simulink.Signal object or Data Store Memory block. The referenced variable must have a module specification and a fixed name and must be global and non-static. Because of its central specification in the Data Dictionary, you can use it across the boundaries of [CGUs](#).

---

**Implementation** Describes how a specific [internal behavior](#) is implemented for a given platform (microprocessor type and compiler). An implementation mainly consists of a list of source files, object files, compiler attributes, and dependencies between the make and build processes.

**Implementation data type (IDT)** According to AUTOSAR, implementation data types are used to define types on the implementation level of abstraction. From the implementation point of view, this regards the storage and manipulation of digitally represented data. Accordingly, implementation data types have data semantics and do consider implementation details, such as the data type.

Implementation data types can be constrained to change the resolution of the digital representation or define a range that is to be considered. Typically, they correspond to `typedef` statements in C code and still abstract from platform specific details such as endianness.

See also [application data type \(ADT\)](#).

**Implementation variable** A variable in the generated [production code](#) to which a [ReplaceableDataItem \(RDI\)](#) object is mapped.

**ImplementationPolicy** A property of [data element](#) and [Calprm](#) elements that specifies the implementation strategy for the resulting variables with respect to consistency.

**Implemented range** The range of a variable defined by its [scaling](#) parameters. To avoid overflows, the implemented range must include the maximum and minimum values the variable can take in the [simulation application](#) and in the ECU.

**Implicit communication** A communication mode in [Classic AUTOSAR](#). The data is exchanged at the start and end of the runnable that requires or provides the data.

**Implicit object** Any object created for the generated code by the TargetLink Code Generator (such as a variable, type, function, or file) that may not have been specified explicitly via a TargetLink block, a Stateflow object, or the TargetLink Data Dictionary. Implicit objects can be influenced via DD templates. For comparison, see [explicit object](#).

**Implicit property** If the property of a [DD object](#) or of a model based object is not directly specified at the object, this property is created by the Code Generator and is based on internal templates or DD Template objects. These properties are called implicit properties. Also see [implicit object](#) and [explicit object](#).

**Included DD file** A [partial DD file](#) that is inserted in the proper point of inclusion in the [DD object tree](#).

**Incremental code generation unit (CGU)** Generic term for [code generation units \(CGUs\)](#) for which you can incrementally generate code. These are:

- Referenced models
  - Subsystems configured for incremental code generation
- Incremental CGUs can be nested in other model-based CGUs.

**Indirect reuse** The Code Generator adds pointers to the reuse structure which reference the indirectly reused [instance-specific variables](#).

Indirect reuse has the following advantages to [direct reuse](#):

- The combination of [shared](#) and [instance-specific variable](#).
- The reuse of input/output variables of neighboring blocks.

**Inline expansion** The process of replacing a function call with the code of the function body. See also [function inlining](#) and [compiler inlining](#).

**Instance-specific variable** A variable that is accessed by one [Reusable system instance](#). Typically, instance-specific variables are used for states and parameters whose value are different across instances.

**Instruction set simulator (ISS)** A simulation model of a microprocessor that can execute binary code compiled for the corresponding microprocessor. This allows the ISS to behave in the same way as the simulated microprocessor.

**Integration model** A model or TargetLink subsystem that contains [modular units](#) which it integrates to make a larger entity that provides its functionality.

**Interface** Describes the [data elements](#), [NvData](#), [event messages](#), [operations](#), or [calibration parameters](#) that are provided or required by a [software component \(SWC\)](#) via a [port \(AUTOSAR\)](#).

**Internal behavior** An element that represents the internal structure of an [atomic software component \(atomic SWC\)](#). It is characterized by the following entities and their interdependencies:

- [Exclusive area](#)
- [Interrunnable variable \(IRV\)](#)
- [Per instance memory](#)
- [Per instance parameter](#)
- [Runnable](#)
- [RTE event](#)
- [Shared parameter](#)

**Interrunnable variable (IRV)** Variable object for specifying communication between the [runnables](#) in one [atomic software component \(atomic SWC\)](#).

**Interrupt service routine (ISR) function** A function that implements an ISR and calls the step functions of the subsystems that are assigned by the user or by the TargetLink Code Generator during multirate code generation.

**Intertask communication** The flow of data between tasks and ISRs, tasks and tasks, and between ISRs and ISRs for multirate code generation.

**Is service** A property of an [interface](#) that indicates whether the interface is provided by a [basic software](#) service.

**ISV** Abbreviation for instance-specific variable.

## L

---

**Leaf bus element** A leaf bus element is a subordinate [bus element](#) that is not a [bus](#) itself.

**Leaf bus signal** See also [leaf bus element](#).

**Leaf struct component** A leaf struct component is a subordinate [struct component](#) that is not a [struct](#) itself.

**Legacy function** A function that contains a user-provided C function.

**Library subsystem** A subsystem that resides in a Simulink® library.

**Local container** A container that is owned by the tool that triggers the container exchange.

The tool that triggers the exchange transfers the files of a [software component](#) to this container when you export a software component. The [external container](#) is not involved.

**Local MATLAB variable** A variable that is generated when used on the left side of an assignment or in the interface of a MATLAB local function. TargetLink does not support different data types and sizes on local MATLAB variables.

**Look-up function** A function for a look-up table that returns a value from the look-up table (1-D or 2-D).

## M

---

**Macro** A literal representing a C preprocessor definition. Macros are used to provide a fixed sequence of computing instructions as a single program statement. Before code compilation, the preprocessor replaces every occurrence of the macro by its definition, i.e., by the code that it stands for.

**Macro AF** The short form for an [access function \(AF\)](#) that is implemented as a function-like preprocessor macro.

**MATLAB code elements** MATLAB code elements include [MATLAB local functions](#) and [local MATLAB variables](#). MATLAB code elements are not available in the Simulink Model Explorer or the Property Manager.

**MATLAB local function** A function that is scoped to a [MATLAB main function](#) and located at the same hierarchy level. MATLAB local functions are treated like MATLAB main functions and have the same properties as the MATLAB main function by default.

**MATLAB main function** The first function in a MATLAB function file.

**Matrix AF** An access function resulting from a DD AccessFunction object whose VariableKindSpec property is set to `APPLY_TO_MATRIX`.

**Matrix signal** Collective term for 2-D signals implemented as [matrix variable](#) in [production code](#).

**Matrix variable** Collective term for 2-D arrays in [production code](#) that implement 2-D signals.

**Measurement** Viewing and analyzing the time traces of [calibration parameters](#) and [measurement variables](#), for example, to observe the effects of ECU parameter changes.

**Measurement and calibration system** A tool that provides access to an [ECU](#) for [measurement](#) and [calibration](#). It requires information on the [calibration parameters](#) and [measurement variables](#) with the ECU code.

**Measurement variable** Any variable type that can be [measured](#) but not [calibrated](#). The term *measurement variable* is independent of a variable type's dimension.

**Memory mapping** The process of mapping variables and functions to different [memory sections](#).

**Memory section** A memory location to which the linker can allocate variables and functions.

**Message Browser** A TargetLink component for handling fatal (F), error (E), warning (W), note (N), and advice (A) messages.

**MetaData files** Files that store metadata about code generation. The metadata of each [code generation unit \(CGU\)](#) is collected in a DD Subsystem object that is written to the file system as a partial DD file called `<CGU>_SubsystemObject.dd`.

**Method Behavior subsystem** An atomic subsystem used to generate code for a method implementation. From the TargetLink perspective, this is an [Adaptive AUTOSAR Function](#) that can take arguments.

It contains a Function block whose AUTOSAR mode is set to **Adaptive** and whose Role is set to **Method Behavior**.

**Method Call subsystem** An atomic subsystem that is used to generate a method call in the code of an [Adaptive AUTOSAR Function](#). The subsystem contains a Function block whose AUTOSAR mode is set to **Adaptive** and whose Role is set to **Method Call**. The subsystem interface is used to generate the function interface while additional model elements that are contained in the subsystem are only for simulation purposes.

**Microcontroller family (MCF)** A group of [microcontroller units](#) with the same processor, but different peripherals.

**Microcontroller unit (MCU)** A combination of a specific processor with additional peripherals, e.g. RAM or AD converters. MCUs with the same processor, but different peripherals form a [microcontroller family](#).

**MIL simulation** A simulation method in which the function model is computed (usually with double floating-point precision) on the host computer as an executable specification. The simulation results serve as a reference for [SIL simulations](#) and [PIL simulations](#).

**MISRA** Organization that assists the automotive industry to produce safe and reliable software, e.g., by defining guidelines for the use of C code in automotive electronic control units or modeling guidelines.

**Mode** An operating state of an [ECU](#), a single functional unit, etc..

**Mode declaration group** Contains the possible [operating states](#), for example, of an [ECU](#) or a single functional unit.

**Mode manager** A piece of software that manages [modes](#). A mode manager can be implemented as a [software component \(SWC\)](#) of the [application layer](#).

**Mode request type map** An entity that defines a mapping between a [mode declaration group](#) and a type. This specifies that mode values are instantiated in the [software component \(SWC\)](#)'s code with the specified type.

**Mode switch event** An [RTE event](#) that specifies to start or continue the execution of a [Runnable](#) as a result of a [mode change](#).

**Model Compare** A dSPACE software tool that identifies and visualizes the differences in the contents of Simulink/TargetLink models (including Stateflow). It can also merge the models.

**Model component** A model-based [code generation unit \(CGU\)](#).

**Model element** A model in MATLAB/Simulink consists of model elements that are TargetLink blocks, Simulink blocks, and Stateflow objects, and signal lines connecting them.

**Model port** A port used to connect a [behavior model](#) in [ConfigurationDesk](#). In TargetLink, multiple model ports of the same kind (data in or data out) can be grouped in a [model port block](#).

**Model port block** A block in [ConfigurationDesk](#) that has one or more [model ports](#). It is used to connect the [behavior model](#) in [ConfigurationDesk](#).

**Model port variable** A DD Variable object that represents a [model port](#) of a [behavior model](#) in [ConfigurationDesk](#).

**Model-dependent code elements** Code elements that (partially) result from specifications made in the model.

**Model-independent code elements** Code elements that can be generated from specifications made in the Data Dictionary alone.

**Modular unit** A submodel containing functionality that is reusable and can be integrated in different [integration models](#). The [production code](#) for the modular unit can be generated separately.

**Module** A DD object that specifies code modules, header files, and other arbitrary files.

**Module specification** The reference of a DD Module object at a [Function Block](#) ([TargetLink Model Element Reference](#)) block or DD object. The resulting code elements are generated into the [module](#). See also [production code](#) and [stub code](#).

**ModuleOwnership** A DD object that specifies an owner for a module (module owner) or module group, i.e. the owning [code generation unit \(CGU\)](#) that generates the [production code](#) for it or declares the [module](#) as external code that is not generated by TargetLink.

---

**Nested bus** A nested bus is a [bus](#) that is a subordinate [bus element](#) of another bus.

**Nested struct** A nested struct is a [struct](#) that is a subordinate [struct component](#) of another struct.

**Non-scalar signal** Collective term for vector and [matrix signals](#).

**Non-standard scaling** A [scaling](#) whose LSB is different from  $2^0$  or whose Offset is not 0.

**Nv receiver port** A require port in NvData communication as described by [Classic AUTOSAR](#). In the Data Dictionary, nv receiver ports are represented as DD NvReceiverPort objects.

**Nv sender port** A provide port in NvData communication as described by [Classic AUTOSAR](#). In the Data Dictionary, nv sender ports are represented as DD NvSenderPort objects.

**Nv sender-receiver port** A provide-require port in NvData communication as described by [Classic AUTOSAR](#). In the Data Dictionary, nv sender-receiver ports are represented as DD NvSenderReceiverPort objects.

**NvData** Data that is exchanged between an [atomic software component \(atomic SWC\)](#) and the [ECU's NVRAM](#).

**NvData interface** An [interface](#) used in [NvData](#) communication.

**NVRAM** Abbreviation of *non volatile random access memory*.

**NVRAM manager** A piece of software that manages an [ECU's NVRAM](#). An NVRAM manager is part of the [basic software](#).

## O

**Offline simulation application (OSA)** An application that can be used for offline simulation in VEOS.

**Online parameter modification** The modification of parameters in the [production code](#) before or during a [SIL simulation](#) or [PIL simulation](#).

**Operation** Defined in a [client-server interface](#). A [software component \(SWC\)](#) can request an operation via a [client port](#). A software component can provide an operation via a [server port](#). Operations are implemented by [server runnables](#).

**Operation argument** Specifies a C-function parameter that is passed and/or returned when an [operation](#) is called.

**Operation call subsystem** A collective term for [synchronous operation call subsystem](#) and [asynchronous operation call subsystem](#).

**Operation call with runnable implementation subsystem** An atomic subsystem that contains a Function block whose AUTOSAR mode property is set to **Classic** and whose Role is set to **Operation call with runnable implementation**.

**Operation invoked event** An [RTE event](#) that specifies to start or continue the execution of a [Runnable](#) as a result of a client call. A Runnable that is related to an [Operation invoked event](#) represents a server.

**Operation result provider subsystem** A subsystem used when modeling [asynchronous](#) client-server communication. It is used to generate the call of the [Rte\\_Result API](#) function and for simulation purposes.

See also [Asynchronous operation call subsystem](#).

**Operation subsystem** A collective term for [Operation call subsystem](#) and [Operation result provider subsystem](#).

**OSEK Implementation Language (OIL)** A modeling language for describing the configuration of an OSEK application and operating system.

## P

---

**Package** A structuring element for grouping elements of [Software components](#) in any hierarchy. Using package information, software components can be spread across or combined from several [Software component description \(SWC-D\)](#) files during [AUTOSAR import/export](#) scenarios.

**Parent model** A model containing references to one or more other models by means of the Simulink Model block.

**Partial DD file** A [DD file](#) that contains only a DD subtree. If it is included in a [DD project file](#), it is called [Included DD file](#). The partial DD file can be located on a central network server where all team members can share the same configuration data.

**Per instance memory** The definition of a data prototype that is instantiated for each [atomic software component instance](#) by the [RTE](#). A data type instance can be accessed only by the corresponding instance of the [atomic SWC](#).

**Per instance parameter** A parameter for measurement and calibration unique to the instance of a [Software component \(SWC\)](#) that is instantiated multiple times.

**Physical evaluation board (physical EVB)** A board that is equipped with the same target processor as the [ECU](#) and that can be used for validation of the generated [production code](#) in [PIL simulation](#) mode.

**PIL simulation** A simulation method in which the TargetLink control algorithm ([Production code](#)) is computed on a [microcontroller target](#) ([Physical](#) or [Virtual](#)).

**Plain data type** A data type that is not struct, union, or pointer.

**Platform** A specific target/compiler combination. For the configuration of platforms, refer to the Code generation target settings in the TargetLink Main Dialog Block block.

**Pool area** A DD object which is parented by the DD root object. It contains all data objects which can be referenced in TargetLink models and which are used for code generation. Pool data objects allow common data specifications to be reused across different blocks or models to easily keep consistency of common properties.

**Port (AUTOSAR)** A part of a [software component \(SWC\)](#) that is the interaction point between the component and other software components.

**Port-defined argument values** Argument values the RTE can implicitly pass to a server.

**Preferences Editor** A TargetLink tool that lets users view and modify all user-specific preference settings after installation has finished.

**Production code** The code generated from a [code generation unit \(CGU\)](#) that owns the module containing the code. See also [stub code](#).

**Project folder** A folder in the file system that belongs to a TargetLink code generation project. It forms the root of different [artifact locations](#) that belong to this project.

**Property Manager** The TargetLink user interface for conveniently managing the properties of multiple model elements at the same time. It can consist of menus, context menus, and one or more panes for displaying property-related information.

**Provide calprm port** A provide port in parameter communication as described by [Classic AUTOSAR](#). In the Data Dictionary, provide calprm ports are represented as DD ProvideCalPrmPort objects.

## R

---

**Read/write access function** An [access function \(AF\)](#) that *encapsulates the instructions* for reading or writing a variable.

**Real-time application** An application that can be executed in real time on dSPACE real-time hardware such as SCALEXIO.

**Receiver port** A require port in sender-receiver communication as described by [Classic AUTOSAR](#). In the Data Dictionary, receiver ports are represented as DD ReceiverPort objects.

**ReplaceableDataItem (RDI)** A ReplaceableDataItem (RDI) object is a DD object that describes an abstract interface's basic properties such as the data type, scaling and width. It can be referenced in TargetLink block dialogs and is generated as a global [macro](#) during code generation. The definition of the RDI macro can then be generated later, allowing flexible mapping to an [implementation variable](#).

**Require calprm port** A require port in parameter communication as described by [Classic AUTOSAR](#). In the Data Dictionary, require calprm ports are represented as DD RequireCalPrmPort objects.

**RequirementInfo** An object of a DD RequirementInfo object. It describes an item of requirement information and has the following properties: Description, Document, Location, UserTag, ReferencedInCode, SimulinkStateflowPath.

**Restart function** A production code function that initializes the global variables that have an entry in the RestartfunctionName field of their [variable class](#).

**Reusable function definition** The function definition that is to be reused in the generated code. It is the code counterpart to the [Reusable system definition](#) in the model.

**Reusable function instance** An instance of a [Reusable function definition](#). It is the code counterpart to the [Reusable system instance](#) in the model.

**Reusable model part** Part of the model that can become a [Reusable system definition](#). Refer to [Basics on Function Reuse](#) ( [TargetLink Customization and Optimization Guide](#)).

**Reusable system definition** A model part to which the function reuse is applied.

**Reusable system instance** An instance of a [Reusable system definition](#).

**Root bus** A root bus is a [bus](#) that is not a subordinate part of another bus.

**Root function** A function that represents the starting point of the TargetLink-generated code. It is called from the environment in which the TargetLink-generated code is embedded.

**Root model** The topmost [parent model](#) in the system hierarchy.

**Root module** The [module](#) that contains all the code elements that belong to the [production code](#) of a [code generation unit \(CGU\)](#) and do not have their own [module specification](#).

**Root step function** A step function that is called only from outside the [production code](#). It can also represent a non-TargetLink subsystem within a TargetLink subsystem.

**Root struct** A root struct is a [struct](#) that is not a subordinate part of another struct.

**Root style sheet** A root style sheet is used to organize several style sheets defining code formatting.

**RTE event** The abbreviation of [run-time environment event](#).

**Runnable** A part of an [atomic SWC](#). With regard to code execution, a runnable is the smallest unit that can be scheduled and executed. Each runnable is implemented by one C function.

**Runnable execution constraint** Constraints that specify [runnables](#) that are allowed or not allowed to be started or stopped before a runnable.

**Runnable subsystem** An atomic subsystem that contains a Function block whose AUTOSAR mode property is set to **Classic** and whose Role is set to **Runnable**.

**Run-time environment (RTE)** A generated software layer that connects the [application layer](#) to the [basic software](#). It also interconnects the different [SWCs](#) of the application layer. There is one RTE per [ECU](#).

**Run-time environment event** A part of an [internal behavior](#). It defines the situations and conditions for starting or continuing the execution of a specific [Runnable](#).

## S

---

**Scaling** A parameter that specifies the fixed-point range and resolution of a variable. It consists of the data type, least significant bit (LSB) and offset.

**Sender port** A provide port in sender-receiver communication as described by [Classic AUTOSAR](#). In the Data Dictionary, sender ports are represented as DD SenderPort objects.

**Sender-receiver interface** An [interface](#) that describes the [data elements](#) and [event messages](#) that are provided or required by a [software component \(SWC\)](#) via a [port \(AUTOSAR\)](#).

**Sender-receiver port** A provide-require port in sender-receiver communication as described by [Classic AUTOSAR](#). In the Data Dictionary, sender-receiver ports are represented as DD SenderReceiverPort objects.

**Server port** A provide port in client-server communication as described by [Classic AUTOSAR](#). In the Data Dictionary, server ports are represented as DD ServerPort objects.

**Server runnable** A [Runnable](#) that provides an [operation](#) via a [server port](#). Server runnables are triggered by [operation invoked events](#).

**Shared parameter** A parameter for measurement and calibration that is used by several instances of the same [software component \(SWC\)](#).

**Shared variable** A variable that is accessed by several [reusable system instances](#). Typically, shared variables are used for parameters whose values are the same across instances. They increase code efficiency.

**SIC runnable function** A void (void) function that is called in a [task](#). Generated into the [Simulink implementation container \(SIC\)](#) to call the [root function](#) that is generated by TargetLink from a TargetLink subsystem. In [ConfigurationDesk](#), this function is called *Runnable function*.

**SIL simulation** A simulation method in which the control algorithm's generated [production code](#) is computed on the host computer in place of the corresponding model.

**Simple TargetLink model** A simple TargetLink model contains at least one TargetLink Subsystem block and exactly one MIL Handler block.

**Simplified initialization mode** The initialization mode used when the Simulink diagnostics parameter Underspecified initialization detection is set to **Simplified**.

See also [? classic initialization mode](#).

**Simulation application** An application that represents a graphical model specification (implemented control algorithm) and simulates its behavior in an offline Simulink environment.

**Simulation code** Code that is required only for simulation purposes. Does not belong to the [? production code](#).

**Simulation S-function** An S-function that calls either the [? root step functions](#) created for a TargetLink subsystem, or a user-specified step function (only possible in test mode via API).

**Simulink data store** Generic term for a memory region in MATLAB/Simulink that is defined by one of the following:

- A Simulink.Signal object
- A Simulink Data Store Memory block

**Simulink function call** The location in the model where a Simulink function is called. This can be:

- A Function Caller block
- The action language of a Stateflow Chart
- The MATLAB code of a MATLAB function

**Simulink function definition** The location in the model where a Simulink function is defined. This can be one of the following:

- [? Simulink Function subsystem](#)
- Exported Stateflow graphical function
- Exported Stateflow truthtable function
- Exported Stateflow MATLAB function

**Simulink function ports** The ports that can be used in a [? Simulink Function subsystem](#). These can be the following:

- TargetLink ArgIn and ArgOut blocks  
These ports are specific for each [? Simulink function call](#).
- TargetLink InPort/OutPort and Bus Import/Bus Outport blocks  
These ports are the same for all [? Simulink function calls](#).

**Simulink Function subsystem** A subsystem that contains a Trigger block whose Trigger Type is **function-call** and whose Treat as Simulink Function checkbox is selected.

**Simulink implementation container (SIC)** A file that contains all the files required to import [? production code](#) generated by TargetLink into [? ConfigurationDesk](#) as a [? behavior model](#) with [? model ports](#).

**Slice** A section of a vector or [matrix signal](#), whose elements have the same properties. If all the elements of the vector/matrix have the same properties, the whole vector/matrix forms a slice.

**Software component (SWC)** The generic term for [atomic software component \(atomic SWC\)](#), [compositions](#), and special software components, such as [calprm software components](#). A software component logically groups and encapsulates single functionalities. Software components communicate with each other via [ports](#).

**Software component description (SWC-D)** An XML file that describes [software components](#) according to AUTOSAR.

**Stateflow action language** The formal language used to describe transition actions in Stateflow.

**Struct** A struct (short form for [structure](#)) consists of subordinate [struct components](#). A struct component can be a struct itself.

**Struct component** A struct component is a part of a [struct](#) and can be a struct itself.

**Structure** A structure (long form for [struct](#)) consists of subordinate [struct components](#). A struct component can be a struct itself.

**Stub code** Code that is required to build the simulation application but that belongs to another [code generation unit \(CGU\)](#) than the one used to generate [production code](#).

**Subsystem area** A DD object which is parented by the DD root object. This object consists of an arbitrary number of Subsystem objects, each of which is the result of code generation for a specific [code generation unit \(CGU\)](#). The Subsystem objects contain detailed information on the generated code, including C modules, functions, etc. The data in this area is either automatically generated or imported from ASAM MCD-2 MC, and must not be modified manually.

**Supported Simulink block** A TargetLink-compliant block from the Simulink library that can be directly used in the model/subsystem for which the Code Generator generates [production code](#).

**SWC container** A [container](#) for files of one [SWC](#).

**Synchronous operation call subsystem** A subsystem used when modeling *synchronous* client-server communication. It is used to generate the call of the `Rte_Call` API function and for simulation purposes.

## T

---

**Table function** A function that returns table output values calculated from the table inputs.

**Target config file** An XML file named `TargetConfig.xml`. It contains information on the basic data types of the target/compiler combination such as the byte order, alignment, etc.

**Target Optimization Module (TOM)** A TargetLink software module for optimizing [production code](#) generation for a specific [microcontroller](#)/compiler combination.

**Target Simulation Module (TSM)** A TargetLink software module that provides support for a number of evaluation board/compiler combinations. It is used to test the generated code on a target processor. The TSM is licensed separately.

**TargetLink AUTOSAR Migration Tool** A software tool that converts classic, non-AUTOSAR TargetLink models to AUTOSAR models at a click.

**TargetLink AUTOSAR Module** A TargetLink software module that provides extensive support for modeling, simulating, and generating code for AUTOSAR software components.

**TargetLink Base Suite** The base component of the TargetLink software including the [ANSI C](#) Code Generator and the Data Dictionary Manager.

**TargetLink base type** One of the types used by TargetLink instead of pure C types in the generated code and the delivered libraries. This makes the code platform independent.

**TargetLink Blockset** A set of blocks in TargetLink that allow [production code](#) to be generated from a model in MATLAB/Simulink.

**TargetLink Data Dictionary** The central data container that holds all relevant information about an ECU application, for example, for code generation.

**TargetLink simulation block** A block that processes signals during simulation. In most cases, it is a block from standard Simulink libraries but carries additional information required for production code generation.

**TargetLink subsystem** A subsystem from the TargetLink block library that defines a section of the Simulink model for which code must be generated by TargetLink.

**Task** A code section whose execution is managed by the real-time operating system. Tasks can be triggered periodically or based on events. Each task can call one or more [SIC runnable functions](#).

**Task function** A function that implements a task and calls the functions of the subsystems which are assigned to the task by the user or via the TargetLink Code Generator during multirate code generation.

**Term function** A function that contains the code to be executed when the simulation finishes or the ECU application terminates.

**Terminate function** A [Runnable](#) that finalizes a [SWC](#), for example, by calling code that has to run before the application shuts down.

**Timing event** An [RTE event](#) that specifies to start or continue the execution of a [Runnable](#) at constant time intervals.

**tllib** A TargetLink block library that is the source for creating TargetLink models graphically. Refer to [How to Open the TargetLink Block Library](#) ([TargetLink Orientation and Overview Guide](#)).

**Transformer** The [Classic AUTOSAR](#) entity used to perform a [data transformation](#).

**TransformerError** The parameter passed by the [run-time environment \(RTE\)](#) if an error occurred in a [data transformation](#). The `Std_TransformerError` is a struct whose components are the transformer class and the error code. If the error is a hard error, a special runnable is triggered via the [TransformerHardErrorEvent](#) to react to the error.

In AUTOSAR releases prior to R19-11 this struct was named `Rte_TransformerError`.

**TransformerHardErrorEvent** The [RTE event](#) that triggers the [Runnable](#) to be used for responding to a hard [TransformerError](#) in a [data transformation](#) for client-server communication.

**Type prefix** A string written in front of the variable type of a variable definition/declaration, such as `MyTypePrefix Int16 MyVar`.

## U

---

**Unicode** The most common standard for extended character sets is the Unicode standard. There are different schemes to encode Unicode in byte format, e.g., UTF-8 or UTF-16. All of these encodings support all Unicode characters. Scheme conversion is possible without losses. The only difference between these encoding schemes is the memory that is required to represent Unicode characters.

**User data type (UDT)** A data type defined by the user. It is placed in the Data Dictionary and can have associated constraints.

**Utility blocks** One of the categories of TargetLink blocks. The blocks in the category keep TargetLink-specific data, provide user interfaces, and control the simulation mode and code generation.

## V

---

**Validation Summary** Shows unresolved model element data validation errors from all model element variables of the Property View. It lets you search, filter, and group validation errors.

**Value copy AF** An [access function \(AF\)](#) resulting from DD AccessFunction objects whose AccessFunctionKind property is set to READ\_VALUE\_COPY or WRITE\_VALUE\_COPY.

**Variable access function** An [access function \(AF\)](#) that *encapsulates the* access to a variable for reading or writing.

**Variable class** A set of properties that define the role and appearance of a variable in the generated [production code](#), e.g. CAL for global calibratable variables.

**VariantConfig** A DD object in the [Config area](#) that defines the [code variants](#) and [data variants](#) to be used for simulation and code generation.

**VariantItem** A DD object in the DD [Config area](#) used to variant individual properties of DD Variable and [ExchangeableWidth](#) objects. Each variant of a property is associated with one variant item.

**V-ECU implementation container (VECU)** A file that consists of all the files required to build an [offline simulation application \(OSA\)](#) to use for simulation with VEOS.

**V-ECU Manager** A component of TargetLink that allows you to configure and generate a V-ECU implementation.

**Vendor mode** The operation mode of RTE generators that allows the generation of RTE code which contains vendor-specific adaptations, e.g., to reduce resource consumption. To be linkable to an RTE, the object code of an SWC must have been compiled against an application header that matches the RTE code generated by the specific RTE generator. This is the case because the data structures and types can be implementation-specific.

See also [compatibility mode](#).

**VEOS** A dSPACE software platform for the C-code-based simulation of [virtual ECUs](#) and environment models on a PC.

**Virtual ECU (V-ECU)** Software that emulates a real [ECU](#) in a simulation scenario. The virtual ECU comprises components from the application and the [basic software](#), and provides functionalities comparable to those of a real ECU.

**Virtual ECU testing** Offline and real-time simulation using [virtual ECUs](#).

**Virtual evaluation board (virtual EVB)** A combination of an [instruction set simulator \(ISS\)](#) and a simulated periphery. This combination can be used for validation of generated [production code](#) in [PIL simulation](#) mode.

---

**Worst-case range limits** A range specified by calculating the minimum and maximum values which a block's output or state variable can take on with respect to the range of the inputs or the user-specified [constrained range limits](#).



# Appendix

---

## Where to go from here

## Information in this section

Comprehensive Modeling of RTE Code Pattern.....	374
AUTOSAR-Related Code Generator Options.....	387

# Comprehensive Modeling of RTE Code Pattern

## Where to go from here

## Information in this section

Overview of RTE API Functions.....	374
Calibration and Measurement.....	375
Client-Server Communication.....	376
Interrunnable Communication.....	377
Mode management.....	380
NvData Communication (Explicit).....	380
NvData Communication (Implicit).....	381
Per Instance Memory.....	383
Sender-Receiver Communication (Explicit).....	383
Sender-Receiver Communication (Implicit).....	385

## Overview of RTE API Functions

### Overview

RTE API Function	Restricted to ClassicAUTOSAR Release	Supported by TargetLink
Rte_Call	-	Client-Server Communication on page 376
Rte_CData	-	Calibration and Measurement on page 375
Rte_DRead	-	-
Rte_Enter	-	- <sup>1)</sup>
Rte_Exit	-	- <sup>1)</sup>
Rte_Feedback	-	Sender-Receiver Communication (Explicit) on page 383
Rte_IFeedback	-	-
Rte_IInvalidate	-	Sender-Receiver Communication (Implicit) on page 385
Rte_Invalidate	-	Sender-Receiver Communication (Explicit) on page 383
Rte_IRead	-	NvData Communication (Implicit) on page 381 Sender-Receiver Communication (Implicit) on page 385
Rte_IrTrigger	4.x	-
Rte_IrvIRead	-	Interrunnable Communication on page 377
Rte_IrvIWrite	-	Interrunnable Communication on page 377

RTE API Function	Restricted to ClassicAUTOSAR Release	Supported by TargetLink
Rte_IrvIWriteRef	≥ 4.2.2	<a href="#">Interrunnable Communication on page 377</a>
Rte_IrvRead	-	<a href="#">Interrunnable Communication on page 377</a>
Rte_IrvWrite	-	<a href="#">Interrunnable Communication on page 377</a>
Rte_IStatus	-	<a href="#">Sender-Receiver Communication (Implicit) on page 385</a>
Rte_IsUpdated	≥ 3.2.2	<a href="#">Sender-Receiver Communication (Explicit) on page 383</a>
Rte_IWrite	-	<a href="#">Sender-Receiver Communication (Implicit) on page 385</a> <a href="#">NvData Communication (Implicit) on page 381</a>
Rte_IWriteRef	-	<a href="#">NvData Communication (Implicit) on page 381</a> <a href="#">Sender-Receiver Communication (Implicit) on page 385</a>
Rte_Mode	-	<a href="#">Mode management on page 380</a>
Rte_NPorts	-	-
Rte_Pim	-	<a href="#">Per Instance Memory on page 383</a>
Rte_Port	-	-
Rte_Ports	-	-
Rte_Prm	4.x	<a href="#">Calibration and Measurement on page 375</a>
Rte_Read	-	<a href="#">Sender-Receiver Communication (Explicit) on page 383</a> <a href="#">NvData Communication (Explicit) on page 380</a>
Rte_Receive	-	<a href="#">Sender-Receiver Communication (Explicit) on page 383</a>
Rte_Result	-	<a href="#">Client-Server Communication on page 376</a>
Rte_Send	-	<a href="#">Sender-Receiver Communication (Explicit) on page 383</a>
Rte_Switch	-	<a href="#">Mode management on page 380</a>
Rte_SwitchAck	4.x	-
Rte_Trigger	4.x	-
Rte_Write	-	<a href="#">Sender-Receiver Communication (Explicit) on page 383</a> <a href="#">NvData Communication (Explicit) on page 380</a>

<sup>1)</sup> Instead of modeling runnables that can enter exclusive areas, you can model runnables that run in exclusive areas with TargetLink. The RTE API functions can then be generated by an RTE code generator.

## Calibration and Measurement

### Supported RTE API Functions

TargetLink supports the following RTE API functions for calibration and measurement:

RTE API Function	Purpose	Modeling in TargetLink
Rte_CData	To access SWC-internal calibration parameters.	<ul style="list-style-type: none"> <li>▪ <a href="#">How to Model Per Instance Parameters for Calibration on page 239</a></li> <li>▪ <a href="#">How to Model Shared Parameters for Calibration on page 241</a></li> </ul>

RTE API Function	Purpose	Modeling in TargetLink
Rte_Prm	To access calibration parameters that are provided via ports and interfaces of an SWC of the ParameterSwComponentType type.	<a href="#">How to Model Parameter Communication for Calibration</a> on page 244

**Related topics****Basics**

[Preparing SWCs for Measurement and Calibration.....](#) 235

## Client-Server Communication

**Supported RTE API Functions**

TargetLink supports the following RTE API functions for client-server communication:

RTE API Function	Purpose	Modeling in TargetLink
Rte_Call	To call a server operation. This API function is used for synchronous and asynchronous client-server communication.	<ul style="list-style-type: none"> <li>▪ <a href="#">How to Model Operation Calls via Synchronous Operation Call Subsystems</a> on page 144</li> <li>▪ <a href="#">How to Model Operation Calls via Asynchronous Operation Call Subsystems</a> on page 146</li> <li>▪ <a href="#">How to Combine a Synchronous Operation Call with a Server Operation's Implementation</a> on page 157</li> <li>▪ <a href="#">How to Model a Unidirectional Get or Set Operation in Synchronous Client-Server Communication via a Port Block</a> on page 141</li> </ul>
Rte_Result	To get the result of a server operation. This API function is used for asynchronous client-server communication.	<ul style="list-style-type: none"> <li>▪ <a href="#">How to Model Operation Results via Operation Result Provider Subsystems</a> on page 149</li> </ul>

**Related topics****Basics**

[Modeling Client-Server Communication.....](#) 133

## Interrunnable Communication

### Supported RTE API Functions

TargetLink supports the following RTE API functions for interruptible communication:

RTE API Function	Purpose	Modeling in TargetLink
Rte_IrvIRead	To read implicitly from an interruptible variable that is shared by different runnables of the same SWC.	<p>1. Create a DD InterRunnableVariable object in the following DD subtree:  <code>/Pool/Autosar/SoftwareComponents/&lt;SoftwareComponent&gt;/InterRunnableVariables/</code></p> <p>2. Set the CommunicationMode property to <b>Implicit</b>.</p> <p>3. Create a DD Typedef object that meets your requirements and reference it at the DD InterRunnableVariable created in step 1.</p> <p>4. Optionally, specify an initialization value by referencing a DD Variable object via the InterRunnableVariable object's InitValueRef property.</p> <p><b>Note</b></p> <p><b>Avoid ambiguous initializations of data prototypes</b></p> <p>Specify an initialization value for each <a href="#">data prototype</a> within the Data Dictionary whenever possible. This avoids ambiguities between <a href="#">Classic AUTOSAR</a> initialization values and initialization values of model elements that are required by Simulink.</p> <p>5. Reference the DD InterRunnableVariable object at an InPort, Bus Import, or Data Store Memory block.  If you used a Data Store Memory block, add a Data Store Read block to the model that uses the same data store and connect it as required.</p>
Rte_IrvIWrite <sup>1)</sup>	To write implicitly to an interruptible variable that is shared by different runnables of the same SWC.	<p>1. Create a DD InterRunnableVariable object in the following DD subtree:  <code>/Pool/Autosar/SoftwareComponents/&lt;SoftwareComponent&gt;/InterRunnableVariables/</code></p> <p>2. Set the CommunicationMode property to <b>Implicit</b>.</p> <p>3. Create a DD Typedef object that meets your requirements and reference it at the DD InterRunnableVariable created in step 1.</p> <p>4. Optionally, specify an initialization value by referencing a DD Variable object via the InterRunnableVariable object's InitValueRef property.</p> <p><b>Note</b></p> <p><b>Avoid ambiguous initializations of data prototypes</b></p> <p>Specify an initialization value for each <a href="#">data prototype</a> within the Data Dictionary whenever possible. This avoids ambiguities between <a href="#">Classic AUTOSAR</a> initialization values and initialization values of model elements that are required by Simulink.</p> <p>5. Reference the DD InterRunnableVariable object at an OutPort, Bus Outport, or Data Store Memory block.  If you used a Data Store Memory block, add a Data Store Write block to the model that uses the same data store and connect it as required.</p>

RTE API Function	Purpose	Modeling in TargetLink
Rte_IrvIWriteRef <sup>2</sup>	To write implicitly to a <i>non-scalar</i> interruptable variable that is shared by different runnables of the same SWC.	<p>1. Create a DD InterRunnableVariable object in the following DD subtree:  <code>/Pool/Autosar/SoftwareComponents/&lt;SoftwareComponent&gt;/InterRunnableVariables/</code></p> <p>2. Set the CommunicationMode property to <b>Implicit</b>.</p> <p>3. Create a DD Typedef object that is non-scalar and reference it at the DD InterRunnableVariable created in step 1.</p> <p>4. Optionally, specify an initialization value by referencing a DD Variable object via the InterRunnableVariable object's InitValueRef property.</p> <p><b>Note</b></p> <p><b>Avoid ambiguous initializations of data prototypes</b></p> <p>Specify an initialization value for each <a href="#">data prototype</a> within the Data Dictionary whenever possible. This avoids ambiguities between <a href="#">Classic AUTOSAR</a> initialization values and initialization values of model elements that are required by Simulink.</p> <p>5. Reference the DD InterRunnableVariable object at an OutPort, Bus Outport, or Data Store Memory block.  If you used a Data Store Memory block, add a Data Store Write block to the model that uses the same data store and connect it as required.</p>
Rte_IrvRead	To read explicitly from an interruptable variable that is shared by different runnables of the same SWC.	<p>1. Create a DD InterRunnableVariable object in the following DD subtree:  <code>/Pool/Autosar/SoftwareComponents/&lt;SoftwareComponent&gt;/InterRunnableVariables/</code></p> <p>2. Set the CommunicationMode to <b>Explicit</b>.</p> <p>3. Create a DD Typedef object that meets your requirements and reference it at the DD InterRunnableVariable created in step 1.</p> <p>4. Optionally, specify an initialization value by referencing a DD Variable object via the InterRunnableVariable object's InitValueRef property.</p> <p><b>Note</b></p> <p><b>Avoid ambiguous initializations of data prototypes</b></p> <p>Specify an initialization value for each <a href="#">data prototype</a> within the Data Dictionary whenever possible. This avoids ambiguities between <a href="#">Classic AUTOSAR</a> initialization values and initialization values of model elements that are required by Simulink.</p> <p>5. Reference the DD InterRunnableVariable object at an InPort, Bus Import, or Data Store Memory block.  If you used a Data Store Memory block, add a Data Store Read block to the model that uses the same data store and connect it as required.</p>

RTE API Function	Purpose	Modeling in TargetLink
Rte_IrvWrite	To write explicitly to an interruptible variable that is shared by different runnables of the same SWC.	<p>1. Create a DD InterRunnableVariable object in the following DD subtree:  <code>/Pool/Autosar/SoftwareComponents/&lt;SoftwareComponent&gt;/InterRunnableVariables/</code></p> <p>2. Set the CommunicationMode to <b>Explicit</b>.</p> <p>3. Create a DD Typedef object that meets your requirements and reference it at the DD InterRunnableVariable created in step 1.</p> <p>4. Optionally, specify an initialization value by referencing a DD Variable object via the InterRunnableVariable object's InitValueRef property.</p> <p><b>Note</b></p> <p><b>Avoid ambiguous initializations of data prototypes</b></p> <p>Specify an initialization value for each <a href="#">data prototype</a> within the Data Dictionary whenever possible. This avoids ambiguities between <a href="#">Classic AUTOSAR</a> initialization values and initialization values of model elements that are required by Simulink.</p> <p>5. Reference the DD InterRunnableVariable object at an OutPort, Bus Outport, or Data Store Memory block.  If you used a Data Store Memory block, add a Data Store Write block to the model that uses the same data store and connect it as required.</p>

<sup>1)</sup> For non-scalar interruptible variables, TargetLink generates calls to this API function only if the GenerateIWriteForNonScalarIrvs Code Generator option is set to **on**.

<sup>2)</sup> For non-scalar interruptible variables, TargetLink does not generate calls to this API function if the GenerateIWriteForNonScalarIrvs Code Generator option is set to **on**.

## Related topics

### Basics

[Modeling Interruptible Communication](#)..... 171

### References

[GenerateIWriteForNonScalarIrvs](#)..... 390

## Mode management

### Supported RTE API Functions

TargetLink supports the following RTE API functions for mode management:

RTE API Function	Purpose	Modeling in TargetLink
Rte_Mode	To get the active mode. ② <a href="#">Classic AUTOSAR 4.x</a> enhanced feature: If the mode machine instance is in transition the values of the previous and the next mode are also provided. <sup>1)</sup>	<ol style="list-style-type: none"> <li>1. Create a DD ModeSwitchInterface object with a DD ModeElement object that has a reference to a DD ModeDeclarationGroup object.</li> <li>2. In the DD, create a ModeReceiverPort and reference the DD ModeSwitchInterface object.</li> <li>3. In the TargetLink model, select the DD ModeReceiverPort and ModeElement object at an InPort block.</li> </ol>
Rte_Switch	To request a mode switch.	<ol style="list-style-type: none"> <li>1. Create a DD ModeSwitchInterface object with a DD ModeElement object that has a reference to a DD ModeDeclarationGroup object.</li> <li>2. Create a DD ModeSenderPort object and reference the DD ModeSwitchInterface.</li> <li>3. In the TargetLink model, select the DD ModeSenderPort object and the DD ModeElement object at an OutPort block.</li> </ol>

<sup>1)</sup> TargetLink does not support the command's enhanced feature as introduced with ② [Classic AUTOSAR 4.x](#).

### Related topics

#### Basics

[Modeling Modes.....](#) 217

## NvData Communication (Explicit)

### Supported RTE API Functions

TargetLink supports the following RTE API functions for ② [explicit NvData](#) communication:

RTE API Function	Purpose	Modeling in TargetLink
Rte_Read	To read an NvData element with explicit data semantics from the NvBlockSwComponent.	<ol style="list-style-type: none"> <li>1. Create a DD NvDataInterface object with at least one DD NvDataElement object.</li> <li>2. Create a DD NvReceiverPort or NvSenderReceiverPort object and reference the DD NvDataInterface object.</li> <li>3. In the TargetLink model, select the DD NvReceiverPort or NvSenderReceiverPort object and a DD NvDataElement object at one of the following objects: <ul style="list-style-type: none"> <li>▪ InPort or Bus Inport</li> <li>▪ Data Store Memory</li> </ul> </li> <li>4. On the AUTOSAR page of the block dialog, set Mode to <b>Explicit</b>.</li> <li>5. If you used the Data Store Memory block, add a corresponding Data Store Read block to generate the RTE API call in production code.</li> </ol>

RTE API Function	Purpose	Modeling in TargetLink
Rte_Write	To write an NvData element with explicit data semantics to the NvBlockSwComponent.	<ol style="list-style-type: none"> <li>1. Create a DD NvDataInterface object with at least one DD NvDataElement object.</li> <li>2. Create a DD NvSenderPort or NvSenderReceiverPort object and reference the DD NvDataInterface object.</li> <li>3. In the TargetLink model, select the DD NvSenderPort or NvSenderReceiverPort object and a DD NvDataElement object at one of the following objects: <ul style="list-style-type: none"> <li>▪ OutPort or Bus Outport</li> <li>▪ Data Store Memory</li> </ul> </li> <li>4. On the AUTOSAR page of the block dialog, set Mode to <b>Explicit</b>.</li> <li>5. If you used the Data Store Memory block, add a corresponding Data Store Write block to generate the RTE API call in production code.</li> </ol>

**Related topics****HowTos**

Modeling NvData Communication..... 178

## NvData Communication (Implicit)

**Supported RTE API Functions**

TargetLink supports the following RTE API functions for [implicit](#) NvData communication:

RTE API Function	Purpose	Modeling in TargetLink
Rte_IRead	To read an NvData element with implicit data semantics from the NvBlockSwComponent.	<ol style="list-style-type: none"> <li>1. Create a DD NvDataInterface object with at least one DD NvDataElement object.</li> <li>2. Create a DD NvReceiverPort or NvSenderReceiverPort object and reference the DD NvDataInterface object.</li> <li>3. In the TargetLink model, select the DD NvReceiverPort or NvSenderReceiverPort object and a DD NvDataElement object at one of the following objects: <ul style="list-style-type: none"> <li>▪ InPort or Bus Import</li> <li>▪ Data Store Memory</li> </ul> </li> <li>4. On the AUTOSAR page of the block dialog, set Mode to <b>Implicit</b>.</li> <li>5. If you used the Data Store Memory block, add a corresponding Data Store Read block to generate the RTE API call in production code.</li> </ol> <p>Instead of step 3 and 4 you can create a variable via the DD NvDataInterface object's Synchronize Interface Settings context menu command and reference it as a block parameter at one or more supported blocks.</p>

RTE API Function	Purpose	Modeling in TargetLink
Rte_IWrite	To access a data element that is to be written to the NvBlockSwComponent.	<ol style="list-style-type: none"> <li>1. Create a DD NvDataInterface object with at least one DD NvDataElement object.</li> <li>2. In the DD, create a DD NvSenderPort or NvSenderReceiverPort object and reference the DD NvDataInterface object.</li> <li>3. In the TargetLink model, select the DD NvSenderPort or NvSenderReceiverPort object and a DD NvDataElement object at one of the following objects: <ul style="list-style-type: none"> <li>▪ OutPort or Bus Outport</li> <li>▪ Data Store Memory</li> </ul> </li> <li>4. On the AUTOSAR page of the block dialog, set Mode to <b>Implicit (IWrite)</b>.</li> <li>5. If you used the Data Store Memory block, add a corresponding Data Store Write block to generate the RTE API call in production code.</li> </ol>
Rte_IWriteRef	To access a data element that is to be written to the NvBlockSwComponent.	<ol style="list-style-type: none"> <li>1. Create a DD NvDataInterface object with at least one DD NvDataElement object.</li> <li>2. Create a DD NvSenderPort or NvSenderReceiverPort object and reference the DD NvDataInterface object.</li> <li>3. In the TargetLink model, select the DD NvSenderPort or NvSenderReceiverPort object and a DD NvDataElement object at one of the following objects: <ul style="list-style-type: none"> <li>▪ OutPort or Bus Outport</li> <li>▪ Data Store Memory</li> </ul> </li> <li>4. On the AUTOSAR page of the block dialog, set Mode to <b>Implicit (IWriteRef)</b>.</li> <li>5. If you used the Data Store Memory block, add a corresponding Data Store Write block to generate the RTE API call in production code.</li> </ol>

**Related topics****HowTos**

[Modeling NvData Communication.....](#) 178

**References**

[Synchronize Interface Settings \(TargetLink Data Dictionary Manager Reference\)](#)

## Per Instance Memory

**Supported RTE API Functions** TargetLink supports the following RTE API functions for per instance memories:

RTE API Function	Purpose	Modeling in TargetLink
Rte_Pim	To access a per instance memory variable.	<ol style="list-style-type: none"> <li>1. Create a DD VariableGroup object, name it as required, and reference it at the DD SoftwareComponent object's RelatedVariables child object via the PerInstanceMemoriesRef property.</li> <li>2. From the context menu of the DD VariableGroup object, select AUTOSAR - Create PIMVariable and rename the created DD Variable object as required.</li> <li>3. Specify the DD Variable object's Type property as required.</li> <li>4. In the TargetLink model, select the DD Variable object as a parameter at a block such as a Gain block.</li> </ol>

### Related topics

#### Basics

[Modeling Per Instance Memories.....](#) 213

## Sender-Receiver Communication (Explicit)

**Supported RTE API Functions** TargetLink supports the following RTE API functions for explicit sender-receiver communication:

RTE API Function	Purpose	Modeling in TargetLink
Rte_Feedback	To get an acknowledgement notification for explicit sender-receiver communication.	<ol style="list-style-type: none"> <li>1. Model explicit sender communication with data/event semantics (<a href="#">Rte_Write/Rte_Send</a>).</li> <li>2. At a DD SenderPort/SenderReceiverPort object, create a DataSenderComSpec/EventSenderComSpec object and an Acknowledgment object. Specify the acknowledgment's Timeout property.</li> <li>3. In the TargetLink model, place a SenderComSpec block before the port block that transfers the data element and connect it.</li> <li>4. Connect the Feedback port of the SenderComSpec block as required.</li> </ol>
Rte_Invalidate	To invalidate a data element instead of writing a data element with <a href="#">data semantics</a> ( <a href="#">Rte_Write</a> ). The receiver can be configured to either keep the last value or replace the last value by the initial value.	<ol style="list-style-type: none"> <li>1. Model explicit sender-receiver communication with data semantics (<a href="#">Rte_Write</a>).</li> <li>2. At the DD SenderPort/SenderReceiverPort object create a DataSenderComSpec object, enable its CanInvalidate property, and reference DD DataElement object.</li> <li>3. In the TargetLink model, place a SenderComSpec block before the port block that transfers the data element and connect it.</li> <li>4. Connect the Invalidate port of the SenderComSpec block as required.</li> </ol>

RTE API Function	Purpose	Modeling in TargetLink
Rte_IsUpdated	To provide access to the update flag of an explicit receiver.	<ol style="list-style-type: none"> <li>1. Model explicit sender communication with data semantics.</li> <li>2. At a DD ReceiverPort/SenderReceiverPort object, create a DataReceiverComSpec object.</li> <li>3. At the DataReceiverComSpec object, reference the DD DataElement object and set its EnableUpdate property to <b>on</b>.</li> <li>4. To the <a href="#">runnable subsystem</a>, add a Data Store Memory and a Data Store Read block.</li> <li>5. On the AUTOSAR page of the Data Store Memory block's dialog, set AUTOSAR mode to <b>Classic</b>, set Kind to <b>DataElementUpdated</b>, and reference the DD ReceiverPort and DD DataElement objects whose update flag you want to check.</li> </ol>
Rte_Read	To read a data element with <a href="#">data semantics</a> .	<ol style="list-style-type: none"> <li>1. Create a DD SenderReceiverInterface object with one or more DD DataElement objects.</li> <li>2. Create a DD ReceiverPort or SenderReceiverPort object and reference the DD SenderReceiverInterface object.</li> <li>3. In the TargetLink model, select the DD ReceiverPort/SenderReceiverPort object and a DD DataElement object at an InPort/Bus Import/Data Store Memory block.</li> <li>4. On the AUTOSAR page of the block dialog, set Mode to <b>Explicit</b>.</li> <li>5. If you used the Data Store Memory block, add a corresponding Data Store Read block to generate the RTE API call in production code.</li> </ol>
Rte_Receive	To receive a data element with <a href="#">event semantics</a> .	<ol style="list-style-type: none"> <li>1. Create a DD SenderReceiverInterface object with one or more DD DataElement objects.</li> <li>2. Set the DD DataElement object's ImplementationPolicy property to <b>queued</b>.</li> <li>3. Create a DD ReceiverPort object and reference the DD SenderReceiverInterface object.</li> <li>4. In the TargetLink model, select the DD ReceiverPort object and a DD DataElement object at an InPort/Bus Import/Data Store Memory block.</li> <li>5. On the AUTOSAR page of the block dialog, set Mode to <b>Explicit</b>.</li> <li>6. If you used the Data Store Memory block, add a corresponding Data Store Read block to generate the RTE API call in production code.</li> </ol>
Rte_Send	To send a data element with <a href="#">event semantics</a> to another SWC.	<ol style="list-style-type: none"> <li>1. Create a DD SenderReceiverInterface object with one or more DD DataElement objects.</li> <li>2. Set the DD DataElement object's ImplementationPolicy property to <b>queued</b>.</li> <li>3. Create a DD SenderPort object and reference the DD SenderReceiverInterface object.</li> <li>4. In the TargetLink model, select the DD SenderPort object and a DD DataElement object at an OutPort/Bus Outport/Data Store Memory block.</li> <li>5. On the AUTOSAR page of the block dialog, set Mode to <b>Explicit</b>.</li> <li>6. If you used the Data Store Memory block, add a corresponding Data Store Write block to generate the RTE API call in production code.</li> </ol>
Rte_Write	To write a data element with <a href="#">data semantics</a> .	<ol style="list-style-type: none"> <li>1. Create a DD SenderReceiverInterface object with one or more DD DataElement objects.</li> <li>2. Create a DD SenderPort or SenderReceiverPort object and reference the DD SenderReceiverInterface object.</li> <li>3. In the TargetLink model, select the DD SenderPort/SenderReceiverPort object and a DD DataElement object at an OutPort/Bus Outport/Data Store Memory block.</li> <li>4. On the AUTOSAR page of the block dialog, set Mode to <b>Explicit</b>.</li> <li>5. If you used the Data Store Memory block, add a corresponding Data Store Write block to generate the RTE API call in production code.</li> </ol>

**Related topics****Basics**

[Modeling Sender-Receiver Communication.....](#) 108

## Sender-Receiver Communication (Implicit)

**Supported RTE API Functions**

TargetLink supports the following RTE API functions for implicit sender-receiver communication:

RTE API Function	Purpose	Modeling in TargetLink
<code>Rte_IInvalidate</code>	To invalidate a data element instead of writing a data element with <a href="#">data semantics</a> ( <code>Rte_IWrite/Rte_IWriteRef</code> ). The receiver can be configured to either keep the last value or replace the last value by the initial value.	<ol style="list-style-type: none"> <li>1. Model implicit sender-receiver communication with data semantics (<code>Rte_IWrite/Rte_IWriteRef</code>).</li> <li>2. At a DD SenderPort/SenderReceiverPort object, create a DD DataSenderComSpec object and enable its CanInvalidate property.</li> <li>3. In the TargetLink model, place a SenderComSpec block before the port block that transfers the data element.</li> <li>4. Connect the Invalidate port of the SenderComSpec block as required.</li> </ol>
<code>Rte_IRead</code>	To read a data element with <a href="#">data semantics</a> .	<ol style="list-style-type: none"> <li>1. Create a DD SenderReceiverInterface object with one or more DD DataElement objects.</li> <li>2. Create a DD ReceiverPort/SenderReceiverPort object and reference the DD SenderReceiverInterface object.</li> <li>3. In the TargetLink model, select the DD ReceiverPort/SenderReceiverPort object and a DD DataElement object at an InPort/Bus Import/Data Store Memory block.</li> <li>4. On the AUTOSAR page of the block dialog, set Mode to <b>Implicit (IWrite)</b> or <b>Implicit (IWriteRef)</b>.</li> <li>5. If you used the Data Store Memory block, add a corresponding Data Store Write block to generate the RTE API call in production code.</li> </ol>
<code>Rte_IStatus</code>	To provide access to the RTE status of a data element that is to be read via <code>Rte_IRead</code> . Additionally, it provides the transformer error, if specified.	<ol style="list-style-type: none"> <li>1. Model implicit sender-receiver communication with data semantics (<code>Rte_IRead</code>).</li> <li>2. At a DD ReceiverPort/SenderReceiverPort object, create a DD DataReceiverComSpec object.</li> <li>3. At the DD DataReceiverComSpec object, specify at least one of the following properties: <ul style="list-style-type: none"> <li>▪ Set AliveTimeOut to a value that is greater than zero.</li> <li>▪ Set HandleInvalid to KEEP.</li> <li>▪ Set EnableNeverReceived to on.</li> <li>▪ Set HandleDataStatus to on.</li> </ul> </li> <li>4. In the TargetLink model, place a ReceiverComSpec block after the port block that transfers the data element and connect it.</li> <li>5. Connect the RTE_Status or Trafo Error port of the ReceiverComSpec block as required.</li> </ol>

RTE API Function	Purpose	Modeling in TargetLink
Rte_IWrite	To write a data element with <a href="#">data semantics</a> .	<ol style="list-style-type: none"> <li>1. Create a DD SenderReceiverInterface object with one or more DD DataElement objects.</li> <li>2. Create a DD SenderPort/SenderReceiverPort object and reference the DD SenderReceiverInterface object.</li> <li>3. In the TargetLink model, select the DD SenderPort/SenderReceiverPort object and a DD DataElement object at an OutPort/Bus Outport/Data Store Memory block.</li> <li>4. On the AUTOSAR page of the block dialog, set Mode to <b>Implicit (IWrite)</b>.</li> <li>5. If you used the Data Store Memory block, add a corresponding Data Store Write block to generate the RTE API call in production code.</li> </ol>
Rte_IwriteRef	To write a data element with <a href="#">data semantics</a> .	<ol style="list-style-type: none"> <li>1. Create a DD SenderReceiverInterface object with one or more DD DataElement objects.</li> <li>2. Create a DD SenderPort/SenderReceiverPort object and reference the DD SenderReceiverInterface object.</li> <li>3. In the TargetLink model, select the DD SenderPort/SenderReceiverPort object and a DD DataElement object at an OutPort/Bus Outport/Data Store Memory block.</li> <li>4. On the AUTOSAR page of the block dialog, set Mode to <b>Implicit (IWriteRef)</b>.</li> <li>5. If you used the Data Store Memory block, add a corresponding Data Store Write block to generate the RTE API call in production code.</li> </ol>

**Related topics****Basics**

[Modeling Sender-Receiver Communication.....](#) 108

# AUTOSAR-Related Code Generator Options

## Where to go from here

## Information in this section

AllowDuplicationOfImplicitAUTOSARDATAccess.....	387
AssumeFunctionCallSemanticsForRteAPIArguments.....	388
AssumeOperationCallsHaveNoUnknownDataFlow.....	389
GenerateIWriteForNonScalarIrvs.....	390
ARDATAPrototypeActualParamName.....	391
StrictRunnableInterfaceChecks.....	392
SuppressNoRunnableRestartCodeError.....	392

## AllowDuplicationOfImplicitAUTOSARDATAccess

### Purpose

For optimization purposes, treat implicit-reading AUTOSAR data access API calls like reading from a variable.

### Description

The RTE API is implemented by an RTE code generator, for example, SystemDesk, and not by TargetLink.

E.g. "Rte\_IRead\_<RunnableName>\_<ReceiverPortName>\_<DataElementName>()" can be implemented by the RTE as a macro or as a function.

If an implicit-reading AUTOSAR data access is implemented by a macro, then other variables might be replaced by this macro.

In this case, the API call can be used wherever this variable's value is needed. This can lead to the generation of code that is more efficient.

Example:

```
a = Rte_IRead_MyRunnable_MyPort_A();
if (a <= b) {
    c = a;
} else {
    ....
}
.... /* hundreds of lines of code */
d = x;
x = a;
```

is not modified if "AllowDuplicationOfImplicitAUTOSARDataAccess"='off' but can become

```
if (Rte_IRead_MyRunnable_MyPort_A() <= b) {
    c = Rte_IRead_MyRunnable_MyPort_A();
} else {
    ....
}
.... /* hundreds of Lines of code */
d = x;
x = Rte_IRead_MyRunnable_MyPort_A();
```

if "AllowDuplicationOfImplicitAUTOSARDataAccess"='on'.

If the implicit-reading AUTOSAR data access API call is implemented as a function or as a more complex macro, then multiple identical API calls are less efficient than storing the desired value in a local variable and reading from this variable. In this case TargetLink may not perform the described optimization.

Data type	Numeric ID - Literal ID
0 - off	
1 - on (default)	

## AssumeFunctionCallSemanticsForRteAPIArguments

Purpose	Determines the allowed contents of RTE API call argument lists.
Description	<p>The RTE API is implemented by an RTE code generator, for example SystemDesk, and not by TargetLink.</p> <p>E.g. "Rte_IWrt..._&lt;RunnableName&gt;_&lt;SenderPortName&gt;_&lt;DataElementName&gt;()" can be implemented by the RTE as a macro or as a function.</p> <p>If it is implemented as a macro, then after macro replacement the argument of the RTE API call might be used more than once.</p> <p>In this case, the argument of the call must not have side effects (e.g. a call to a function with an internal state) and should not contain expensive operations.</p> <p>If it is implemented as a function or as a macro that evaluates its arguments only once, then treating the call like a function call can lead to the generation of more efficient code.</p> <p>Switching on "AssumeFunctionCallSemanticsForRteAPIArguments" enables TargetLink to work under the latter assumption.</p>

Example:

```
c = (a + b) * 5;
d = (uint16) c;
Rte_IWrite_MyRunnable_MyPort_D(d);
```

can become

```
d = (uint16) ((a + b) * 5);
Rte_IWrite_MyRunnable_MyPort_D(d);
```

or

```
c = (a + b) * 5;
Rte_IWrite_MyRunnable_MyPort_D((uint16) c);
```

if "AssumeFunctionCallSemanticsForRteAPIArguments"='off' and

```
Rte_IWrite_MyRunnable_MyPort_D((uint16) ((a + b) * 5));
```

if "AssumeFunctionCallSemanticsForRteAPIArguments"='on'.

Data type	Numeric ID - Literal ID
0	- off
1	- on (default)

## AssumeOperationCallsHaveNoUnknownDataFlow

Purpose	For optimization purposes, treat every AUTOSAR operation in the same way as if the Data Dictionary Operation object has the NoDataFlowWithOtherOperations property set (or if the underlying function has a function class with the SIDE_EFFECT_FREE Optimization property set).
---------	--

Description	<p>Generally, the implementation of an interaction between AUTOSAR operation calls and/or operation results is unknown. For other functions, TargetLink lets you specify the function's behavior via the function class Optimization property:</p> <ul style="list-style-type: none"> <li>▪ "There is no unknown data flow via global variables, apart from the interface defined in the model" (SIDE_EFFECT_FREE): Accesses to global variables and function calls accessing global variables can be moved past a call of the respective function.</li> </ul>
-------------	--

- "There are no internal states, and the number of function calls along an execution path can be changed" (MOVABLE): The function call can be moved into a conditionally executed control flow branch or to the second operand of a logical AND or OR operation.

For operation calls, there is usually no way to specify a function class. Exceptions are synchronous operation call systems that are also server runnables. Instead, TargetLink offers the "NoDataFlowWithOtherOperations" and "NoStatesOrSideEffects" properties at the Data Dictionary Operation object. This option allows you to assure that all operation calls can be treated as if the function class has the SIDE\_EFFECT\_FREE Optimization property set or the operation has the NoDataFlowWithOtherOperations property set: i.e. this option overrides the NoDataFlowWithOtherOperations property of all operations.

If the option is switched off, then the value of the NoDataFlowWithOtherOperations property of each operation is evaluated.

Note that TargetLink always assumes certain data flow for asynchronous operation calls and operation results. See the description of the Data Dictionary Operation object's "NoDataFlowWithOtherOperations" and "NoStatesOrSideEffects" properties for details.

Data type	Numeric ID - Literal ID
	0 - off
	1 - on (default)

## GenerateWriteForNonScalarIrvs

Purpose	Generates Rte_IrvlWrite instead of Rte_IrvlWriteRef for non-scalar implicit InterRunnableVariables in AUTOSAR code.
Description	If this option is set to 'on', TargetLink does not generate calls to Rte_IrvlWriteRef RTE API functions for non-scalar implicit InterRunnableVariables. Instead, it generates calls to Rte_IrvlWrite RTE API functions.

Data type	Numeric ID - Literal ID
	0 - off (default)
	1 - on

## ARDATAPrototypeActualParamName

<b>Purpose</b>	Contains the name template for an actual parameter of an AUTOSAR RTE API function call performing a read/write access to a data prototype.
<b>Description</b>	<p>A call of an AUTOSAR RTE API function accessing a data prototype requires an actual parameter. You can use this option to specify the template for the actual parameter's name.</p> <p>The name macro <code>\$(DataPrototype)</code> is replaced by the name of the DD object that is used to specify the data prototype. The following list shows the RTE API functions that evaluate this option's name template together with the object kind of the corresponding DD objects:</p> <ul style="list-style-type: none"> <li>▪ Sender-receiver communication (DD DataElement object): <ul style="list-style-type: none"> <li>▪ <code>Rte_IRead</code></li> <li>▪ <code>Rte_IWrite</code></li> <li>▪ <code>Rte_IWriteRef</code></li> <li>▪ <code>Rte_Read</code></li> <li>▪ <code>Rte_Receive</code></li> <li>▪ <code>Rte_Send</code></li> <li>▪ <code>Rte_Write</code></li> </ul> </li> <li>▪ NvData communication (DD NvDataElement object): <ul style="list-style-type: none"> <li>▪ <code>Rte_IRead</code></li> <li>▪ <code>Rte_IWrite</code></li> <li>▪ <code>Rte_IWriteRef</code></li> <li>▪ <code>Rte_Read</code></li> <li>▪ <code>Rte_Write</code></li> </ul> </li> <li>▪ Interrunnable communication (DD InterRunnableVariable object): <ul style="list-style-type: none"> <li>▪ <code>Rte_IrvIRead</code></li> <li>▪ <code>Rte_IrvIWrite</code></li> <li>▪ <code>Rte_IrvIWriteRef</code></li> <li>▪ <code>Rte_IrvRead</code></li> <li>▪ <code>Rte_IrvWrite</code></li> </ul> </li> </ul>

The actual parameters of other AUTOSAR RTE API functions are not affected. This option does not specify the name template for other actual parameters that are used in sender-receiver communication, e.g., the actual parameter that belongs to a status variable.

If the actual parameter is a pointer, TargetLink will add a prefix to this template. For pointers with local scope, TargetLink will add 'p\_', while for pointers with global scope TargetLink will add 'p\_\${I}\_' as a prefix.

Data type	Data Type	Default
	String	\$(DataPrototype)\$R

## StrictRunnableInterfaceChecks

Purpose	Activates strict checking of the runnable interface.
Description	<p>If this option is enabled, TargetLink performs the following checks:</p> <ul style="list-style-type: none"> <li>▪ It checks if all the interface ports of a runnable system are AUTOSAR ports. If this condition is not fulfilled, TargetLink terminates with an error.</li> <li>▪ It is possible to place an AUTOSAR port in a subsystem that is placed in the runnable system. TargetLink checks if the AUTOSAR port has a one-to-one connection to the runnable system's border. If this condition is not fulfilled, TargetLink displays a warning but continues the code generation.</li> </ul> <p>If this option is disabled, TargetLink does not perform any of these checks.</p> <p>Note, that disabling this option may result in code that is not compilable with RTE code, because it is not conform with the AUTOSAR standard.</p>

Data type	Numeric ID - Literal ID
	0 - off
	1 - on (default)

## SuppressNoRunnableRestartCodeError

Purpose	Emit a warning instead of an error if there is restart code for a NoRestartCode runnable.
Description	The generation of runnable-specific restart functions can be omitted by setting a runnable's NoRestartCode property to 'on'. If restart code is necessary for such a runnable, TargetLink will issue an error message.

Setting this option changes this error to a warning, which makes it possible to analyze the generated code and to find the reason for the unwanted restart code.

---

**Data type**

Numeric ID - Literal ID
0 - off (default)
1 - on



**A**

access to the RTE status  
modeling 122  
acknowledgment notifications  
modeling 122  
simulating 301  
acknowledgments  
basics 117  
activation trigger 218  
AllowDuplicationOfImplicitAUTOSARDataAccess 387  
application errors of operations  
modeling 166  
application layer 63  
AR\_Fuelsys demo  
mode management 324  
ARDATAPrototypeActualParamName 391  
arginout argument 103  
ASAM MCD-2 MC standard 236  
AssumeFunctionCallSemanticsForRteAPIArguments 388  
AssumeOperationCallsHaveNoUnknownDataFlow 389  
asynchronous server call returns event 74  
atomic software component 64  
AUTOSAR  
application layer 63  
demo model 23  
interface 91  
introduction 15  
AUTOSAR and non-AUTOSAR controllers in one model  
modeling 269  
AUTOSAR development partnership  
basics 15

**B**

basic software 17  
basics  
acknowledgments 117  
AUTOSAR development partnership 15  
client-server communication 133  
code generation modes 276  
communication attributes 117  
dual block data 269  
generating a frame model from AUTOSAR data 314  
interrunnable communication 171  
mode management 322  
modes 217  
per instance memories 213  
preparing block parameters of several SWCs for measurement and calibration 244  
preparing SWCs for measurement and calibration 235  
RTE events 74  
runnables 69  
sender-receiver communication 108  
simulating AUTOSAR-compliant SWCs 296

software components 63  
tools for developing ECU software (Classic AUTOSAR) 18

**C**

calibration parameters  
creating 103  
Classic AUTOSAR  
basic software 17  
Code Generator Options  
AllowDuplicationOfImplicitAUTOSARDataAccess 387  
ARDATAPrototypeActualParamName 391  
AssumeFunctionCallSemanticsForRteAPIArguments 388  
AssumeOperationCallsHaveNoUnknownDataFlow 389  
GenerateIWriteForNonScalarIrvs 390  
StrictRunnableInterfaceChecks 392  
SuppressNoRunnableRestartCodeError 392

run time environment 17

Classic AUTOSAR-compliant code  
generating 275

Classic-AUTOSAR-compliant code  
generating 278

client port 91

client-server communication

basics 133

modeling 133

Code Generation

Code Generator Options

AllowDuplicationOfImplicitAUTOSARDataAccess (Classic AUTOSAR) 387

ARDATAPrototypeActualParamName (Classic AUTOSAR) 391

AssumeFunctionCallSemanticsForRteAPIArguments (Classic AUTOSAR) 388

AssumeOperationCallsHaveNoUnknownDataFlow (Classic AUTOSAR) 389

GenerateIWriteForNonScalarIrvs (Classic AUTOSAR) 390

StrictRunnableInterfaceChecks (Classic AUTOSAR) 392

SuppressNoRunnableRestartCodeError (Classic AUTOSAR) 392

code generation modes

basics 276

Code Generator Options

AllowDuplicationOfImplicitAUTOSARDataAccess 387

ARDATAPrototypeActualParamName 391

AssumeFunctionCallSemanticsForRteAPIArguments 388

AssumeOperationCallsHaveNoUnknownDataFlow 389

GenerateIWriteForNonScalarIrvs 390

StrictRunnableInterfaceChecks 392

SuppressNoRunnableRestartCodeError 392

Common Program Data folder 12

CommonProgramDataFolder 12

communication attribute 91

communication attributes

basics 117

communication specifications for operations

creating 164

composition 64

container for each controller SWCs (SystemDesk)

exporting 334

controller SWCs (TargetLink)

implementing 338

importing 336

creating calibration parameters 103

creating communication specifications for operations 164

creating data elements 103

creating interfaces 100

creating interruptable variables 173

creating operations with arguments 103

creating ports 101

creating software components 66

**D**

data elements

creating 103

data receive error event 74

data received event 74

data send completed event 74

defining

SWC internal behaviors (SystemDesk) 332

demo model

AUTOSAR 23

Documents folder 12

DocumentsFolder 12

dual block data

basics 269

**E**

ECU software architecture (Classic AUTOSAR)

16

ECU state manager 218

event

RTE 74

event for a runnable

specifying 76

example

generating a frame model from an AUTOSAR file 319

exclusive area 69

executing a runnable in an exclusive area 79

exporting

container for each controller SWCs (SystemDesk) 334

**F**

frame model from AUTOSAR data

generating 316

frame model from Classic AUTOSAR data

generating 314

updating 318

**G**

GenerateWriteForNonScalarIrvs 390  
 generating a frame model from an AUTOSAR file  
     example 319  
 generating a frame model from AUTOSAR data  
     basics 314  
 generating a frame model from Classic  
 AUTOSAR data 314, 316  
 generating Classic AUTOSAR-compliant code  
 275  
 generating Classic-AUTOSAR-compliant code  
 278

**I**

implementing  
     controller SWCs (TargetLink) 338  
 implementing mode management 327  
 importing  
     controller SWCs (TargetLink) 336  
 initial mode 218  
 initialization of data elements  
     modeling 126  
 interface 91  
 interfaces  
     creating 100  
 interruptable communication  
     basics 171  
     modeling 171, 175  
 interruptable variables  
     creating 173  
 introduction  
     AUTOSAR 15  
     mode management 322  
 invalidation  
     modeling 122  
 is service 91

**L**

Local Program Data folder 12  
 LocalProgramDataFolder 12

**M**

mode management  
     AR\_Fuelsys demo 324  
     basics 322  
     implementing 327  
     introduction 322  
     modeling 327  
     SystemDesk 322  
     TargetLink 322  
 mode manager 218  
 mode switch event 74  
 modeling  
     access to the RTE status 122  
     acknowledgment notifications 122  
     invalidation 122  
     software architecture (SystemDesk) 329  
 modeling application errors of operations 166

**N**

modeling AUTOSAR and non-AUTOSAR  
 controllers in one model 269  
 modeling client-server communication 133  
 modeling from scratch  
     starting 29  
 modeling initialization of data elements 126  
 modeling interruptable communication 171, 175  
 modeling mode management 327  
     workflow 327  
 modeling modes 217, 219  
 modeling per instance memories 213, 214  
 modeling runnables 69, 71  
 modeling sender-receiver communication 108  
 modeling software components (SWCs) 63  
 modes  
     basics 217  
     modeling 217, 219

**O**

operation invoked event 74  
 operations with arguments  
     creating 103

**P**

per instance memories  
     basics 213  
     modeling 213, 214  
 port 91  
 ports  
     creating 101  
 preparing block parameters of several SWCs for  
 measurement and calibration  
     basics 244  
 preparing SWCs for calibration 235  
 preparing SWCs for measurement 235  
 preparing SWCs for measurement and  
 calibration  
     basics 235

**R**

receiver port 91  
 returnvalue argument 103  
 RTE 17  
     event 74  
     simulation frame 296  
 RTE events  
     basics 74  
 RTE status  
     simulating 299  
 run time environment 17  
 runnable in an exclusive area  
     executing 79  
 runnables  
     basics 69  
     modeling 69, 71

**S**

sender port 91

sender-receiver communication  
     basics 108  
     modeling 108  
     modeling via data store blocks 115  
     modeling via port blocks 112  
 server  
     communication specification 164  
 server port 91  
 signal invalidation  
     simulating 303  
 simulating acknowledgment notifications 301  
 simulating AUTOSAR-compliant SWCs  
     basics 296

simulating signal invalidation 303  
 simulating SWCs 295  
 simulating the RTE status 299  
 simulation frame 296  
 software architecture (SystemDesk)  
     modeling 329

**Software Components**

    basics 63  
     creating 66  
 software components (SWCs)  
     modeling 63  
 software tool together with TargetLink  
     using 313  
 specifying an event for a runnable 76  
 starting modeling from scratch 29  
 StrictRunnableInterfaceChecks 392  
 SuppressNoRunnableRestartCodeError 392  
 SWC internal behaviors (SystemDesk)  
     defining 332  
 SWCs  
     simulating 295  
 SWCs for calibration  
     preparing 235  
 SWCs for measurement  
     preparing 235  
 SystemDesk  
     mode management 322

**T**

TargetLink  
     mode management 322  
 TargetLink's modeling features  
     dual block data 269  
 timing event 74  
 tools for developing ECU software (Classic  
 AUTOSAR)  
     basics 18

**U**

updating a frame model from Classic AUTOSAR  
 data 318  
 using a software tool together with TargetLink  
 313  
 using Classic AUTOSAR in TargetLink 21

**W**

workflow

modeling mode management 327

