

TargetLink

# Code Generation Guide for MATLAB<sup>®</sup> Code in Simulink<sup>®</sup> Models

For TargetLink 5.1

Release 2020-B – November 2020

## 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.

© 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	5
Introduction to Code Generation for MATLAB Code	9
Basics on Code Generation for MATLAB® Code.....	9
Working with Model Elements	11
Basics on Working with Model Elements.....	11
Basics on Logging MATLAB® Function Data.....	12
Working with MATLAB Code Elements	15
Basics on Working with MATLAB® Code Elements.....	15
Basics on Constant Variables in MATLAB® Code.....	17
Basics on Mapping of Local MATLAB® Variable Data Types.....	19
Basics on Specifying DD Properties for MATLAB Code Elements.....	21
How to Create DD Objects for MATLAB Code Elements (script-based).....	22
How to Create DD Objects for Local MATLAB Variables.....	23
How to Create DD Objects for MATLAB Local Functions.....	24
Working with MATLAB Functions	25
Basics on Working with MATLAB Functions.....	25
Basics on Exporting Stateflow® MATLAB Functions.....	26
Basics on Embedding External Code in a MATLAB® Function.....	27
Basics on Commenting MATLAB Functions.....	27
Example of Transferring MATLAB Function Comments to Production Code.....	29
Glossary	31
Index	61



# About This Guide

---

**Content**

This guide introduces you to the script-based implementation of function algorithms by means of adding MATLAB® code functions to Simulink® models.

**Note**

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.

---

**Target group**

This guide is primarily targeted at function developers and software specialists who want to integrate MATLAB functions into Simulink models.









---

**Required knowledge**

Knowledge in handling MATLAB, the MATLAB language, Simulink, and Stateflow is 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 Code Generation for MATLAB Code

## Basics on Code Generation for MATLAB® Code

### MATLAB® code

MATLAB code lets you implement the function algorithm in textual form instead of a graphical language. The function algorithm is called a MATLAB function.

#### MATLAB code example

```
function my_main_function
    algorithm body
end

function my_local_function
    algorithm body
end
```

### MATLAB functions

MATLAB functions consist of one [MATLAB main function](#) and can contain any number of [MATLAB local functions](#).

### MATLAB code in TargetLink

TargetLink supports the modeling of MATLAB code as follows:

- MATLAB functions in Stateflow® charts
- MATLAB Function blocks in Simulink® models

TargetLink does not support MATLAB nested functions. In addition, not all MATLAB function statements and MATLAB function operations are supported. Refer to [Supported MATLAB® Code Function Statements and Function Operators](#) ([TargetLink Code Generation Reference for MATLAB® Code in Simulink® Models](#)).

### Demo model

The following demo model introduces you to working with MATLAB code:

- [MATLAB\\_CODE\\_STATEFLOW](#) ([TargetLink Demo Models](#))

## Related topics

### Basics

[MATLAB\\_CODE\\_STATEFLOW](#) ( TargetLink Demo Models)

### References

[Supported MATLAB® Code Function Statements and Function Operators](#)  
( TargetLink Code Generation Reference for MATLAB® Code in Simulink® Models)

# Working with Model Elements

## Where to go from here

## Information in this section

Basics on Working with Model Elements.....	11
Basics on Logging MATLAB® Function Data.....	12

## Basics on Working with Model Elements

### Model elements

When implementing MATLAB® code in your model, you work with [model elements](#), such as [MATLAB main functions](#) and variables with defined data.

### Specifying model elements

You can specify the behavior during code generation and model elements properties in the Data Dictionary Manager and the [Property Manager](#). Refer to:

- [Basics on the Data Dictionary Manager](#) ([TargetLink Data Dictionary Basic Concepts Guide](#))
- [Basics on the TargetLink Property Manager](#) ([TargetLink Preparation and Simulation Guide](#))

## Related topics

### Basics

[Basics on the Compatibility of Buses and Predefined Structs](#) (📖 TargetLink Customization and Optimization Guide)  
[Basics on the Data Dictionary Manager](#) (📖 TargetLink Data Dictionary Basic Concepts Guide)  
[Basics on the TargetLink Property Manager](#) (📖 TargetLink Preparation and Simulation Guide)  
[Modifying Multiple Properties at Once via the Property Manager](#) (📖 TargetLink Preparation and Simulation Guide)

## Basics on Logging MATLAB® Function Data

### Logging MATLAB® function data

MATLAB® function data can be logged in all simulation modes.

Logging can be specified with the [Property Manager](#). TargetLink uses an individual subplot for each signal and each channel of a signal. Therefore, you have to specify which elements of a signal you want to plot.

#### Note

MATLAB code input variables and [local MATLAB variable](#) cannot be logged.

### Logging options

Logging options are available only for the MATLAB Function block output variables. Refer to [Basics on Logging](#) (📖 TargetLink Preparation and Simulation Guide).

**Counting** There are two ways of counting when specifying plot channels:

- Zero-based indexing
- One-based indexing

The counting type depends on where counting is specified:

Counting Specified In	Access in Code	Plot Channels
Stateflow action language	Zero-based	Zero-based
MATLAB function in Stateflow charts	One-based	Zero-based
MATLAB function in MATLAB Function blocks	One-based	One-based

---

## Related topics

### Basics

[Basics on Logging](#) ( [TargetLink Preparation and Simulation Guide](#))



# Working with MATLAB Code Elements

## Where to go from here

## Information in this section

Basics on Working with MATLAB® Code Elements.....	15
Basics on Constant Variables in MATLAB® Code.....	17
Basics on Mapping of Local MATLAB® Variable Data Types.....	19
Basics on Specifying DD Properties for MATLAB Code Elements.....	21
How to Create DD Objects for MATLAB Code Elements (script-based).....	22
How to Create DD Objects for Local MATLAB Variables.....	23
How to Create DD Objects for MATLAB Local Functions.....	24

## Basics on Working with MATLAB® Code Elements

### 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](#).

Depending on the Generate optimized code option, the Code Generator tries to eliminate some of the C code variables generated for MATLAB code elements. Refer to [How to Enable Code Optimization](#) ([TargetLink Customization and Optimization Guide](#)).

### Specifying MATLAB code elements

Settings and behavior for [MATLAB code elements](#) cannot be specified as usual. TargetLink uses the names of the MATLAB code elements to look up the [DD objects](#). If the name of a DD object matches, the settings defined at the DD

object are used. Refer to [Basics on Specifying DD Properties for MATLAB Code Elements](#) on page 21.

If MATLAB code elements are not specified explicitly, a set of rules is used to map data types and dimensions for MATLAB code elements automatically. Refer to [Basics on Mapping of Local MATLAB® Variable Data Types](#) on page 19.

### Specifying interface signals for MATLAB local functions

If you assigned a DD FunctionClass object whose Storage property is set to **extern** to a [MATLAB local function](#), TargetLink cannot automatically determine the interface signals for the MATLAB local function. Therefore, you have to specify signal dimensions and data type as follows:

- Connect the inputs and outputs to the DD Variable objects.
- Set the Width property and the Type property accordingly.

Make sure that the properties of the referenced DD Variable objects are consistent with the Simulink output signal dimension and the Simulink output data type. The following tables help you map the properties correctly:

#### Output signal dimension

Output Simulink Signal Dimension (Compiled Size)	Width of the Associated DD Variable Object
[1 1] (scalar)	'' (scalar)
[n 1] (row matrix)	n (vector) OR [n 1] (matrix)
[1 n] (column matrix)	[1 n] (matrix)
[n m] (matrix)	[n m] (matrix)

#### Examples for Simulink output data type value

Simulink Output Data Type	TargetLink Data Type / LSB / Offset of the Associated DD Variable Object
double	Float64 Int8 with LSB unequal to 1, for example $2^{-10}$ UInt32 with Offset unequal to 0, for example 4
single	Float32
logical	Bool
int8	Int8 with LSB = 1 and Offset = 0
uint8	UInt8 with LSB = 1 and Offset = 0




## Related topics

## Basics

Basics on Mapping of Local MATLAB® Variable Data Types.....	19
Basics on Specifying DD Properties for MATLAB Code Elements.....	21

## HowTos

How to Enable Code Optimization ( [TargetLink Customization and Optimization Guide](#))

## Basics on Constant Variables in MATLAB® Code

### Constant variables in MATLAB® code

In TargetLink, you can use constant variables in MATLAB® code to increase code readability. This facilitates the following:

- Lower error rate
- Faster development speed
- Greater re-usability

#### Example

```
function out = myfcn( in )
    N = length( in );
    tmp = zeros( N, 1 );
    for i = 1:N
        ...
    end
    ...
end
```

### Using constant variables in MATLAB code

You can use constant value expressions and constant value variables in MATLAB code.

**Constant value expression** An expression for which the Code Generator can determine the variable values during code generation.

The following expressions are recognized as constant value expressions:

- Numerical constants, for example **42** or **3.14**
- Stateflow® data and MATLAB variables with **constant** or **parameter** Simulink® data scope and a DD VariableClass object with the following settings:

Property	Value
Alias	off
Info	none
Optimization	ERASABLE

Property	Value
Scope	global or local
Volatile	off

- `length( ... )`
- `size( ... )`
- Constant value variables assigned to a DD VariableClass object whose Optimization property is set to ERASABLE.

In addition, the following expressions can be used if all arguments are constant value expressions:

- `ones( ... )`
- `zeros( ... )`
- `false( ... )`
- `true( ... )`
- `ranges ':'`
- Simple mathematical operations, for example `4+MyParam`
- Concatenations, for example `[ 4 MyParam 5* [ ones( 1,3 ) 6 ]`

**Constant value variable** TargetLink handles [local MATLAB variables](#) as constant value variables if the following conditions are met:

- The local MATLAB variable is not part of a [reusable system definition](#).
- The defining expression must be a [constant value expression](#).
- The local MATLAB variable has only one assignment.
- The assigned DD VariableClass object must comply the following settings:

Property	Value
Alias	off
Info	none
Scope	global or local
Volatile	off



### Declaring a constant local MATLAB variable as macro or constant C variable

You can declare a constant [local MATLAB variable](#) as a macro or constant C variable as follows:


1. Create a DD Variable object for the MATLAB local variable. Refer to [How to Create DD Objects for Local MATLAB Variables](#) on page 23.
2. Assign a DD VariableClass object whose Const or Macro property is set to on. Refer to [Basics on Variable Classes](#) ([TargetLink Customization and Optimization Guide](#)).

## Related topics








### Basics

Basics on Variable Classes ( TargetLink Customization and Optimization Guide)  
 Details on Preconfigured VariableClass Objects in the Data Dictionary  
 ( TargetLink Customization and Optimization Guide)

### HowTos

How to Define New Variable Classes ( TargetLink Customization and Optimization Guide)

### References

Alias ( TargetLink Data Dictionary Reference)  
 Const ( TargetLink Data Dictionary Reference)  
 Info ( TargetLink Data Dictionary Reference)  
 Macro ( TargetLink Data Dictionary Reference)  
 Optimization ( TargetLink Data Dictionary Reference)  
 Scope ( TargetLink Data Dictionary Reference)  
 Volatile ( TargetLink Data Dictionary Reference)

## Basics on Mapping of Local MATLAB® Variable Data Types

### Mapping

TargetLink has a set of rules to determine and map the dimension and data type of each [local MATLAB® variable](#). Therefore, it is not necessary to explicitly specify each local MATLAB variable. The Code Generator tries to apply the following rules in the specified order and finishes the code generation process if a rule can be applied:

1. DD look-up
2. Boolean assignment
3. Loop index variable
4. Simulink® data type (default)
5. Floating-point data type (fall-back)

**1. DD look-up** Explicitly specified data types for local MATLAB variables are always prioritized. Refer to [How to Create DD Objects for Local MATLAB Variables](#) on page 23.

**2. By the first defining Boolean (logical) assignment** The following code patterns are recognized as Boolean data types by the Code Generator and are mapped to the TargetLink Bool data type. ... represents an arbitrary expression:

```

a=false;
b=false(...);
c=true;
d=true(...);
e=logical(...);
f= ... & ...;
g= ... | ...;
h= ... && ...;
i= ... || ...;
j=any(...);
k=all(...);
l=cast(...,'logical');
m=isequal(...);
n = xor(...);
o= ... == ...;
p= ... ~= ...;
q= ... <= ...;
r= ... < ...;
s= ... >= ...;
t= ... > ...;
u= ~...;

```

### 3. By the first use as loop index variable

```

for i=start:end; (with step=1)
for i=start:step:end;

```

TargetLink can determine a smallest possible signed integer data type for loop variables under the following conditions:

- The loop variable does not have write access outside the loop.
- **start** is an integer [constant value expression](#) or is a variable with integer data type.
- **step** is an integer [constant value expression](#).
- **end** must be an integer data type if **end** is not a [constant value expression](#).

**4. By the underlying Simulink data type (default)** All local MATLAB variables are mapped to a suitable TargetLink data type on the basis of their Simulink data type if the following Code Generator Options are set to on (default):


- AutoSelectBoolTypesForLocalMATLABVariables
- AutoSelectIntegerTypesForLocalMATLABVariables
- AutoSelectFloatTypesForLocalMATLABVariables

For more information on Code Generator Options, refer to [Basics on Configuring the Code Generator for Production Code Generation](#) ([TargetLink Customization and Optimization Guide](#)).

**5. By the platform's default floating-point data type (fall-back)** The platform's default floating-point data type is used as a fall-back if the Code Generator cannot apply other rules.

## Related topics




### Basics

[Basics on Configuring the Code Generator for Production Code Generation](#)  
( [TargetLink Customization and Optimization Guide](#))

### HowTos



[How to Create DD Objects for Local MATLAB Variables..... 23](#)

### References



[AutoSelectBoolTypesForLocalMATLABVariables](#) ( [TargetLink Model Element Reference](#))  
[AutoSelectFloatTypesForLocalMATLABVariables](#) ( [TargetLink Model Element Reference](#))  
[AutoSelectIntegerTypesForLocalMATLABVariables](#) ( [TargetLink Model Element Reference](#))

# Basics on Specifying DD Properties for MATLAB Code Elements




## DD properties for MATLAB® code elements

 **DD objects** and properties cannot be assigned as usual for  **MATLAB® code elements**.

## Assigning DD objects and properties

To assign  **DD objects** and properties to  **MATLAB code elements**, you have to create DD objects for MATLAB code elements. Refer to:


- [How to Create DD Objects for MATLAB Code Elements \(script-based\)](#) on page 22
- [How to Create DD Objects for Local MATLAB Variables](#) on page 23
- [How to Create DD Objects for MATLAB Local Functions](#) on page 24

**Local MATLAB variables** After the DD objects for  **local MATLAB variables** are created, you have to link the MATLAB Function block to the DD objects in the  **Property Manager**. You can do this by adding the MATLABFunction column to the Property View using the Column Chooser. Refer to [How to Customize Columns in the Property View and Validation Summary](#) ( [TargetLink Preparation and Simulation Guide](#)).

**MATLAB local functions** The created DD Function objects must be referenced by the Local function property of an existing DD MATLABFunction object. Refer to [How to Create DD Objects for Local MATLAB Variables](#) on page 23.

## Related topics

## HowTos

How to Create DD Objects for Local MATLAB Variables.....	23
How to Create DD Objects for MATLAB Code Elements (script-based).....	22
How to Create DD Objects for MATLAB Local Functions.....	24
How to Customize Columns in the Property View and Validation Summary (  TargetLink Preparation and Simulation Guide)	



## How to Create DD Objects for MATLAB Code Elements (script-based)

## Objective

To create  DD objects for  MATLAB® code elements (script-based).


## Workflow

This workflow consists of the following parts:

- [Part 1](#) on page 22 - Create XML files for each MATLAB function
- [Part 2](#) on page 22 - Create  DD objects for  MATLAB code elements

## Part 1

### To create XML files for each MATLAB function in the code generation unit

- 1 In the navigation pane of the Code Generator Options dialog, select **General Settings - OutputMATLABCodeInfo**.
- 2 Select **Output XML files with MATLAB code information**.
- 3 Generate production code. Refer to [How to Generate Production Code for Selected TargetLink Subsystems](#) ( TargetLink Preparation and Simulation Guide).

## Interim result

You created XML files for each MATLAB function in the code generation unit with following file names:


MATLABCodeInfo\_<TLSubsystemID>\_<MATLABFcnName>.xml

## Part 2

### To create DD objects for MATLAB code elements

- 1 In the MATLAB Command Window, type:

```
tlCreateMATLABFunctionDDObjects('XmlFiles', ...  
'MATLABCodeInfo_<TLSubsystemID>_<MATLABFcnName>.xml')
```

- 2 Repeat step 1 for each MATLAB function created in part 1 for which you want to create  DD objects.

## Result

You created  DD objects for  MATLAB code elements (script-based).

**Related topics****HowTos**

[How to Generate Production Code for Selected TargetLink Subsystems](#)  
 (📖 [TargetLink Preparation and Simulation Guide](#))

## How to Create DD Objects for Local MATLAB Variables

**Objective**

To create [DD objects](#) for [local MATLAB® variables](#).

**Method****To create DD objects for local MATLAB variables**

- 1 In the Data Dictionary Manager, right-click /Pool/Variables, select **Create VariableGroup**, and enter a unique name, for example **MATLAB\_Variables**.
- 2 Right-click /Pool/Variables/MATLAB\_Variables and select **Create Variable**. Give the DD Variable object the same name as the [local MATLAB variable](#), for example **My\_MATLABVariable**.
- 3 Right-click /Pool and select **Create MATLABFunctions**.
- 4 Right-click /Pool/MATLABFunctions and select **Create MATLABFunction**. Give the DD Function object the same name as the MATLAB function you want to specify, for example **My\_MATLABFunction**.
- 5 Right-click /Pool/MATLABFunctions/MyMATLABFunction and select **Create Variables**.
- 6 At the VariableRef property, reference the DD Variable object you created in step 2, for example **My\_MATLABVariable**.


**Result**

You created [DD objects](#) for [local MATLAB variables](#).

**Related topics****HowTos**

[How to Customize Columns in the Property View and Validation Summary](#)  
 (📖 [TargetLink Preparation and Simulation Guide](#))

## How to Create DD Objects for MATLAB Local Functions

<b>Objective</b>	To create <a href="#">DD objects</a> for a <a href="#">MATLAB® local function</a> .
<b>Preconditions</b>	You created a DD MATLABFunction object and linked it to a Stateflow® MATLAB function or a MATLAB Function block. Refer to <a href="#">How to Create DD Objects for Local MATLAB Variables</a> on page 23
<b>Method</b>	<p><b>To create DD objects for a MATLAB local function</b></p> <ol style="list-style-type: none"> <li>1 In the Data Dictionary Manager, right-click /Pool and select Create Functions.</li> <li>2 Right-click /Pool/Functions, select Create FunctionGroup, and enter a unique name, for example MATLAB_Functions.</li> <li>3 Right-click /Pool/Functions/MATLAB_Functions and select Create Function. Give the DD Function object the same name as the <a href="#">MATLAB local function</a>, for example MATLABLocalFcn.</li> <li>4 Right-click /Pool/Functions/MATLAB_Functions/MATLABLocalFcn/InterfaceVariables and select Create InterfaceVariable for each MATLAB local function input and MATLAB local function output.</li> <li>5 In the Property Value List, set the Kind property to a suitable value and reference a DD Variable object from /Pool/Variables/MATLAB_Variables.</li> </ol>
<b>Result</b>	You created <a href="#">DD objects</a> for a <a href="#">MATLAB local function</a> .
<b>Related topics</b>	<p>Basics</p> <p><a href="#">Details on Synchronizing Simulink and TargetLink Data</a> ( <a href="#">TargetLink Preparation and Simulation Guide</a>)</p>



# Working with MATLAB Functions

## Where to go from here

## Information in this section

Basics on Working with MATLAB Functions.....	25
Basics on Exporting Stateflow® MATLAB Functions.....	26
Basics on Embedding External Code in a MATLAB® Function.....	27
Basics on Commenting MATLAB Functions.....	27
Example of Transferring MATLAB Function Comments to Production Code.....	29

## Basics on Working with MATLAB Functions

### Implementing MATLAB functions

MATLAB® functions can be modeled as follows:

- A Stateflow® MATLAB function via a Stateflow chart
- A MATLAB Function block in a Simulink® model

#### Note

MATLAB nested functions are not supported by TargetLink.

#### Difference between MATLAB Function blocks and Stateflow MATLAB functions in TargetLink

It is useful to differentiate between MATLAB Function blocks and Stateflow MATLAB functions for the following reasons:

- The inputs of MATLAB Function blocks are comparable with Stateflow chart inputs and therefore also have special properties. For example, they can inherit scaling data.
- Stateflow MATLAB functions can be called from different places in a chart.

**Executing MATLAB functions**

Stateflow® MATLAB functions and MATLAB Function blocks are executed in different ways.

**Stateflow MATLAB function** A Stateflow MATLAB function is executed if the function is called from a Stateflow object.

**MATLAB Function block** A MATLAB Function block is executed if the following conditions are met:

- The trigger condition is met (for edge-triggered MATLAB Function blocks).
- The function call occurs (for function-call-triggered MATLAB Function blocks).
- The input data is available (for MATLAB Function blocks without trigger or function call inputs).

Only the Discrete (fixed sample time) and Inherited update methods are supported.

**Influencing function signature**

TargetLink generates C code functions for each [MATLAB main function](#) and [MATLAB local function](#). The C code signature can be influenced by defining variable classes for MATLAB variables, Stateflow data, and [local MATLAB variables](#).

**Default variable class** If the default variable class is used, the input variables of Stateflow MATLAB functions are implemented as formal parameters (SFFcnInput) and the output variables are implemented as formal parameters of the corresponding C function (SFFcnOutput). The default classes of MATLAB Function blocks are SFGlobal and SFGlobalInit. Variables with a ref\_param scope, value\_param scope, or variables of a pointer-to-struct variable are added to the interface of the MATLAB function automatically. The scope of a struct root must be ref\_param.

**Order of formal parameters** The order of the formal parameters specified in the MATLAB function might be different from the generated C function. Therefore, you can change the order of formal parameters in generated C functions via the Function ArgumentList property in the [Property Manager](#). Refer to [Column Chooser Dialog](#) ([TargetLink Tool and Utility Reference](#)).

**Related topics****References**

[Column Chooser Dialog](#) ([TargetLink Tool and Utility Reference](#))

## Basics on Exporting Stateflow® MATLAB Functions

**Exporting Stateflow® MATLAB® functions**

Stateflow® MATLAB® functions on the root level of a Stateflow chart can be exported in the same way as root-level graphical functions. Exporting a Stateflow MATLAB function is useful to make a MATLAB function accessible outside the

chart and across the TargetLink subsystem boundaries. Exported functions are not scope-reduced.

The generated production code contains a C function. Calls to exported MATLAB functions are supported in every chart that belongs to the same Stateflow machine.

#### Excluding a Stateflow MATLAB function from export

You can exclude Stateflow MATLAB functions from export by inlining the Stateflow MATLAB function in the generated production code. To inline a Stateflow MATLAB function, select **Inline** from the **Function Inline Option** option list in the Simulink® Model Explorer.

## Basics on Embedding External Code in a MATLAB® Function

#### Embedding external code in a MATLAB® Function

You can embed external code in a [MATLAB® main function](#) via a [MATLAB local function](#) as follows:

1. Insert the external code as a MATLAB local function into your MATLAB main function.
2. Create [DD objects](#) for the MATLAB local function. Refer to [How to Create DD Objects for MATLAB Local Functions](#) on page 24.
3. Set the value of the DD FunctionClass property of the created DD object to a DD FunctionClass object whose Storage property is set to **extern**.

#### Related topics

##### HowTos

[How to Create DD Objects for MATLAB Local Functions.....](#) 24

## Basics on Commenting MATLAB Functions

#### Code comments

Comments in MATLAB describe the written code. Commenting code helps other users understand the code.

#### MATLAB code comments in TargetLink

TargetLink allows code comments in MATLAB code and transfers them to the production code. MATLAB comments are associated with the subsequent MATLAB construct and transferred to the corresponding construct in the production code.

You can use the following comment types:

- Single-line comments
- Multi-line comments
- Inline comments

**Single-line comments** Place one or more single-line comments before a construct. Empty lines are not transferred to generated production code.

MATLAB Code	Production Code
<pre>% short description % % another short description y = x;</pre>	<pre>/* short description */ /* another short description */ Sa1_fcn_y = Sa1_fcn_x;</pre>

**Multi-line comments** Place multi-line comments before a construct. Empty lines are transferred to generated production code. All spaces in front of the opening block comment operator are deleted to prevent double indentation. Spaces inside the multi-line comment block are kept to allow for user-defined formatting.

MATLAB Code	Production Code
<pre>%{ Long description:  text with explanation and details %} y = x + 42;</pre>	<pre>/* Long description:      text with explanation     and details */ Sa1_fcn_y = Sa1_fcn_x + 42.;</pre>

**Inline comments** Place inline comments within the construct. In the generated production code, inline comments are transferred before the construct.

MATLAB Code	Production Code
<pre>% split description y = x ... x: description + z ... z: description ;</pre>	<pre>/* split description x: description z: description */ Sa1_fcn_y = Sa1_fcn_x + Sa1_fcn_z;</pre>

### Specific rules

Comments that are assigned to a function must be placed in front of the function so that they are transferred in the C function in the production code.

TargetLink uses specific rules for processing spaces, tabs, and escape sequences in MATLAB code comments.

**Spaces** Spaces are treated differently depending on the comment type and location of the spaces:

- Single-line and inline comments: The leading space is deleted.
- Multi-line comments: For each line, all spaces in front of the opening block comment operator are deleted to prevent double indentation.

**Tabs** Tabs in MATLAB code comments are replaced by single spaces.

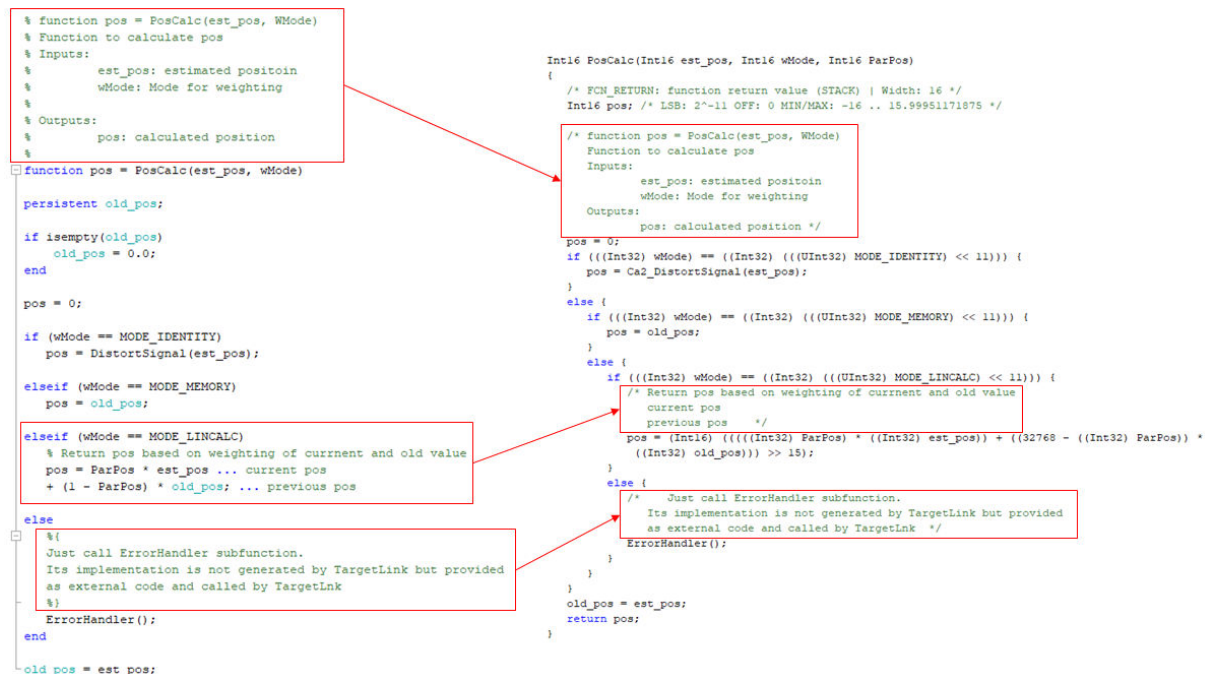
**Escape sequences** Escape sequences in MATLAB code comments are replaced by `__`.

## Example of Transferring MATLAB Function Comments to Production Code

### Introduction

This examples shows how TargetLink transfers MATLAB® code comments in generated production code. Refer to [Basics on Commenting MATLAB Functions](#) on page 27.

### Example



### Related topics

#### Basics

[Basics on Commenting MATLAB Functions..... 27](#)



# 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.....	32
A.....	33
B.....	36
C.....	37
D.....	40
E.....	42
F.....	43
G.....	44
I.....	44
L.....	46
M.....	47
N.....	49
O.....	50
P.....	51
R.....	52
S.....	54
T.....	56
U.....	58
V.....	58
W.....	59

## Numerics

**1-D look-up table**

output value (y).

A look-up table that maps one input value (x) to one

**2-D look-up table**

output value (z).

A look-up table that maps two input values (x,y) to one



**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 Inport, 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
- Multipoint 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., variant 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.

**DataltemMapping (DIM)** A DataltemMapping object is a DD object that references a [ReplaceableDataltem \(RDI\)](#) 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](#).

## I

---

**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.

## M

**Look-up function** A function for a look-up table that returns a value from the look-up table (1-D or 2-D).

**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.

## N

**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 InPort/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.

**tlilib** 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.

## W

**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](#).



**C**

Common Program Data folder 6  
CommonProgramDataFolder 6

**D**

Documents folder 6  
DocumentsFolder 6

**L**

Local Program Data folder 6  
LocalProgramDataFolder 6

