TargetLink Demo Models

For TargetLink 5.1

Release 2020-B – November 2020

dSPACE

How to Contact dSPACE

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

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

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.

© 2009 - 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 Document 7

Model Preparation 9

PREPARATION 9
CUSTOM_BLOCKS 14
CUSTOM_ENHANCEMENT 15

Basic Features 21

PIPT1 22
VOICE_COIL_FLEXIBLE_POSITIONER (VCFP) 25
ENABLSYS 31
POSCONTROL 35
STRUCT_INTERFACES 41
MATRIX 44
BUS_STRUCT 50
VARIABLE_VECTOR_WIDTH 55
FUNCTION_REUSE 56
MULTIPLE_INSTANCES_REFMODEL 60
EMBEDDING_EXTERNAL_CODE 64
AUTODOC_GENERATION 67
VARIABLE_INITIALIZATION 72
FUNCTION_VARIANTS 75
ENUM_TYPES 78
ENUM_BLINKER 81
BUS_CC 85
MATRIX_INVERSE 92

Supported Stateflow Features 97

STATEFLOW 97
FUELSYS 99
FUELSYS_FLOAT 101

Lookup Tables, User-Written Lookup Functions 103

TABLE1D 103

TABLE2D 105
TABLE1D_USR 106
TABLE1D_USR_LOCAL 108
TABLE2D_USR 110
TABLE1D_FCN 112
PRELOOKUP_USR 113
DYNAMIC_LUT 117

AUTOSAR Software Components 119

AR_POSCONTROL 119
DCMC 128
AUTOSAR_FUELSYS 130
AR_NVDATA_TRANSFORMER 135
AR_COLLISION_DETECTION 139
AR_MEMORY_MAPPING 147
AR_ARRAY_OF_STRUCT_DATA 152
AR_INCREMENTAL_CODEGEN 157
AAR_COMMUNICATION 165

Multirate Code Generation 177

MULTIRATE 177

Component-Based Development 181

MODEL_REFERENCING 181
CG_FROM_DD 185
REPLACEABLE_DATA_ITEMS 187
MODULAR_DEVELOPMENT 192

Code Testing 199

ONLINE_PARAMETER_MODIFICATION 199
SIGNAL_INJECTION 203
REAL_TIME_PLATFORM_SUPPORT 206
VECU_EVENT_SIMULATION 213

Data Dictionary Releated Features 219

DD_FILTER 219
DD_ML_API 221

DD_ML_IMPORTEXPORT..... 222
DD_CUSTOM_COMMAND..... 225

MATLAB Code 229

MATLAB_CODE_STATEFLOW..... 229

Glossary 235

Index 265

About This Document

Contents

A convenient way to familiarize yourself with TargetLink's features is to experiment with the demo models. These show possible implementation methods with frequently used blocks.

Symbols

dSPACE user documentation uses the following symbols:

Symbol	Description
DANGER	Indicates a hazardous situation that, if not avoided, will result in death or serious injury.
WARNING	Indicates a hazardous situation that, if not avoided, could result in death or serious injury.
CAUTION	Indicates a hazardous situation that, if not avoided, could result in minor or moderate injury.
NOTICE	Indicates a hazard that, if not avoided, could result in property damage.
Note	Indicates important information that you should take into account to avoid malfunctions.
Tip	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.

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.

Model Preparation

Where to go from here

Information in this section

PREPARATION..... 9
CUSTOM_BLOCKS..... 14
CUSTOM_ENHANCEMENT..... 15

PREPARATION

Access

In the MATLAB Command Window, enter the following command:

```
tl_demos preparation
```

You can also enter **`tl_demos`** to access all available demos in the MATLAB Help browser. Select a demo and click **Open this example** at the top right.

Description

The `tl_demos preparation` model shows you how TargetLink can enhance Simulink blocks by adding TargetLink production code options to them.

Model Reference (preparing model referencing)

Simulink model referencing is a mechanism that lets you include models in other models by using Model blocks. Each Model block represents a reference to another model and displays the inputs and outputs of the referenced model.

This sample demonstrates how to prepare a Simulink subsystem that includes a referenced model for TargetLink. You can start system preparation either interactively with the **System Preparation** dialog, or by typing

```
» tl_prepare_system( ...
'System','preparation_with_mdl_referencing/Model_Reference', ...
'PrepareRefMdls','on');
```

in the MATLAB Command Window. This will prepare the referenced model together with this subsystem. To have the prepared models renamed to <modelName>_TL, use

```
» tl_prepare_system( ...
'System','preparation_with_mdl_referencing/Model_Reference', ...
'PrepareRefMdls','on','Save','on','prepSystem','$_TL');
```

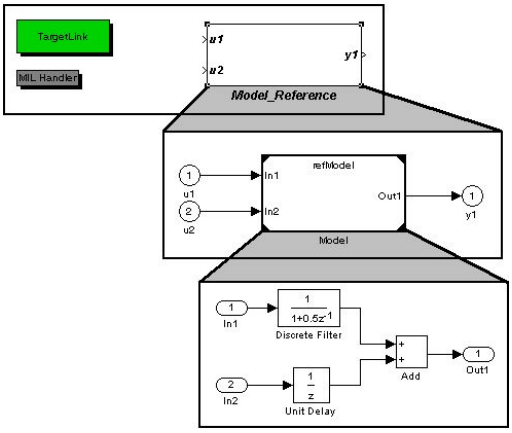
This will introduce a TargetLink-compliant referenced system and leave the original unmodified.

The **System Preparation** dialog can be opened with the **Prepare System** button on the Tools page of the TargetLink Main Dialog, or by typing

```
» tl_prepare_system
```

in the MATLAB Command Window.

This is the **Model Reference** sample:



S-function

Custom code S-functions run your custom code in MIL simulation mode. They also log block outputs by providing an interface to the TargetLink Data Server, and check if inputs remain within the associated limits.

This sample demonstrates how to prepare a Simulink subsystem for TargetLink using custom code.

The subsystem contains an S-function for which no corresponding TargetLink block exists. Preparation will replace this S-function by a generated TargetLink **Custom Code** block whose interface corresponds to the interface of the S-function it has replaced.

You can start system preparation either interactively with the **System Preparation** dialog, or by typing

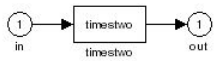
```
» tl_prepare_system('SubSystem','preparation/S_Function')
```

in the MATLAB Command Window. The **System Preparation** dialog can be opened with the **Prepare System** button on the Tools page of the TargetLink Main Dialog, or by typing

```
» tl_prepare_system
```

in the MATLAB Command Window.

This is the **S-function** sample:



DeadZone

This sample demonstrates how to prepare a Simulink subsystem for TargetLink using libmaps. You can start system preparation either interactively with the **System Preparation** dialog, or by typing

```
» tl_prepare_system('SubSystem','preparation/DeadZone_Blocks')
```

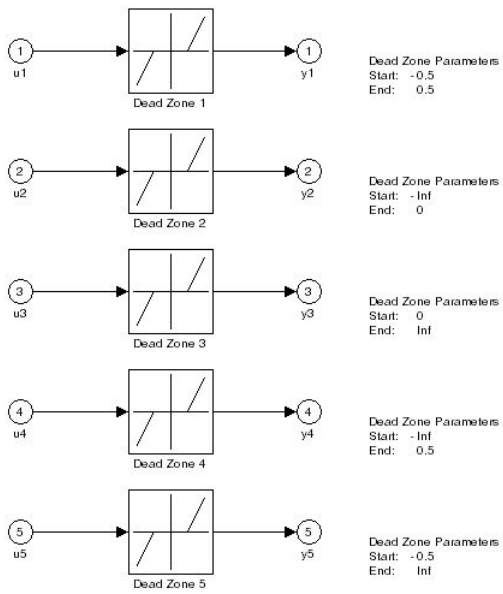
in the MATLAB Command Window. The **System Preparation** dialog can be opened with the **Prepare System** button on the Tools page of the TargetLink Main Dialog, or by typing

```
» tl_prepare_system
```

in the MATLAB Command Window.

Note: This sample interferes with libmaps defined in the `tlsamples` library. To enable preparation, the tlsamples_libmap.m file must be disabled.

This is the **DeadZone** sample:



UserLib (1)

This sample demonstrates how to prepare a Simulink subsystem for TargetLink with blocks that originate from a custom library. A libmap in `tllib_ext_libmap.m` defines that the picontroller block should be replaced by a corresponding block from the tllib_ext library. Production code for the subsystem reuses functions for the PI controller blocks.

You can start system preparation either interactively with the System Preparation dialog, or by typing

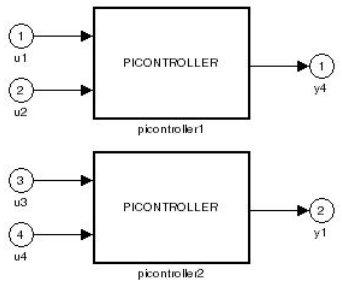
```
» tl_prepare_system('SubSystem','preparation/UserLib_1')
```

in the MATLAB Command Window. The System Preparation dialog can be opened with the Prepare System button on the Tools page of the TargetLink Main Dialog, or by typing

```
» tl_prepare_system
```

in the MATLAB Command Window.

This is the UserLib(1) sample:



UserLib (2)

This sample demonstrates how to prepare a Simulink subsystem for TargetLink that contains a block from a custom library. There are two methods:

1. Prepare the subsystem with its library. This approach results in a prepared subsystem and a prepared library, i.e. the original custom library is modified for TargetLink.

   To do this, type

   ```
   tl_prepare_system('SubSystem','preparation_with_customlib/UserLib_
   2', ...
   'PrepareLibs', 'on')
   ```

   The model and library are not saved after preparation. When you save the library, all blocks in all models that link into the custom library are TargetLink-compliant.

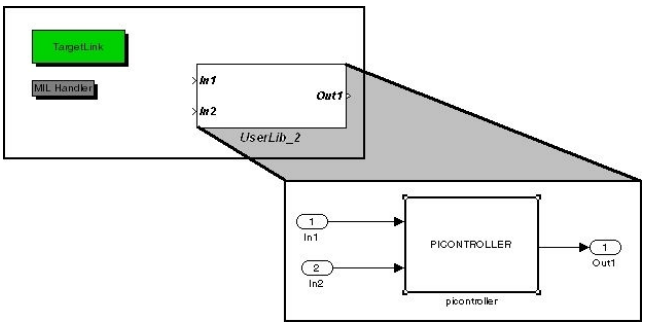
2. Prepare the subsystem with its library, but save them to new model files. This approach results in a prepared subsystem in a new model and two libraries: the original which remains unmodified, and the prepared TargetLink-compliant copy. Additionally, a generated libmap file states that during preparation, blocks that link into the original library should be relinked into the new TargetLink-compliant library.

To do this, type

```
tl_prepare_system('SubSystem','preparation_with_customlib/UserLib_
2', ...
'PrepareLibs', 'on', 'save','on','prepSystem','$_tl');
```

in the MATLAB Command Window. The System Preparation dialog can be opened with the **Prepare System** button on the **Tools** page of the TargetLink Main Dialog, or by typing `tl_prepare_system` in the MATLAB Command Window.

This is the **UserLib(2)** sample:



Bitwise

This sample model demonstrates how to prepare a Simulink subsystem for TargetLink using custom code. You can start system preparation either interactively with the **System Preparation** dialog, or by typing

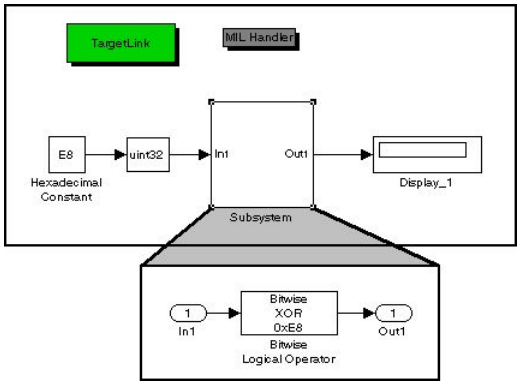
```
» tl_prepare_system('SubSystem','bw_preparation/Subsystem')
```

in the MATLAB Command Window. The **System Preparation** dialog can be opened with the **Prepare System** button on the **Tools** page of the TargetLink Main Dialog, or by typing

```
» tl_prepare_system
```

in the MATLAB Command Window. Note that libmaps and functions from the tlsamples library are needed for this sample.

This is the **Bitwise** sample:



Related topics

Basics

Basics on Model Referencing (📖 TargetLink Customization and Optimization Guide)
Basics on Preparing Simulink Systems for TargetLink Code Generation (📖 TargetLink Preparation and Simulation Guide)

CUSTOM_BLOCKS

Access

In the MATLAB Command Window, enter the following command:

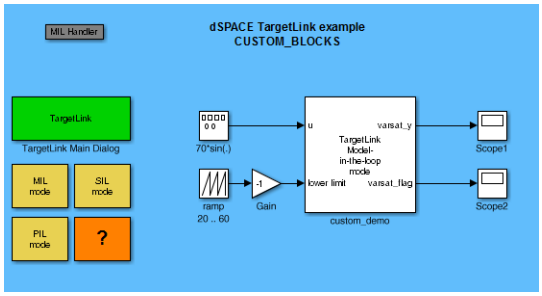
```
tl_demos custom_blocks
```

You can also enter **`tl_demos`** to access all available demos in the MATLAB Help browser. Select a demo and click **Open this example** at the top right.

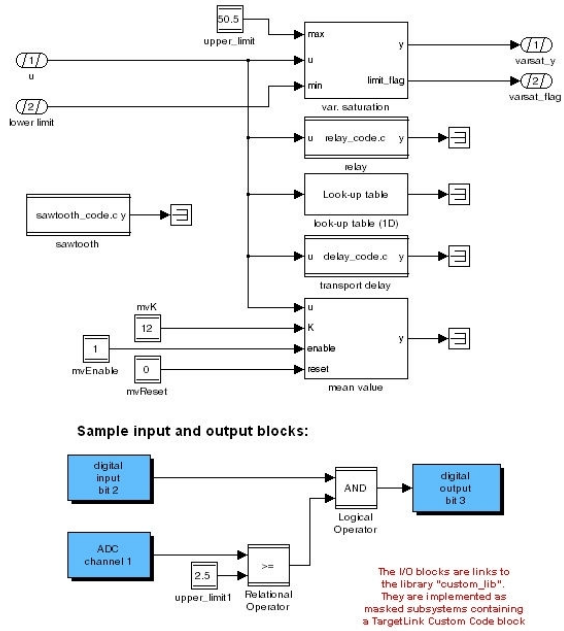
Description

The **`custom_blocks`** demo shows you sample implementations, e.g., sawtooth signal generator, relay, variable saturation, etc.

The illustration below shows the root level of the custom_blocks demo model.



The custom_demo subsystem contains **Custom Code** blocks.



Custom code S-functions

With a **Custom Code** block, you can insert parts of your own code directly into generated code. Generally, the custom code S-function is required for MIL simulation only. However, for production code simulation, it has to be available to make the model Simulink-compliant. This means that there must be custom code S-functions on the MATLAB search path for each **Custom Code** block in the model. In addition, they must be executable without a TargetLink version installed on your PC. Custom Code is treated as black box where only the interface variables are allowed to be optimized. The custom code (external code) is intentionally left unchanged as the Code Generator assumes that it was already optimized.

Related topics

HowTos

How to Build a Custom Code S-Function (Type I) (📖 TargetLink Preparation and Simulation Guide)

CUSTOM_ENHANCEMENT

Access

In the MATLAB Command Window, enter the following command:

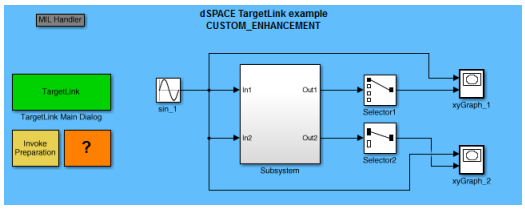
```
tl_demos custom_enhancement
```

You can also enter **`tl_demos`** to access all available demos in the MATLAB Help browser. Select a demo and click **Open this example** at the top right.

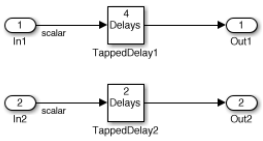
Description

The `custom_enhancement` demo model shows you a subsystem that is to be prepared for TargetLink, and some TargetLink utility blocks.

The following illustration shows the root level of the custom_enhancement demo model.



The **Subsystem** subsystem which is to be prepared contains two Simulink **Tapped Delay** blocks. The **TappedDelay1** block delays its scalar input by 4 sample periods, the **TappedDelay2** block delays its scalar input by 2 sample periods.



Enhancing unsupported Simulink blocks to TargetLink Custom Code blocks

When you prepare the **Subsystem** subsystem of the `custom_enhancement.slx` demo model for TargetLink (use the **Invoke Preparation** button), the two Simulink **Tapped Delay** blocks are enhanced to TargetLink **Custom Code** blocks. The preparation follows the rules defined in the `tappeddelay_libmap.m` file.

The following files come into play during Simulink-to-TargetLink preparation of the `custom_enhancement.slx` demo model:

- `tappeddelay_libmap.m`:

  This M file provides the libmap for a Simulink **Tapped Delay** block. It contains the following block references:

  ```
  libmap(m).slblock = 'built-in/Tapped Delay';
  libmap(m).tlblock = 'USE_CUSTOM_CODE_BLOCK';
  ```

  `USE_CUSTOM_CODE_BLOCK` is a predefined keyword with which the Simulink block is not replaced by but enhanced to a TargetLink **Custom Code** block (type II). The original Simulink **Tapped Delay** block is kept (under the mask) and used for MIL simulation. After model preparation is finished you can easily check this by looking under the **Custom Code** block's mask (Ctrl+U). The dialog of the Simulink **Tapped Delay** block is displayed.

- `enhanceTappedDelay.m`:

  This M file sets various **Custom Code** block properties during the preparation process, for example, it introduces all the **Custom code symbols** that are used

in the custom code. The file also specifies the width of the **delayBuffer** state variable via the **getDelayBufferSize** expression. This expression is resolved at code generation time and provides the actual variable width.

Associating the **Custom code symbols** with the properties of the underlying Simulink block is a characteristic of **Custom Code** blocks (type II). This ensures that Simulink and TargetLink always use the same property values. For example, the **delayBuffer** state variable is associated with the **vinit** Simulink property. Since the association does not become active until the block is enhanced, the value must be transferred from Simulink to TargetLink, as you can see in the following code line:

```
'state.initial', get_param(hBlock, 'vinit')
```

Hence, the property value obtained via the Simulink API, `get_param(<block>, 'vinit')`, is equal to the value obtained via the TargetLink API, `tl_get(<block>, param(1).initial`.

The file is referenced in the `tappeddelay_libmap.m` file:

```
libmap(m).funcSL2TL = 'enhanceDelay';
```

- `prodcode_tappeddelay.c`:

  This C file represents the custom code template. It provides the production code pattern for a Simulink **Tapped Delay** block. During code generation, the code of the custom code template is copied to the associated source code file. The width macros are replaced by the width values they represent, and the variables are replaced by their corresponding interface variables, specifically for each block instance.

  This is the content of the `prodcode_tappeddelay.c` custom code template:

```
/* fxp_output_begin */

#define NUM_CHANNELS (OUT_1_NUMELEMS)

#if NUM_CHANNELS == 1

    y = delayBuffer[ringBufIdx];

#else

    for (outIdx = 0; outIdx < NUM_CHANNELS; outIdx++)

    {

        y[outIdx] = delayBuffer[(ringBufIdx + outIdx) %
NUM_CHANNELS];

    }

#endif

#undef NUM_CHANNELS

/* fxp_output_end */
```

```
/* fxp_update_begin */

#define NUM_CHANNELS (OUT_1_NUMELEMS)

delayBuffer[ringBufIdx] = u;

ringBufIdx = (ringBufIdx + 1) % NUM_CHANNELS;

#undef NUM_CHANNELS

/* fxp_update_end */
```

This file is referenced in the `enhanceTappedDelay.m` file:

```
tl_set(hBlock, 'codefile', 'prodcode_tappeddelay.c');
```

Note

Both TargetLink Custom Code blocks reference the same `prodcode_tappeddelay.c` custom code template.

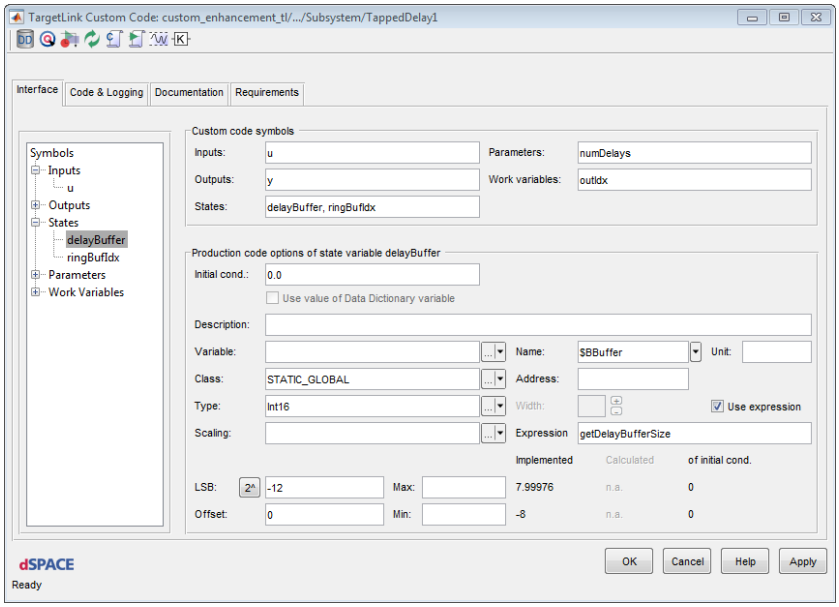
Width-invariant custom code template

Although the two TargetLink Custom Code blocks reference the same custom code template, they can handle output signals of varying width (TappedDelay1: width=4, TappedDelay2: width=2). This is possible because the `prodcode_tappeddelay.c` custom code template is width-invariant for the output variables because it uses a width macro for the block's output variable, i.e., `OUT_1_NUMELEMS`.

The width of the delayBuffer state variable depends on the number of delays as follows:

$\text{Width}_{\text{Buffer}} = \text{Delays}$

The width of the delayBuffer state variable is specified via the `getDelayBufferSize` expression. The analogously named script (`getDelayBufferSize.m`) determines the actual width according to the formula shown above by using the appropriate width macros.



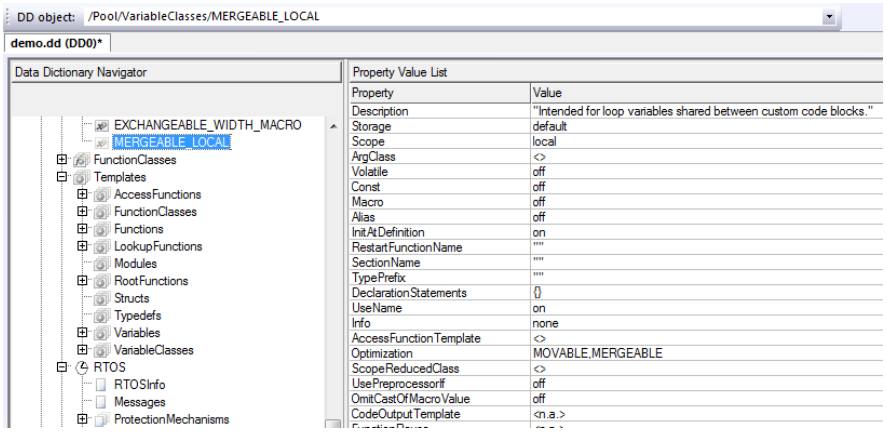
Using work variables as loop indices in custom code templates

The outIdx work variable which is used as a loop index references the MERGEABLE_LOCAL variable class. This variable class allows outIdx to be shared by the two Custom Code block instances.

Note

The MERGEABLE_LOCAL variable class is not predefined. You must define it in the TargetLink Data Dictionary.

The following illustration shows the required property settings of the MERGEABLE_LOCAL variable class in the TargetLink Data Dictionary:



Related topics

Basics

Basics on Defining the Interface of Custom Code Blocks (📖 TargetLink Preparation and Simulation Guide)
Basics on Using Width Macros in Custom Code (Type II) Block (📖 TargetLink Preparation and Simulation Guide)
Details on Masking Simulink Blocks with TargetLink Custom Code (Type II) Blocks (📖 TargetLink Preparation and Simulation Guide)

Basic Features

Where to go from here

Information in this section

PIPT1 22
VOICE_COIL_FLEXIBLE_POSITIONER (VCFP) 25
ENABLSYS 31
POSCONTROL 35
STRUCT_INTERFACES 41
MATRIX 44
BUS_STRUCT 50
VARIABLE_VECTOR_WIDTH 55
FUNCTION_REUSE 56
MULTIPLE_INSTANCES_REFMODEL 60
EMBEDDING_EXTERNAL_CODE 64
AUTODOC_GENERATION 67
VARIABLE_INITIALIZATION 72
FUNCTION_VARIANTS 75
ENUM_TYPES 78
ENUM_BLINKER 81
BUS_CC 85
MATRIX_INVERSE 92

PIPT1

Access

In the MATLAB Command Window, enter the following command:

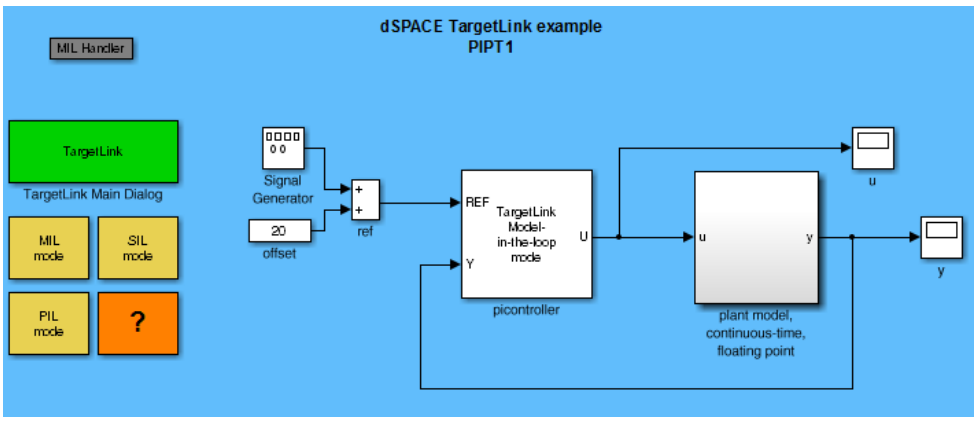
```
tl_demos pipt1
```

You can also enter **`tl_demos`** to access all available demos in the MATLAB Help browser. Select a demo and click Open this example at the top right.

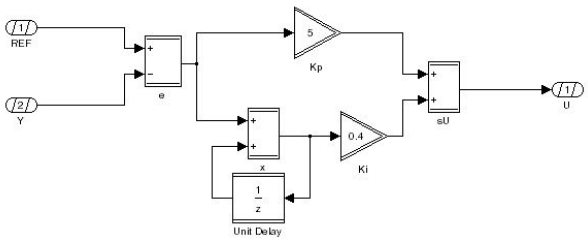
Description

The `PIPT1` demo model shows you a continuous-time first-order lag system with fixed-point PI controller.

The illustration below shows the root level of the PIPT1 demo.



The picontroller subsystem shows some simple TargetLink blocks such as InPort, OutPort and Unit Delay blocks.



Variable classes

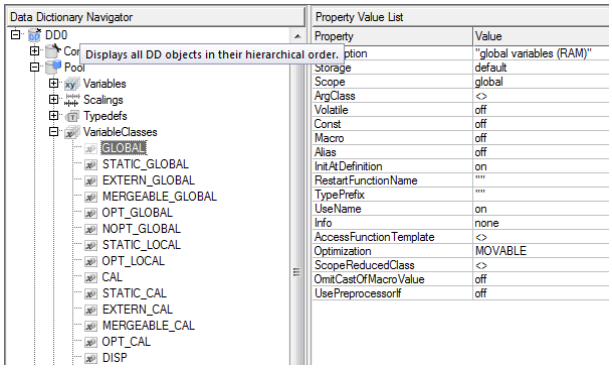
A variable class is a set of properties that define the appearance of a variable in the generated production code.

During code generation, the Code Generator uses sophisticated optimization routines to generate optimum code for the model. Block outputs, states, and parameters are represented by variables in the generated production code if they could not be eliminated to reduce the memory consumption.

For example, parameters are usually inlined in the operations with which they are associated , except for tables, which are usually represented by global C arrays.

Superfluous block outputs are removed, if possible. If not, the Code Generator tries to allocate them with the smallest possible scope, for example, as static local variables of functions.

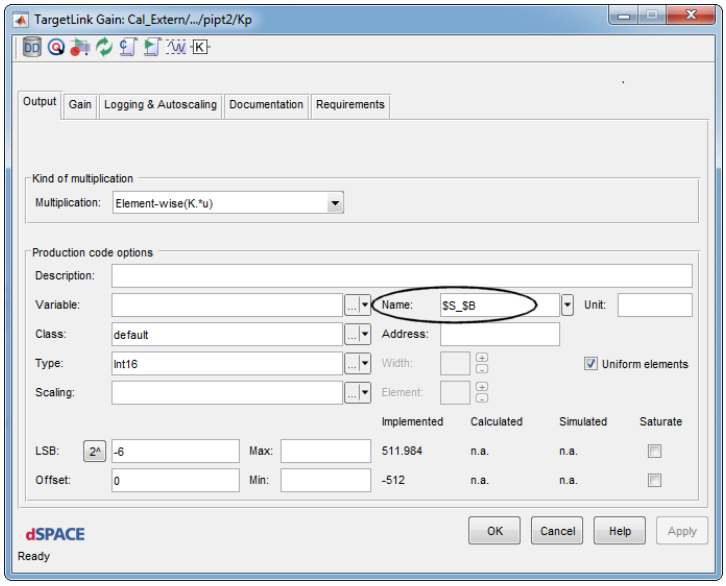
Variable classes are defined in the **TargetLink Data Dictionary**, where each VariableClass object corresponds to one variable class definition.



Name macros

In TargetLink, the name of a variable (a block output, state, or parameter) in the generated production code is set in the dialog of the block or the Property Manager.

You can either define names manually or let TargetLink generate them automatically by using name macros. Name macros begin with a $ character, followed by a letter or an identifier enclosed in parentheses. During code generation, the macro is replaced by the string it represents.

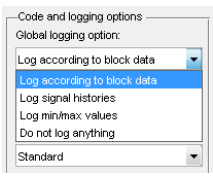


Logging

TargetLink provides data display and logging methods. For example, logging of block outputs is required if you want to acquire signal data during the simulation.

The various TargetLink block properties (for example, scaling data) have an impact on the generated production code. This means that choosing suitable parameters can lead to optimal code, whereas unsuitable parameters can lead to inefficient code. To gain information on this impact, you can log and plot selected signals during simulations in the MIL, SIL or PIL simulation modes. By comparing the signal histories, you can evaluate the fixed-point effects of the different block properties on the production code (for example, quantization and overflows). This allows you to find better-suited block properties (for example, the scaling data) and gradually improve your model until the generated code is ready for production.

TargetLink also lets you specify the plot interval. It is recommended to set the time interval to `inf` so that the plot interval is updated only at the end of data acquisition. The logging speed basically depends on the number of logged signals and the specified time interval for plot updates. The simulation duration has only minor relevance for the logging performance.



Worst-case autoscaling

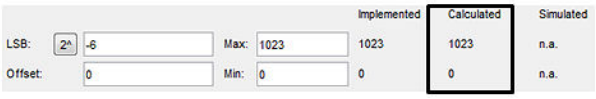
Blocks with user-defined range limits are used as data points to calculate the worst-case ranges of neighboring blocks.

To calculate the worst-case ranges for this demo model, you require additional range information inside a loop. For this purpose, you must specify maximum and minimum values for the Unit Delay block inside the picontroller subsystem.

To avoid warnings you can also define the upper and lower limit of the gain parameter of both Gain blocks.

The Autoscaling Tool calculates the worst-case output range of a block by determining the minimum and maximum values a block's output or state variable can take on (with respect to the range of the input).

Both inport blocks inside the picontroller subsystem need additional range information to enable autoscaling.



Related topics

Basics

Basics on Logging (📖 TargetLink Preparation and Simulation Guide)
Basics on Using Name Macros (📖 TargetLink Customization and Optimization Guide)
Basics on Variable Classes (📖 TargetLink Customization and Optimization Guide)
Basics on Worst-Case Range Calculation (📖 TargetLink Preparation and Simulation Guide)
Details on Preconfigured VariableClass Objects in the Data Dictionary (📖 TargetLink Customization and Optimization Guide)

VOICE_COIL_FLEXIBLE_POSITIONER (VCFP)

Access

In the MATLAB Command Window, enter the following command:

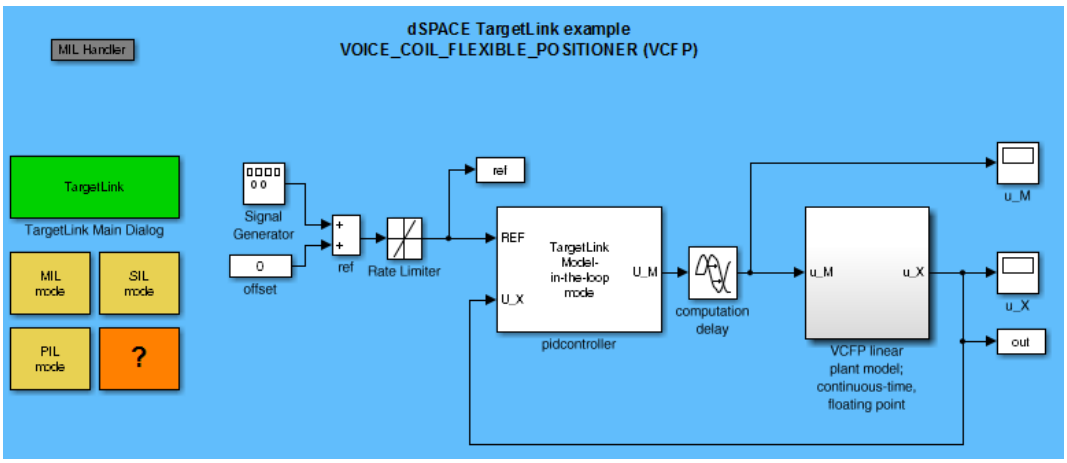
```
tl_demos vcfp
```

You can also enter **`tl_demos`** to access all available demos in the MATLAB Help browser. Select a demo and click **Open this example** at the top right.

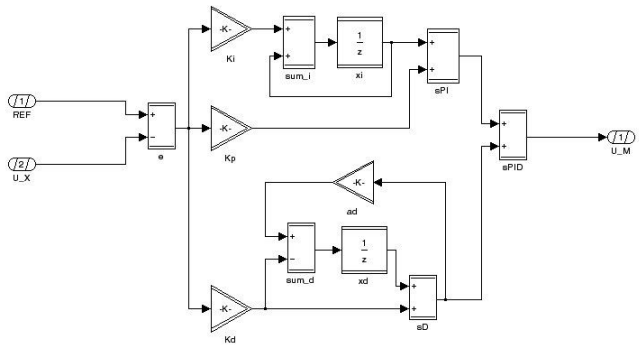
Description

The `voice-coil-driven flexible positioner (VCFP)` demo model shows you a 7th-order plant controlled by a fixed-point PID controller.

The illustration below shows the root level of the VCFP demo.



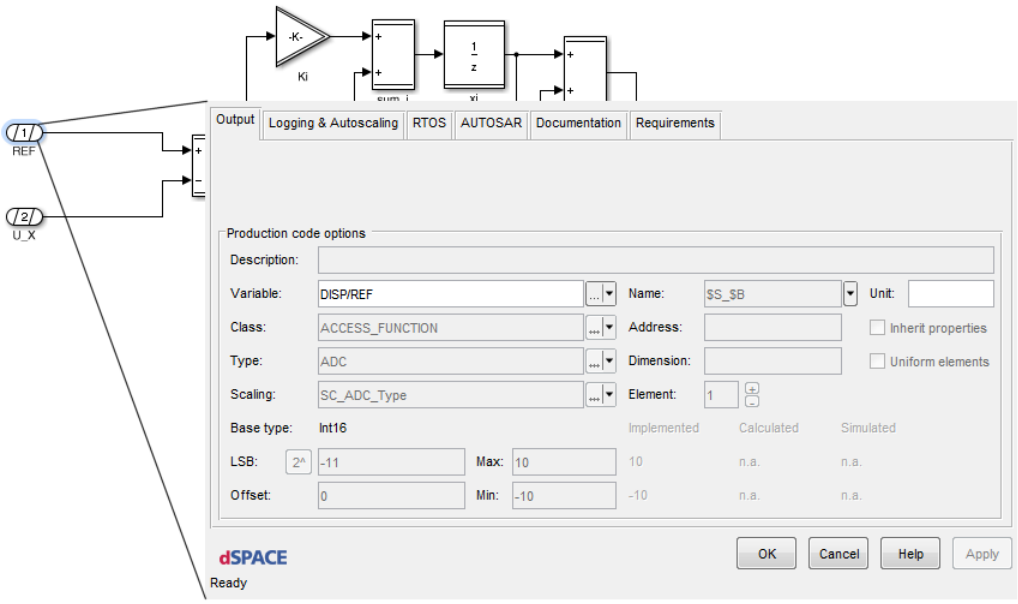
The **pidcontroller** subsystem shows some simple TargetLink blocks such as **InPort**, **OutPort**, **Sum**, **Unit Delay** or **Gain** blocks.



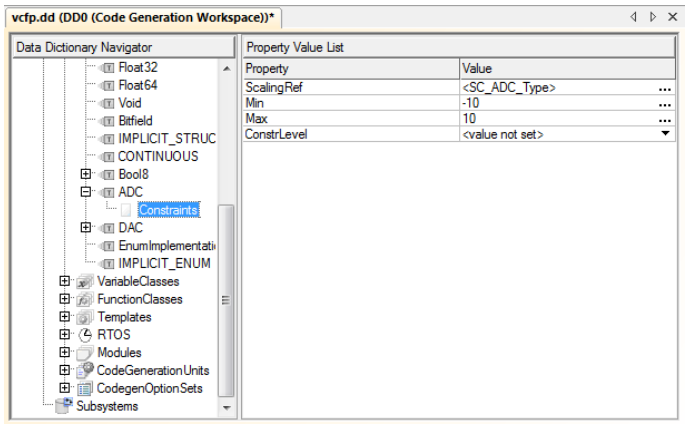
Applying user-defined data types

User-defined data types improve the code readability and are also useful if you often have to assign identical constraints (fixed scaling parameters and min/max values) to multiple signals of one kind in your model.

The following illustration shows the TargetLink InPort **DISP/REF**, which references the user-defined data type **ADC**.

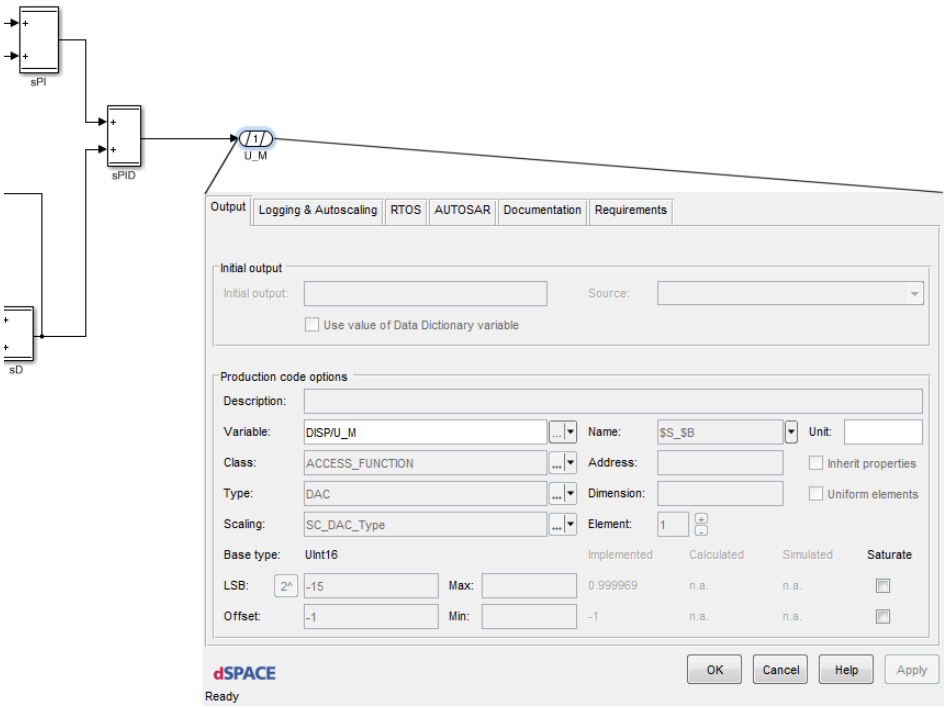


The user-defined ADC data type is defined in the **/Pool/Typedefs** area in the Data Dictionary Manager in addition to DAC.

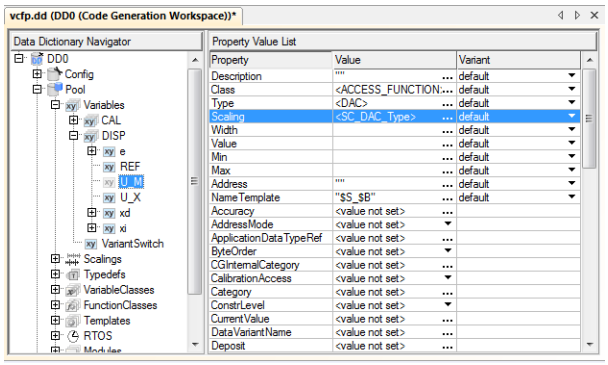


Applying block output variables

Block output variables can be predefined in the Data Dictionary Manager, which can be useful if you need to change properties such as Type or Class globally. The following illustration shows the TargetLink OutPort U_M, which references a set of predefined properties.

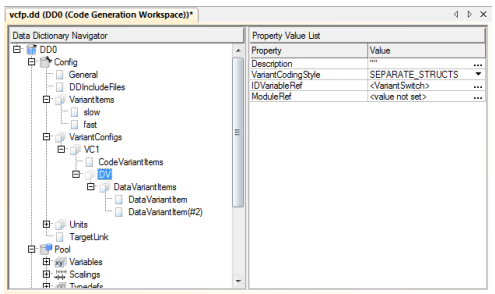


The output variable U_M is defined in the **`/Pool/Variables`** area in the Data Dictionary Manager as shown below.



Applying variant coding

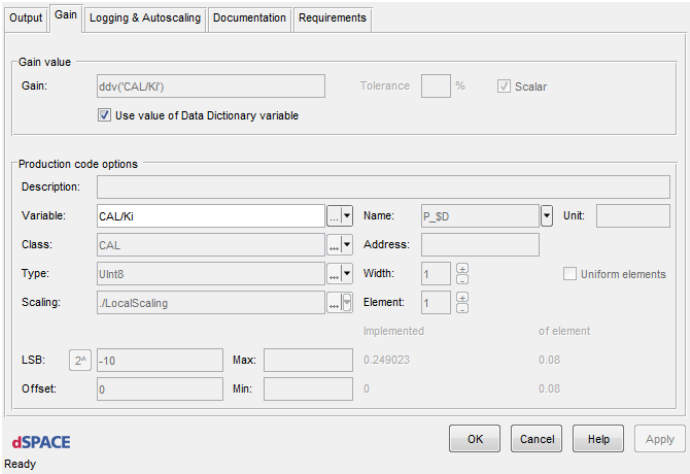
A variant is a variation of a product that has the basic product features plus its own specific characteristics. Variants are a powerful means to cope with the increasing number of variations in the field of vehicle control applications, i.e., they allow you to adapt a basic product to the needs of specific vehicle control applications. The following illustration shows the settings necessary if you want to switch between variants at ECU run time.



Referencing value property of DD variable objects

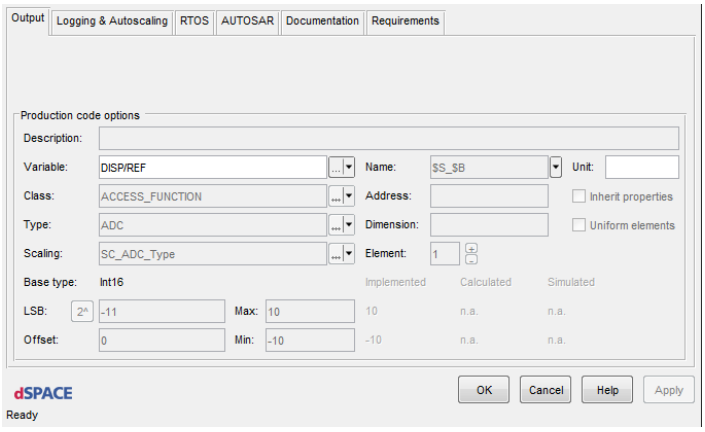
When variants are used, the value assigned to the initial variant item must be the same as the value specified in the dialog of the TargetLink block. If it is not, an error occurs during code generation. To guarantee consistency, you can use the API function **`ddv`**, which reads/returns the current value of a DD variable object according to the active variant configuration.

Enter **`ddv('<ddObject>')`** in the value edit field of the block parameter. For example, enter **`ddv('Cal/Ki')`** in the Gain edit field of a Gain block's dialog in your model.

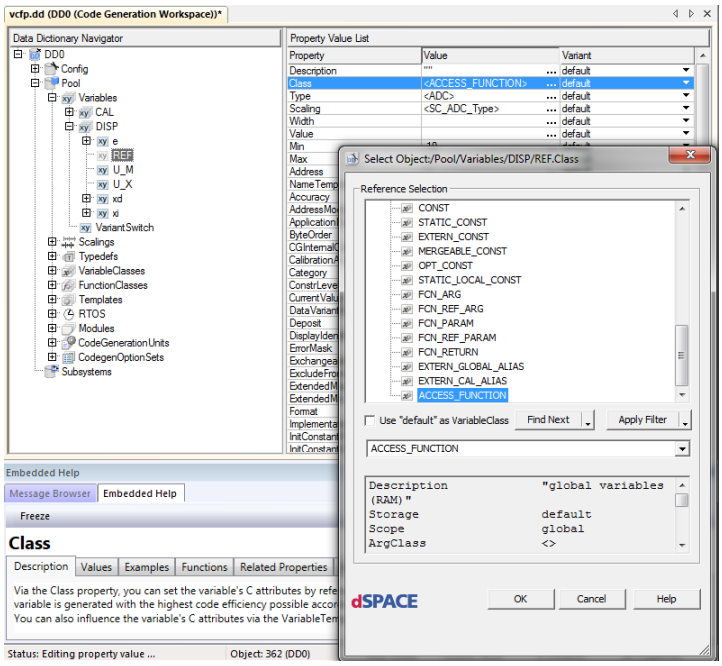


Usage of access function templates

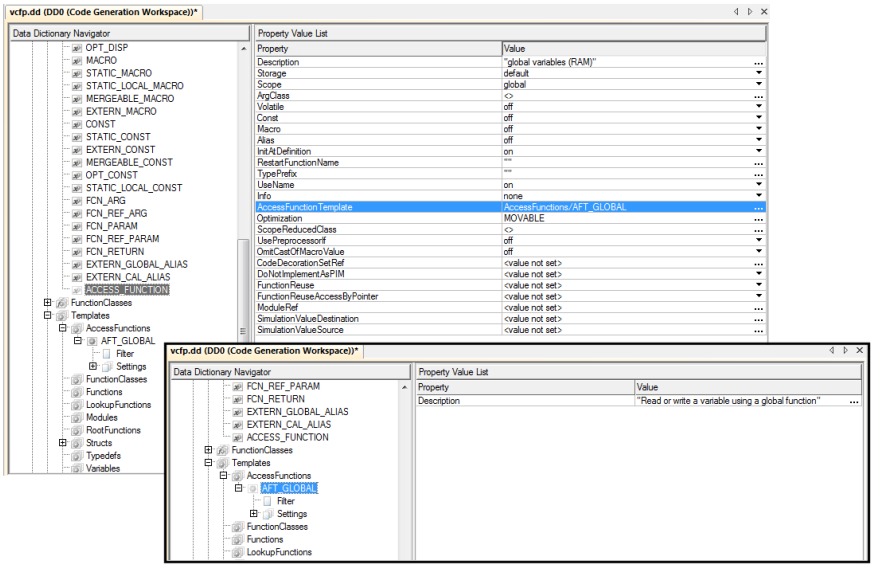
Access functions provide access to interface signals, i.e., they can be specified for variables that represent the interface of a subsystem. This is done by providing access functions for reading or writing the signals. Using access functions is advantageous if code is delivered to external users and you want to hide the way in which data signals are implemented by global variables. The following illustration shows the output variable **DISP/REF**, which is referenced from the DD. Variables defined in the DD cannot be edited in the block dialog, which is why the edit fields are disabled.



The output variable DISP/REF itself references the variable class ACCESS_FUNCTION.



The variable class references the access function template AFT_Global as shown below.



Related topics

Basics

Basics on Access Functions (📖 TargetLink Customization and Optimization Guide)

HowTos

How to Create User-Defined Data Types (📖 TargetLink Customization and Optimization Guide)
How to Specify Data Variants (📖 TargetLink Customization and Optimization Guide)

ENABLSYS

Access

In the MATLAB Command Window, enter the following command:

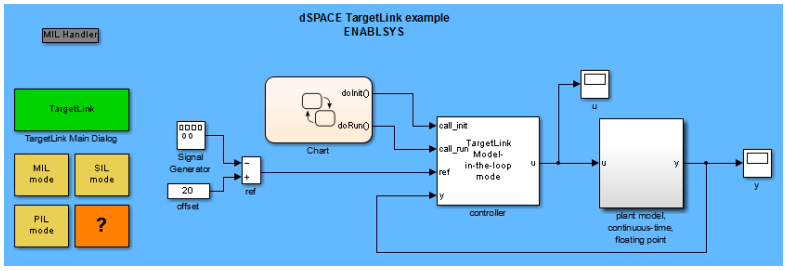
```
tl_demos enablsys
```

You can also enter **`tl_demos`** to access all available demos in the MATLAB Help browser. Select a demo and click **Open this example** at the top right.

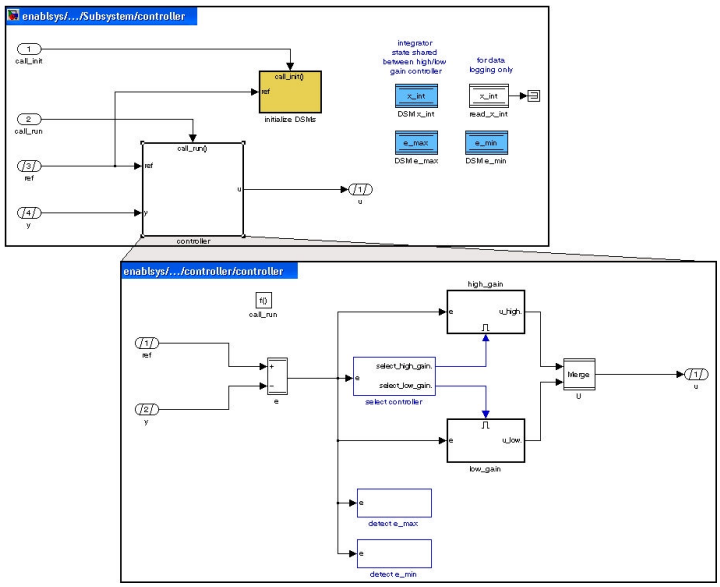
Description

The `enablsys` demo model shows you a first-order lag plant and a PI controller with gain scheduling, which is modeled by means of enabled subsystems.

The illustration below shows the root level of the ENABLSYS demo model.

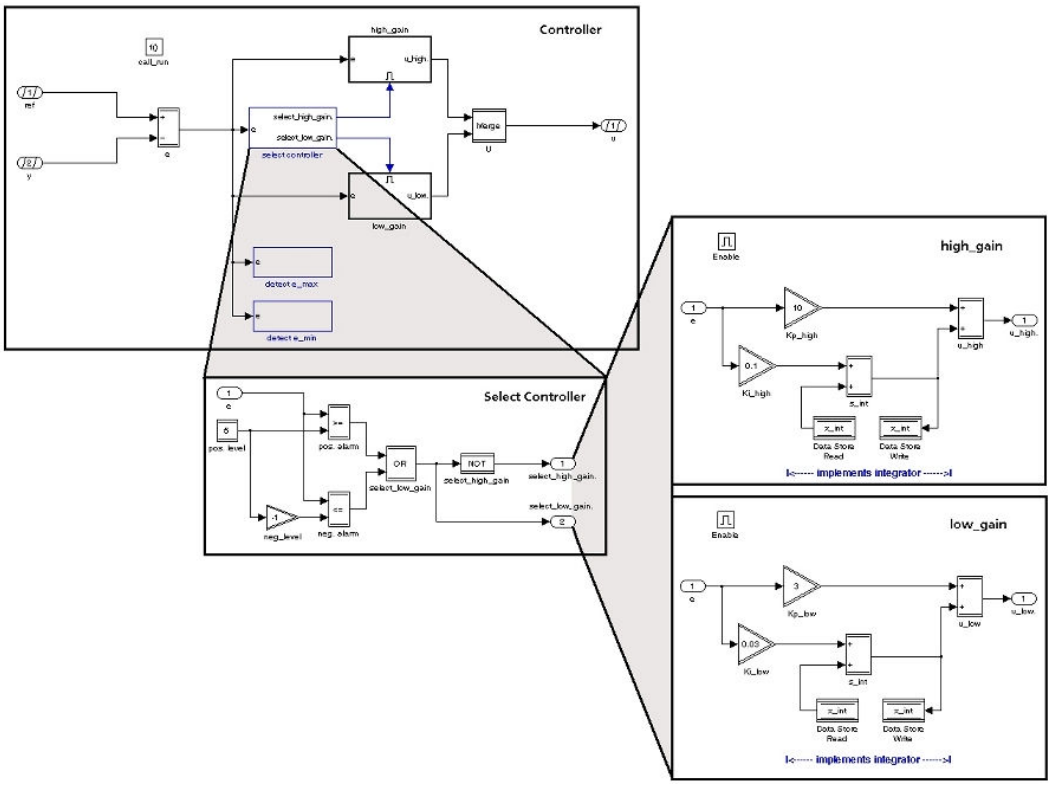


The **controller** subsystem(s) contains Logical, Relational and Merge Enabled/Triggered subsystems.



Enabled subsystems

Enabled subsystems are subsystems that execute at each simulation step if the external input/control signal (scalar or element of a vectorized input signal) has a positive value. Adding an Enable block to a subsystem makes it an enabled subsystem and implements a data-based control flow for controlling its execution. Enabled subsystems also give you control over the initial value of the output signals and whether or not to reset states on each enable. Enabled subsystems require subsystem output variables to be implemented as **static**.



The subsystems high_gain and low_gain are implemented as enabled subsystems. They are executed according to their control signals (select_high_gain, select_low_gain). Both subsystems also implement an integrator using Data Store Read and Data Store Write blocks that exchange data at each simulation step.

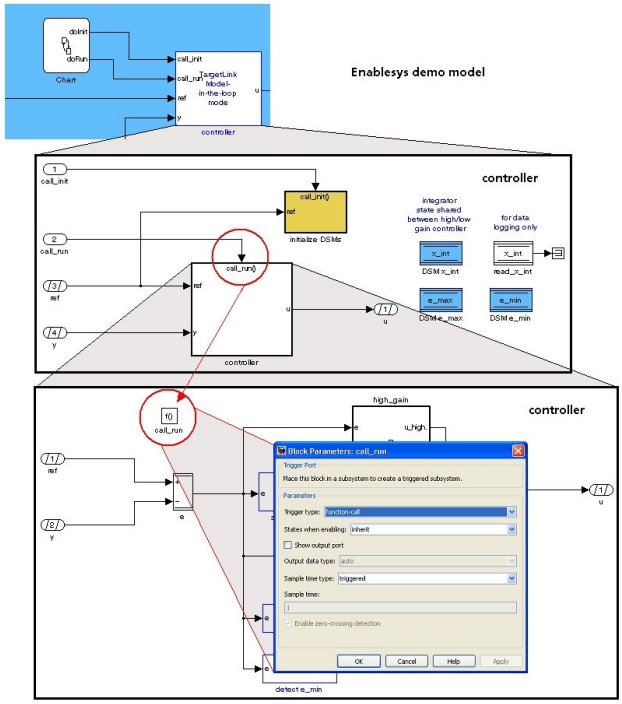
Function-call-triggered subsystems

Triggered subsystems are subsystems that execute each time a trigger event occurs. Adding a Trigger block to a subsystem makes it a triggered subsystem and implements an event-based control flow for controlling its execution.

In an event-based control flow, there is a distinction between events that originate outside a TargetLink subsystem and control functions inside it, and events that originate and end inside a TargetLink subsystem.

A function-call event is sent from its source to its sink. Stateflow charts can act as both the source and the sink. A function-call generator can act as the source and a function-call-triggered subsystem as the sink.

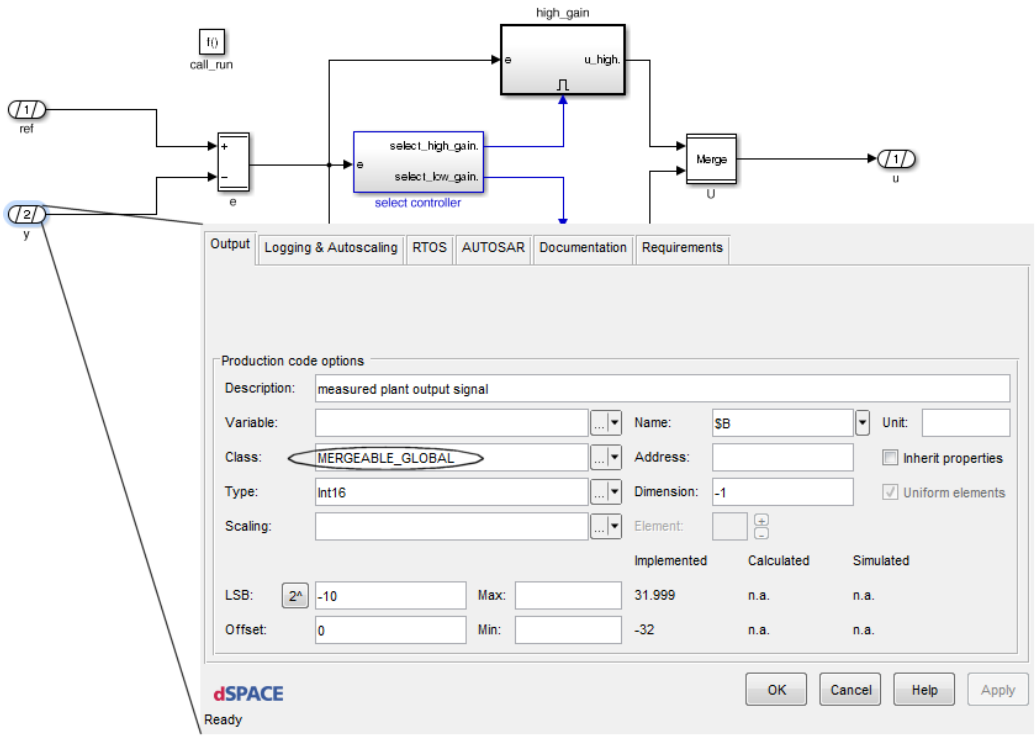
The excerpt from the ENABLSYS demo model below shows how to realize a function-call-triggered subsystem and also how to implement state sharing between a high-gain and low-gain controller.



Mergeable variables

Mergeable variables are identified and mapped to one another, if possible, during code generation. This means multiple definitions of a variable are identified by its name and mapped to one variable. They can be referenced from multiple blocks in the model.

To see an example of using mergeable variables, open the ENABLSYS demo model and navigate to the controller subsystem. Open an InPort block and set the Class Attribute to MERGEABLE_GLOBAL.



Related topics

Basics

Details on Executing Enabled Subsystems (📖 TargetLink Customization and Optimization Guide)
Details on Implementing Externally Function-Call-Triggered Subsystems (📖 TargetLink Customization and Optimization Guide)
Details on Implementing Function-Call-Triggered Subsystems (📖 TargetLink Customization and Optimization Guide)
Details on Preconfigured VariableClass Objects in the Data Dictionary (📖 TargetLink Customization and Optimization Guide)

POSCONTROL

Access

In the MATLAB Command Window, enter the following command:

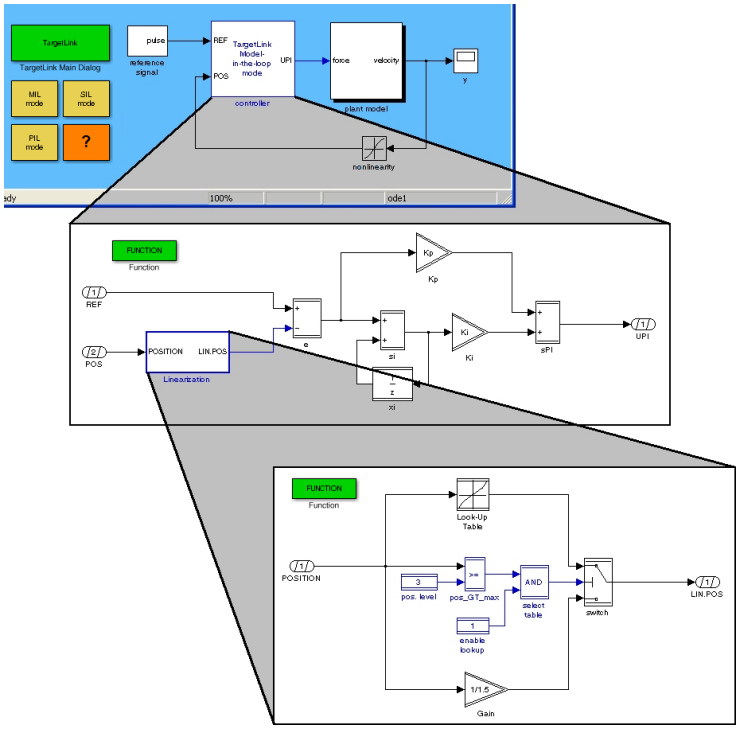
```
tl_demos poscontrol
```

You can also enter **`tl_demos`** to access all available demos in the MATLAB Help browser. Select a demo and click **Open this example** at the top right.

Description

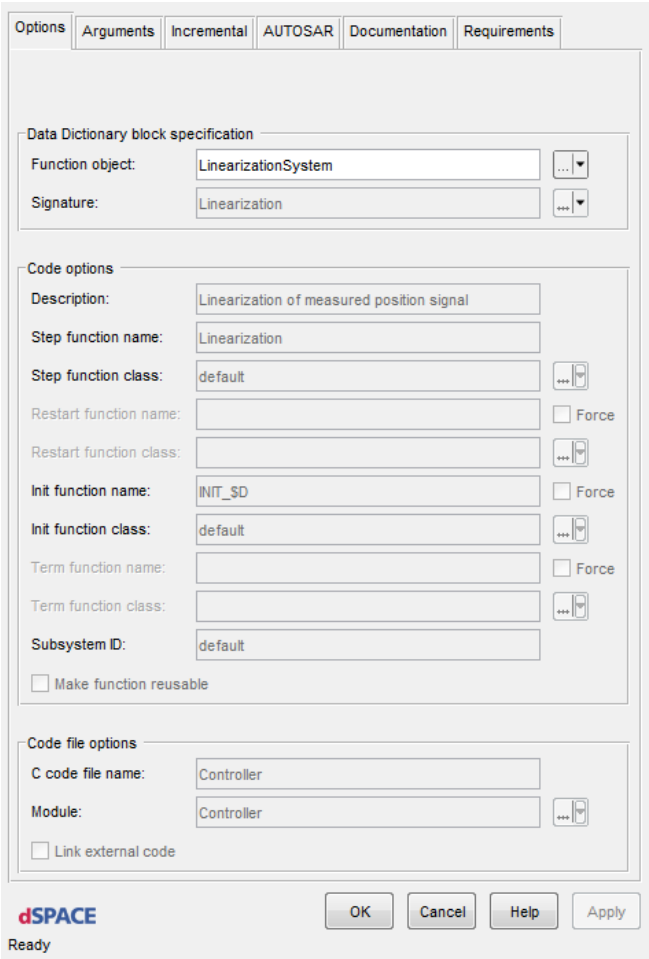
The `poscontrol` demo model shows you a PI controller with compensation of nonlinearity.

The followig illustration shows the different levels of the demo model.



Function block

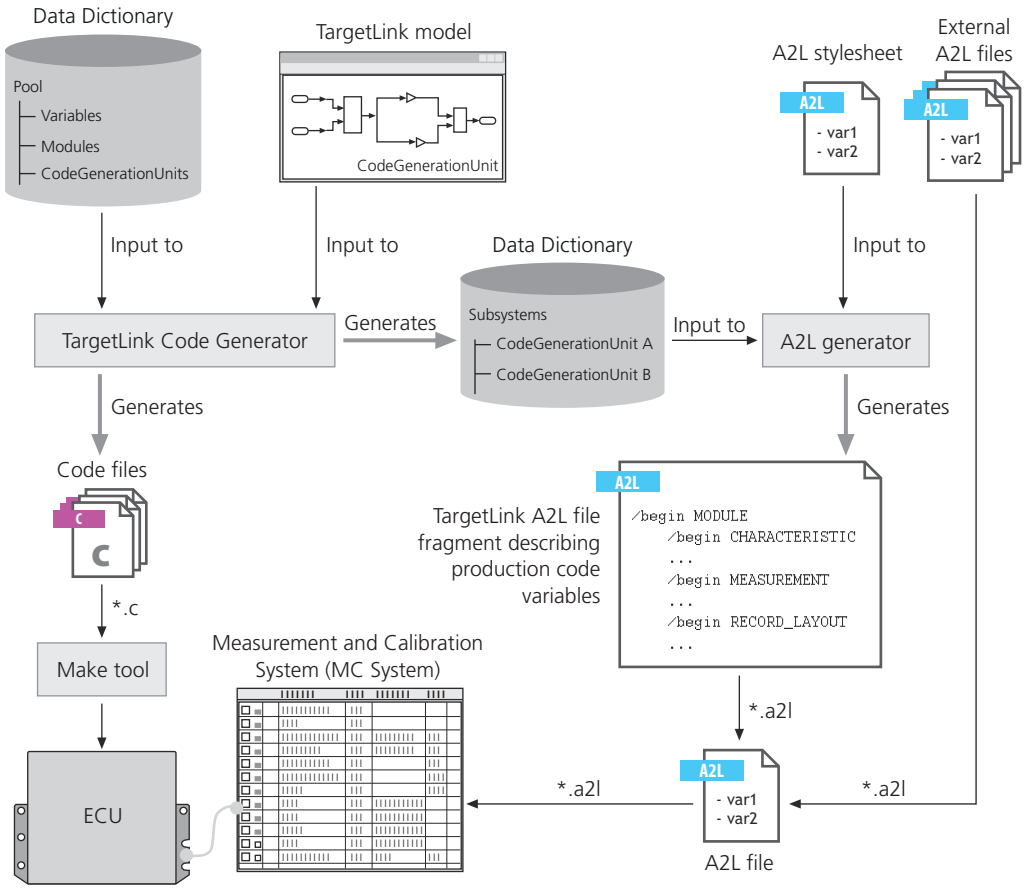
To let TargetLink generate separate C functions for a specific subsystem, simply insert a Function block into the corresponding subsystem. This prevents the subsystem from being inlined in the generated production code. Once you have instructed the Code Generator to implement a subsystem by C functions, you have to specify the functions to be generated, i.e., you have to specify the C function subsystem interface precisely. You can specify the interface in the Function block or in the corresponding Function block object in the Data Dictionary.



ASAP 2 file generation

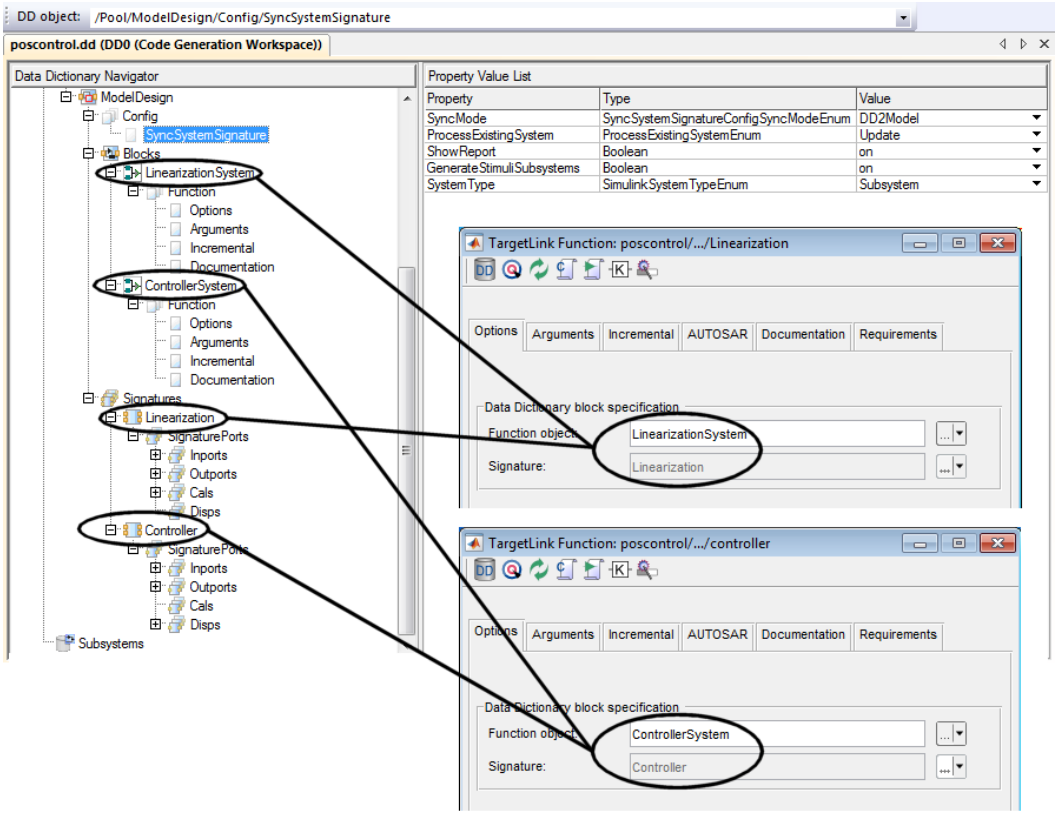
If you want to access an ECU for calibration, you need ASAM MCD-2MC (A2L) files. They provide a measurement and calibration (MC) system with all information required for accessing ECUs. You can generate an A2L file for the parts of your application for which you generated production code with TargetLink. Generating A2L files is possible by using the ASAM MCD-2MC export module of the **TargetLink Data Dictionary**. During A2L file generation, the export module maps properties of TargetLink blocks to corresponding A2L file elements.

The illustration below shows the relationship between production code generation, A2L file generation, and MC systems.

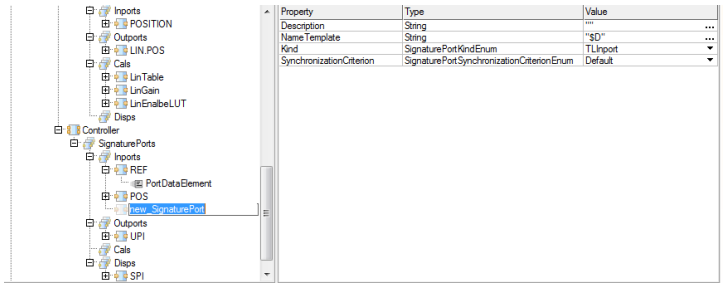


Synchronization of a function system's signature

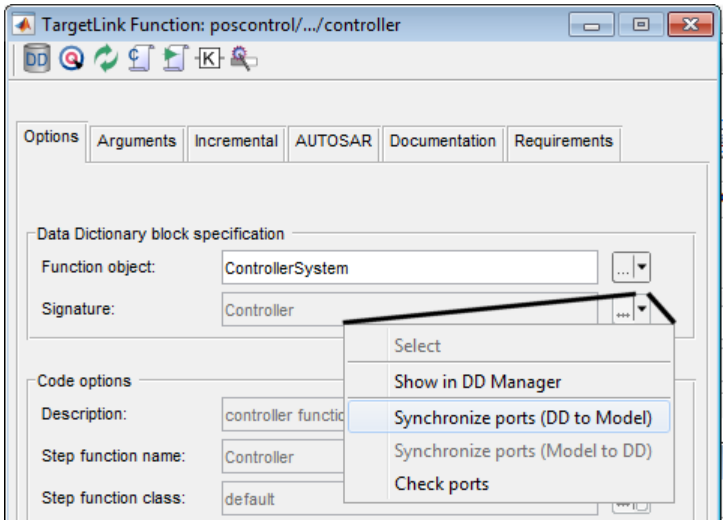
The demo contains two subsystems (Controller and Linearization). Both subsystems contain a Function block connected with a DD Function block object and a DD Signature object. DD Signature objects contain interface specifications (port data). Refer to the following illustration:



You are recommended to change the port data specified in the DD Signature object to observe TargetLink's synchronization mechanics. You can add, change or delete port data according to your needs.



You can then synchronize the data between the interface (port data) specified in the DD Signature object and the modeled interface of the function system.



Synchronization is useful if:

- You want to perform consistency checks between the interface (port data) specified in the DD Signature object and the modeled interface of the function system (Check ports).
- You imported or specified port data in your Data Dictionary and want to synchronize it with the subsystem's interface.

  You can update or synchronize the interfaces in the model with the specifications of the DD Signature object via the DD to Model command in the Function block, as shown above. TargetLink generates new Port blocks automatically according to the specifications in the DD Signature object. However, superfluous Port blocks are not deleted automatically during synchronization. Instead, you are informed about these blocks.
- If you do not have DD Signature object specifications but want to transfer your modeled interface data to the Data Dictionary, you can also synchronize the port data to the Data Dictionary (Model to DD).
- You can also synchronize the data via the Synchronize system signature command in the context menu of DD Signature objects in the Data Dictionary Manager. You can specify the synchronization behavior in the `/Pool/ModelDesign/Config/` by editing the properties of the SyncSystemSignature object.

  Also refer to the `tlSyncSystemSignature` API function.
- An HTML report is generated whenever you check or synchronize ports. The report contains detailed information about differences in the specification of the DD Signature object and the interface of the function system.

Related topics

Basics

Centrally Specifying and Synchronizing Function System Signatures (📖 TargetLink Customization and Optimization Guide)
Introducing A2L Files (📖 TargetLink Interoperation and Exchange Guide)

References

Function Block (📖 TargetLink Model Element Reference)
tlSyncSystemSignature (📖 TargetLink API Reference)

STRUCT_INTERFACES

Access

In the MATLAB Command Window, enter the following command:

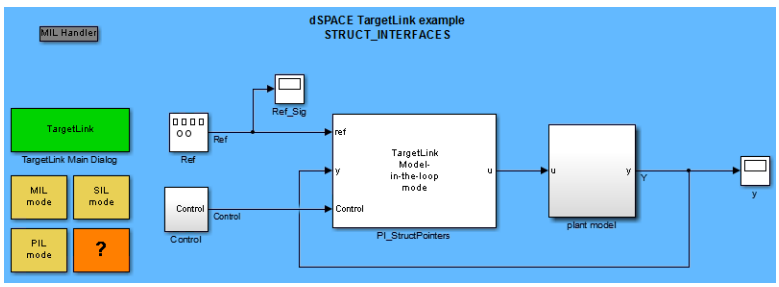
```
tl_demos struct_interfaces
```

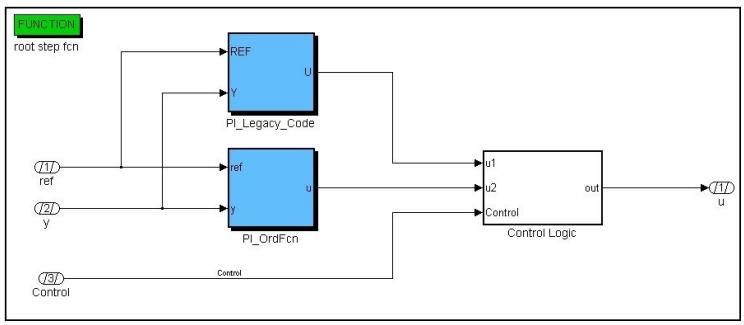
You can also enter **`tl_demos`** to access all available demos in the MATLAB Help browser. Select a demo and click **Open this example** at the top right.

Description

The **`struct_interfaces`** demo model shows you the use of pointers to structures in function interfaces. The model contains two different realizations of PI controllers (all in floating-point arithmetic).

The illustration below shows the root level of the Struct_Interfaces demo model.





Pointers to structures

TargetLink supports pointers to structured variables (structs) as formal parameters in function signatures.

These are some of the benefits of using pointers to structs in function signatures to structure production code:

- You can keep the function signature concise.
- You can minimize stack consumption, as TargetLink generates the actual parameters for the structs only once in the production code.
- You can avoid using global variables for parameter passing.

Additional information

- Struct_Interfaces/PI_StructPointers/Subsystem/PI_StructPointers:

  PI_Legacy_Code: The block constitutes an external subsystem with a function interface containing pointers to structures. The subsystem is used to integrate legacy code with pointers to structures for production code generation as well as SIL/PIL simulation. The MIL simulation is carried out entirely on the block level as usual with external subsystems.

  PI_OrdFcn: The block contains an ordinary PI controller for which TargetLink creates a separate function. It's interface contains pointers to structures.

  Root step function: The root step function created for the entire TargetLink subsystem also possesses a function interface with pointers to structures due to the specifications in the Inports and Outports (gray background). Please note also that on the "Arguments tab of the Function block root step fcn additional pointers are specified. Consequently, they appear in the function interface of the root step function also and can be passed on to inner subsystems, although they are not used on the root level. The parameter order of the function interface including additional pointers can also be specified on the Arguments" tab. To navigate quickly from a block to the respective DD variable object, right click on the variable field in the block dialog and select Show in DD Manager from the context menu.
- Struct_Interfaces/PI_StructPointers/Subsystem/PI_StructPointers/PI_Legacy_Code:

The specified step function in the PI_Legacy_Code Function block is an GLOBAL_FCN with C code file name set to PI_Legacy_Code. TargetLink generate only a call to the function. The subsystem is used to integrate external code for the file in PI_Legacy_Code.c with pointers to structures. The legacy code file PI_Legacy_Types.h contains the required types. Please note that a module template in the DD is used to suppress the generation of the code for this function. The gray blocks reference components of structures in the DD, whose variable classes possess the ref_param scope. Consequently, the corresponding structures are passed to the subsystem's function PI_Legacy_Code as pointers to structures. In order to avoid superfluous copy operations, variable merging can be used. The REF and Y blocks reference identical DD components as the ref and y blocks on the root level thereby reducing memory consumption. To enable variable merging, variable classes of the structure components must possess the MERGEABLE optimization attribute. Please note that variable merging is only possible if variable names are identical. Please note also, how the variable class (via its ArgClass property) and the name property of the DD variable p_PI_LegacyCode_Instance_Spec (referenced from Kp, Ki and UnitDelay) are used to specify properties of the actual variable that p_PI_LegacyCode_Instance_Spec points to.

- Struct_Interfaces/PI_StructPointers/Subsystem/PI_StructPointers/PI_Ord Fcn:

  The gray blocks reference components of structures in the DD, whose variable classes possess the ref_param scope. Consequently, the corresponding structures are passed to the subsystem's function PI_Step_Interior as pointers to structures. In order to avoid superfluous copy operations, variable merging can be used. The ref and y blocks reference identical DD components as the ref and y blocks on the root level thereby reducing memory consumption. To enable variable merging, variable classes of the structure components must possess the MERGEABLE optimization attribute. Please note that variable merging is only possible if variable names are identical.
- Struct_Interfaces/PI_StructPointers/Subsystem/PI_StructPointers/Control Logic:

  Apart from referencing DD structure components, pointers to structures also appear in function interfaces if TargetLink BusPorts are used in combination with the "Create implicit struct" option (see the Control block) and a variable class with ref_param scope.

Related topics

Basics

Basics on Specifying Pointers to Structs (📖 TargetLink Customization and Optimization Guide)

MATRIX

Access

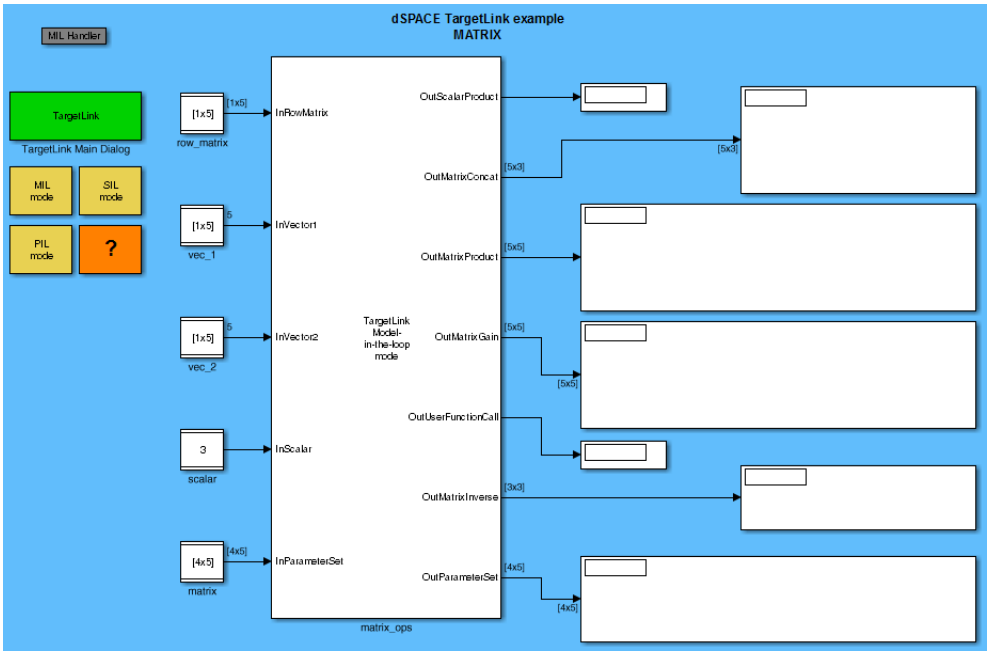
In the MATLAB Command Window, enter the following command:

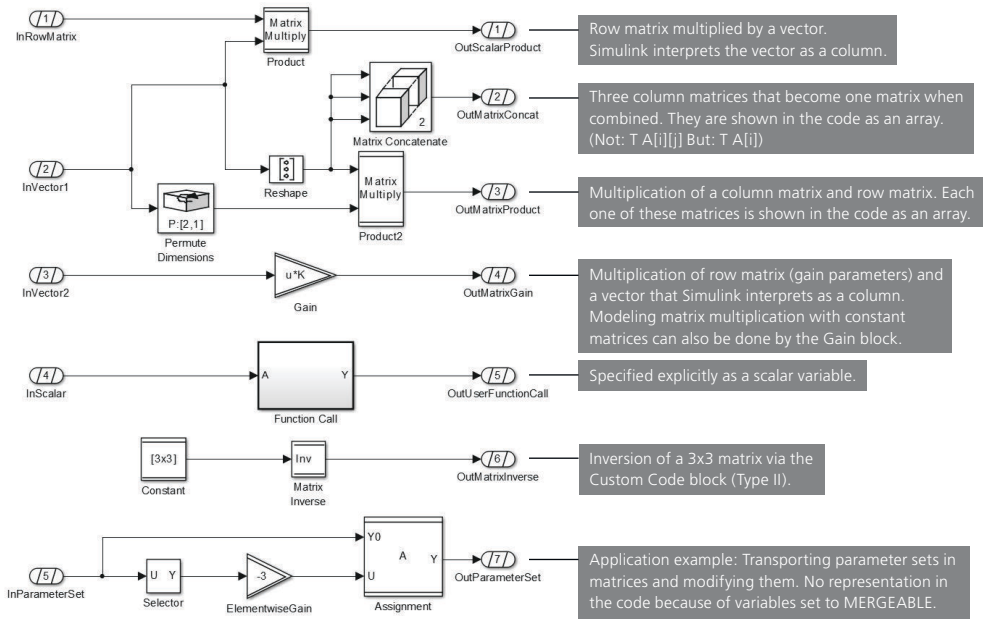
```
tl_demos matrix
```

You can also enter **`tl_demos`** to access all available demos in the MATLAB Help browser. Select a demo and click Open this example at the top right.

Description

The **`matrix`** demo model shows you how to work with matrix signals. The illustration below shows the root level of the demo model.





Dimension Reduction

By default, TargetLink optimizes production code as far as possible to increase efficiency. If you do not explicitly create DD **Variable** objects for signals whose dimensions are explicitly specified, these optimizations includes eliminating dimensions of size **1**. This can cause production code variables to have fewer dimensions than the signals in your model.

Note

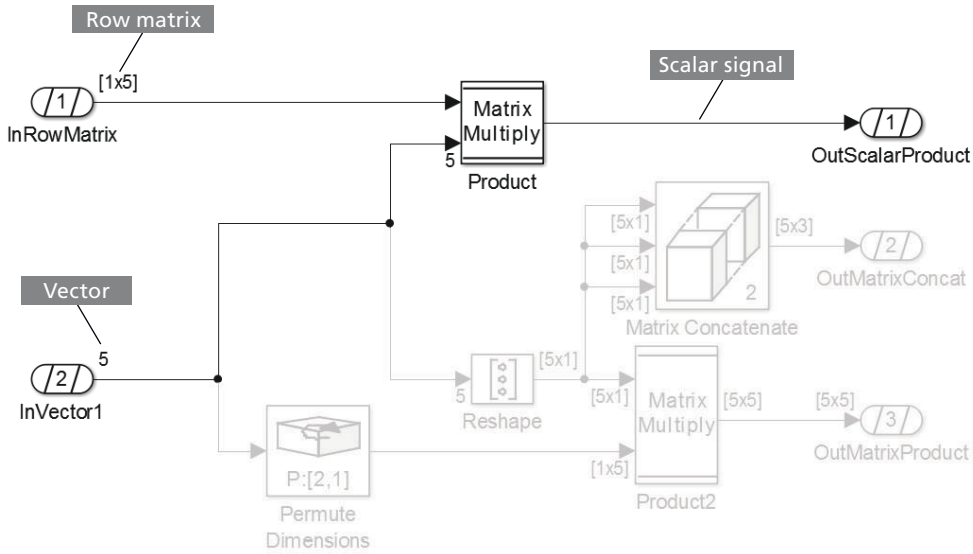
Simulink treats vectors specified via block dialogs and vector signals as follows:

- Vectors treated as parameters, init values, states, etc. that are specified via block dialogs are treated as row matrices.
- Vector signals are treated as column matrices.

Multiplying matrix and vector In this example, the signals **InRowMatrix** and **InVector1** are multiplied.

Note that Simulink treats **InVector1** as a column matrix to make multiplication possible.

The product results from the multiplication of a 1x5 row matrix with a 5x1 column matrix that results in a scalar signal.



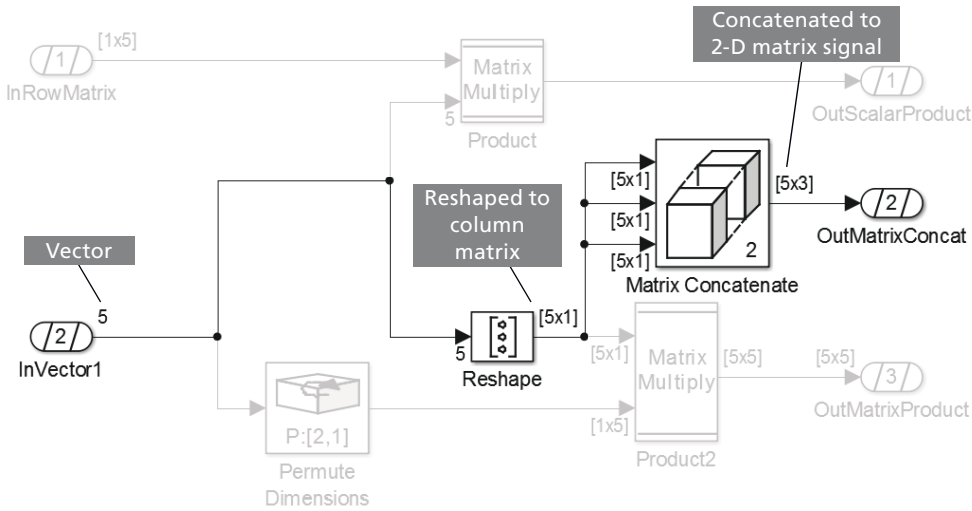
Matrix Operations

Note

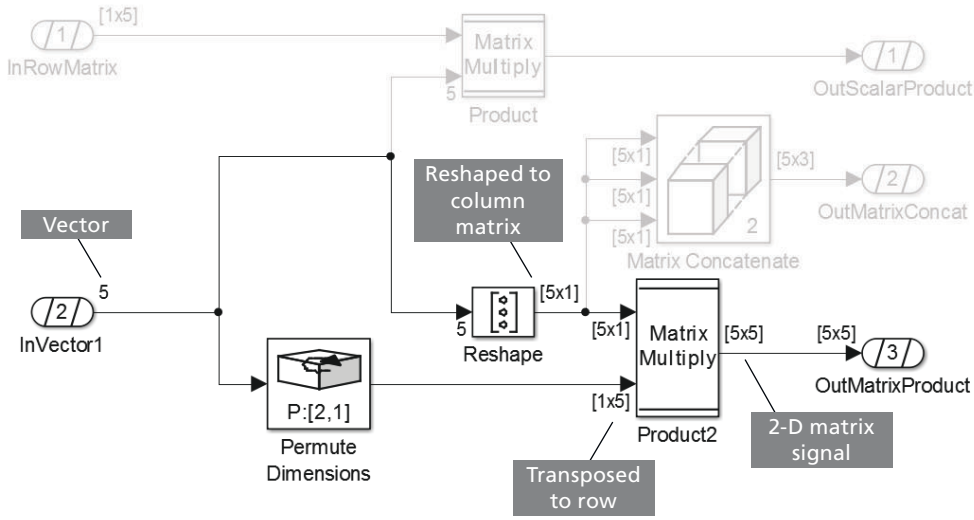
Simulink treats vectors specified via block dialogs and vector signals as follows:

- Vectors treated as parameters, init values, states, etc. that are specified via block dialogs are treated as row matrices.
- Vector signals are treated as column matrices.

Reshaping and concatenation This example shows the reshaping of InVector1 from a vector of length 5 to a 5x1 column matrix. The signal is split three times to concatenate each 5x1 signal to form a 5x3 matrix.



Multiplying matrices



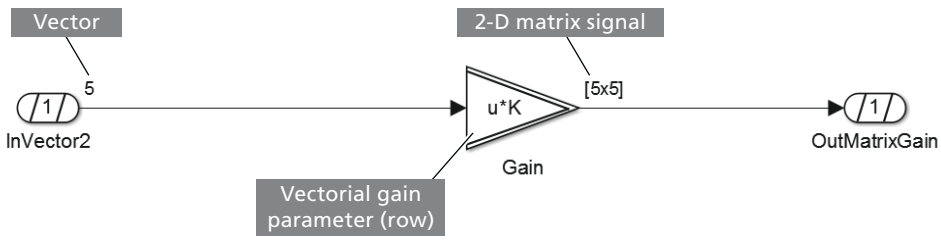
In this example, InVector1 is transformed twice to get two signals:

- Reshaping it to a 5x1 column matrix via a SL Reshape block.
- Transposing it to a 1x5 row matrix via a SL Permute Dimensions block.

Here, Simulink takes both signals at face value. Accordingly, Product2 is the result of multiplying a 5x1 column matrix with a 1x5 row matrix, i.e. a 5x5 matrix.

Multiplying vectors In this example, the InVector2 signal is multiplied by a vectorized gain parameter.

Note that Simulink treats InVector2 as a column matrix to make multiplication possible.



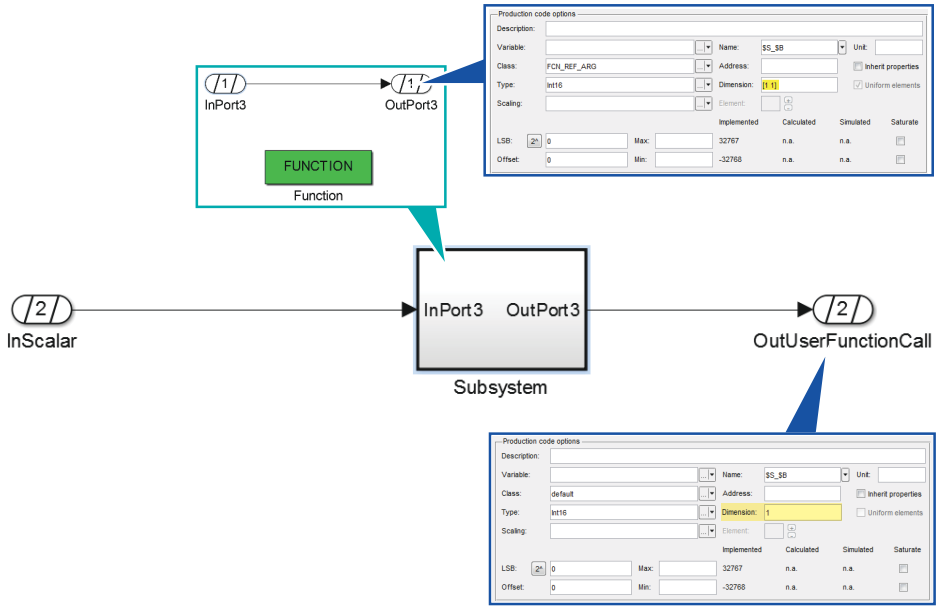
Accordingly, the gain operation forms the multiplication of a 5x1 column matrix with a 1x5 row matrix (gain parameter). OutMatrixGain is a 5x5 matrix signal that abides mathematics.

Function Interface using matrix signals

Suppose you have to model a function with a given prototype. Its interface is as follows:

- One scalar input argument
- One matrix output argument

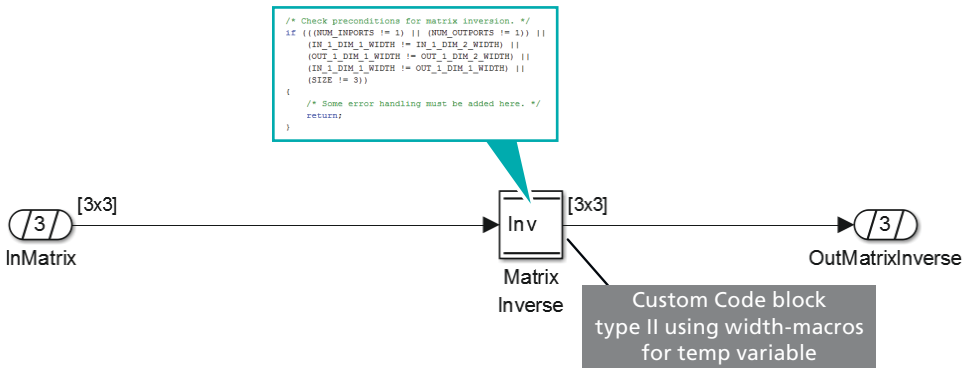
TargetLink lets you explicitly specify the dimensions of the function's arguments via the Dimension edit field of the InPort/OutPort blocks that represents the function's arguments:



Here, the Dimension of OutPort3 is explicitly specified as a matrix of size 1x1.

Using custom code containing matrices

TargetLink does not natively support calculating the matrix inverse. This example uses a Custom Code block (type II) to include custom code for calculating the matrix inverse of a [3x3] matrix via its determinant.



Note

TargetLink stores matrix elements in memory according to row-major order. This is important if you want to use external FORTRAN code, external code generated with MathWorks tools, or any other code using column-major order.

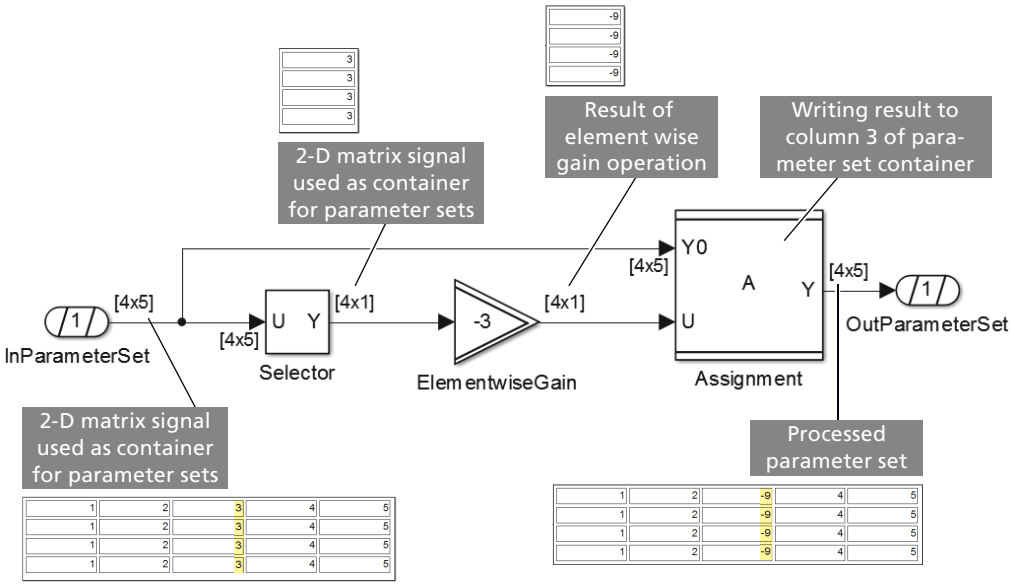
Principles of Row-Major and Column-Major Order (Storage of Matrices in Memory) (🕮 TargetLink Preparation and Simulation Guide)

Embedding External Functions That Use Column-Major Order (🕮 TargetLink Customization and Optimization Guide)

Details on Using Matrices in Customer-Specific C Functions (🕮 TargetLink Preparation and Simulation Guide)

Using matrices as containers for parameters

The MATRIX demo model shows how to use a matrix signal as a container for a parameter set:



InParameterSet is a matrix signal with 4 rows and 5 columns. Each of its columns represents a set of parameters to process.

The Selector block is configured such that it selects all the elements contained in the third column of InParameterSet. The Assignment block is configured such that it writes the result of the element-wise gain operation to the third column of InParameterSet and outputs the new OutParameterSet.

To save memory, the InPort, OutPort, and Assignment block use a fixed variable name with a variable class set to `MERGEABLE_GLOBAL`. In cases where the variable is not mergeable, the code will consist of one or two additional matrix variables, depending on the model.

Related topics

Basics

Details on Using Matrices in Customer-Specific C Functions (📖 TargetLink Preparation and Simulation Guide)
Embedding External Functions That Use Column-Major Order (📖 TargetLink Customization and Optimization Guide)
Principles of Row-Major and Column-Major Order (Storage of Matrices in Memory) (📖 TargetLink Preparation and Simulation Guide)
Working With Matrix Signals (📖 TargetLink Preparation and Simulation Guide)

Examples

Example of a Function Interface That Uses Matrix Signals (📖 TargetLink Preparation and Simulation Guide)
Example of Dimension Reduction (📖 TargetLink Preparation and Simulation Guide)
Example of Using Custom Code Containing Matrices (📖 TargetLink Preparation and Simulation Guide)
Example of Using Matrices as Containers for Parameters (📖 TargetLink Preparation and Simulation Guide)
Examples of Matrix Operations (📖 TargetLink Preparation and Simulation Guide)

References

Bus Inport Block (📖 TargetLink Model Element Reference)

BUS_STRUCT

Access

In the MATLAB Command Window, enter the following command:

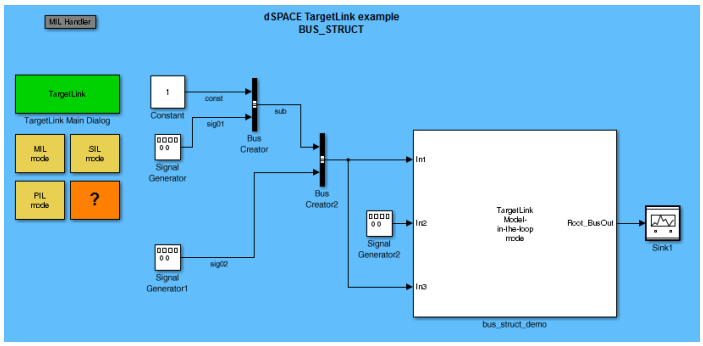
```
tl_demos bus_struct
```

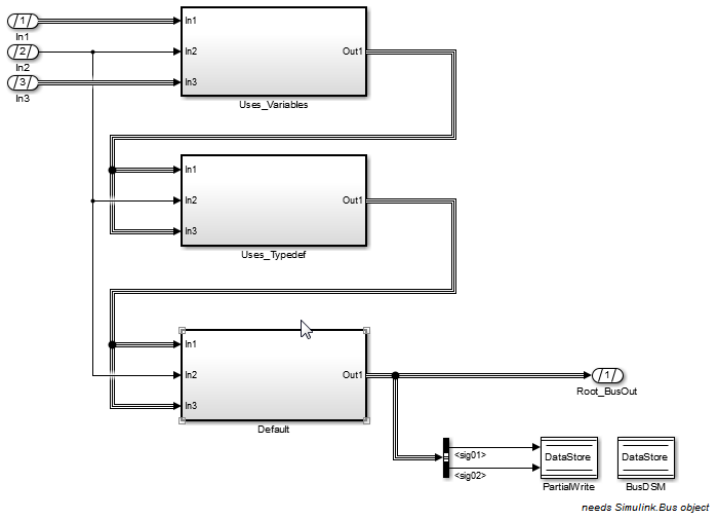
You can also enter **`tl_demos`** to access all available demos in the MATLAB Help browser. Select a demo and click **Open this example** at the top right.

Description

The **`bus_struct`** demo model shows you how to implement busses. It consists of a single top-level TargetLink subsystem. This subsystem contains three other subsystems to demonstrate various typical configuration options with respect to code generation for Simulink busses.

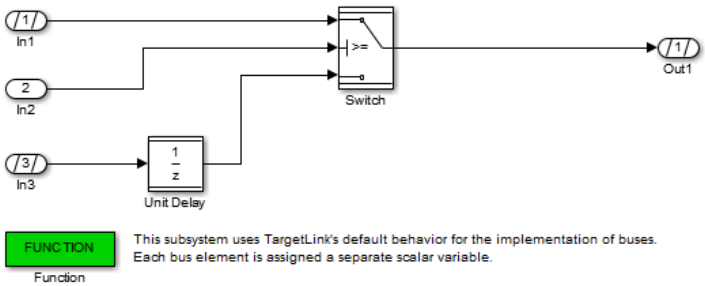
Refer to the illustrations below:



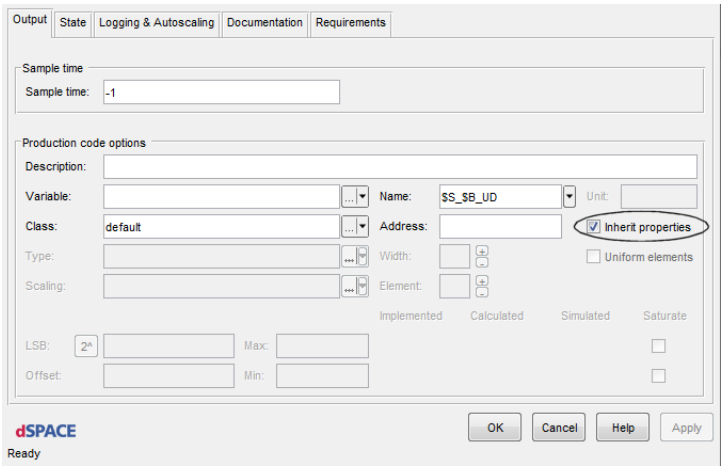


Implementing a bus as separate variables

Refer to the Default subsystem:

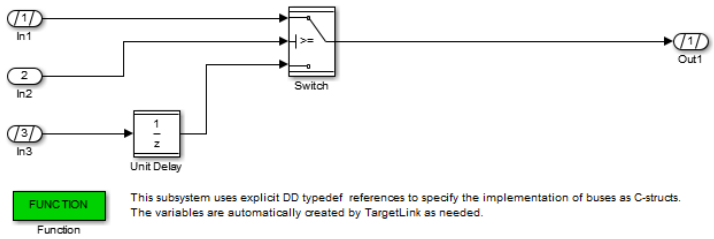


A subsystem uses TargetLink's default configuration to implement the bus. TargetLink generates individual variables at the lowest possible visibility for each bus signal. It also contains a Function block. TargetLink generates a function, and the implementation variables for the BusInport and the BusOutport are generated as global variables. In this configuration, the **Inherit Properties** checkbox of **Unit Delay** and **Switch** blocks must be set in order to generate code:

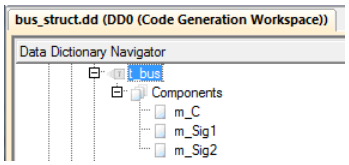


Using a predefined Data Dictionary Typedef object to implement a bus

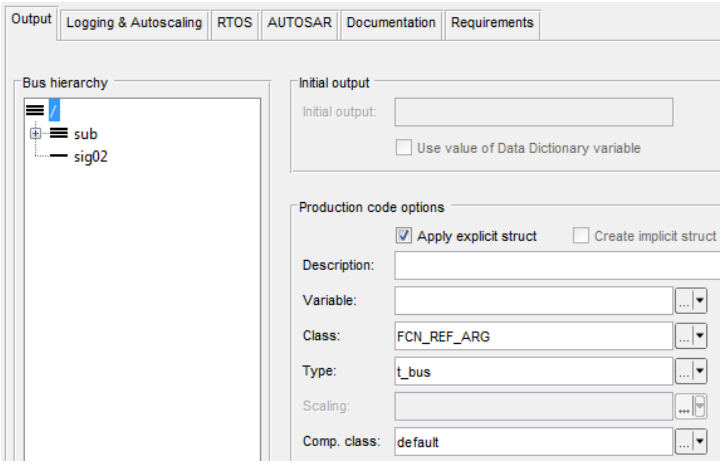
Refer to the Uses_Typdef subsystem:



A subsystem uses a DD Typedef object (see below) to specify the implementation of the buses.



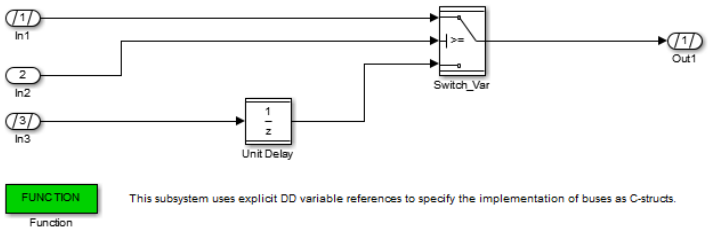
This is achieved by selecting the Apply explicit struct option. In this case, the Type field is set to the corresponding Typedef object. The Uses_Typedef subsystem also contains a TargetLink Function block. The VariableClass option in the BusPort dialogs is set to FCN_REF_ARG, so the arguments to the functions are passed as pointers to structs. See the following screenshot:



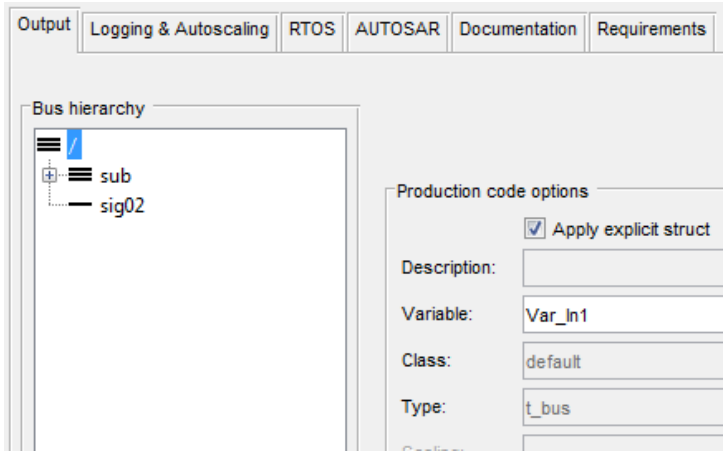
The Unit Delay and Switch blocks are also configured to use the DD typedef. The actual implementation variables are generated as needed.

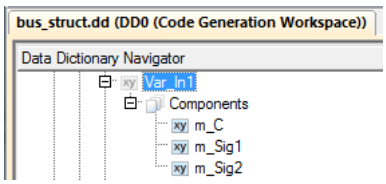
Using a predefined Data Dictionary Variable object to implement a bus

Refer to the Uses_Variables subsystem:



The Uses_Variables subsystem uses explicit DD variables to configure the BusPorts. The variables are defined as structures in the Data Dictionary. A TargetLink Function block is used to generate the code for this subsystem into a separate function. The VariableClass of the DD Variables is set to **default**. Therefore, the inputs and outputs of the function are generated as global variables. The function itself has a prototype of Void func(Void);. To achieve this behavior, the **Apply explicit struct** option must be set in the Output pane of the TargetLink BusPort Dialog, and the **Variable** field must reference the appropriate DD object. See the following screenshots:

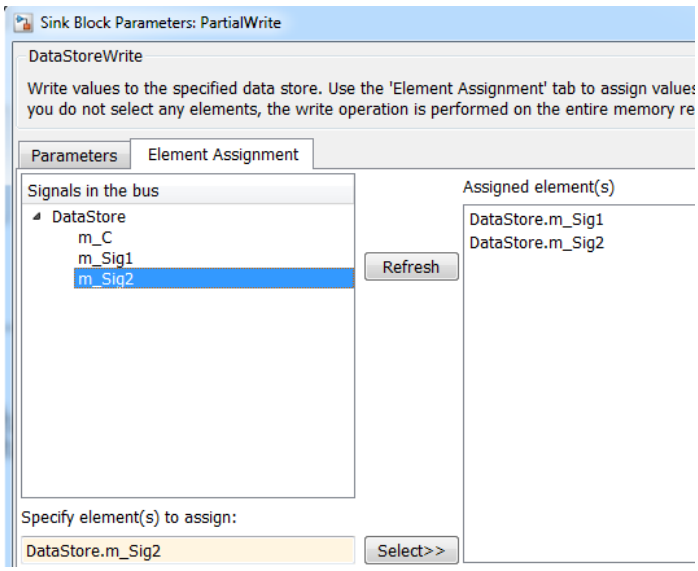




Configure the additional bus-capable blocks, Unit Delay and Switch, to use DD variables by referencing them from their respective block dialogs.

Accessing a bus data store and partial writing

The actual data to be processed is specified by the entries in the Assigned element(s) field in the *Simulink block dialog* of of data store read and data store write blocks. The screenshot below shows an example of a data store block, which specifies a bus struct that consists of three bus components but only two components are written by the data store write block (partial writing).



The number of **Assigned element(s)** determines the number of block ports (here: two inports).

For read access, the assignment of data store elements follows the same mechanism via the **Assigned element(s)**.

Related topics

Basics

Mapping Entire Buses to Explicit Structs in the Code (📖 TargetLink Customization and Optimization Guide)
Working with Bus Signals (📖 TargetLink Preparation and Simulation Guide)

VARIABLE_VECTOR_WIDTH

Access

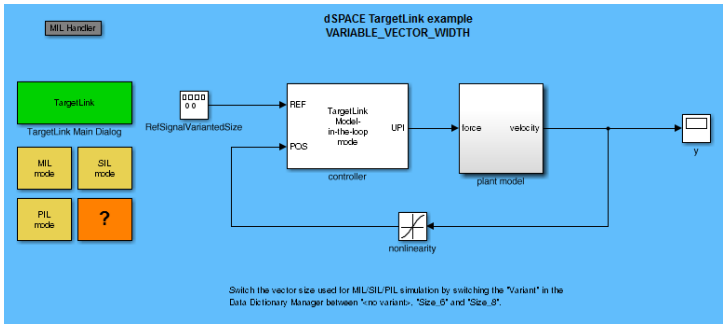
In the MATLAB Command Window, enter the following command:

```
tl_demos variable_vector_width
```

You can also enter **`tl_demos`** to access all available demos in the MATLAB Help browser. Select a demo and click **Open this example** at the top right.

Description

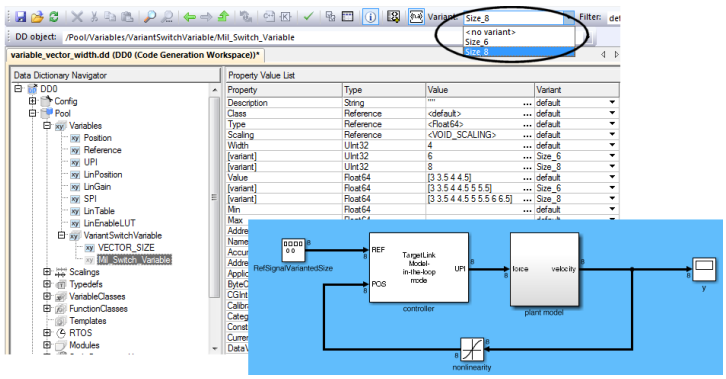
The **`variable_vector_width`** demo model shows you how to work with vectorized variables that have width variants. Using preprocessor macros for vector widths, the same model and generated production code can be used for all widths. The code of complete subsystems can become width-varying at code compile time.



Variable Vector Widths via Width Macros

In the `/Pool/Variables/VariantSwitchVariable/` object tree of the Data Dictionary Manager, the Mil_Switch_Variable object contains the variant settings. You can select the desired variant in the Variant list of the Data Dictionary Manager.

- If you select <no variant>, the variable vector width is 4.
- If you select Size_6, the variable vector width is 6.
- If you select Size_8, the variable vector width is 8.



Related topics

Basics

Introduction to Variable Vector Widths (📖 TargetLink Customization and Optimization Guide)

FUNCTION_REUSE

Access

In the MATLAB Command Window, enter the following command:

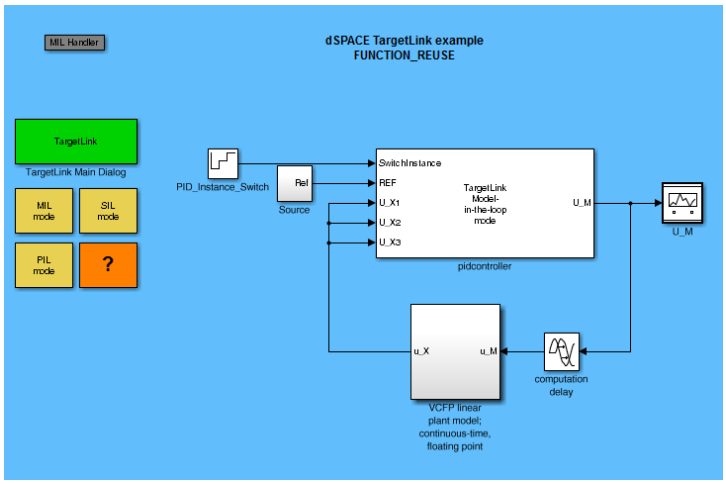
```
tl_demos function_reuse
```

You can also enter `tl_demos` to access all available demos in the MATLAB Help browser. Select a demo and click **Open this example** at the top right.

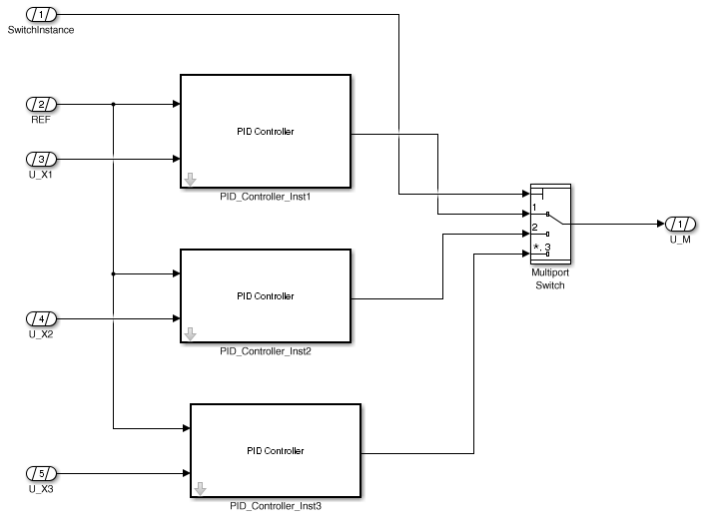
Description

The `function_reuse` model shows how to reuse certain model parts in a model. C functions that represent such reused model parts can be reused as well in order to reduce code size. This leads to one reused function but multiple function call instances in the code. TargetLink can implement function interface variables instance-specifically.

The following illustration shows the root level of the demo model.

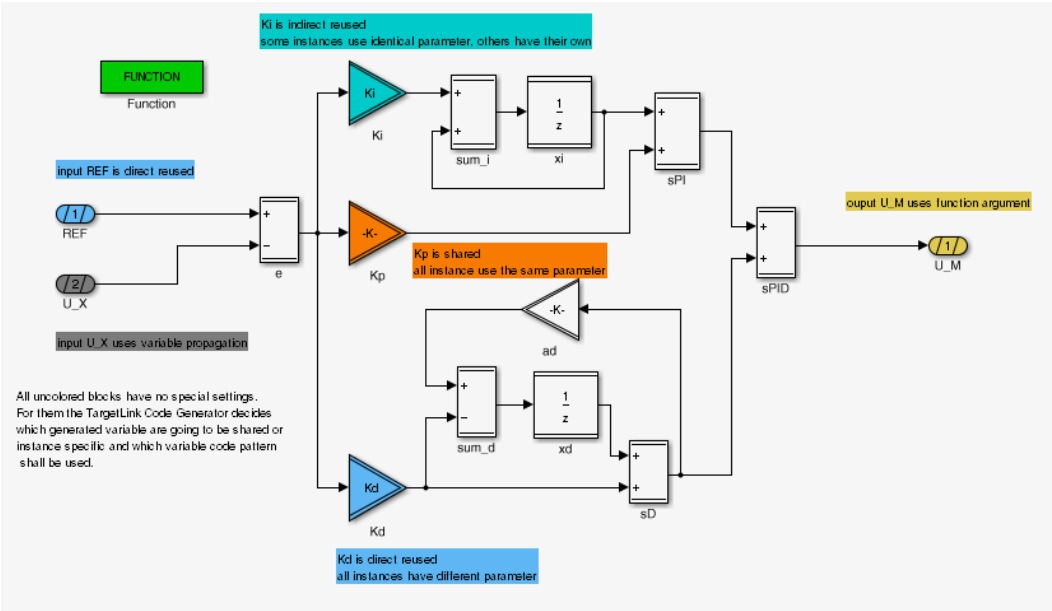


The PID Controller subsystem is reused three times with different initial values and parameters:



Reusing code for different instances of a subsystem (Function Reuse)

The following illustration shows a PID Controller subsystem which is used in the model three times. The colored and labeled blocks have special settings. The following reuse mechanisms are shown:



Modeling aspects The reusable system definition shown above contains a Function block. To make Kd, Ki, and Xd instance-specific, the reusable system definition is masked with the following parameters, which agrees with to Simulink modeling requirements.

Mask Parameter	Value for Instance ...		
	#1	#2	#3
Instance-specific parameter Ki	0.05	0.05	0.06
Name template parameter Ki	'Ki1'	'Ki1'	'Ki2'
Instance-specific parameter Kd	7.6196	7.8	7.927
Instance-specific init value Xd	-0.5	0.5	0.5

Generated code For an example of the code that TargetLink generates in the header file that relates to the reusable function definition (here: `PID_Controller_Reuse.h`), refer to the following illustration:

Sub Reuse Structs

```
typedef struct tagt_FcnReuse_InstanceSubStruct_Spid1_0 {
   Float32 * const pU_X;
   Float32 * const p_FcnReuseSubStruct_Spid1_Ki;
} t_FcnReuse_InstanceSubStruct_Spid1_0; /* Reuse linker section structure */
typedef struct tagt_FcnReuse_InstanceSubStruct_Spid1_1 {
   Float32 Spid1_REF;
} t_FcnReuse_InstanceSubStruct_Spid1_1; /* Reuse linker section structure */
```

Main Reuse Struct

```
typedef struct tagt_FcnReuse_InstanceStruct_Spid1 {
   Float32 Kd;
   Float32 X_Spid1_xd;
   Float32 X_Spid1_xi;
   t_FcnReuse_InstanceSubStruct_Spid1_0 * pt_FcnReuse_InstanceSubStruct_Spid1_0;
   t_FcnReuse_InstanceSubStruct_Spid1_1 * pt_FcnReuse_InstanceSubStruct_Spid1_1;
} t_FcnReuse_InstanceStruct_Spid1; /* Reuse structure */
```

Shared Variable

```
extern CAL_SHARED Float32 Kp;
```

Function Definition

```
extern Void Reusable_PID_Controller(Float32 * Spid1_U_M, t_FcnReuse_InstanceStruct_Spid1
* pISV);
```

The relevant code characteristics are:

- The first sub reuse struct applies indirect reuse. It holds pointers to the instance-specific Ki gain parameter and the instance-specific U_X input variable.

  The second sub reuse struct applies direct reuse. It holds the instance-specific REF input variable.
- The main reuse struct holds the instance-specific Kd gain parameter (direct reuse), the two xd and xi state parameters (direct reuse), and two pointers to the two sub reuse structs.
- The Kp gain parameter is a shared variable.
- The function definition holds a pointer to the U_M output variable, and a pointer (pISV) to the main reuse struct.

Related topics

Basics

Basics on Reusing Variables of Preceding and Subsequent Blocks (📖 TargetLink Customization and Optimization Guide)
Introduction to Function Reuse (📖 TargetLink Customization and Optimization Guide)
MULTIPLE_INSTANCES_REFMODEL 60

HowTos

How to Implement Instance-Specific Variables as Function Arguments (📖 TargetLink Customization and Optimization Guide)
How to Use Shared Variables (📖 TargetLink Customization and Optimization Guide)

MULTIPLE_INSTANCES_REFMODEL

Access

In the MATLAB Command Window, enter the following command:

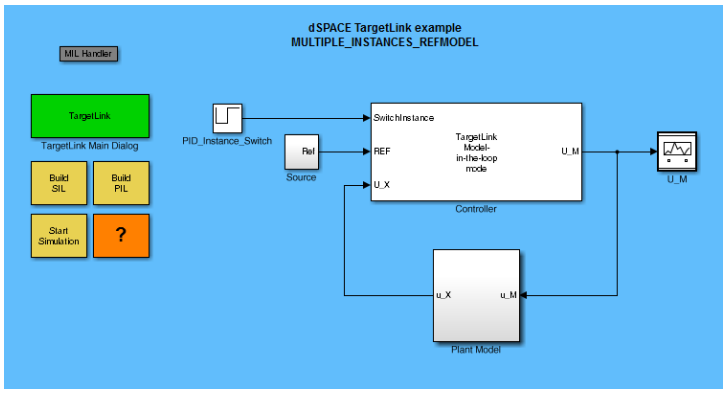
```
tl_demos multiple_instances_refmodel
```

You can also enter **`tl_demos`** to access all available demos in the MATLAB Help browser. Select a demo and click **Open this example** at the top right.

Description

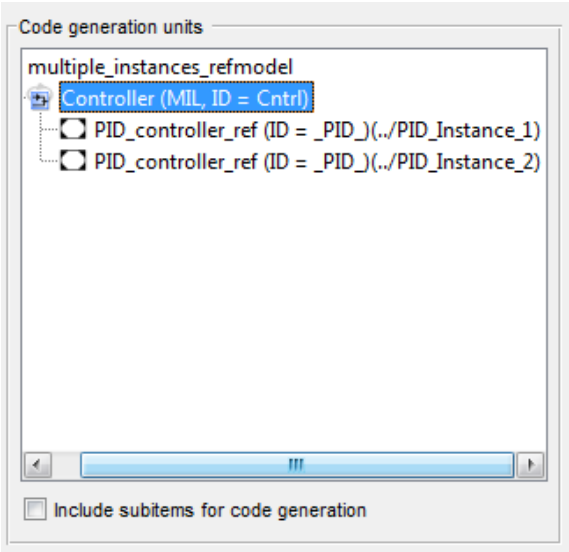
The `multiple_instances_refmodel` demo shows you how to reuse code functions generated for referenced models with instance-specific initial parameter values by defining Simulink model arguments.

The following illustration shows the root level of the demo model:

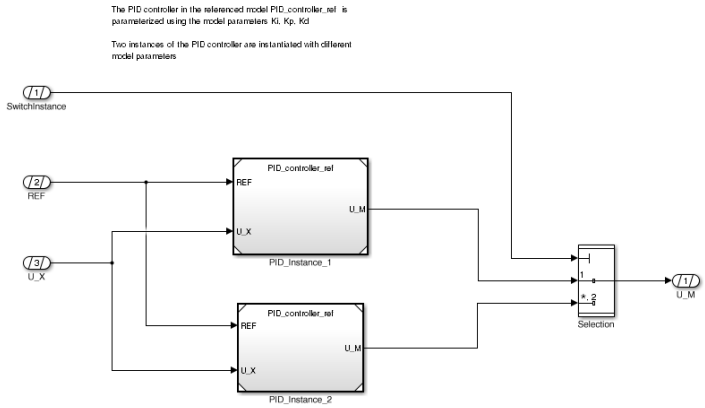


Reusing referenced models

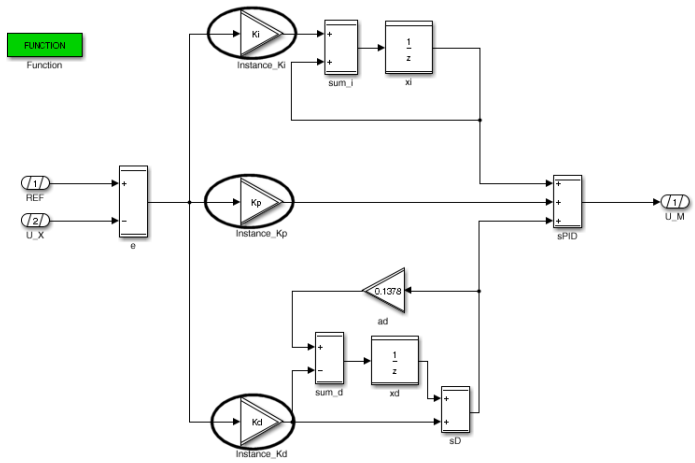
TargetLink supports referenced models if they reside in a parent TargetLink subsystem on any level and contain a **Function** block. The referenced models can be embedded multiple times in a code generation unit, if the **Make function reusable** checkbox is selected in the corresponding Function block. In this demo, a referenced model `PID_controller_ref` is referenced twice in the parent TargetLink subsystem. Each instance of the referenced model is shown in the **Code generation units** list in the TargetLink Main Dialog.



The Controller (= parent TargetLink subsystem) looks like this:

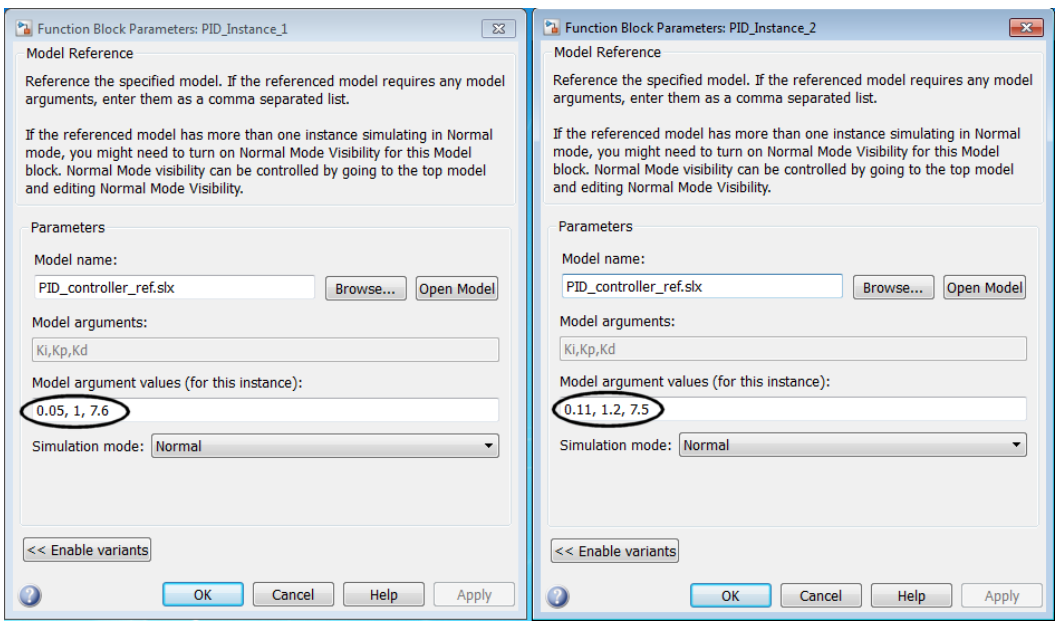


The referenced model has model arguments (the gain values Ki, Kp, Kd).



Referenced model of the PID controller.
The model contains the three model parameter
Ki, Kp and Kd all used in the Gain blocks.

A look at the parameters of Model blocks referencing the `PID_controller_ref` model shows that the model arguments have *instance-specific* initial values:



The production code for the referenced model is generated by using TargetLink's function reuse mechanism. This mechanism reduces the size of the generated code (ROM) because one function is reused for several instances in the model. Data is duplicated as instance-specific only where necessary. The signature of a generated function contains a pointer to structure with instance-specific parameters. The reusable function definition is coded as follows:

```
 96
 97  /*****************************************************************************************\
 98   ***  FUNCTION:
 99   ***      PID_Controller
100   ***
101   ***  DESCRIPTION:
102   ***      Function Block Object for the reusable PID Controller
103   ***      Implementation function for the reusable PID Controller
104   ***
105   ***  PARAMETERS:
106   ***      Type              Name                Description
107   ***      ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
108   ***      Float32 *         REF
109   ***      Float32 *         U_X
110   ***      t_FcnReuse_InstanceStruct_S_PID_1 *pISV
111   ***
112   ***  RETURNS:
113   ***      Float32
114   ***
115   ***  SETTINGS:
116   ***
117  \*****************************************************************************************/
118  Float32 PID_Controller(Float32 * REF, Float32 * U_X, t_FcnReuse_InstanceStruct_S_PID_1 * pISV)
119  {
120     /* SLLocal: Default storage class for local variables | Width: 32 */
121     Float32 S_PID_1_Instance_Kd;
122     Float32 S_PID_1_e;
123     Float32 S_PID_1_sD;
124     Float32 S_PID_1_sPID;
125
```

The code generated for the parent TargetLink subsystem contains definitions of reused structures of the instance-specific parameters:

```
88  /*****************************************************************************\
89     FCN_REUSE_INST_STRUCT__PID__aSrToLocal_Cntrl: TL CG ScopeReducedClass : derived from FCN_REUSE_IN
90     ST_STRUCT__PID__a | Width: N.A.
91  \*****************************************************************************/
92  static t_FcnReuse_InstanceStruct_S_PID_1 FcnReuse_InstanceStruct_S_PID_1_PID_controller_ref_Cntrl =
93      {
94     7.6F /* Kd_:  */,
95     0.05F /* Ki_:  */,
96     1.F /* Kp_:  */,
97     0.F /* X_S_PID_1_xd:  */,
98     0.F /* X_S_PID_1_xi:  */
99  };
.00
.01  static t_FcnReuse_InstanceStruct_S_PID_1 FcnReuse_InstanceStruct_S_PID_1_PID_controller_ref_Cntrl_a
.02      = {
.03     7.5F /* Kd_:  */,
.04     0.11F /* Ki_:  */,
.05     1.2F /* Kp_:  */,
.06     0.F /* X_S_PID_1_xd:  */,
.07     0.F /* X_S_PID_1_xi:  */
.08  };
```

In addition, the code also contains the function calls of the reused function instances:

```
158      /* Multiport switch: Controller/Selection */
159      switch (SCntrl1_SwitchInstance) {
160         case 1: {
161            /* # combined # TargetLink outport: Controller/U_M
162               # combined # call of function: Controller/PID_Instance_1 */
163            SCntrl1_U_M = PID_Controller(&SCntrl1_REF, &SCntrl1_U_X,
164             &FcnReuse_InstanceStruct_S_PID_1_PID_controller_ref_Cntrl);
165            break;
166         }
167         default: {
168            /* # combined # TargetLink outport: Controller/U_M
169               # combined # call of function: Controller/PID_Instance_2 */
170            SCntrl1_U_M = PID_Controller(&SCntrl1_REF, &SCntrl1_U_X,
171             &FcnReuse_InstanceStruct_S_PID_1_PID_controller_ref_Cntrl_a);
172            break;
173         }
```

Related topics

Basics

FUNCTION_REUSE 56
Introduction to Function Reuse (📖 TargetLink Customization and Optimization Guide)
MODEL_REFERENCING 181

EMBEDDING_EXTERNAL_CODE

Access

In the MATLAB Command Window, enter the following command:

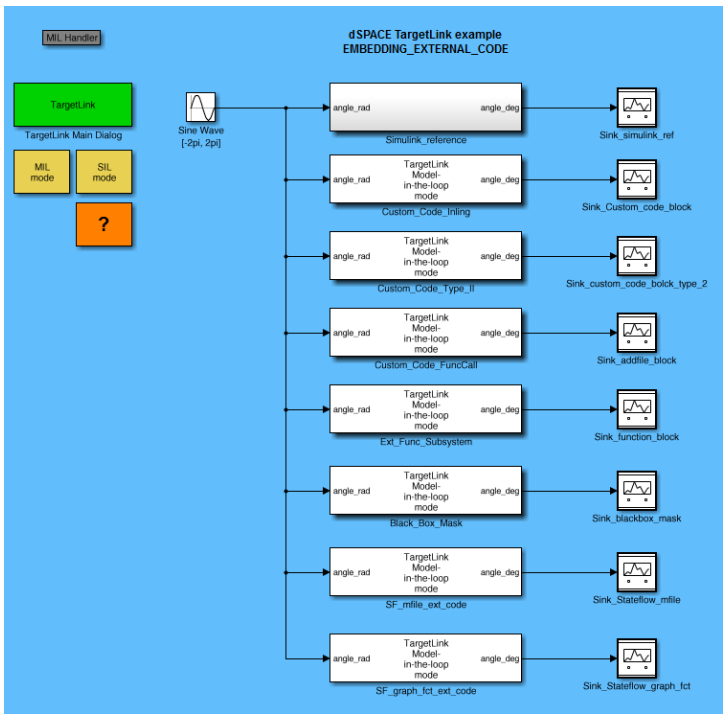
```
tl_demos embedding_external_code
```

You can also enter **`tl_demos`** to access all available demos in the MATLAB Help browser. Select a demo and click **Open this example** at the top right.

Description

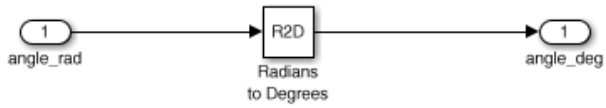
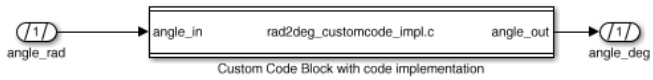
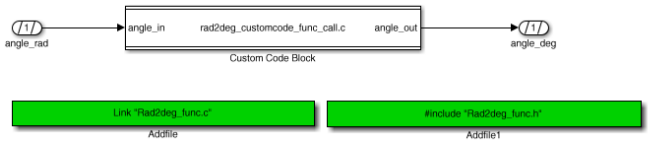
The `embedding_external_code` demo model shows you seven methods for embedding external code in TargetLink models or Stateflow charts. The topmost subsystem of the demo model is the Simulink reference model (radians to degrees).

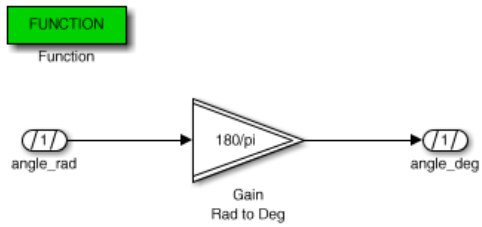
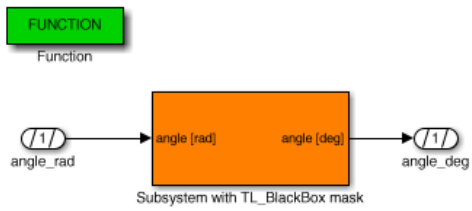
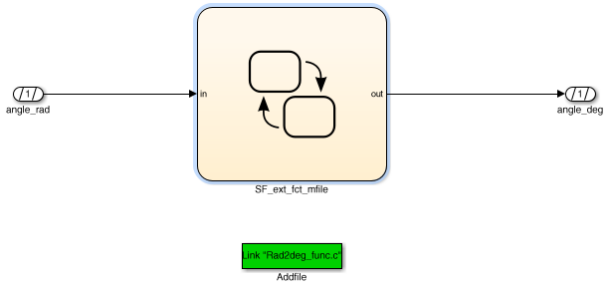
The illustration below shows the root level of the embedding_external_code demo.

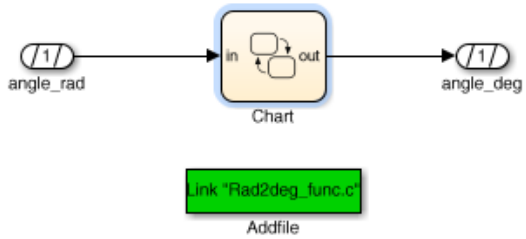


Methods for embedding external code

The following table decribes the different variants for embedding external code contained in this demo model.

Subsystem name	Description
Simulink_reference	The Simulink reference model for the variants to be shown. The Radians to Degrees block from the Simulink Extras/Transformation library is not supported by TargetLink, but can easily be substituted by a Gain block. 
Custom_Code_Inlining	This variant shows a Custom Code block with direct code implementation from within the block (code inlining). The inserted custom code is included directly in the production code. In this case, the inlined code is also used for the MIL simulation (Custom Code Type I block). 
Custom_Code_Type_II	This variant shows an unsupported Simulink block replaced by a TargetLink compliant block via the custom code approach. The unsupported block is converted into a Custom Code Type II block and the C code for the underlying function is provided by the user. During MIL simulation mode, the Simulink block is used. During production code generation TargetLink inlines the external Code (code inlining).
Custom_Code_FuncCall	In contrast to other variants with Custom Code blocks, in this example the external function is not inlined into the code, but called by the production code. During SIL simulation mode the external C function is called by the production code (function call). 

Subsystem name	Description
Ext_Func_Subsystem	This variant shows a Function block with an embedded external function in the production code. No code is generated for the TargetLink blocks inside this subsystem. In MIL simulation mode, the TargetLink blocks are simulated. However, you can only use supported TargetLink blocks. 
Black_Box_Mask	This variant shows a Function block with an embedded external function in the production code. No code is generated for the TargetLink blocks inside this subsystem. The **Subsystem with TL_BlackBox mask** subsystem is not analyzed by TargetLink's Code Generator. As a consequence, unsupported Simulink blocks can be used inside the subsystem and are executed in MIL simulation mode. No code is generated for subsystems masked with the BlackBox mask. 
SF_mfile_ext_code	This variant shows a Stateflow chart calling an external function (via an .m file for Stateflow User Functions). During MIL simulation mode, a floating-point version of the external function is called (`rad2degf.c`). For the production code, this function is mapped to a fixed-point function (`rad2deg.c`) by an additional .m-File. In the production code, `rad2deg.c` is implemented via a function call. 

Subsystem name	Description
SF_graph_fct_ext_code	This variant shows a Stateflow chart calling a graphical function (EXTERN_FCN). The function calls a floating-point version of the external C function (`rad2degf`). During MIL simulation mode this version is used. For the production code the graphical function call is replaced by the fixed-point external C-function (`rad2deg`) and is implemented into the production code as a function call. 

Related topics

Basics

Overview of Methods for Embedding External Code (📖 TargetLink Customization and Optimization Guide)

AUTODOC_GENERATION

Access

In the MATLAB Command Window, enter the following command:

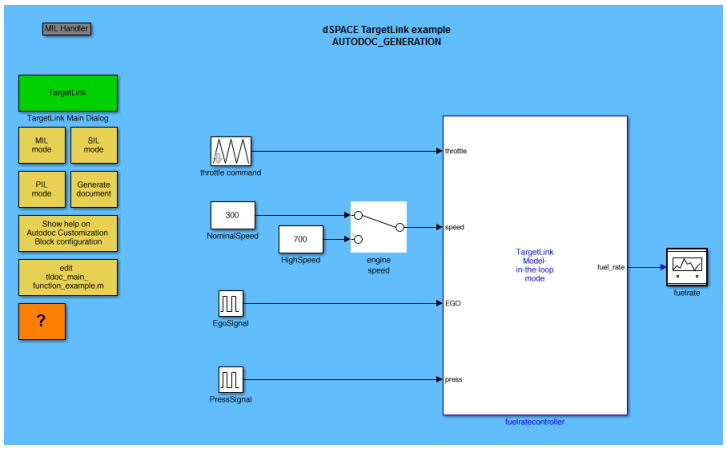
```
tl_demos autodoc_generation
```

You can also enter **`tl_demos`** to access all available demos in the MATLAB Help browser. Select a demo and click **Open this example** at the top right.

Description

The `autodoc_generation` demo model shows you how to generate documentation for your model that is always up-to-date with the generated code and demonstrates various settings and functionalities that lets you manage and customize documentation generation.

The illustration below shows the root level of the autodoc_generation demo demonstrating a fuelrate controller.



In particular, the demo shows how to

- control the general structure and content of the document by using a dedicated M script to call the documentation generation
- work with Autodoc_Customization blocks to control which functions should be documented
- insert user-provided content (texts, images, HTML commands, ...) into the documentation generation process

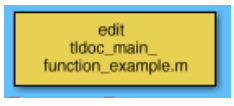
Generating and customizing documentation generation

The documentation generation process is controlled by the `tldoc` API function. You can put the calls to the `tldoc` API function in separate M files. TargetLink provides the following predefined M scripts used for the generation of documentation:

Predefined Script	Description
`tldoc_default.m`	File used to generate HTML documentation.
`tldoc_pdf.m`	File used to generate PDF documentation.

You can call these scripts in the MATLAB Command Window or specify them in the Tools page of the TargetLink Main Dialog.

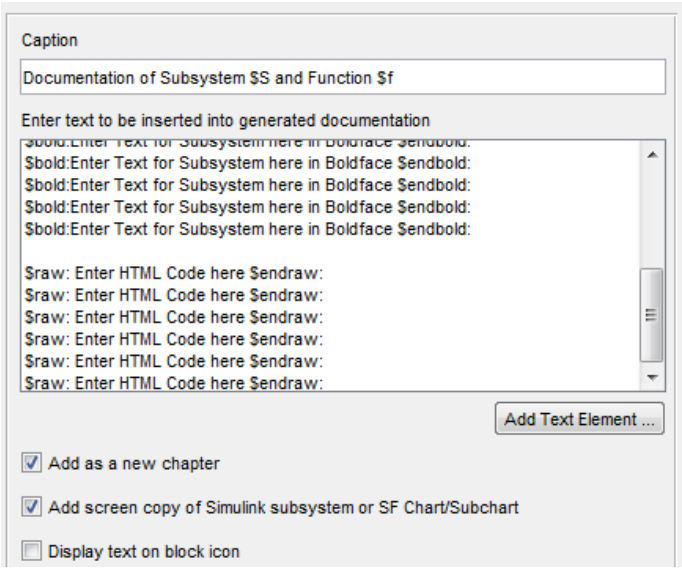
Edit tldoc_main_function_example.m This control lets you open a modified copy of the `tldoc_default.m` script that lets you manage and customize your documentation generation. You should not work with the `tldoc_default.m` original script located in `..\Matlab\Tl\autodoc` in your TargetLink installation folder. Refer to the annotations in the script for detailed information on the settings you can customize.



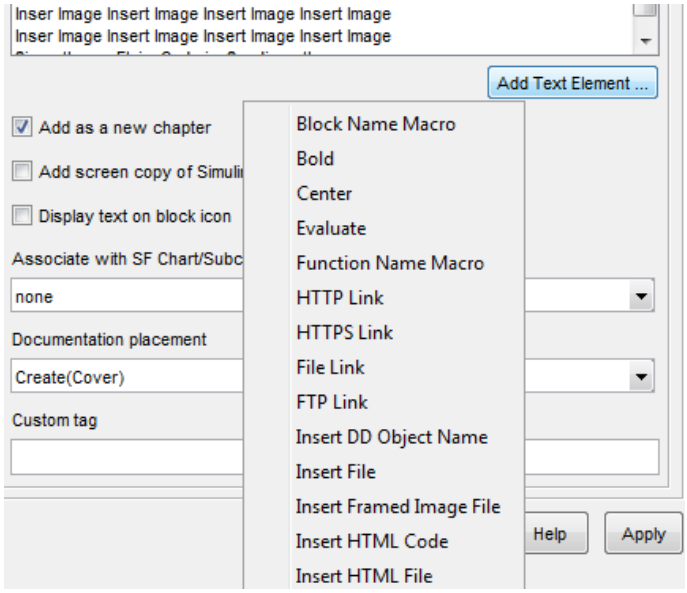
```
function tldoc_main_function_example(documentationBasis, documentationDestDir)

% tldoc_default generates HTML documentation of the TargerLink production code
%
% SYNTAX tldoc_default(documentationBasis,documentationDestDir)
%   Function for automatic documentation of production code generated for
%   selected TargetLink code generation units, which are:
%     - TargetLink subsystems
%     - subsystems configured for incremental code generation
%     - referenced models
%     - Data Dictionary CodeGenerationUnit objects
%
%   The function invokes "tldoc" command with several appropriate arguments. To modify the
%   document style, please copy this file and edit the options described below.
%
%   You can further customize the document generation by your own hook function which
%   is invoked at various points during automatic document generation (see
%   "tl_autodoc_hook.m").
```

For further customization of the documentation generation, you need to insert an Autodoc_Customization block into the model or a subsystem.

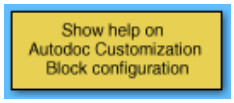
Embedding additional elements via Autodoc_Customization block You can insert additional elements in your generated documentation, such as hyperlinks, images, and name macros for subsystems. Refer to the Autodoc Customization blocks in the demo model.



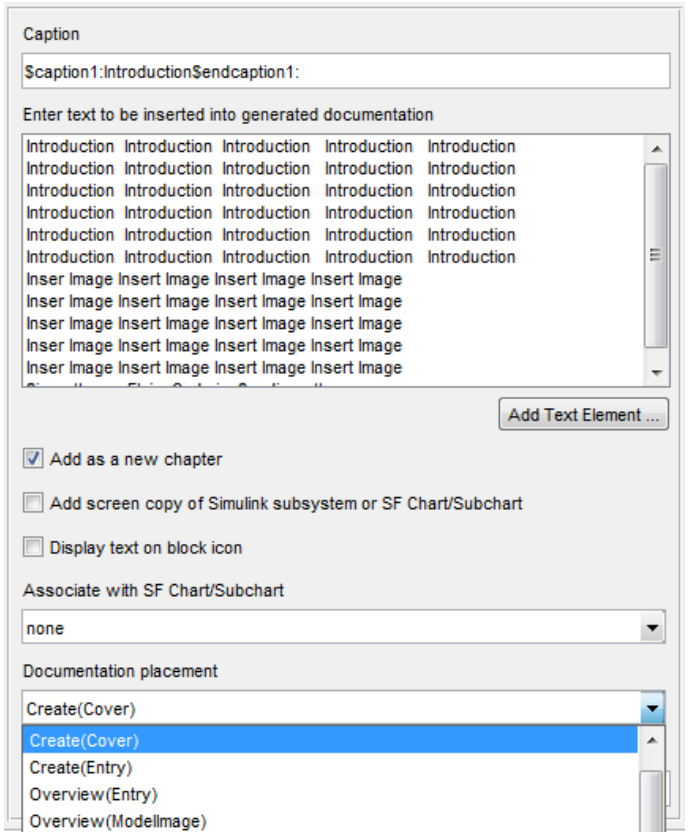
Insertions via Embedded Commands You can use embedded commands to handle text formatting and elements to be inserted. Click the Add Text Element button and select an embedded command to be inserted into the text field.



For detailed information on all commands, click the control in the root level of the demo:

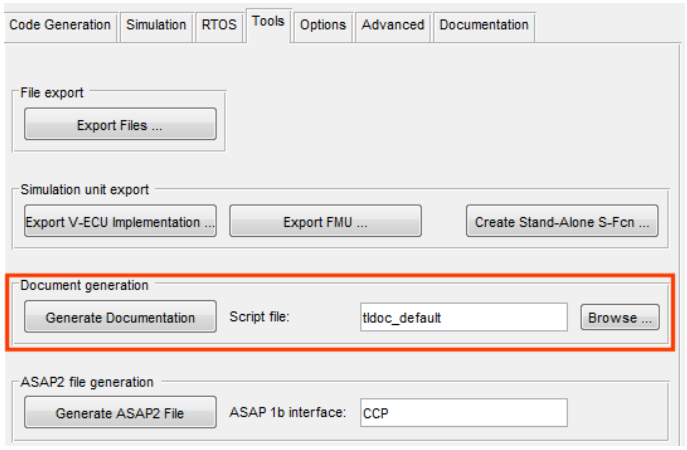


Selective documentation placement You can specify the position of the additional elements via an Autodoc Customization block. In the fuelratecontroller model, an AutoDoc Customization block adds a new chapter and an image at the top of the generated documentation. To do this, select the position in Documentation placement. In this demo, text and image are placed on the cover.



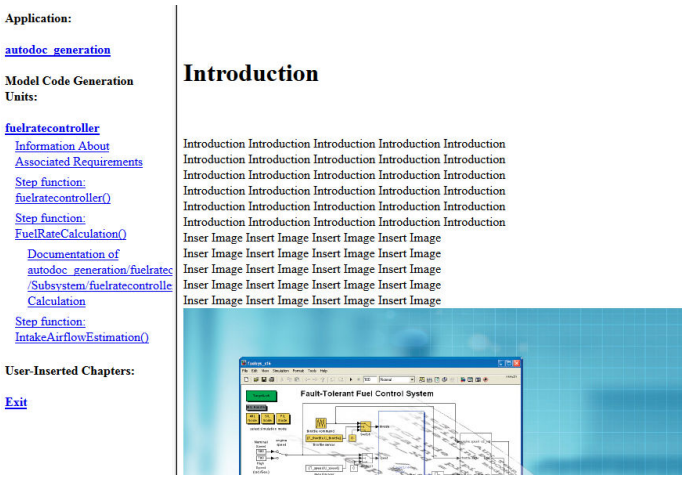
An additional Autodoc Customization block adds elements at the end of the documentation using the same method.

Generating HTML and PDF documentation The `tldoc_main_function_example.m` script generates HTML and PDF documentation according to the settings of the script. To start the generation of the HTML and PDF output, open the TargetLink Main Dialog, change to the Tools page and select Generate Documentation.



You can also call `tldoc_main_function_example` in the MATLAB Command Window.

The HTML and PDF documentation is generated.



Selective documentation generation for specific functions The `tldoc_main_function_example.m` allows you to specify the step functions documentation is to be generated for. In this demo, the script lets you specify that only step functions generated from subsystems with an Autodoc_Customization block are selected for documentation generation. Use the `AutodocFunctionSelection` property:

```
%                                   also described in the parented function part. No function prototype is generated.
%                                   reference to the parent function is generated.
%                                   Default: 'off'
%   AutodocFunctionSelection        If 'on', only functions generated for subsystems and Stateflow charts with an
%                                   AutoDoc Customization block are documented.
%                                   Default: 'off'
%   Functions                       Functions generated by TargetLink to be included in generated documentation.
%                                   Equivalent to using AutodocFunctionSelection property.
```

However, you can also filter specific functions by listing their names using the `Functions` property.

Related topics

Basics

Basics on Customizing the Generated Documentation (📖 TargetLink Interoperation and Exchange Guide)

VARIABLE_INITIALIZATION

Access

In the MATLAB Command Window, enter the following command:

```
tl_demos variable_initialization
```

You can also enter **`tl_demos`** to access all available demos in the MATLAB Help browser. Select a demo and click **Open this example** at the top right.

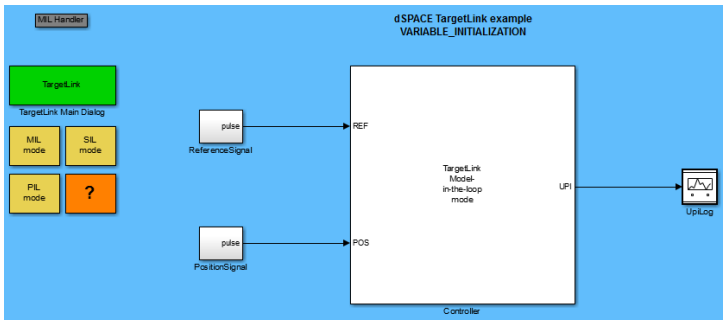
Description

The `variable_initialization` demo model shows you different ways to initialize variables in TargetLink, especially Custom Code blocks for initializing variables by other variables:

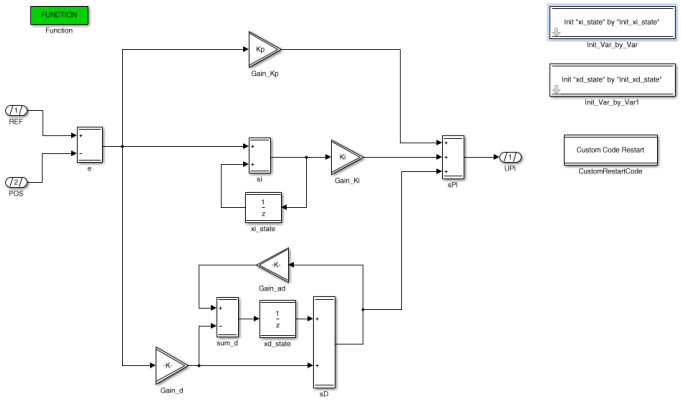
- Initializing variables at their definitions
- Initializing variables in restart functions
- Initializing variables by user-defined content of TargetLink Custom Code blocks. This mechanism also allows for more complex initialization mechanisms.

The demo shows all three mechanisms and focuses on the last method where Custom Code (type II) blocks are used to initialize a variable by using another variable.

The illustration below shows the root level of the variable_initialization demo.



Also refer to the Controller subsystem:



Ways to initialize variables in TargetLink

Variable initialization at definition The first illustration shows variables initialized at their definition which is carried out by TargetLink if the respective variable class has the `InitAtDefinition` property set.:

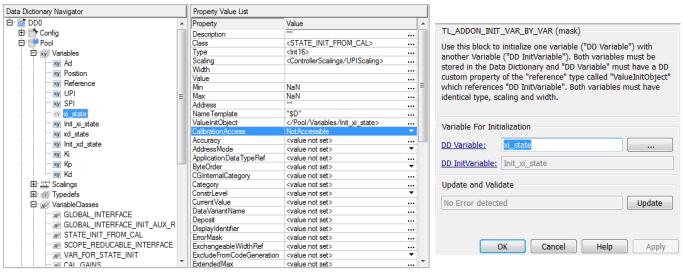
```
/*----------------------------------------------------------------------------*\
  VARIABLES
\*----------------------------------------------------------------------------*/

/*****************************************************************************\
   CAL_GAINS: global calibratable parameters (ROM) | Width: 8
\*****************************************************************************/
volatile UInt8 P_Ad = 35 /* 0.1378 */; /* LSB: 2^-8 OFF: 0 MIN/MAX: 0 .. 0.99609375 */
volatile UInt8 P_Kd = 244 /* 7.6196 */; /* LSB: 2^-5 OFF: 0 MIN/MAX: 0 .. 7.96875 */
```

Initializing variables in restart functions TargetLink initializes variables in a main restart function and a subfunction, as shown in the following code pattern of this demo.This behavior is used by TargetLink if the `InitAtDefinition` property of the variable class is set to `off` and the `RestartFunctionName` property specifies a function name. Use the `default` string to initialize the variable in the main restart function.

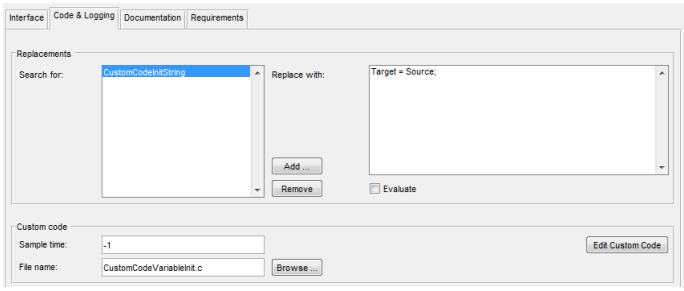
```
void RESTART_Controller(void)
{
   /* Custom code: Controller/CustomRestartCode << restart code (top) >> */
   // Dummy initilizations for state variables which will be overwritten during restart function
   // You can avoid the initialization by not specifying restart function for initialization and set
   //ting InitAtDefinition to "off"


   xd_state = -16384 /* -1. */;
   xi_state = -512 /* -1. */;
   OutputRestart();

   /* Custom code: Controller/CustomRestartCode << restart code >> */
   // Start of other initializations


   /* Custom code: Controller/Init_Var_by_Var1/VariableInit_with_CustomCode << restart code (bottom)
       >> */
    xd_state = Init_xd_state;

   /* Custom code: Controller/Init_Var_by_Var/VariableInit_with_CustomCode << restart code (bottom)
      >> */
    xi_state = Init_xi_state;

   /* Custom code: Controller/CustomRestartCode << restart code (bottom) >> */
   // Additional Comments or code for restart function
   // Insert whatever code you would like to insert here

}
```

Initializing Variables by user-defined custom code You can create a stand-alone Custom Code (type II) block with a special mask able to automatically initialize a variable by user-defined C code statements, e.g., to initialize a variable by using another variable. In this case you have to provide the necessary code strings in the `Restart` section of the Custom Code block.

- The demo contains an example of a user-defined block for initialization, which consists of a subsystem with a Custom Code (type II) mask and a Simulink dialog. In the dialog, a DD variable object is specified that contains custom properties to define another DD Variable object whose value is used to initialize the other variable.



- The block uses Replacement variables of Custom Code blocks to create the initialization string based on DD properties, which are then used in the restart section of the Custom Code file.



Related topics

Examples

Example of Initializing Variables via Restart Functions (📖 TargetLink Customization and Optimization Guide)

FUNCTION_VARIANTS

Access

In the MATLAB Command Window, enter the following command:

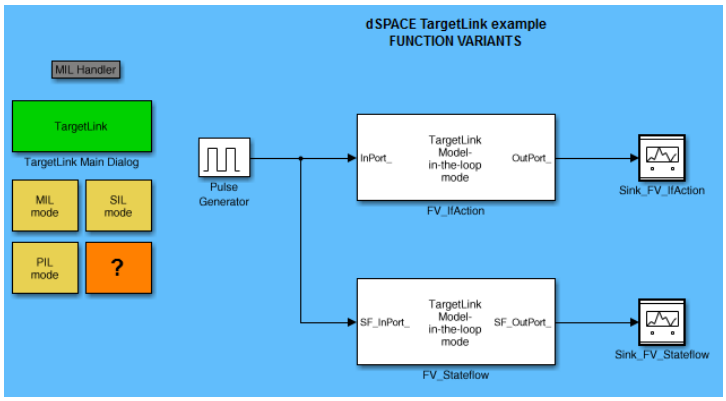
```
tl_demos function_variants
```

You can also enter **`tl_demos`** to access all available demos in the MATLAB Help browser. Select a demo and click **Open this example** at the top right.

Description

The **`function_variants`** demo model shows you how to create function variants with different binding times in TargetLink. It contains an if-Action subsystem and a Stateflow Chart calling a graphical function.

The illustration below shows the root level of the function_variants demo.



You can choose different binding times for function variants. The following binding times apply:

- Modeling

  You can define the variant by adjusting the model so that only one variant is visible both for Simulink/Stateflow and TargetLink. Production code is then generated only for this function variant.

- Production code generation

  Depending on the choice of the variant, new code is generated. The production code includes only the activated function variant.

- Compilation

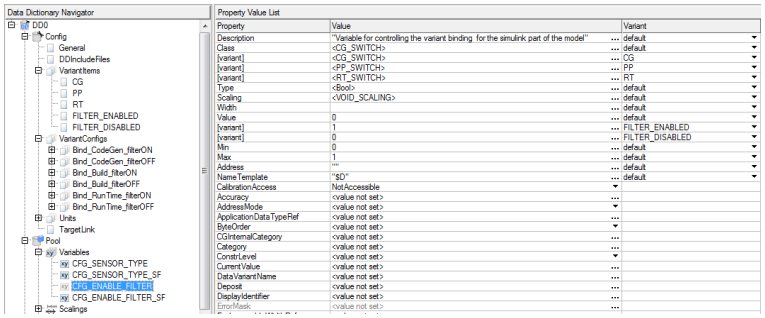
  If you use preprocessor macros (ifdev), the variant is set during compilation time. Therefore, the same C code is used for all cases, since the variant is selected during compilation. A hook script (tl_pre_compile_host_hook.m) determines which variant in the DD is active and which header file has to be compiled when the production code is compiled. You do not have to generate code again.

- Runtime

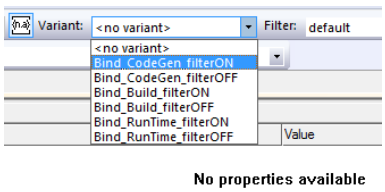
  TargetLink generates code for both variants that can be switched during run time.

  In SIL simulation mode, a hook script (`tl_pre_prodcode_sim_hook.m`) determines which variant in the DD is active and either the properties of a variable or an entire variable class is updated. After that, SIL simulation mode is restarted.

The demo uses an If Action and a Stateflow subsystem for the selective choice of two possible variants. The `SIMULATION_VARS/CFG_ENABLE_FILTER` variable defines the code variant and changes according to the selected version of their class and value. According to the variable class (`CG_SWITCH` , `PP_SWITCH` , `RT_SWITCH`), the binding time of the variant function is determined. The value of the variable sets the filter type - filter (`1`) or filter (`0`).



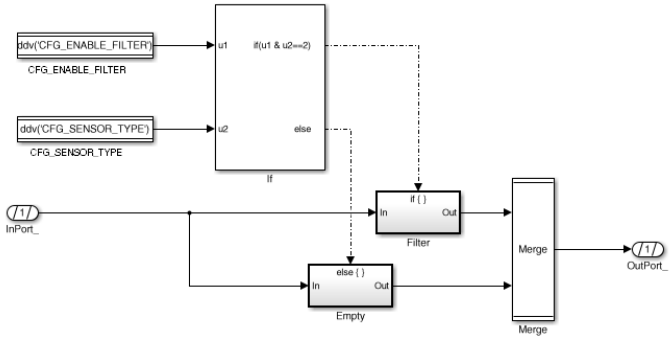
For the binding times code generation, compilation and runtime, the demo allows you to adapt the variants using Data Dictionary-based variant configurations, both for Simulink and Stateflow simulations. You can swich the variants in the Data Dictionary Manager as shown below:



Variant-binding is configured by simply changing the variable classes of the `CFG_ENABLE_FILTER` configuration variable. When binding during code generation is configured, the code items described above are completely removed from the code if the variant is disabled. Only exception: The definition of a DISP variable remains in the generated code, because it can't be specified as ERASABLE. This applies to all variants (`CG_SWITCH`, `PP_SWITCH`_and `RT_SWITCH`) and also for both subsystems (for Stateflow, `CFG_ENABLE_FILTER_SF`).

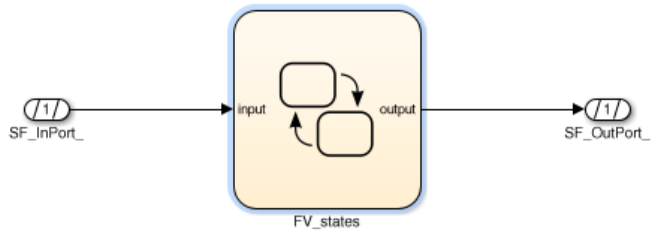
FV_IfAction subsystem

This model demonstrates the use of If Action subsystems for variants.

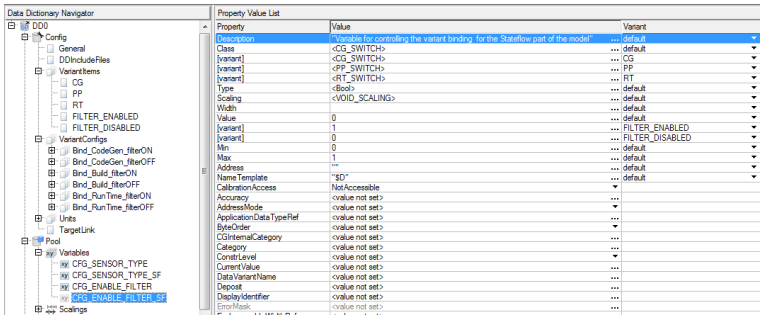


For variant-binding during compilation the **Filter** block contains a lookup table and a PT1 filter. A compound condition is used as switch condition. CAL/DISP variables, lookup table data and lookup table functions are encapsulated by preprocessor conditions.

FV_Stateflow chart



This part of the demo uses a Stateflow chart with transitions calling a graphical function for the selective choice of two possible variants. Similar to the FV_IfAction subsystem, code variants are used to determine the binding timing and the activation of the function variant. The `CFG_ENABLE_FILTER_SF` variable can be modified separately from the Simulink variables. In addition, a constant factor is multiplied to the first transition branch. This modification is for consistency purposes only.



ENUM_TYPES

Access

In the MATLAB Command Window, enter the following command:

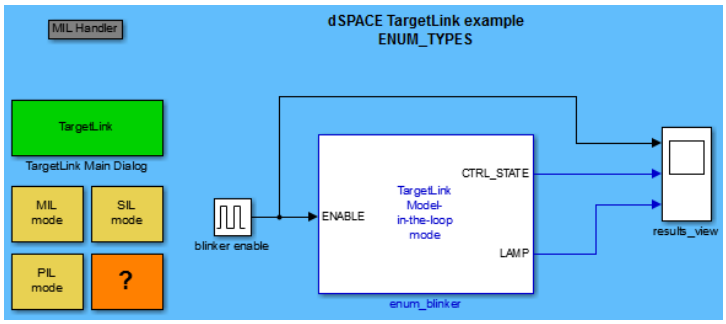
```
tl_demos enum_types
```

You can also enter **`tl_demos`** to access all available demos in the MATLAB Help browser. Select a demo and click **Open this example** at the top right.

Description

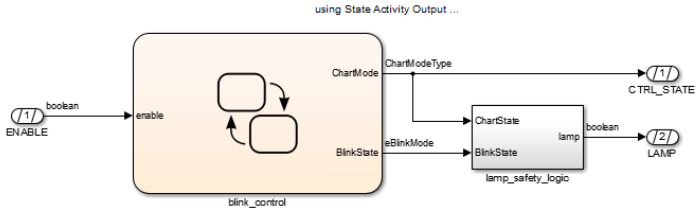
This demo model shows Stateflow Active State Data output (in older Stateflow versions called Active State Output) as Simulink enumeration types in the model. It also shows how to work with enumerations for TargetLink code generation.

The illustration below shows the root level of the enum_types demo.



Stateflow Active State Data and enumerations in model and generated production code

In particular the following TargetLink Code Generation functionalities for enumerations and Stateflow are shown: Outputting Stateflow states' *Child Activity* as an enumeration with optimized code generation, and generating C code enums or macros from the model. Refer to the **enum_blinker** subsystem:

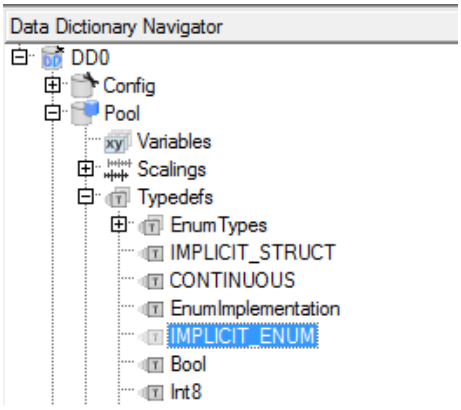


Implementation of Simulink enumerations TargetLink can implement Simulink enumerations in the generated code implicitly and explicitly:

- Via implicit enumeration data type:

  TargetLink automatically implements a data type in the generated code whose characteristics correspond to the Simulink enumeration data type. By default, the Simulink enumerations are implemented as *C enum typedefs* .

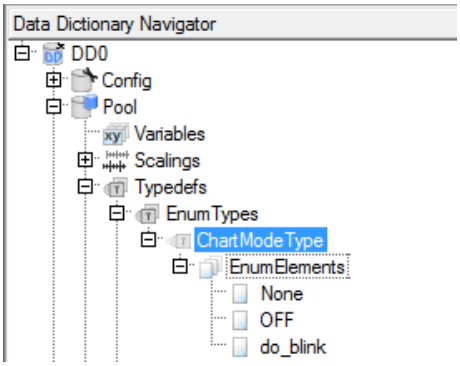
  This approach makes use of the predefined DD **IMPLICIT_ENUM** typedef object:



- Via explicit enumeration data type:

  TargetLink implements a data type in the generated code whose characteristics are specified in a user-defined DD typedef object. The typedef can be implemented either as *C enum typedefs* or as *macros* and also other settings like module, etc. can be specified in the DD.

  This approach makes use of user-defined DD enumeration typedef objects:

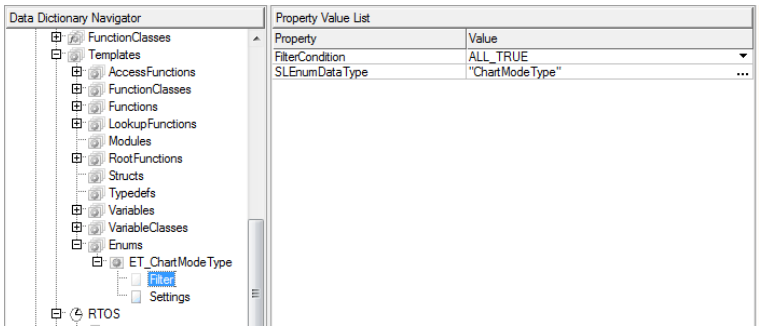


Tip

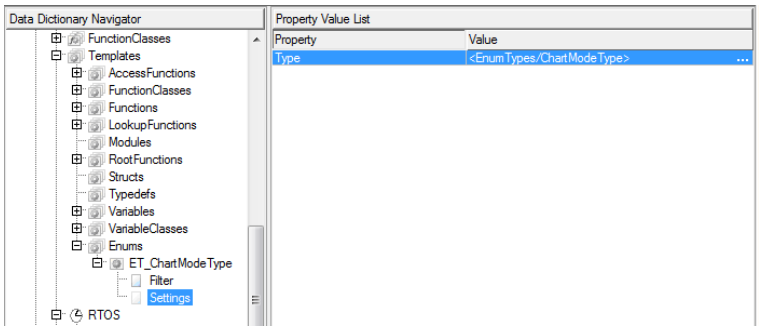
You can conveniently assign DD Typedef objects to Simulink enumeration data types by using DD EnumTemplate objects.

Using EnumTemplates (optionally) Optionally, to map Simulink enumerations on TargetLink enumerations, you can use EnumTemplates. The TargetLink enumeration type has to be defined explicitly in the Data Dictionary. In the model, the default type specification IMPLICIT_ENUM can remain unchanged.

The following illustration shows that the ET_ChartModeType enum template is applied for the Simulink ChartModeType enumeration type:



The following illustration shows that the **ET_ChartModeType** enum template references the explicit **ChartModeType** typedef:



Related topics

Basics

Applying Simulink Enumeration Data Types in TargetLink (📖 TargetLink Preparation and Simulation Guide)
Basics on Making Simulink Enumerations TargetLink-Compliant (📖 TargetLink Preparation and Simulation Guide)
Basics on Working with Active State Data (📖 TargetLink Preparation and Simulation Guide)
ENUM_BLINKER 81

ENUM_BLINKER

Access

In the MATLAB Command Window, enter the following command:

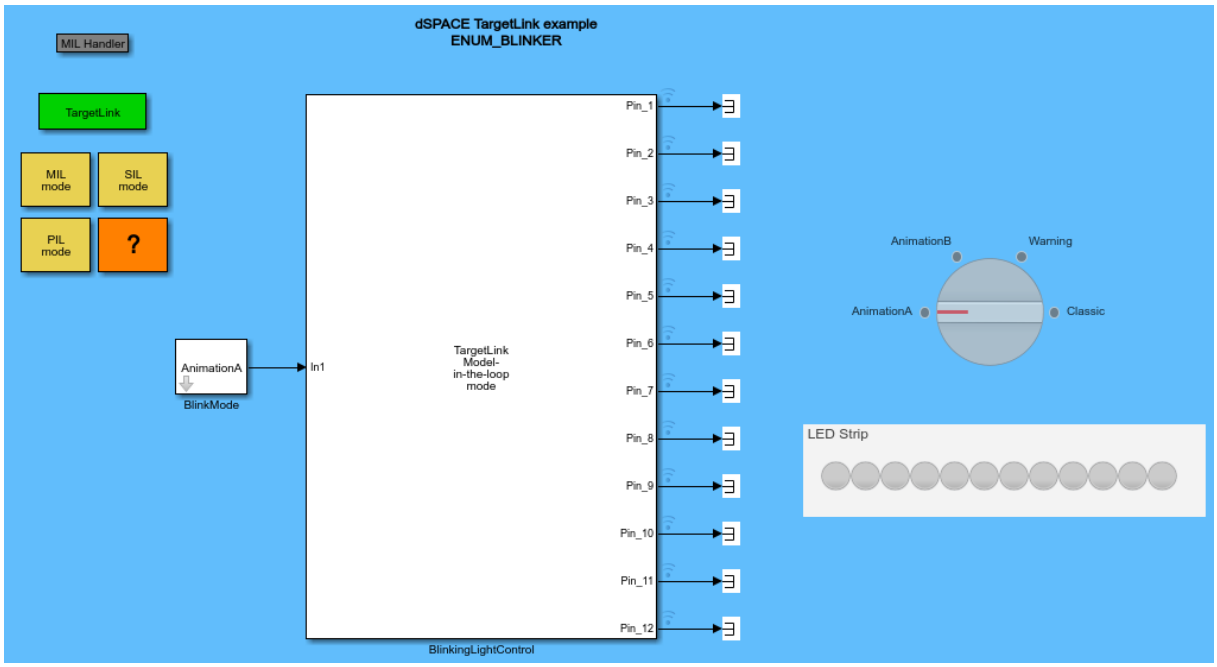
```
tl_demos enum_blinker
```

You can also enter **`tl_demos`** to access all available demos in the MATLAB Help browser. Select a demo and click **Open this example** at the top right.

Description

The **`ENUM_BLINKER`** demo model shows you how to implement a blinker by using Simulink enumerations.

The following illustration shows the root level of the `ENUM_BLINKER` demo:



You can use the rotary switch to select a blink mode and start the simulation. The LED strip shows the selected blink mode.

Enum class definition The demo model uses an enum class definition provided via the `BlinkMode.m` file:

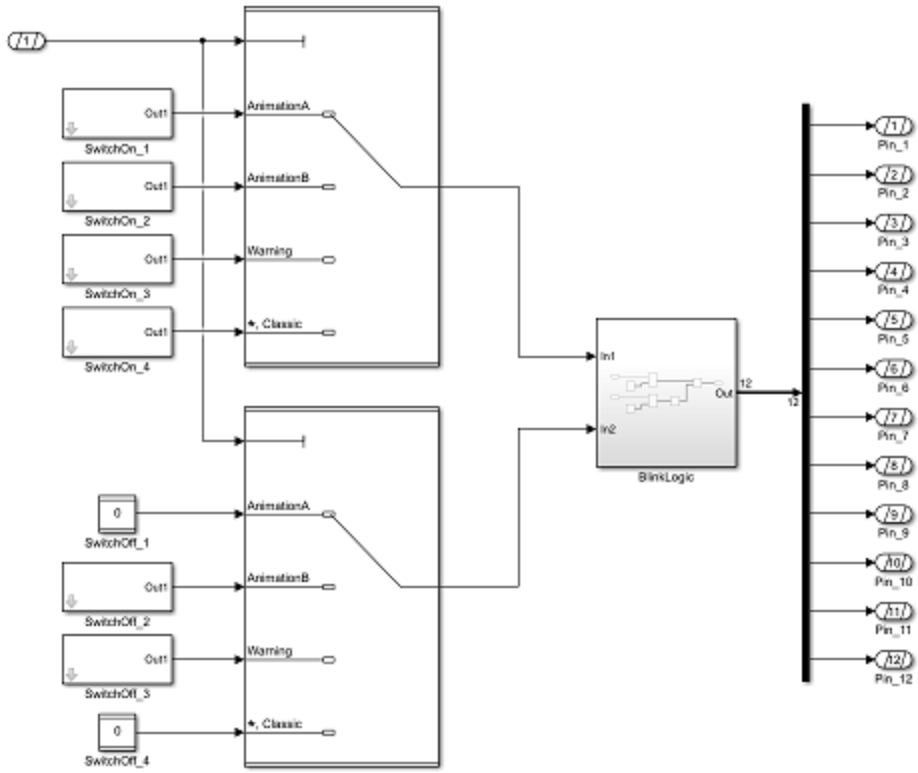
```
classdef BlinkMode < Simulink.IntEnumType

    enumeration
        AnimationA (1)
        AnimationB (2)
        Warning    (3)
        Classic    (4)
    end
end
```

Rotary switch The rotary switch uses integers as its state values that are labeled consistently with the `BlinkMode` enum. The switch is connected with the masked subsystem called BlinkMode that converts the integer to the corresponding enumerated name. This enumerated name is then fed into the TargetLink subsystem.

LED strip The LED strip consists of 12 Lamp elements. Each Lamp element is connected with a signal that leaves the TargetLink subsystem. Each of these signals is a Boolean that represents whether the lamp is on (`1`) or off (`0`).

BlinkingLightControl subsystem

This is the BlinkingLightControl subsystem:

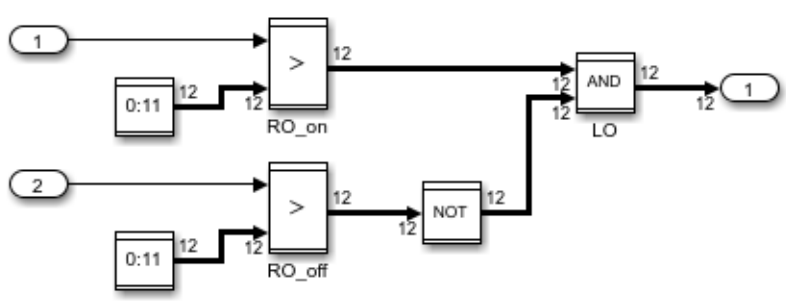


It contains two Multiport Switch blocks.

The upper Multiport Switch block selects which LEDs are switched on, depending on the enumerated name that is fed into the TargetLink subsystem. The lower Multiport Switch block selects which LEDs are switched off, depending on the enumerated name that is fed into the TargetLink subsystem.

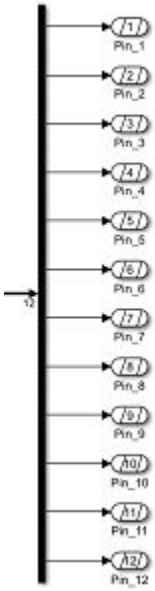
BlinkLogic subsystem

This is the BlinkLogic subsystem:



The subsystem computes a vector whose Boolean elements indicate whether the lamp is on (`1`) or off (`0`).

In the BlinkingLightControl subsystem, this vector signal is demuxed. Each vector element then leaves the TargetLink subsystem:



Enum declaration in production code

TargetLink generates the declaration of the enumerated data type in the `udt_a` header file:

```
typedef enum BlinkMode_tag {
   BLINKMODE_ANIMATIONA = 1,
   BLINKMODE_ANIMATIONB = 2,
   BLINKMODE_WARNING = 3,
   BLINKMODE_CLASSIC = 4
} BlinkMode; /* Description: Enumeration type derived from Simulink type BlinkMode
*/
```

The following listing shows the code of the lower **Multiport Switch** block as an example:

```
/* Multiport switch: BlinkingLightControl/Multiport Switch1 */
   switch (Sa1_In1) {
      case BLINKMODE_ANIMATIONA: {
         Sa1_Multiport_Switch1 = 0;
         break;
      }
      case BLINKMODE_ANIMATIONB: {
         /* SLStaticLocalInit: Default storage class for static local variables
with initvalue | Wid
         th: 16 */
         ...


         /* # combined # Delay: BlinkingLightControl/SwitchOff_2/Delay */
         Sa1_Multiport_Switch1 = X_Sa3_Delay[X_Sa3_Delay_Index];
         break;
      }
      case BLINKMODE_WARNING: {
         /* SLStaticLocalInit: Default storage class for static local variables
with initvalue | Wid
         th: 16 */
         ...

         /* # combined # Delay: BlinkingLightControl/SwitchOff_3/Delay */
         Sa1_Multiport_Switch1 = X_Sa4_Delay[X_Sa4_Delay_Index];
         break;
      }
      case BLINKMODE_CLASSIC:
      default: {
         Sa1_Multiport_Switch1 = 0;
         break;
      }
   }
...
```

Related topics

Basics

Applying Simulink Enumeration Data Types in TargetLink (📖 TargetLink Preparation and Simulation Guide)
ENUM_TYPES 78

BUS_CC

Access

In the MATLAB Command Window, enter the following command:

```
tl_demos bus_cc
```

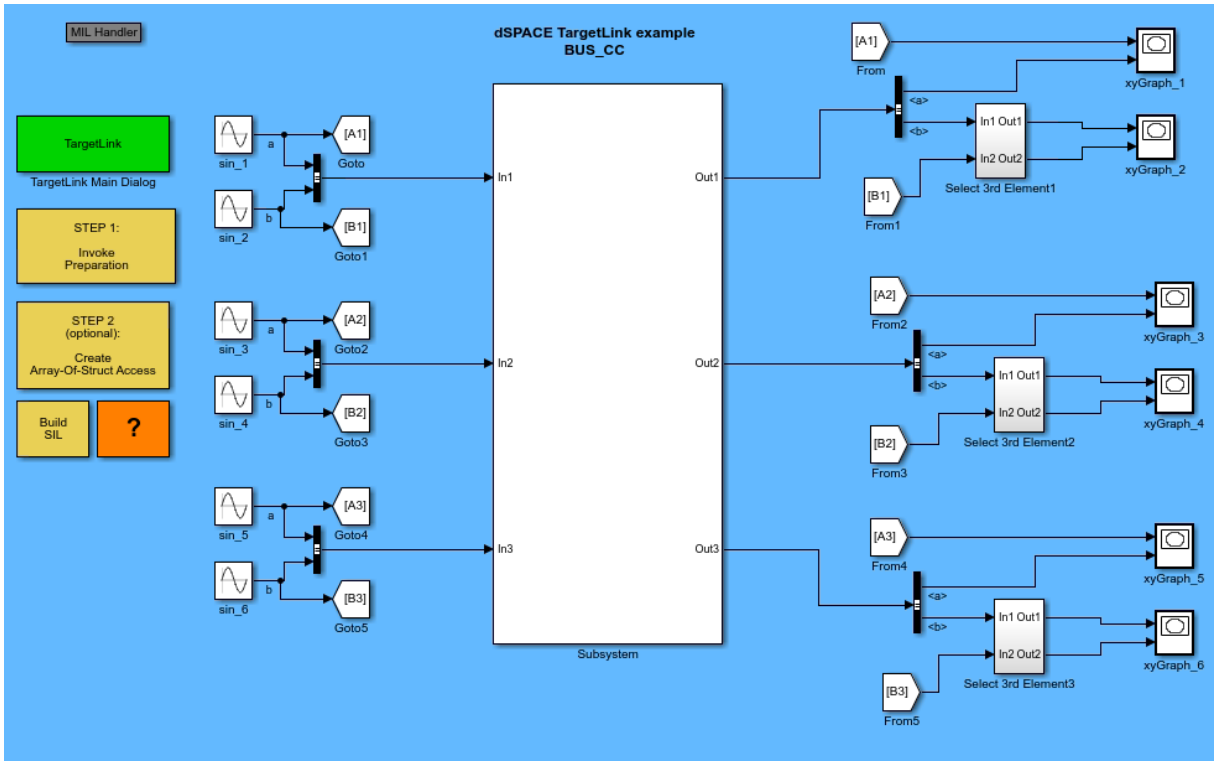
You can also enter **`tl_demos`** to access all available demos in the MATLAB Help browser. Select a demo and click **Open this example** at the top right.

Description

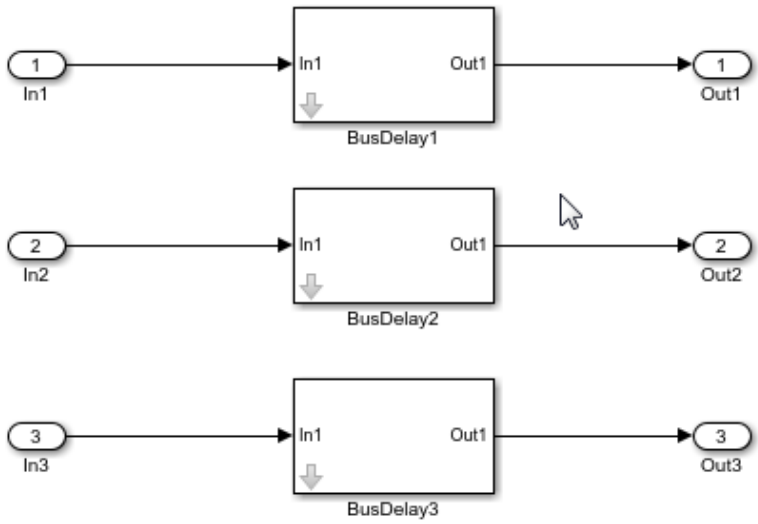
The BUS_CC demo model shows you the use of Simulink buses and TargetLink structured data types with the Custom Code (type II) block. In addition, the specification of array-of-struct variables as DD-based objects are shown. In the model-based CGU, access functions are used to access the elements of the array-of-struct variables.

The demo model contains three masked Simulink subsystems that are used to delay bus signals. During model preparation, the masked subsystems are enhanced to Custom Code (type II) blocks by using a libmap file. The blocks are capable of handling buses.

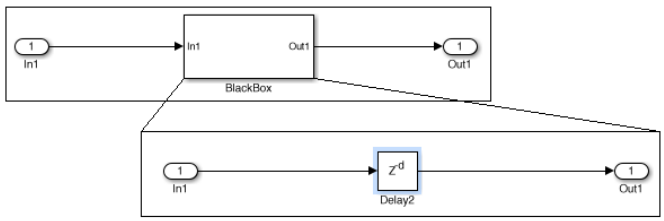
The following screenshot shows the root level of the demo model:



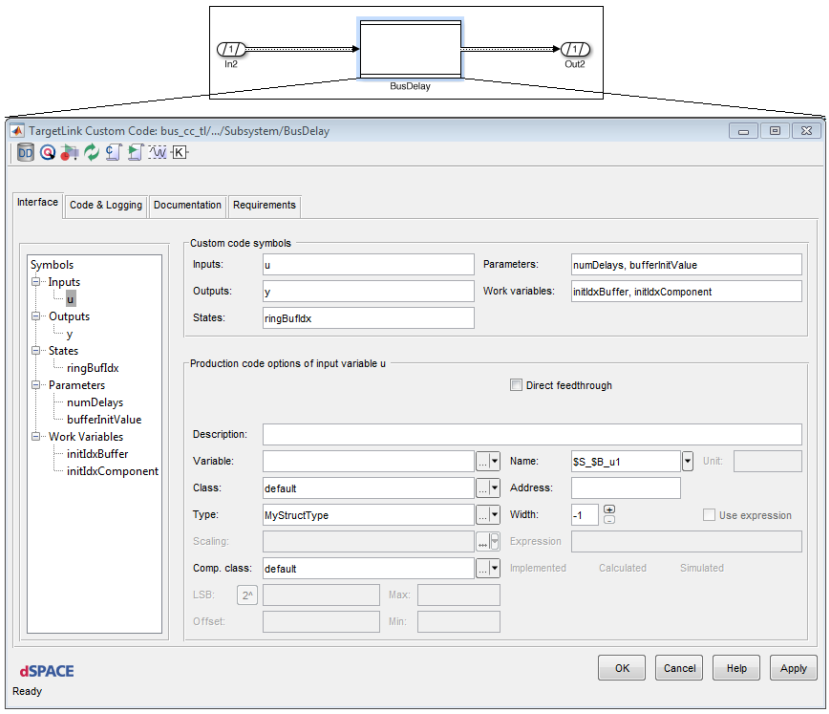
These are the unprepared BusDelay blocks.



Under the mask, an unprepared Delay block contains the following elements:



If you click STEP 1: Invoke Preparation, the `enhanceBusDelay` script in the model's root directory is executed and the BusDelay blocks are enhanced to TargetLink Custom Code (type II) blocks. The `busdelay_libmap` libmap contains the rules for enhancing the blocks.



Using Simulink buses and TargetLink structured data types

Referencing explicitly specified structured data types Explicit structures can be specified at the block variables of the Custom Code (type II) block, in this example at the `u` input, at the `y` output, and at the `bufferInitValue` state as shown in the screenshot before (in this case, the `u` input is shown).

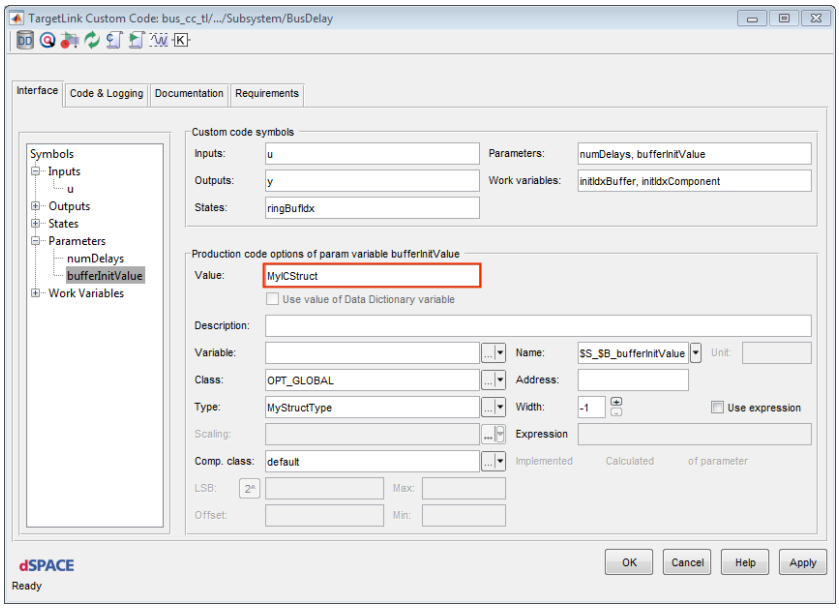
Specifying a mapping to a Simulink property via the slparam block property You must specify a mapping to a Simulink property with the `slparam` block property in order to specify a value ≠ `0` for the block parameter variable. Refer to the `enhanceBusDelay` script as shown in the following screenshot:

```
% create and parameterize the delay buffer
tl_set(hBlock, 'state.varname', 'delayBuffer');
tl_set(hBlock, ...
    'state.slparam',     'vinit', ...
    'state.initial',     get_param(hBlock, 'vinit'), ...
    'state.dimexpr',     'getDelayBufferSize', ...
    'state.class',       'STATIC_GLOBAL', ...
    'state.name',        '$BBuffer');
```

Note

TargetLink does not support the specification of an initial value for a block parameter or block state variable of the Custom Code (type II) block if the variables are not assigned to a parameter of the Simulink block.

Using Simulink initial condition structs A Simulink initial condition struct value is used for the block parameter variable of the Custom Code (type II) block. In this demo, it is `bufferInitValue`, as shown in the following screenshot:



Specifying a Simulink.Bus object via the sldatatypestr block property You must specify a Simulink.Bus object with the `sldatatypestr` property in order to allocate the values of the Simulink initial condition struct to the components of the structured variables. You must specify it in Simulink notation. Refer to the `enhanceBusDelay` script as shown in the following screenshot:

```
% parameter variable for delay buffer initial value

slparam = 'bufferInitValue';

tl_set(hBlock, 'param(2).varname', 'bufferInitValue');
tl_set(hBlock, ...
    'param(2).slparam',          slparam, ...
    'param(2).sldatatypestr',    'Bus: MyBus', ...
    'param(2).value',            get_param(hBlock, slparam), ...
    'param(2).class',            'OPT_GLOBAL', ...
    'param(2).type',             'MyStructType', ...
    'param(2).name',             '$S_$B_bufferInitValue');
```

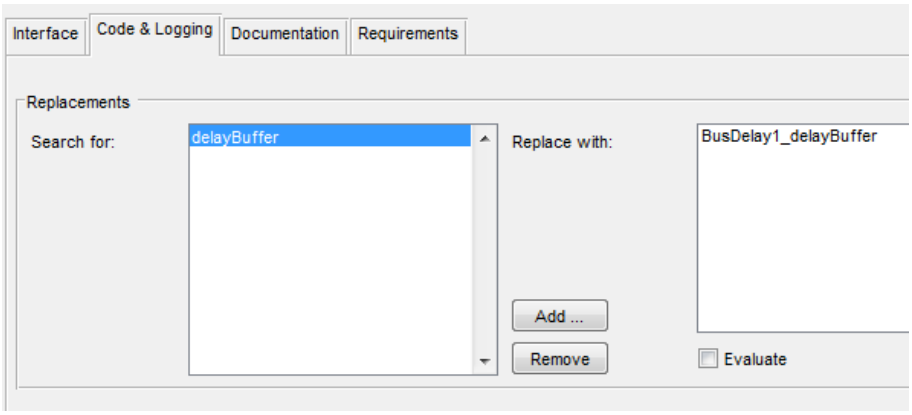
Specifying an array-of-struct variable in the custom code An array-of-struct variable is specified in the custom code as follows: Go to the Code and Logging page of a `BusDelay` Custom Code block and select Edit Custom Code.

```
/* fxp_header_begin */
#include "MyUserTypes.h"
/* fxp_header_end */

/* fxp_decl_begin */
MyStructType delayBuffer[numDelays];
/* fxp_decl_end */
```

The custom code defines an array-of-struct variable that is used as a delay buffer variable.

Because the width of these array-of-struct variables depends on the delay length (`numDelays` parameter) specified at the block, the array-of-struct variable is not specified in the DD. Only array-of-struct variables with a fixed width can be specified in the DD.

A separate delay buffer variable is required for each instance of the Custom Code block, and the name of the delay buffer variable is defined for each instance using custom code replacements. The following illustration shows this replacement for the BusDelay1 block.



Accessing components of structured variables in custom code The following sections of the custom code access components of structured variables and array elements of an array-of-struct variable. Go to the Code and Logging page of a `BusDelay` Custom Code block and select Edit Custom Code.

- The `Restart` section of the custom code contains accesses to components of structured variables:

```
/* fxp_restart_begin */
for (initIdxBuffer = 0; initIdxBuffer < numDelays; initIdxBuffer++)
{
    delayBuffer[initIdxBuffer].a = bufferInitValue.a;

    for (initIdxComponent = 0; initIdxComponent < 3; initIdxComponent++)
    {
        delayBuffer[initIdxBuffer].b[initIdxComponent] = bufferInitValue.b[initIdxComponent];
    }
}
/* fxp_restart_end */
```

- The `Output` and `Update` sections of the custom code contain accesses to the array elements of the array-of-struct variables:

```
/* fxp_output_begin */
y = delayBuffer[ringBufIdx];
/* fxp_output_end */

/* fxp_update_begin */
delayBuffer[ringBufIdx] = u;

ringBufIdx++;
if (ringBufIdx == numDelays)
{
    ringBufIdx = 0;
}
/* fxp_update_end */
```

Specifying array-of-struct variables as DD-based objects and accessing their array elements in the model

The demo model shows how to specify array-of-struct variables as DD objects. Furthermore, if you click STEP 2 (optional): Create Array-Of-Struct Access, access to the array elements of this variable in the model is established by using access functions.

Two array-of-struct variables (ArrayOfStructInputVar, ArrayOfStructOutputVar) are specified in the TargetLink Data Dictionary of the TargetLink demo model. These variables reference the same DD-based module of a DD-based code generation unit. For the array-of-struct variables, DD Variable objects are created that access the three array elements of the array-of-struct variables. The objects (ArrayOfStructInputVar_0..2, ArrayOfStructOutputVar_0..2) are used at the bus inputs and bus outputs of the TargetLink subsystem. The Alias property is set to on so that the variables, e.g., ArrayOfStructInputVar_0, at which the access functions are specified are not generated to the code.

The DD Variable objects reference the same DD-based module as the DD Variable objects representing the array-of-struct variables. This results in a central definition of the array-of-struct variables and the access functions for their array elements in a module that is generated using DD-based code generation.

In the production code for the model-based code generation unit, these access functions are used to access the array elements of the array-of-struct variables specified in the TargetLink Data Dictionary.

The `createAoSFieldElementAccessDDVars.m` M-function is used to create the DD Variable objects. This function is provided with the demo and shows how to create DD Variable objects for the array elements of an array-of-struct variable from the specification of the array-of-struct variables in the TargetLink Data Dictionary. The function creates a set of DD objects for each array element of the array-of-struct variables:

- A `Variable` object
- A `VariableClass` object
- An `AccessFunctionTemplate` object
- An `AccessFunction` object

MATRIX_INVERSE

Access

In the MATLAB Command Window, enter the following command:

```
tl_demos matrix_inv
```

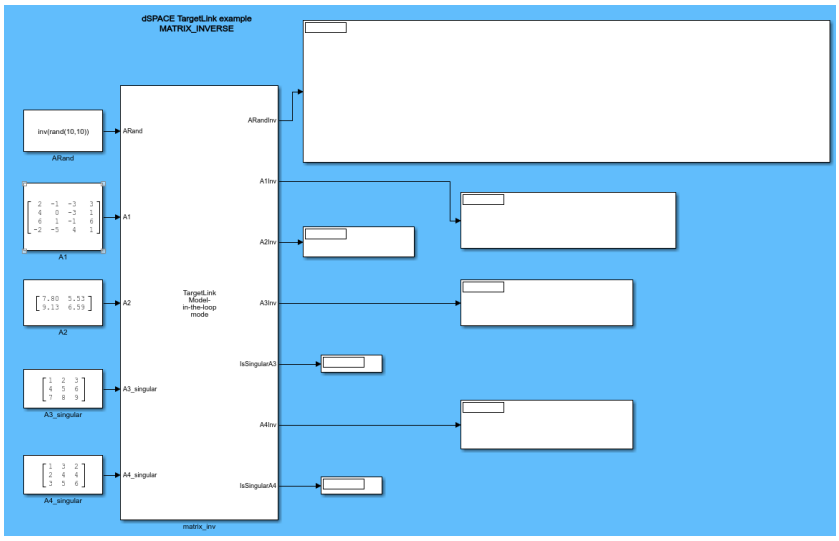
You can also enter **`tl_demos`** to access all available demos in the MATLAB Help browser. Select a demo and click **Open this example** at the top right.

Description

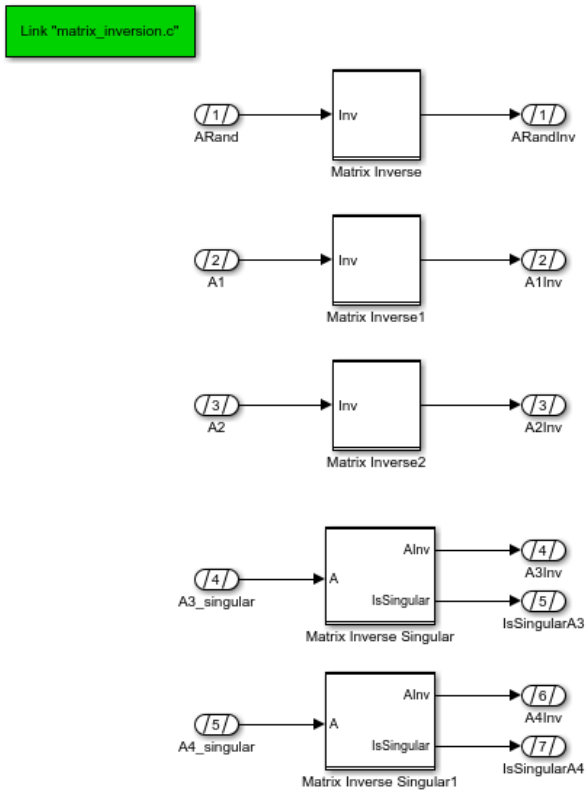
The `MATRIX_INVERSE` demo model shows two possibilities of a matrix inversion:

- a block with only one output, which simply calculates and returns a matrix - without any error check.
- a block with an additional second output that returns a status informing you whether the inverse matrix could be calculated.

The illustration below shows the root level of the matrix_inverse demo model.



Five Custom Code Type II blocks reside in this demo model. The first three demonstrate inverse matrices without a verification. They have different sizes (10x10, 4x4, 2x2). The last ones have an error verification whether the matrix is invertible - both are not invertible.



Inversion of matrices

Two different variants of preconfigured Custom Code Block Type II blocks are used in this demo model. The first variant (first three blocks) without verification, the last one with an error verification. The corresponding custom code template only contains the call of the algorithm implementation provided in a separate source file. The calculation of the inverse matrix in MIL simulation mode is performed by a Simulink Product block specified to calculate the inverse of the input matrix.

You can replace this function call by your own implementation. You can change the following files for your own purposes:

- Demo model with DD, library, and custom code
- C code implementation of the `matrix_inversion.c / *.h`
- The files required for model preparation for converting a Simulink block to a TargetLink Custom Code Type II block and vice versa:
  - `enhance<block>.m`
  - `deenhance<block>.m`
  - `<block>_libmap.m`

Variants of matrix inversions

The first three blocks show the calculation of inverse matrices without checking whether the inverse exists or can be calculated numerically. They have different sizes (10x10, 4x4, 2x2).

The last two blocks return an error status indicating if the inverse exists or can be calculated numerically. However, the `isSingular` output returns different values in MIL and SIL simulation mode:

- In MIL simulation mode, the `isSingular` output returns two values:
  - `true (1)` if Simulink has calculated `+/-inf` or `nan` for a component of the inverse
  - Otherwise: `false (0)`

Note

The `false (0)` value in MIL simulation mode does not mean that the matrix to be inverted can actually be inverted. For example, the matrix on the A3_singular input in the demo model cannot be inverted - but Simulink calculates any values for it with imprecise values.

- In SIL simulation mode, the `isSingular` output returns three values:
  - `0` if the algorithm can calculate an inverse, e.g., with the matrix A1 as input
  - `1` if the algorithm determines that no inverse exists, as it is implemented in the model
  - `2` if the algorithm cannot calculate the inverse due to numerical errors (rounding errors, etc.), not implemented in the model

Note

The implementation of the inverse in the source files receives only one dimension as a parameter. This is sufficient, because only square matrices can be inverted.

Additional rules for using matrix inversion

The following rules apply when working with the TargetLink matrix inversion functionalities:

- If it is not certain whether the matrices to be inverted can be inverted or whether the inverse can be calculated (keywords: *condition number of a matrix* or *ill-conditioned matrix*), the block with the `IsSingular` output must be used.

Note

The Simulink block in MIL simulation mode does not perform checks, but simply calculates a matrix with any values - if necessary with `+/-inf` or `nan`.

- In the demo model, TargetLink only uses a simple algorithm for calculating matrix inversions (more detailed information can be found in the comments in the C files). This algorithm delivers good results for invertible and well-conditioned matrices.

- TargetLink cannot ensure that MIL and SIL/PIL simulation mode results are the same, because the accuracy and correctness of the calculated inverse matrix depends on the algorithm used, among other things. However, MIL and SIL simulation modes use different algorithms.
- The `IsSingular` outport returns different values in MIL and SIL simulation mode. For more information, refer to *Variants of matrix inversions*.

Supported Stateflow Features

Where to go from here

Information in this section

STATEFLOW 97
FUELSYS 99
FUELSYS_FLOAT 101

STATEFLOW

Access

In the MATLAB Command Window, enter the following command:

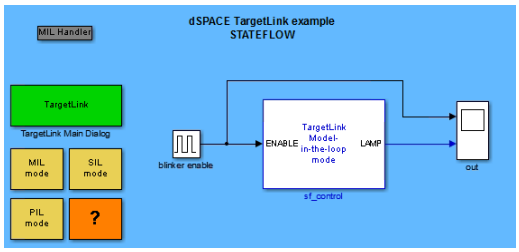
```
tl_demos stateflow
```

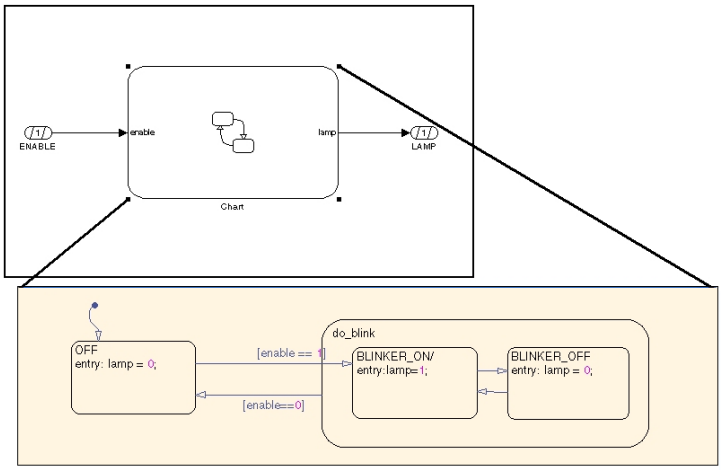
You can also enter **`tl_demos`** to access all available demos in the MATLAB Help browser. Select a demo and click **Open this example** at the top right.

Description

The **`Stateflow`** demo model shows you a simple Stateflow chart.

The illustration below shows the root level of the Stateflow demo model.





Stateflow

TargetLink supports the MathWorks® Stateflow toolbox. It analyzes the control flow part of a Stateflow chart and transforms it into structured C code using conditional and loop statements.

TargetLink-specific properties for standard Stateflow objects are stored in the object's description marked by `$TL$` start and end tags. The order of the properties is not relevant. If a property is not specified, the default value is taken. Each property definition must be unique and have only one object assigned.

TargetLink properties in Stateflow

To access properties in Stateflow, you can use the Model Explorer.

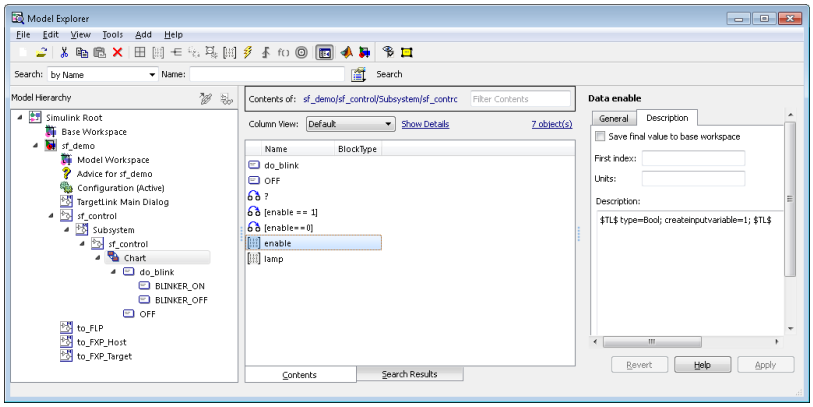
Note

Note that the Model Hierarchy does not necessarily show all elements contained in the model, depending on the settings.
You can specify whether to:

- Show masked subsystems
  (View – Show Masked Subsystems or on the Model Hierarchy pane)
- Show library links
  (View – Show Library Links or on the Model Hierarchy pane)

When working with this demo, it is advisable to select View – Show Masked Subsystems because the Stateflow chart is contained in a masked subsystem.

For example, open the Stateflow demo model via the Model Explorer and navigate to the `sf_control` subsystem as shown below. Select `Chart` to show its properties in the Description tab.



Tip

An easier way to handle a large number of properties is to use the Property Manager.

Related topics

Basics

Basics on the TargetLink Property Manager (📖 TargetLink Preparation and Simulation Guide)
FUELSYS 99

References

Specification of Stateflow Properties in TargetLink (📖 TargetLink Model Element Reference)

FUELSYS

Access

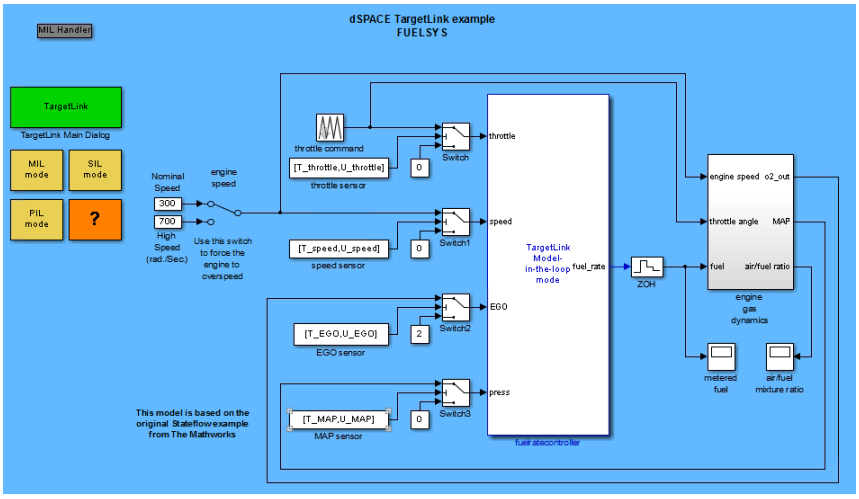
In the MATLAB Command Window, enter the following command:

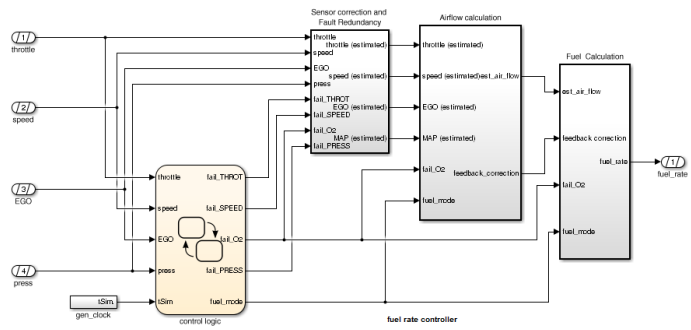
```
tl_demos fuelsys
```

You can also enter **`tl_demos`** to access all available demos in the MATLAB Help browser. Select a demo and click **Open this example** at the top right.

Description

The modeling of a fault-tolerant fuel control system demonstrates how a Stateflow chart in a Simulink model can be used to model hybrid systems that contain both continuous dynamics and complex control logic.





Property Manager & Stateflow

TargetLink-specific properties for standard Stateflow objects are stored in the object's description marked by `$TL$` start and end tags.

The order of the properties is not relevant. If a property is not specified, the default value is taken. The definition of a property must be unique, and a second assignment to the same property is not allowed.

TargetLink parses the description fields for all Stateflow objects. The following results are possible:

- If a description field contains two `$TL$` tags, the text between the two tags is taken as a specification of TargetLink properties: `$TL$` <list of property assignments> `$TL$`.
- Each of the property assignments has the following syntax: <property name>=<property value>.
- If you change the value of a property via the Property Manager, it writes the modifications back to the description field.
- If a wrong value is specified, the Property Manager may replace it with the default value.

- If the text between the two **TL** tags contains anything that is not compliant with the specifications shown in the sections listed below, the Property Manager deletes the erroneous entries when writing modifications back to the description field.

> **Note**
>
> The property keywords can contain both uppercase and lowercase characters.

- The description of a Stateflow object is always interpreted as an intentional variable description. This description is overwritten by a variable description that is available in the DD, but it is not removed if the description of the DD variable is empty. In the Stateflow object it looks like this:

```
'This is a variable description.' $TL$ variable=myVarWithEmpty_DD_Description; $TL$

'' $TL$ variable=myVarWithNonEmpty_DD_Description; $TL$
```

FUELSYS_FLOAT

Access

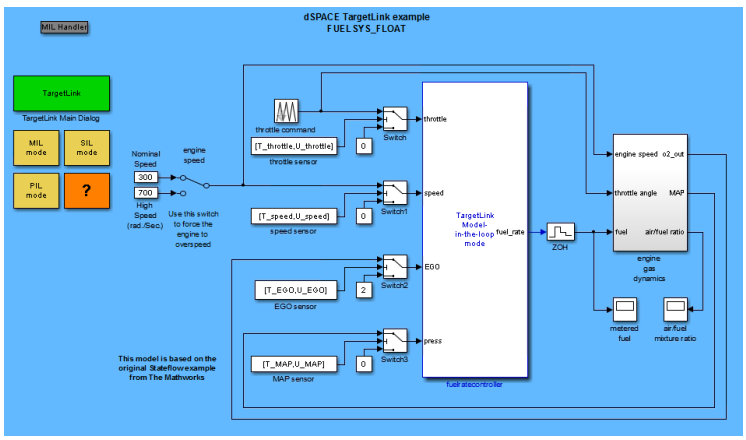
In the MATLAB Command Window, enter the following command:

```
tl_demos fuelsys_float
```

You can also enter **tl_demos** to access all available demos in the MATLAB Help browser. Select a demo and click **Open this example** at the top right.

Description

The fuelsys_float demo model is the floating-point version of the fuelsys demo model and shows you how to do all arithmetic but table maps and control logic in float.

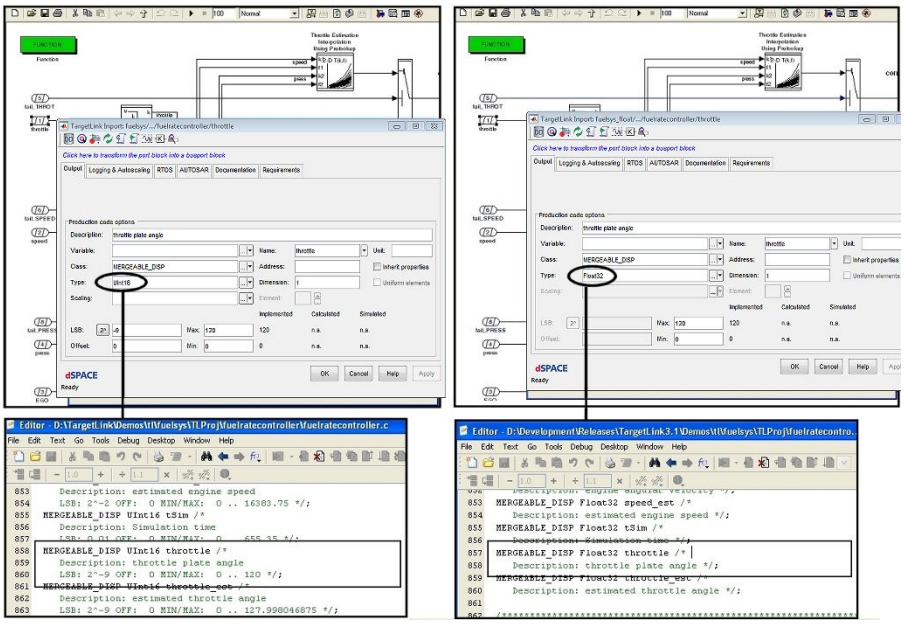


Floating-point code generation

The original fuelsys demo model supports code generation with fixed-point arithmetic. These are used when values are mapped to integers in the programming code by means of a scaling formula (LSB, offset).

The fuelsys_float demo model works with floating-point arithmetic. These are used when variables represent continuously variable physical values (such as speed).

A comparison between the two models and their code output is shown below.



Lookup Tables, User-Written Lookup Functions

Where to go from here

Information in this section

TABLE1D 103
TABLE2D 105
TABLE1D_USR 106
TABLE1D_USR_LOCAL 108
TABLE2D_USR 110
TABLE1D_FCN 112
PRELOOKUP_USR 113
DYNAMIC_LUT 117

TABLE1D

Access

In the MATLAB Command Window, enter the following command:

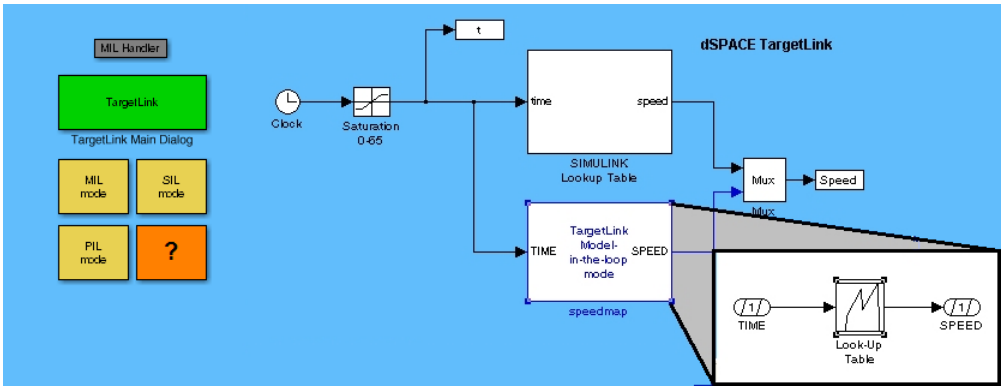
```
tl_demos table1d
```

You can also enter **`tl_demos`** to access all available demos in the MATLAB Help browser. Select a demo and click **Open this example** at the top right.

Description of table1d

The `table1d` demo model shows you how to work with 1-D look-up tables.

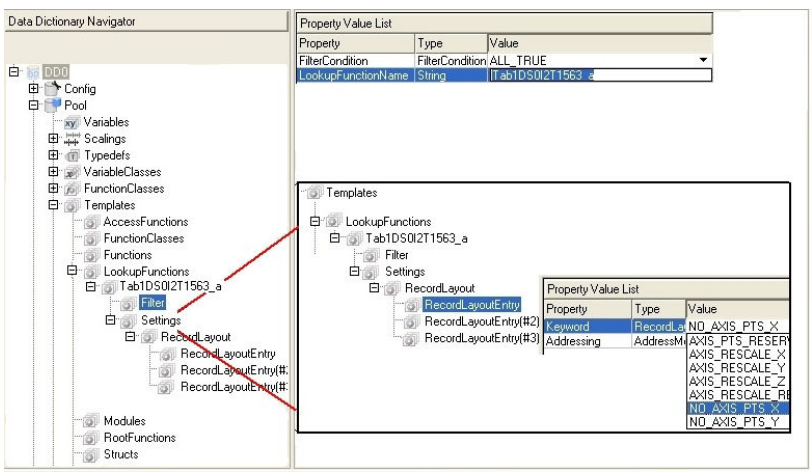
The **Look-Up Table** block maps an input to an output by using linear interpolation of the values defined in the block's parameters. The block icon displays a graph of the input vector (x-axis) versus the output vector (y-axis).



Applying record layouts to customize table data

Table maps are C structs that hold the data of the Look-Up Table and Look-Up Table (2D) blocks. It can be useful to customize table maps, for example, if table code is used by a calibration system that requires a specific structure (record layout) of table data. A record layout provides a scheme that indicates how the objects (e.g., calibration parameters) are to be serialized in the memory of an ECU.

If you want to customize the implemented table code of Look-Up Table and Look-Up Table (2D) blocks, you must reference an existing table map template (record layout) that is defined in `/Pool/Templates/LookUpFunctions` of the TargetLink Data Dictionary.



Related topics

Basics

Basics on Customizing Table Maps (📖 TargetLink Preparation and Simulation Guide)

References

Look-Up Table Block (📖 TargetLink Model Element Reference)

TABLE2D

Access

In the MATLAB Command Window, enter the following command:

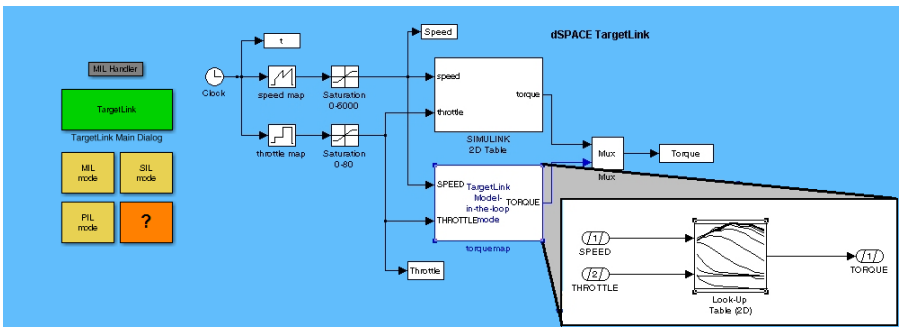
```
tl_demos table2d
```

You can also enter **`tl_demos`** to access all available demos in the MATLAB Help browser. Select a demo and click **Open this example** at the top right.

Description of table2d

The **`table2d`** demo model shows you how to work with 2D look-up tables.

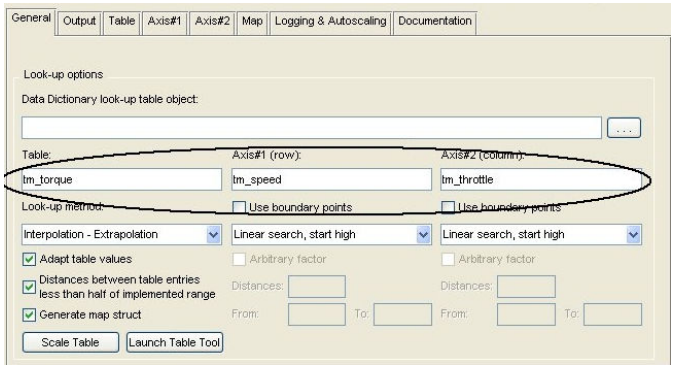
The **Look-Up Table (2D)** block maps two block inputs to one output by using linear interpolation of a table of values defined by the block's parameters. The inputs are compared with the row and column parameters of this table. The first (upper) input identifies a row, the second (lower) input identifies a column.



Workspace variables

TargetLink can access MATLAB´s workspace variables such as **tm_torque**, **tm_speed** and **tm_throttle** instead of specifying the values explicitly in the dialog. To view its values, simply type the variable name in the MATLAB Command Window.

```
>> tm_speed
tm_speed = Columns 1 through 8
     800   1000   1400   1800   2200   2600   3000   3400
           Columns 9 through 14
     3800   4200   4600   5000   5400   6000
```



Related topics

References

Look-Up Table (2D) Block (📖 TargetLink Model Element Reference)

TABLE1D_USR

Access

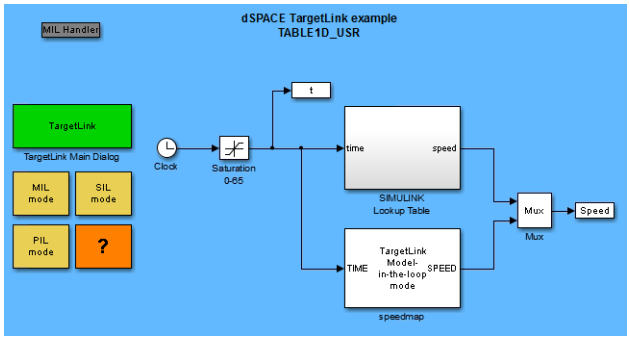
In the MATLAB Command Window, enter the following command:

```
tl_demos table1d_usr
```

You can also enter **`tl_demos`** to access all available demos in the MATLAB Help browser. Select a demo and click **Open this example** at the top right.

Description of table1d_usr

The table1d_usr demo model shows how TargetLink look-up table code can be replaced by custom look-up functions. The MATLAB script lookup1d_script.m and the C-module tab_fcn.c are provided for this purpose.



Custom look-up functions

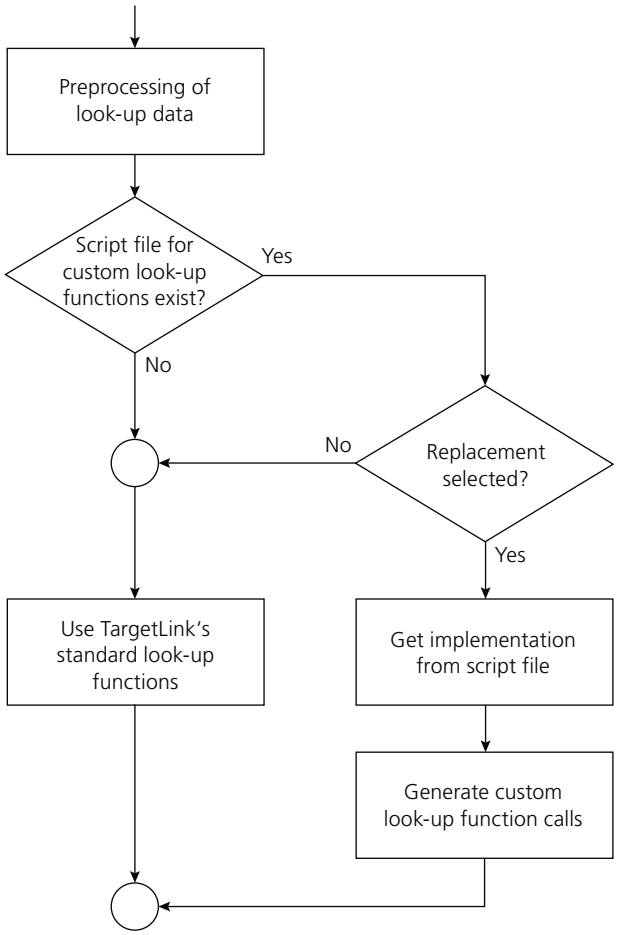
You can replace TargetLink look-up functions with custom look-up functions.

Often the existing ECU software already contains customer-specific functions for look-up tables. In this case you may want to reuse these functions in the code generated by TargetLink to avoid multiple implementations of the same look-up function. TargetLink enables look-up function reuse by providing a scripting mechanism.

Using custom look-up functions you can:

- Call your own look-up functions with an arbitrary name and argument list.
- Create variables to store the look-up table data, including arbitrary data structures.
- Include the necessary header files.

The following illustration shows the code generation process for look-up tables:



Related topics

Basics

Basics on Using Custom Look-Up Functions (TargetLink Preparation and Simulation Guide)

TABLE1D_USR_LOCAL

Access

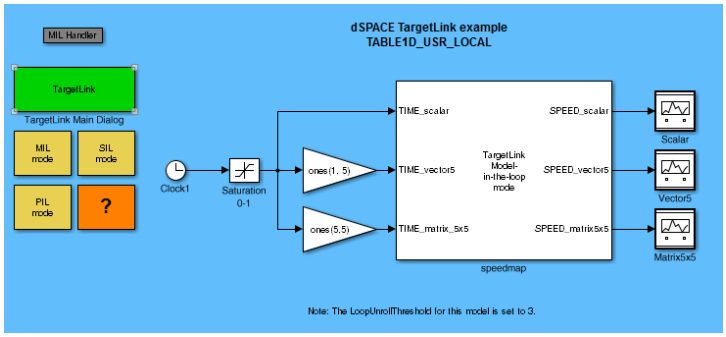
In the MATLAB Command Window, enter the following command:

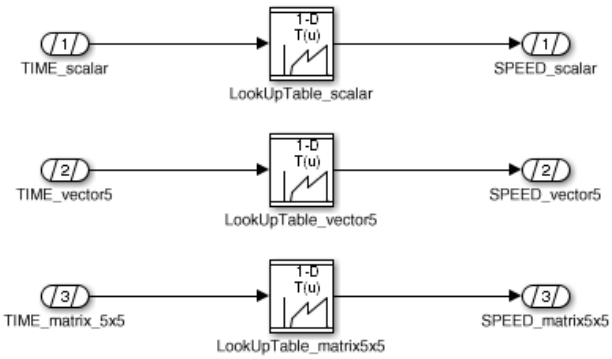
```
tl_demos table1d_usr_local
```

You can also enter **`tl_demos`** to access all available demos in the MATLAB Help browser. Select a demo and click **Open this example** at the top right.

Description of table1d_usr_local

The `table1d_usr_local` demo model shows you how to replace TargetLink look-up table code using a local search algorithm that has nonscalar inputs by custom look-up functions. The `lookup1d_script.m` MATLAB script and the `tab_fcn.c` C file are provided for this purpose.





Accessing nonscalar inputs/outputs in custom look-up functions

To access a nonscalar input or output in a custom look-up function, its scalar placeholder can be used. Refer to the following example:

The template code

```
'out1 = Tab1D_U16_LOCAL(&map,in1,&Last_Idx);'
```

specifies a call to the look-up function Tab1D_U16_LOCAL() with scalar in- and outputs

```
Sb1_LookUpTable_scalar =
Tab1D_U16_LOCAL(&Sb1_LookUpTable_scalar_map,

Sb1_TIME_scalar, …);
```

as well as access to the input and output, if the input of the associated block is a vector:

```
Sb1_LookUpTable_vector5[Aux_S32] = Tab1D_U16_LOCAL(
&Sb1_LookUpTable_vector5_map,Sb1_TIME_vector5[Aux_S32], …);
```

Custom look-up functions with width-inheriting variables

To implement a local search algorithm for a Look-Up Table block with non-scalar inputs, a non-scalar last index variable, that stores the x-axis index value of the last simulation step is necessary. Such a variable must inherit the dimensions from the associated input (the x-axis).

Setting the property `InheritDimensionFromInput` to 1:

```
tlscript('Set','Last_Idx', ...
        …
        'InheritDimensionFromInput' ,1);
```

TABLE2D_USR

Access

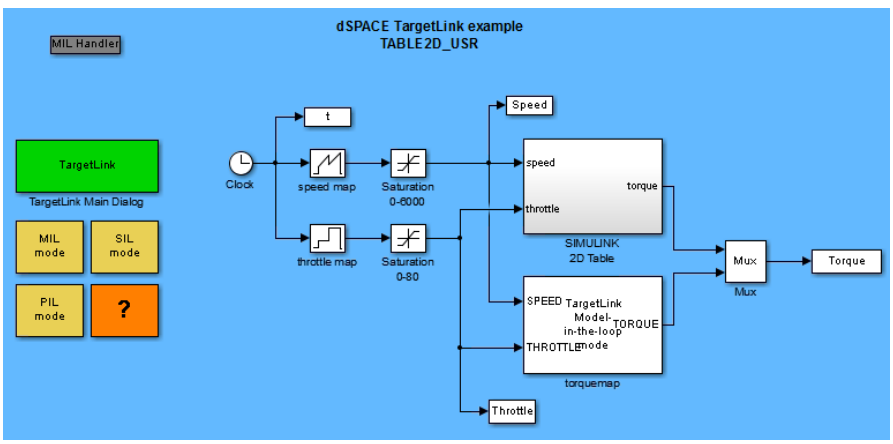
In the MATLAB Command Window, enter the following command:

```
tl_demos table2d_usr
```

You can also enter **`tl_demos`** to access all available demos in the MATLAB Help browser. Select a demo and click **Open this example** at the top right.

Description of table2d_usr

The table2d_usr demo model shows how TargetLink look-up table code can be replaced by custom look-up functions. The MATLAB script lookup2d_script.m and the C-module tab_fcn.c are provided for this purpose.



Custom look-up functions

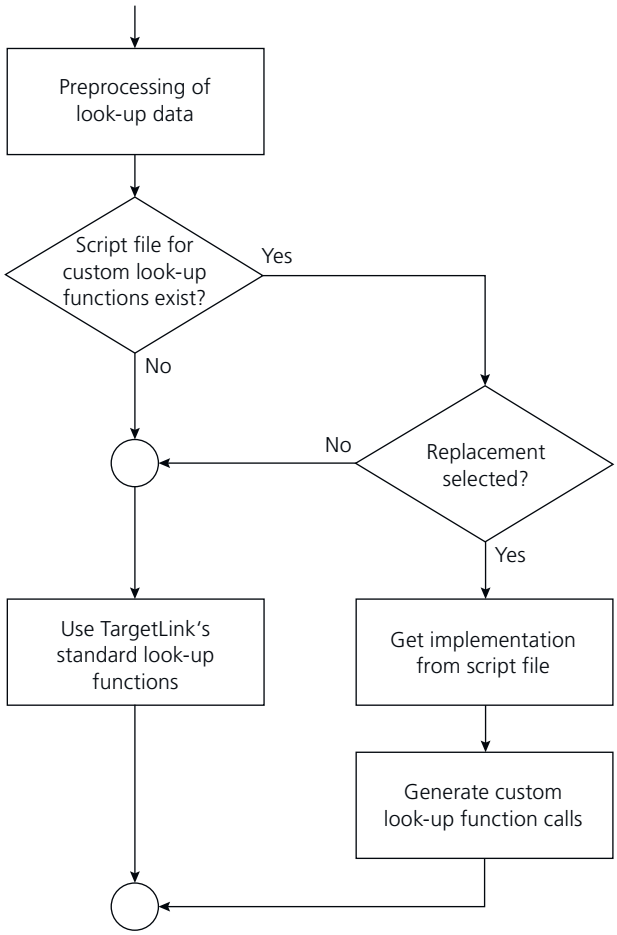
You can replace TargetLink look-up functions with custom look-up functions.

Often the existing ECU software already contains customer-specific functions for look-up tables. In this case you may want to reuse these functions in the code generated by TargetLink to avoid multiple implementations of the same look-up function. TargetLink enables look-up function reuse by providing a scripting mechanism.

Using custom look-up functions you can:

- Call your own look-up functions with an arbitrary name and argument list.
- Create variables to store the look-up table data, including arbitrary data structures.
- Include the necessary header files.

The following illustration shows the code generation process for look-up tables:



Related topics

Basics

Basics on Using Custom Look-Up Functions (📖 TargetLink Preparation and Simulation Guide)

TABLE1D_FCN

Access

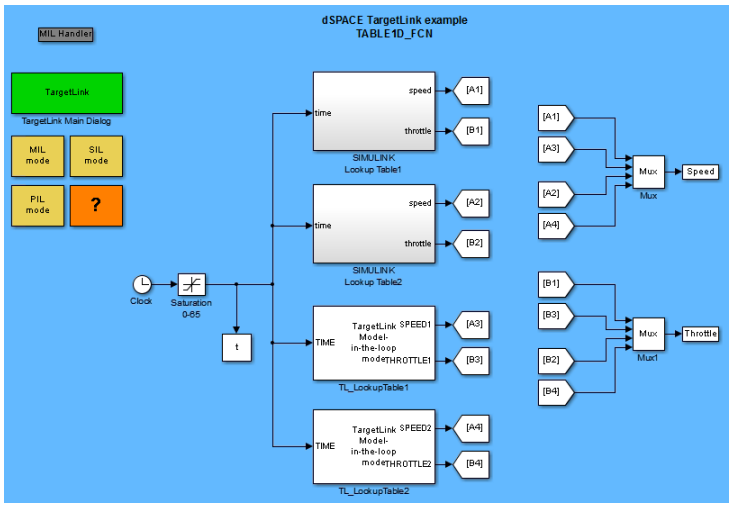
In the MATLAB Command Window, enter the following command:

```
tl_demos table1d_fcn
```

You can also enter **`tl_demos`** to access all available demos in the MATLAB Help browser. Select a demo and click **Open this example** at the top right.

Description

The **`table1d_fcn`** demo model shows you how TargetLink 1D look-up table functions can be shared by several TargetLink subsystems and how to use separate search functions.



Functions shared between subsystems

You can share look-up table functions between TargetLink subsystems to avoid multiple occurrences of the same look-up table function. As a consequence, only one look-up table function, without the TargetLink subsystem ID, is generated into a separate file. This results in less memory consumption when code from several TargetLink subsystems is put together.

Related topics

References

TargetLink Main Dialog Block (📖 TargetLink Model Element Reference)

PRELOOKUP_USR

Access

In the MATLAB Command Window, enter the following command:

```
tl_demos prelookup_usr
```

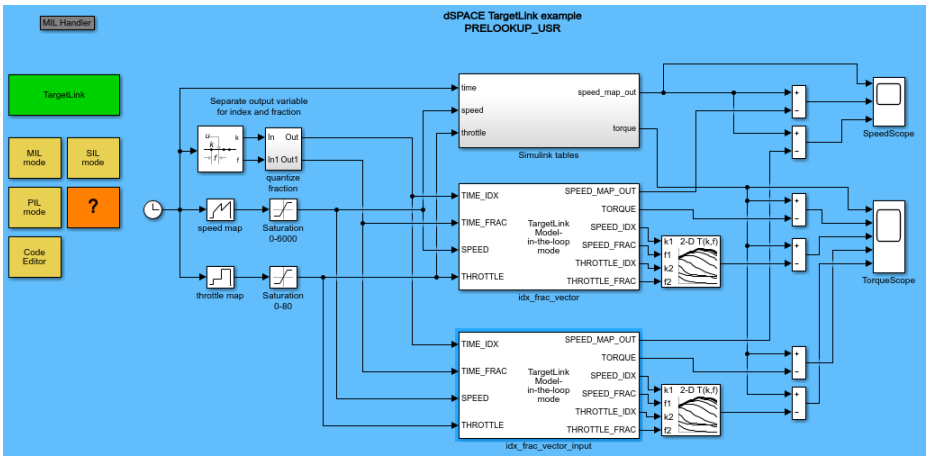
You can also enter **`tl_demos`** to access all available demos in the MATLAB Help browser. Select a demo and click **Open this example** at the top right.

Description

The `PRELOOKUP_USR` demo model shows you how to do the following:

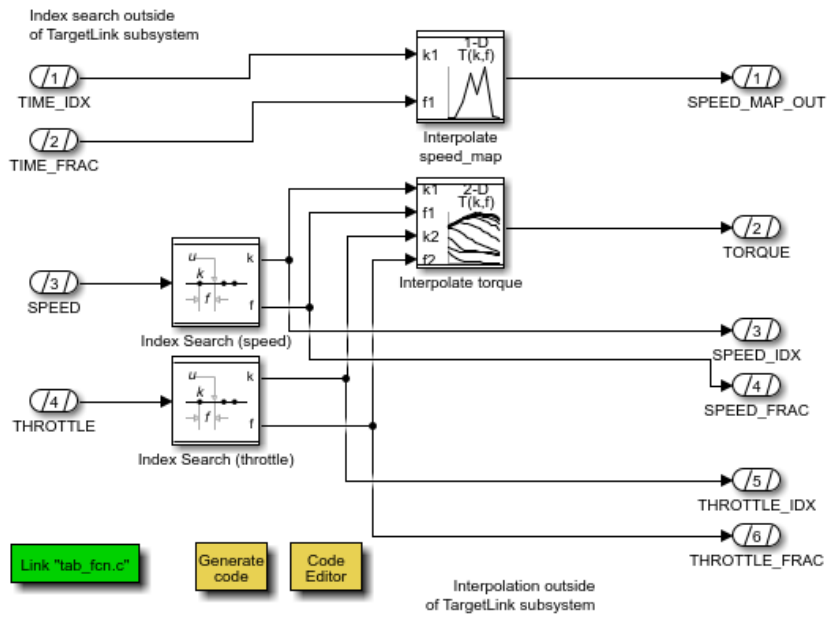
- Specify table data via block dialog or table input ports
- Use custom look-up table functions for **Prelookup** and **Interpolation Using Prelookup** blocks.

The following illustration shows the root level of the `PRELOOKUP_USR` demo model:

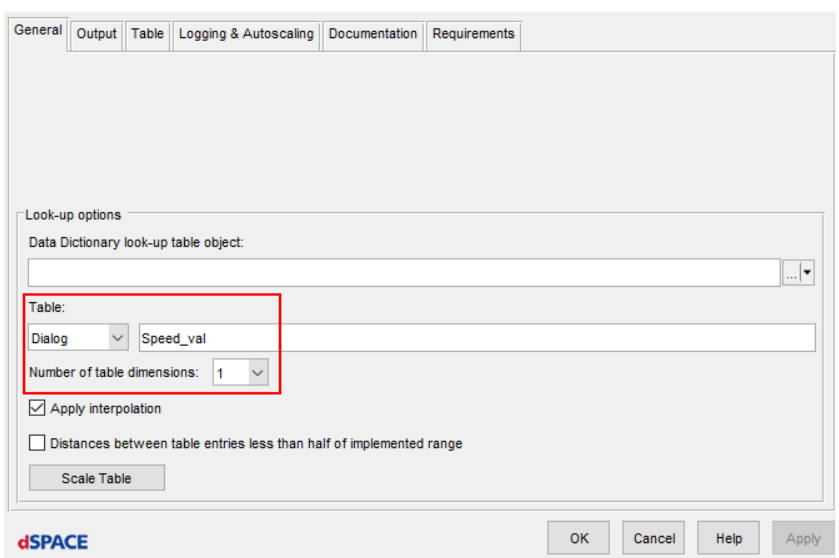


Specifying table data via block dialog

The idx_frac_vector TargetLink subsystem shows how to specify table data via the dialog of Prelookup and Interpolation Using Prelookup blocks:



The following illustration shows the block dialog of the Interpolation Using Prelookup block called `Interpolate speed_map`:



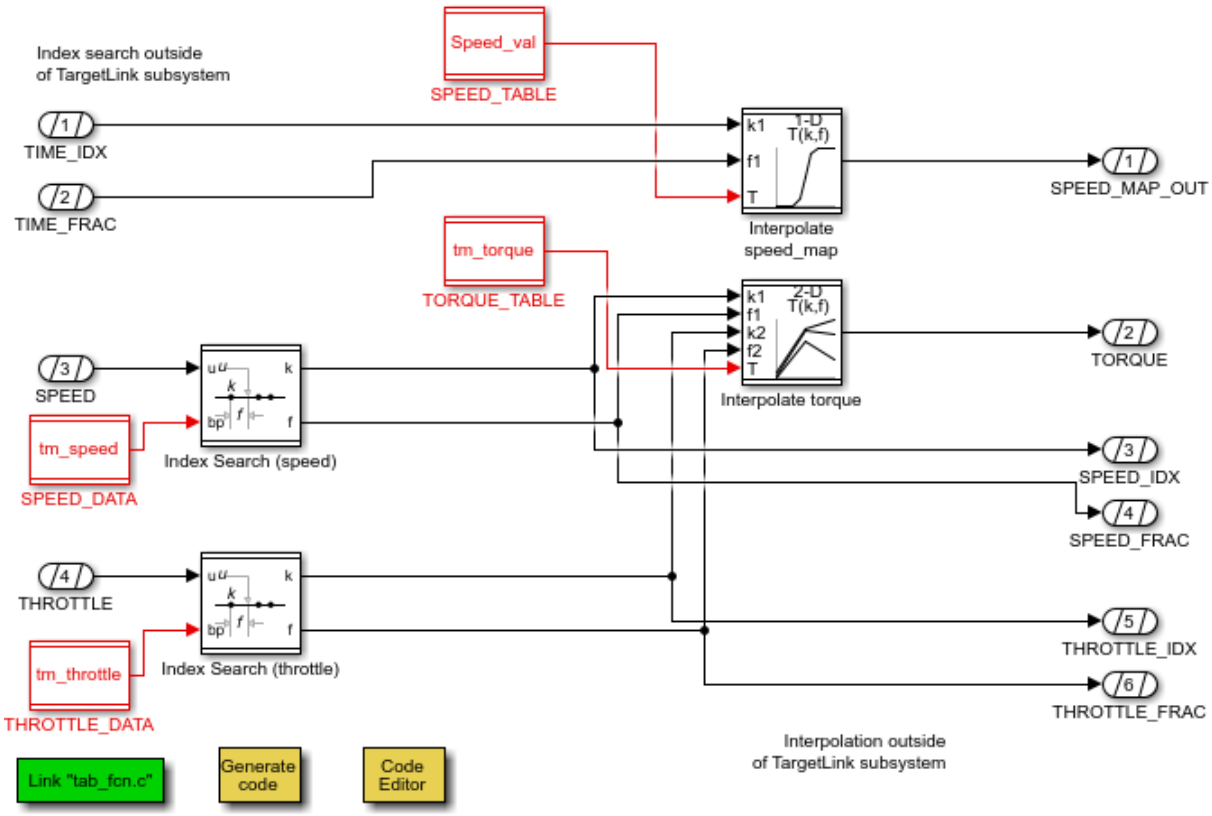
The table data called `Speed_val` is provided in the MATLAB workspace:

Workspace

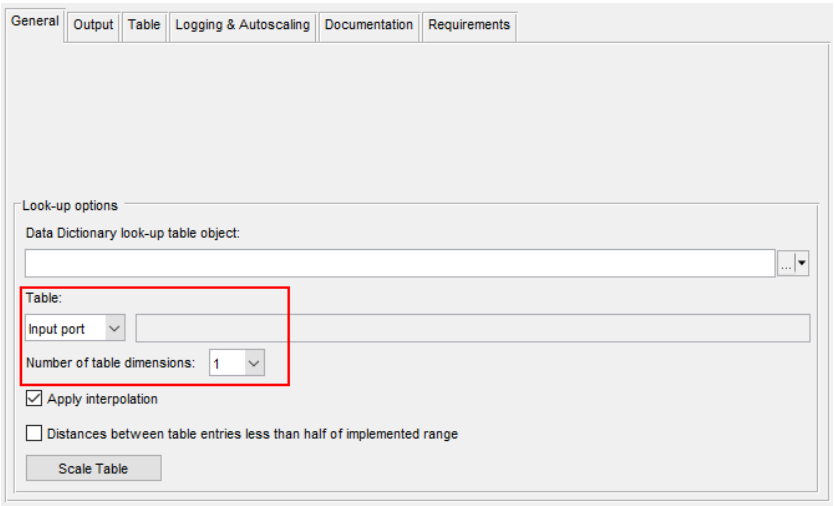
Name	Value
Speed_time	[0,10,12,35,35.5000,62,62.5000,65]
Speed_val	[800,800,2000,4000,2500,4500,900,800]
tEnd	65
Throttle_angle	[15,15,40,40,82,82,11,11]
Throttle_time	[0,10,12,35,35.5000,62,62.5000,65]
tm_speed	*1x14 double*
tm_throttle	*1x11 double*
tm_torque	*14x11 double*
tSample	0.0500
tSim	0.0500
tSim_pre_usr	0.1500

Specifying table data via table input ports

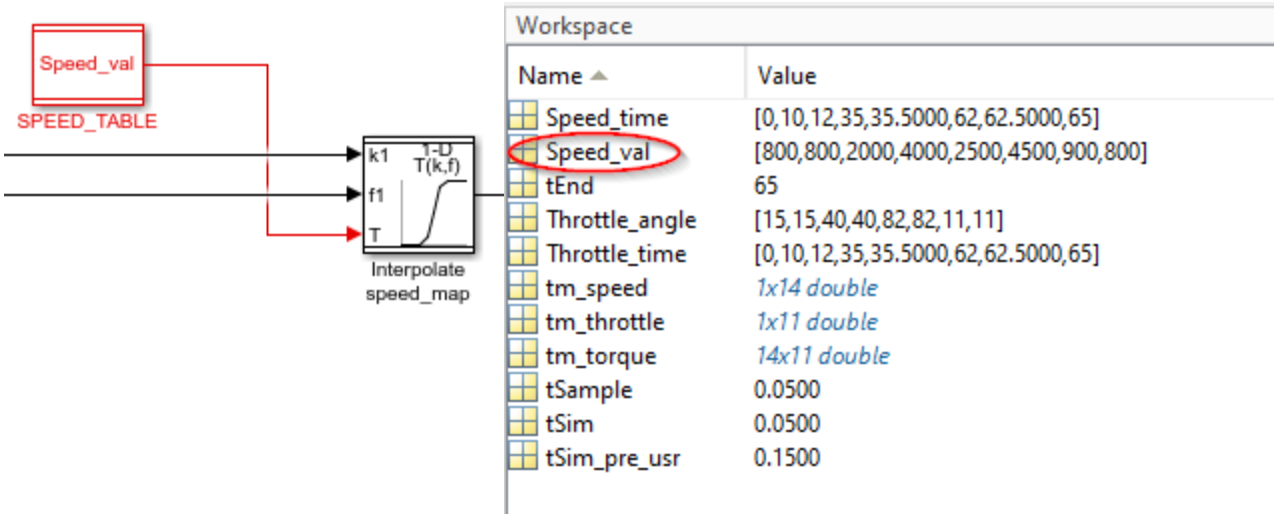
The idx_frac_vector_input TargetLink subsystem shows how to specify table data via table input ports of Prelookup and Interpolation Using Prelookup blocks (color added for clarity):



The following illustration shows the block dialog of the Interpolation Using Prelookup block called `Interpolate speed_map`:



The table data called `Speed_val` is read from the MATLAB workspace at the Constant block called `SPEED_TABLE`:



Custom look-up table functions

During code generation, TargetLink uses the custom look-up functions that are defined in the `tab_fcn` C/H files provided in the working directory of the demo model.

These files are linked via the Addfile block that is placed in each of the TargetLink subsystems:

The mapping between signals and parameters in the model and arguments and variables in the production code is provided via the following scripts:

TargetLink Block in Model	M Script in Working Directory
Interpolation Using Prelookup	`interpolation_script`
Prelookup	`indexsearch_script`

Related topics

Basics

Look-Up Table Blocks (TargetLink Preparation and Simulation Guide)
Using Custom Look-Up Functions (TargetLink Preparation and Simulation Guide)

DYNAMIC_LUT

Access

In the MATLAB Command Window, enter the following command:

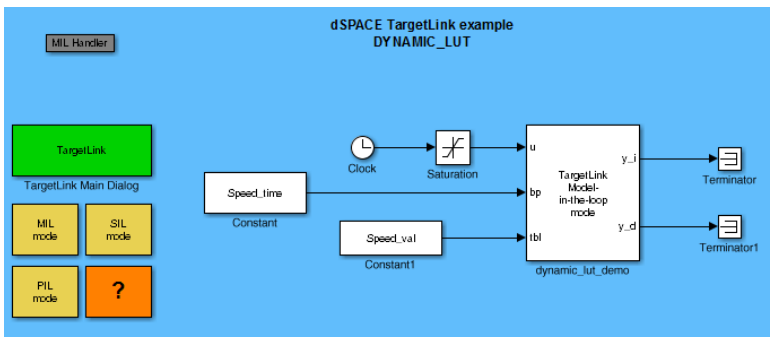
```
tl_demos dynamic_LUT
```

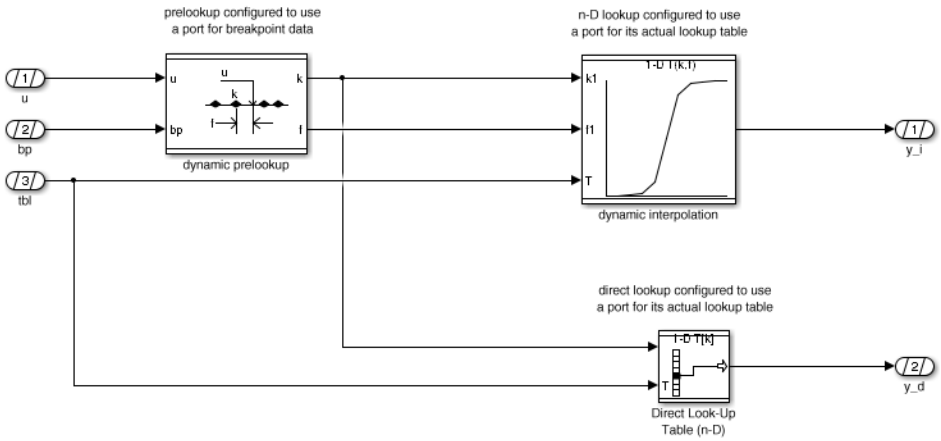
You can also enter **`tl_demos`** to access all available demos in the MATLAB Help browser. Select a demo and click **Open this example** at the top right.

Description

The `dynamic_LUT` demo model shows you how to change table data at run time.The table data is fed into the look-up table block through a table data input port. The model consists of a single top-level TargetLink subsystem. This subsystem contains three TargetLink look-up table blocks.

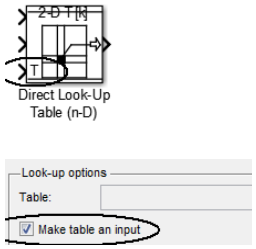
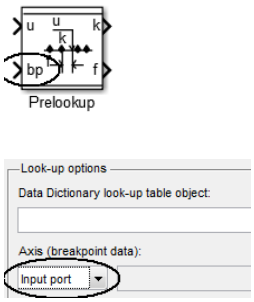
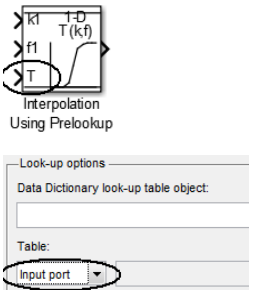
The illustration below shows the root level of the demo model.





Changing table data at run time

You can specify the table data not only in the block dialog but you can also feed the table data into the block through the block's optional table data input port. The latter allows you to change the table data during simulation and runtime of the generated code respectively. Activate the block's table data input port for the three block types as follows:

Direct Look-Up Table (n-D)	Prelookup	Interpolation Using Prelookup
		

Related topics

Basics

Basics on Look-up Tables (📖 TargetLink Preparation and Simulation Guide)

HowTos

How to Prepare Dynamic Look-Up Table Specification (📖 TargetLink Preparation and Simulation Guide)

AUTOSAR Software Components

Where to go from here

Information in this section

AR_POSCONTROL..... 119
DCMC..... 128
AUTOSAR_FUELSYS..... 130
AR_NVDATA_TRANSFORMER..... 135
AR_COLLISION_DETECTION..... 139
AR_MEMORY_MAPPING..... 147
AR_ARRAY_OF_STRUCT_DATA..... 152
AR_INCREMENTAL_CODEGEN..... 157
AAR_COMMUNICATION..... 165

AR_POSCONTROL

Access

In the MATLAB Command Window, enter the following command:

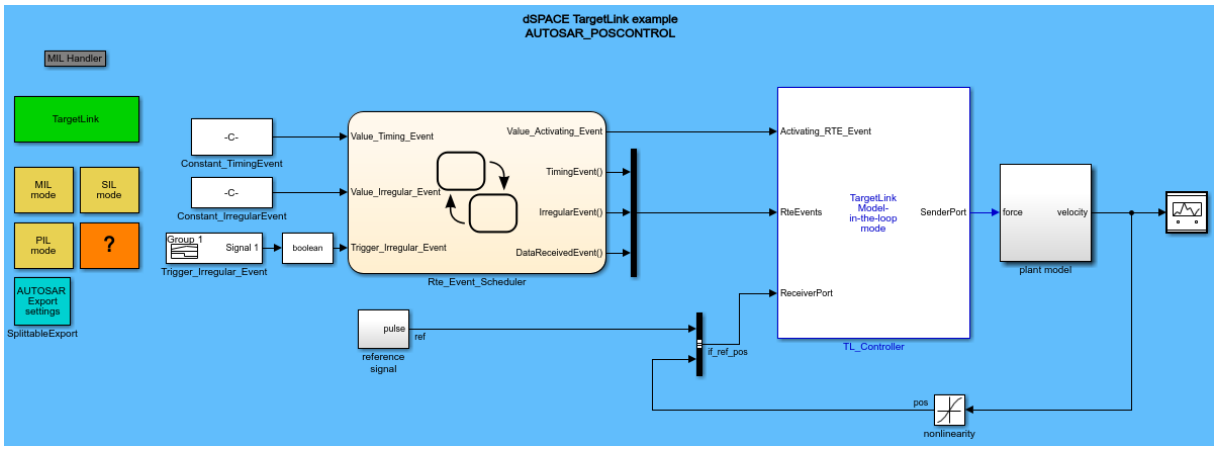
```
tl_demos ar_poscontrol
```

You can also enter **`tl_demos`** to access all available demos in the MATLAB Help browser. Select a demo and click **Open this example** at the top right.

Description

The `ar_poscontrol` demo model shows you a PI controller with one software component (SWC) and two runnables and is prepared for multiple instantiation.

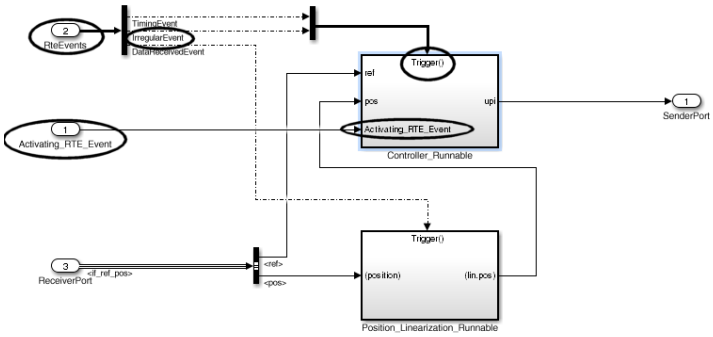
The following illustration shows the root level of the AUTOSAR poscontrol demo model.



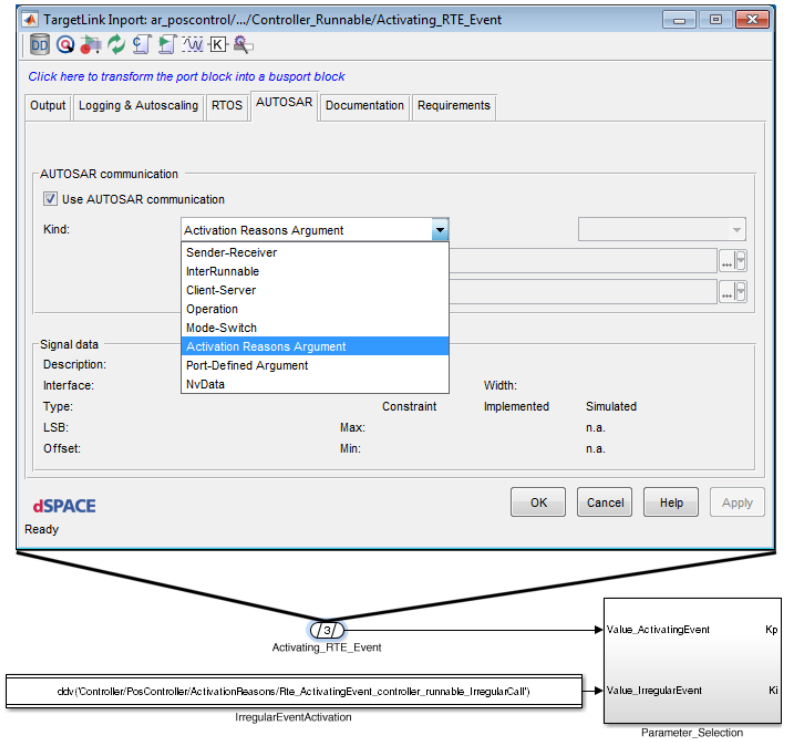
The TL_Controller subsystem contains the Controller SWC, which models the PI controller in the AUTOSAR poscontrol demo.

Activation reasons triggered by RTE events

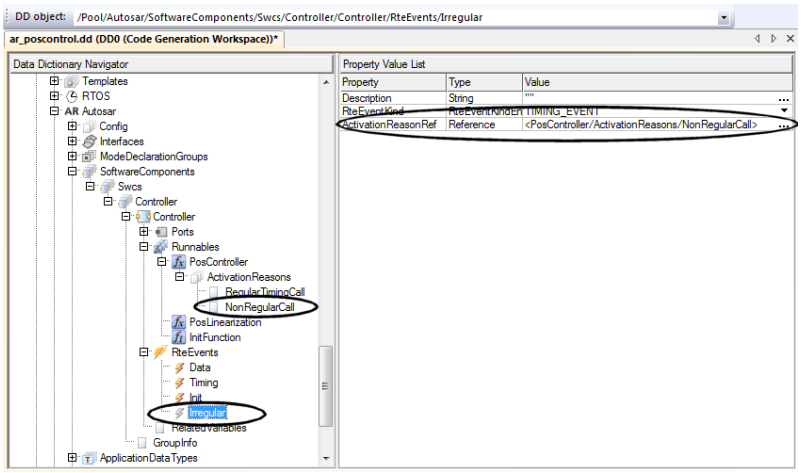
In this demo, a Stateflow chart generates various RTE events that trigger runnables. If the IrregularEvent RTE event triggers Controller_Runnable, the runnable gets the associated activating reason from the RTE. This means that the activating reason is submitted each time an irregular event occurs.



The following screenshot shows how to insert additional inports. The setting determines that an Activation Reasons Argument is read.



For an overview of the specification in the Data Dictionary, refer to the following screenshot:



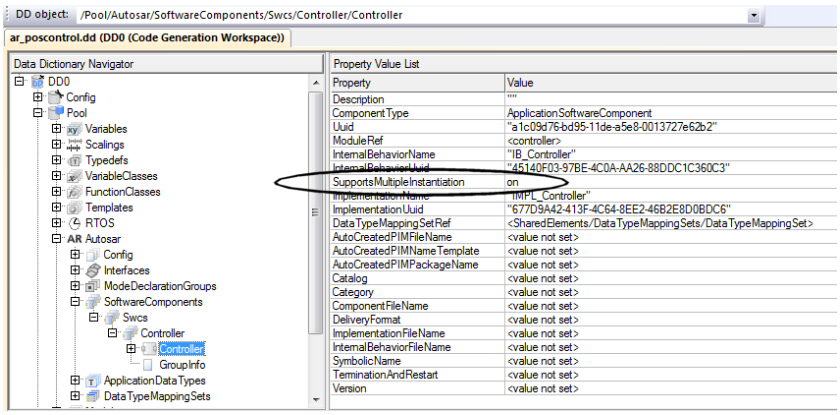
The ActivationReason object is located below the Runnables object. The reference of the ActivationReason object is shown in the RteEvent object (Irregular).

From the context menu of the **Runnables** objects, select **Synchronize Activation Reason Settings** to generate DD macro variables for the connection between the symbolic name of the **Activation Reasons** and the bit positions that are located there.

Develop SWC's for multiple instantiation

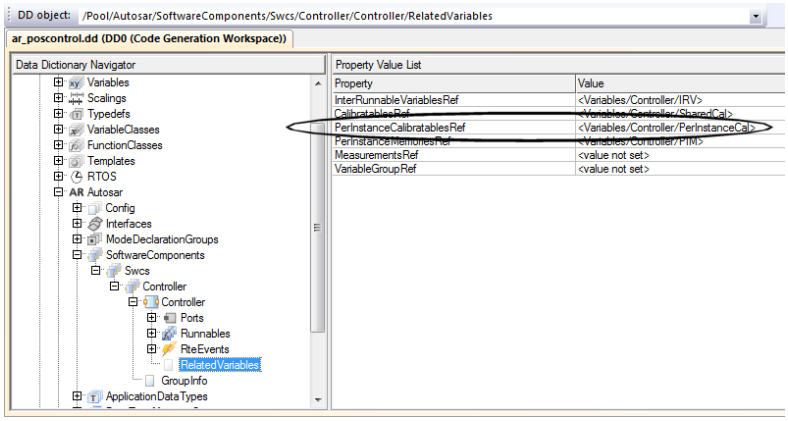
You can save ECU resources by reusing code when instantiating an SWC multiple times.

You instruct TargetLink to prepare an SWC type for multiple instantiation by setting the DD **SoftwareComponent** object's **SupportsMultipleInstantiation** property to **on**. During code generation, TargetLink performs all the further required steps.

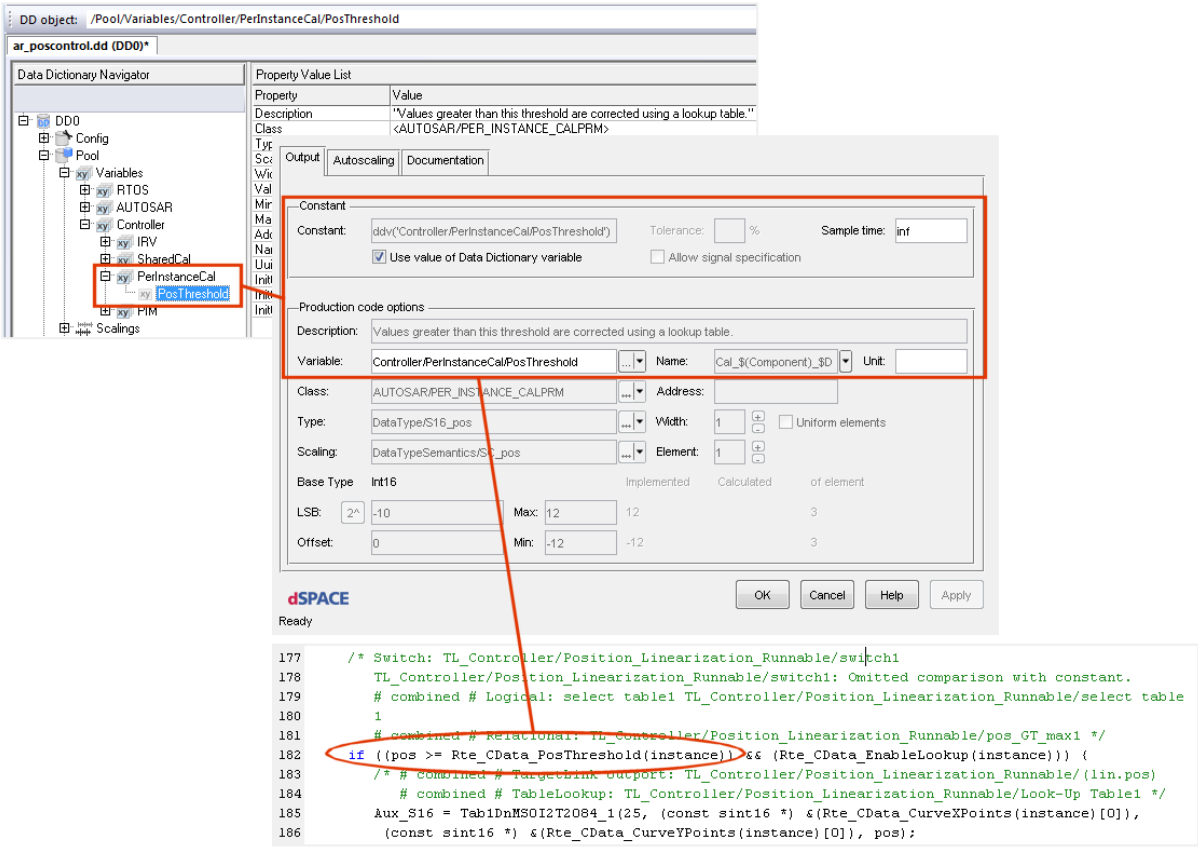


Instance specific calibration parameters TargetLink can create instance specific calibration parameters during code generation. The import and export of instance specific calibration parameters from/to AUTOSAR files is also supported.

You have to explicitly specify how to implement calibration parameters via the corresponding DD **Variable** object's `Class` property. The DD Variable object must be referenced at the DD SoftwareComponent object's **CalibratablesRef/PerInstanceCalibratablesRef** property.

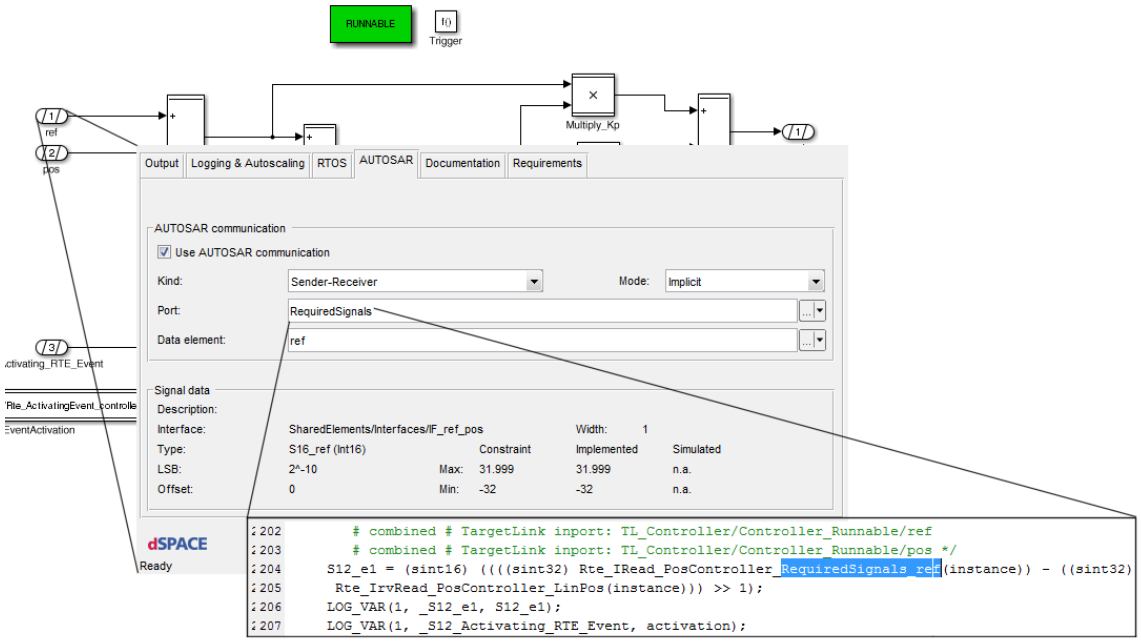


In this demo an instance-specific calibration parameter is modeled:



Modeling sender-receiver communication

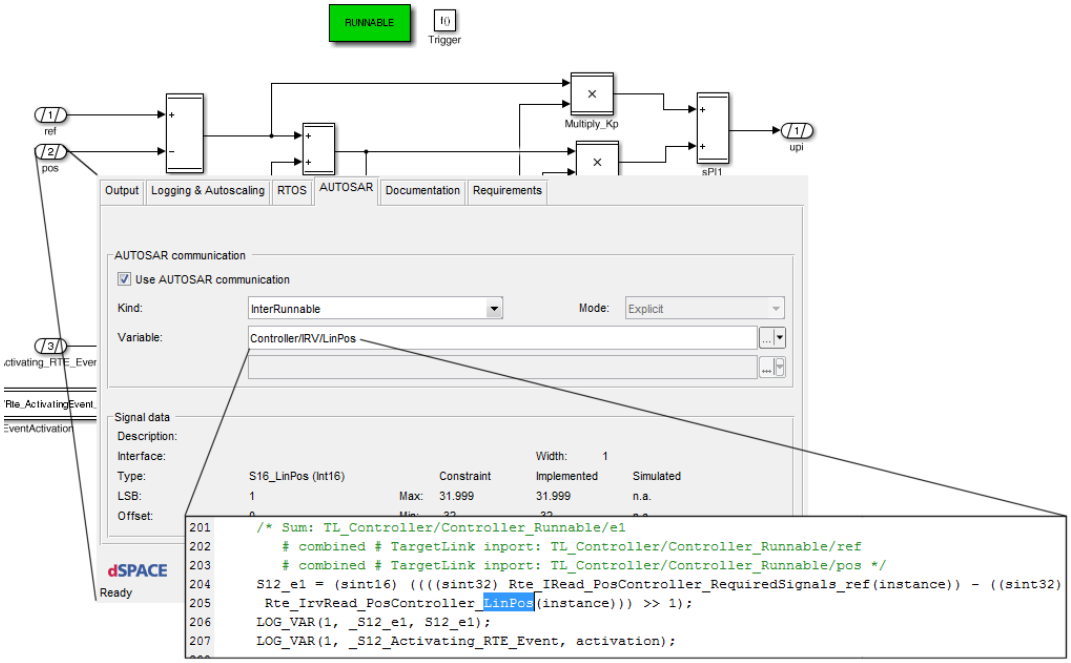
Sender-receiver communication transfers data elements between software components via ports and interfaces according to AUTOSAR. The illustration below shows an example of modeling sender-receiver communication in TargetLink.



The (ref) enhanced InPort block in the `TL_Controller/Controller_Runnable` subsystem is configured to model sender-receiver communication. Predefined DD objects for the port and data element are referenced in the InPort block dialog. The generated code for the software component contains RTE API functions for sender-receiver communication as defined by AUTOSAR.

Modeling interrunnable communication

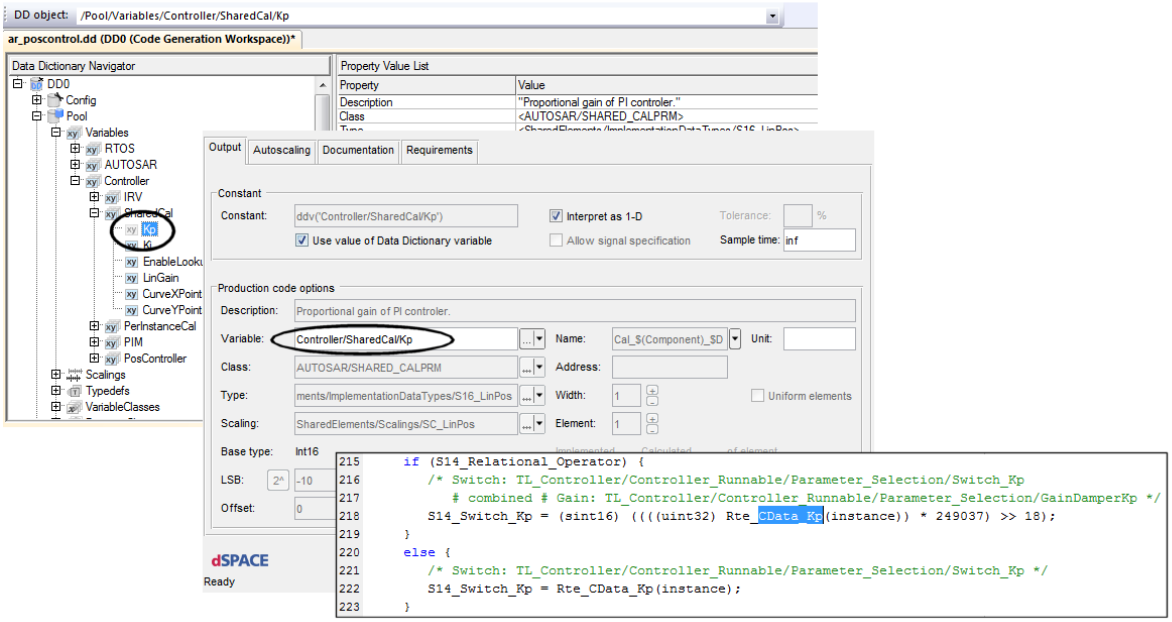
Interrunnable communication transfers the value of scalar variables between runnables of one software component according to AUTOSAR. The illustration below shows an example of modeling interrunnable communication in TargetLink.



The (pos) enhanced InPort block in the `TL_Controller/Controller_Runnable` subsystem is configured to model interrunnable communication. A predefined DD object for the interrunnable variable is referenced in the InPort block dialog. The generated code for the software component contains RTE API functions for interrunnable communication as defined by AUTOSAR.

Preparing SWC-internal parameters for calibration

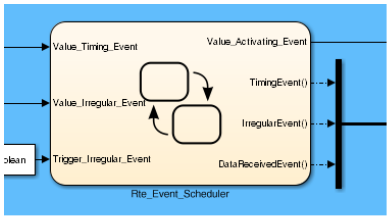
Parameters of a software component can be prepared for calibration and measurement. The illustration below shows an example of preparing a block parameter for calibration and measurement in TargetLink.

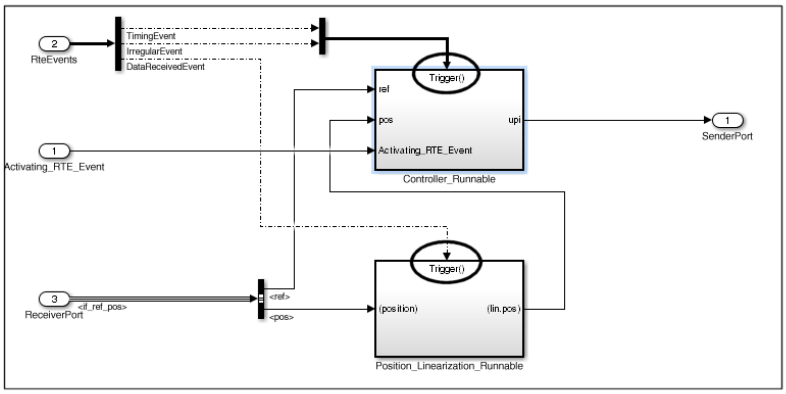


The `Kp` calibration parameter is defined as a DD object. A `Gain` block in the `TL_Controller/Controller_Runnable` subsystem references the calibration parameter from the Data Dictionary as a block parameter. The generated code of the software component contains RTE API functions for calibration and measurement systems to access the parameter.

Simulating runnable triggering

According to AUTOSAR, runnables are triggered by the RTE via RTE events. However, no RTE is present in simulations of software components with TargetLink. You have to make sure that runnable subsystems are executed, either by the data flow or via triggered subsystems. The illustration below shows an example that uses triggered subsystems and Stateflow charts for this purpose.





The `Rte_Event_Scheduler` Stateflow chart creates a sequence of timing and data received RTE events. The events are connected with the `TL_Controller` TargetLink subsystem to trigger the runnable subsystems of the `Controller` software component in simulations.

Exporting splittable elements and constant specification mapping sets

This demo model shows how to export splittable elements and constant specification mapping sets to separate AUTOSAR (ARXML) files. You can create the necessary specifications in the Data Dictionary by running the `tl_set_arexport_splittable.m` script in the demo models working directory or by double-clicking the block at the root-level of the model.

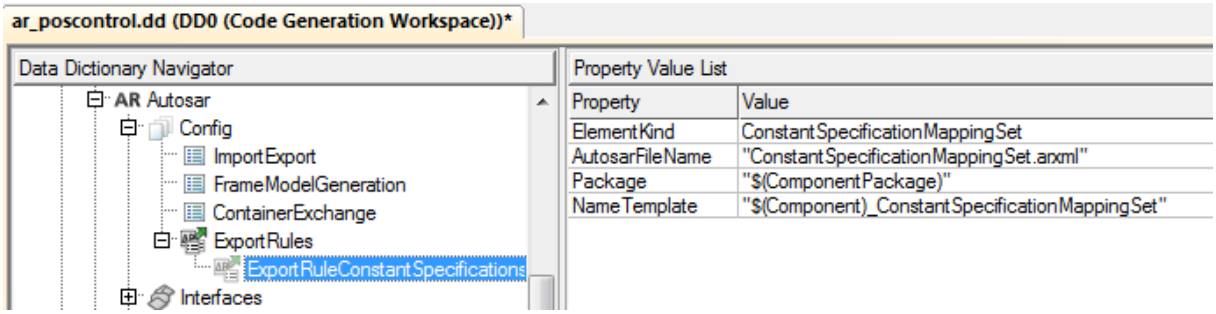
Splittable elements The AUTOSAR standard describes several elements whose description can be distributed over several AUTOSAR (ARXML) files. For example, this lets you separate the description of internal-behavior-specific variables from the description of the internal behavior itself.

This demo model shows how to distribute the following elements across several AUTOSAR (ARXML) files:

Element	AUTOSAR File	Specified At
`<EXPLICIT-INTER-RUNNABLE-VARIABLES>`	`ExplicitInterRunnableVariables.arxml`	DD ExplicitInterRunnableVariablesFileName property
`<SHARED-PARAMETERS>`	`SharedParameters.arxml`	DD SharedParametersFileName property
`<PER-INSTANCE-PARAMETERS>`	`PerInstanceParameters.arxml`	DD PerInstanceParametersFileName property
`<AR-TYPED-PER-INSTANCE-MEMORYS>`	`ArTypedPerInstanceMemories.arxml`	DD ArTypedPerInstanceMemoriesFileName property

Constant specification mapping sets By default, TargetLink implicitly generates constant specification mapping sets into the same AUTOSAR (ARXML) file as the software component description that uses constant specifications which are mapped by the set.

By specifying a DD **ExportRule** object, you can instruct TargetLink to generate constant specification mapping sets into a separate AUTOSAR (ARXML) file:



Related topics

Basics

Basics on Exporting Splittable Elements (📖 TargetLink Interoperation and Exchange Guide)
Basics on Interrunnable Communication (📖 TargetLink Classic AUTOSAR Modeling Guide)
Basics on Preparing SWCs for Measurement and Calibration (📖 TargetLink Classic AUTOSAR Modeling Guide)
Basics on RTE Events (📖 TargetLink Classic AUTOSAR Modeling Guide)
Basics on Sender-Receiver Communication (📖 TargetLink Classic AUTOSAR Modeling Guide)
Modeling Runnables (📖 TargetLink Classic AUTOSAR Modeling Guide)

HowTos

How to Model Per Instance Parameters for Calibration (📖 TargetLink Classic AUTOSAR Modeling Guide)
How to Model Shared Parameters for Calibration (📖 TargetLink Classic AUTOSAR Modeling Guide)

DCMC

Access

In the MATLAB Command Window, enter the following command:

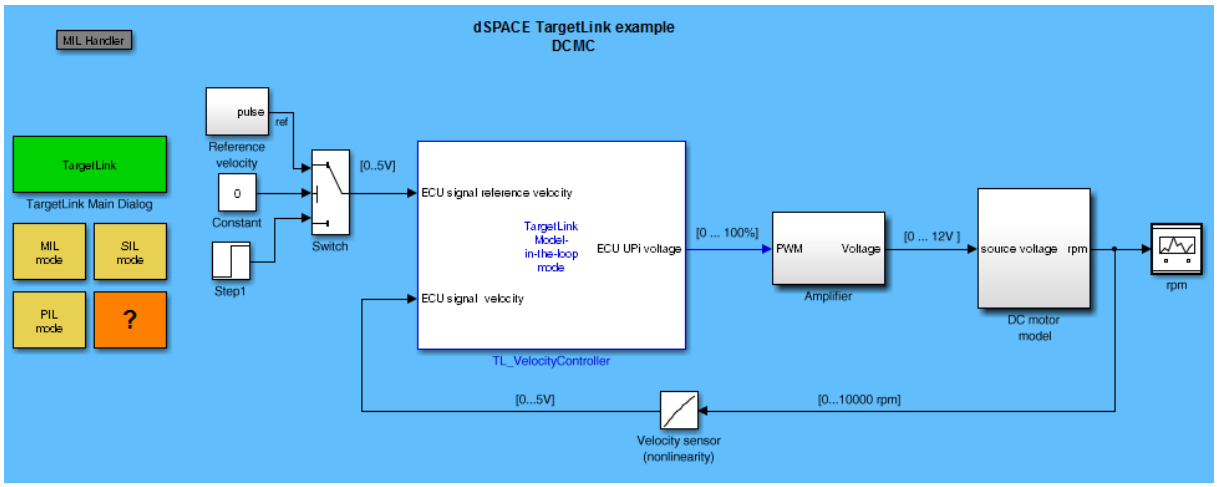
```
tl_demos dcmc
```

You can also enter **`tl_demos`** to access all available demos in the MATLAB Help browser. Select a demo and click **Open this example** at the top right.

Description

The `DCMC` demo model shows a motor controller with four software components that each contain one runnable.

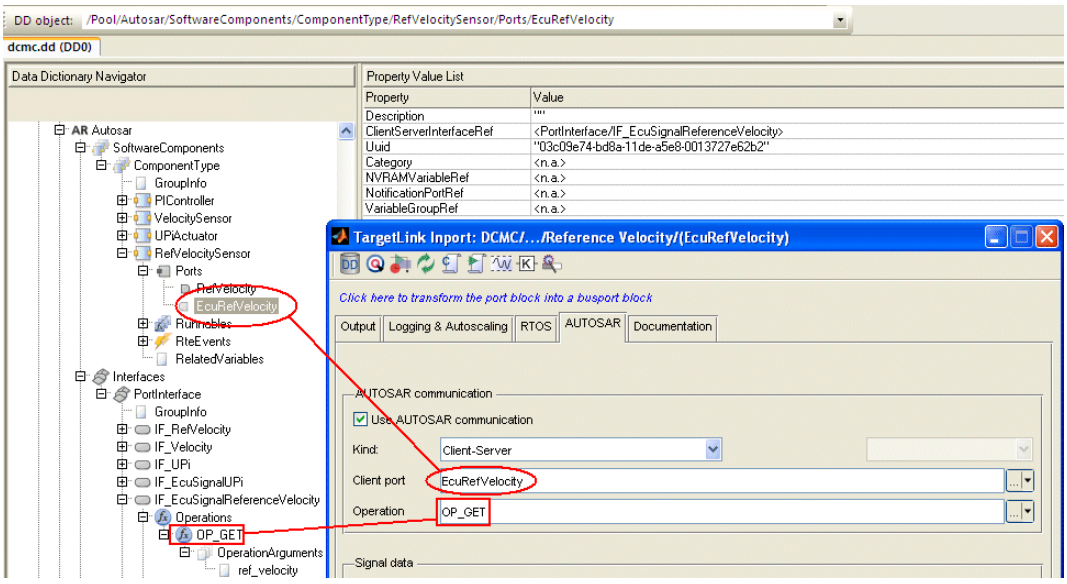
The illustration below shows the TargetLink subsystem of the demo.



The central PI_Controller subsystem implements a controller software component that processes sensor signals to produce an actuator signal. The Reference Velocity and the Velocity Sensor subsystems implement software components that get a sensor signal from the I/O hardware abstraction. The Force Actuator subsystem implements a software component that sets an actuator signal in the I/O hardware abstraction.

Modeling client-server communication using get operations

Client-server communication using get operations allows the reading of data that is provided by an operation. The illustration below shows an example of reading a sensor.

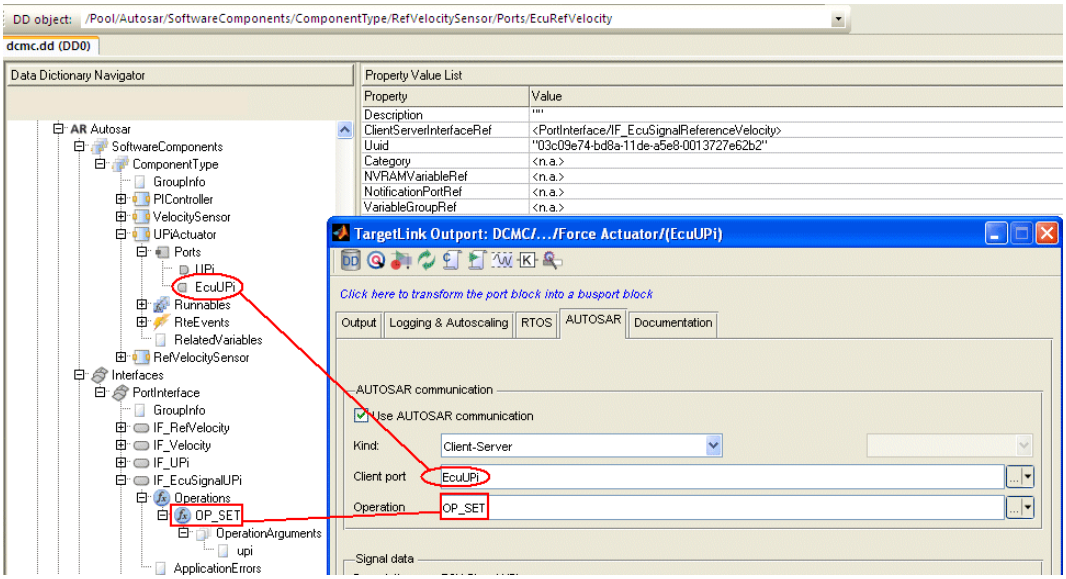


The `RefSensorVelocity` runnable reads the `ref_velocity` operation argument and processes it. The runnable is modeled in the `Reference Velocity` subsystem. It contains a TargetLink **InPort** block that implements the

operation call of the `OP_GET` operation to read the `ref_velocity` ARGOUT argument.

Modeling client-server communication using set operations

Client-server communication using set operations allows the writing of data that is required by an operation. The illustration below shows an example of writing to an actuator.



The `ActuatorUPI` runnable writes the `upi` operation argument. The runnable is modeled in the `Force Actuator` subsystem. It contains a TargetLink OutPort block that implements the operation call of the `OP_SET` operation to write the `upi` ARGIN argument.

Related topics

Basics

Basics on Client-Server Communication (📖 TargetLink Classic AUTOSAR Modeling Guide)

AUTOSAR_FUELSYS

Access

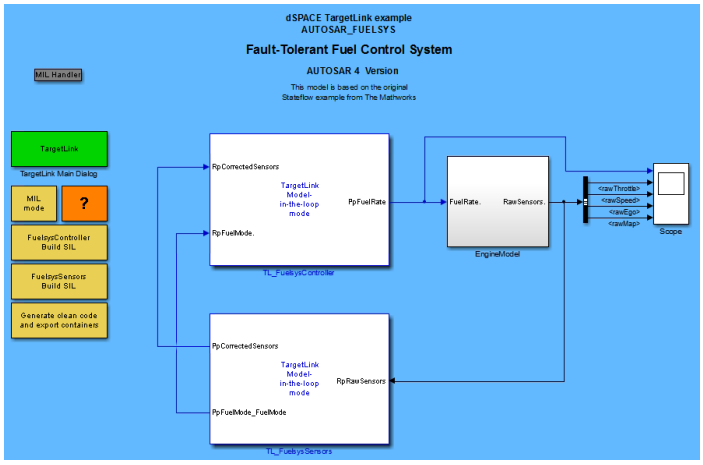
In the MATLAB Command Window, enter the following command:

```
tl_demos ar_fuelsys
```

You can also enter `tl_demos` to access all available demos in the MATLAB Help browser. Select a demo and click Open this example at the top right.

Description

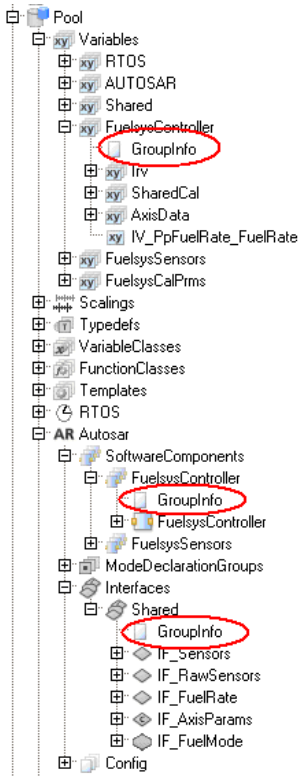
The *AUTOSAR fuelsys* demo implements a fault-tolerant fuel control system with two software components. The illustration below shows the two TargetLink subsystems `TL_FuelsysSensors` and `TL_FuelsysController`, each of which contains a software component.



The `autosar_fuelsys` demo model shows you a fault-tolerant fuel control system with two software components. The `TL_FuelsysSensors` subsystem models one software component with two runnables for processing sensor data. The processed sensor data and a calculated fuel mode are transferred to the `TL_FuelsysController` subsystem, which models a controller software component with four runnables.

Working with packages

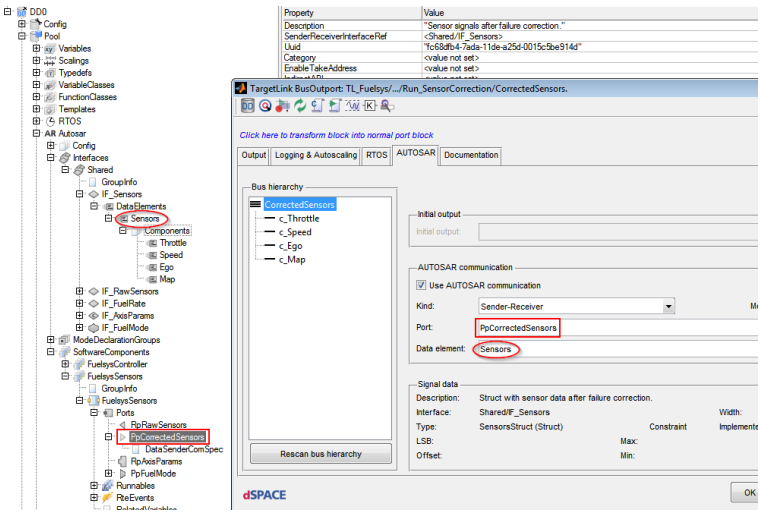
Package information lets you import AUTOSAR files to predefined locations in the DD and export DD objects to structured AUTOSAR files. The illustration below shows how packages are modeled in the Data Dictionary.



Package information is specified at the Package Property of Groupinfo objects. The above illustration shows Groupinfo objects with package information for the variables, interfaces, and the internal behavior of the `FuelsysController` software component.

Modeling sender-receiver communication with struct-based data elements

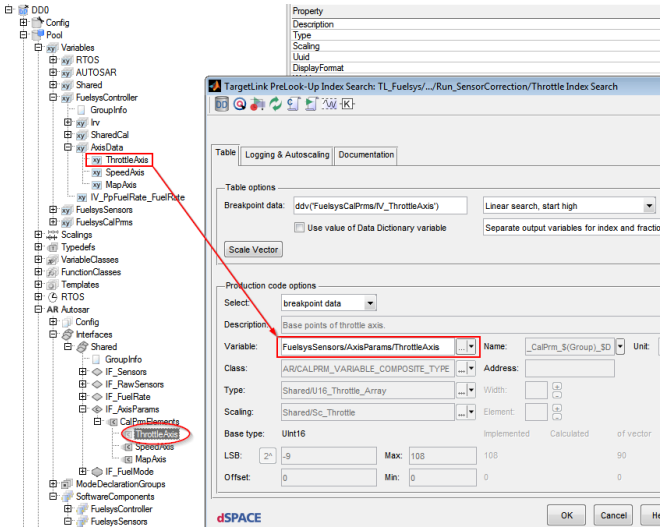
Sender-receiver communication with struct-based data elements lets you transfer structured data in one operation. The illustration below shows an example of modeling sender-receiver communication with struct-based data elements.



The `CorrectedSensors` struct-based data element with the components `Throttle`, `Speed`, `Ego`, and `Map` is transferred from the `FuelSysSensors` to the `FuelSysController` software component via sender-receiver communication. The demo uses a Bus Outport block for modeling the sender. The block references the data element that is defined in the Data Dictionary. The generated code for the software component contains the `Rte_IWrite_SensorCorrection_PpCorrectedSensors_Sensors((const SensorsStruct_IDT *) &Sensors);` RTE API function for transferring the data element.

Preparing parameters of several SWCs for calibration

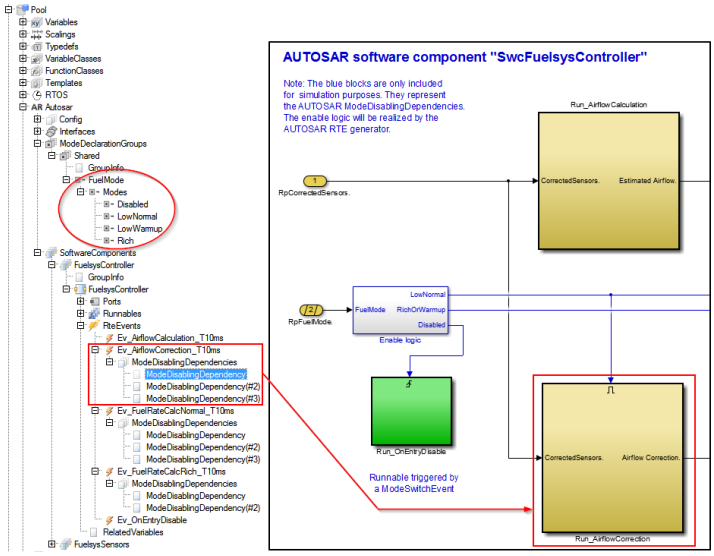
Parameters that are required by several SWCs can be prepared for calibration and measurement via ports and interfaces according to AUTOSAR. The illustration below shows an example from the demo model.



The calibration parameters `ThrottleAxis, SpeedAxis,` and `MapAxis` are transferred via the `IF_AxisParams` interface at the `RpAxisData` port. The port is not represented in the TargetLink model. The variables that represent the calibration parameters are referenced directly at the block.

Modeling mode-dependent activation of runnables

Runnables can be activated on the transition between modes or the activation of runnables can be disabled in certain modes. The illustration below shows an example of modeling the activation of runnables depending on modes.



The `Run_AirflowCorrection` runnable is triggered by the `Ev_AirflowCorrestion_T10ms` RTE event. Three mode disabling dependencies are specified for this RTE event, i.e., the runnable is triggered only in the remaining one of the four defined modes, which is the `LowNormal` mode. However, runnable triggering via RTE events is not performed in TargetLink simulations because there is no RTE. The Enable logic subsystem and using enabled subsystems for the runnables are used for this purpose in the demo model.

The demo model also shows how to generate code for the `Rte_Switch` RTE API function.

Related topics

Basics

Basics on Packages (📖 TargetLink Interoperation and Exchange Guide)
Basics on Preparing SWCs for Measurement and Calibration (📖 TargetLink Classic AUTOSAR Modeling Guide)
Basics on Sender-Receiver Communication (📖 TargetLink Classic AUTOSAR Modeling Guide)
Mode Management with SystemDesk and TargetLink (📖 TargetLink Classic AUTOSAR Modeling Guide)

HowTos

How to Model Parameter Communication for Calibration (📖 TargetLink Classic AUTOSAR Modeling Guide)

AR_NVDATA_TRANSFORMER

Access

In the MATLAB Command Window, enter the following command:

```
tl_demos ar_nvdata_transformer
```

You can also enter **`tl_demos`** to access all available demos in the MATLAB Help browser. Select a demo and click **Open this example** at the top right.

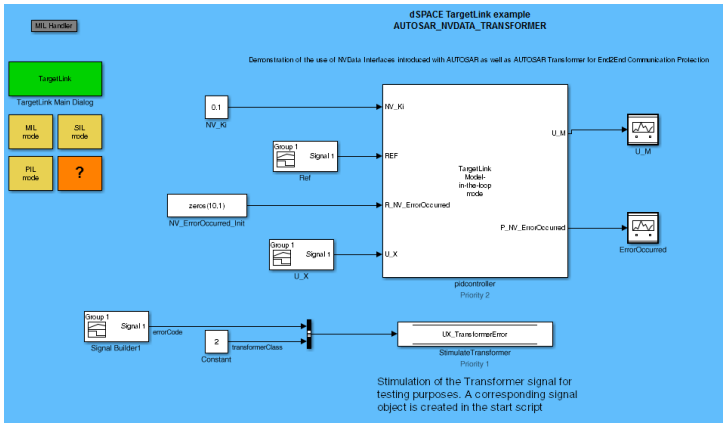
Description

The `ar_nvdata_transformer` demo model shows you NVData communication and transformation error logic.

Refer to the following features:

- Read and Write access to nonvolatile RAM using NVData interfaces
- Transformer error handling by using **ReceiverComSpec** or **SenderComSpec blocks**.

The following illustration shows the root level of the AUTOSAR_NVDATA_TRANSFORMER demo model:



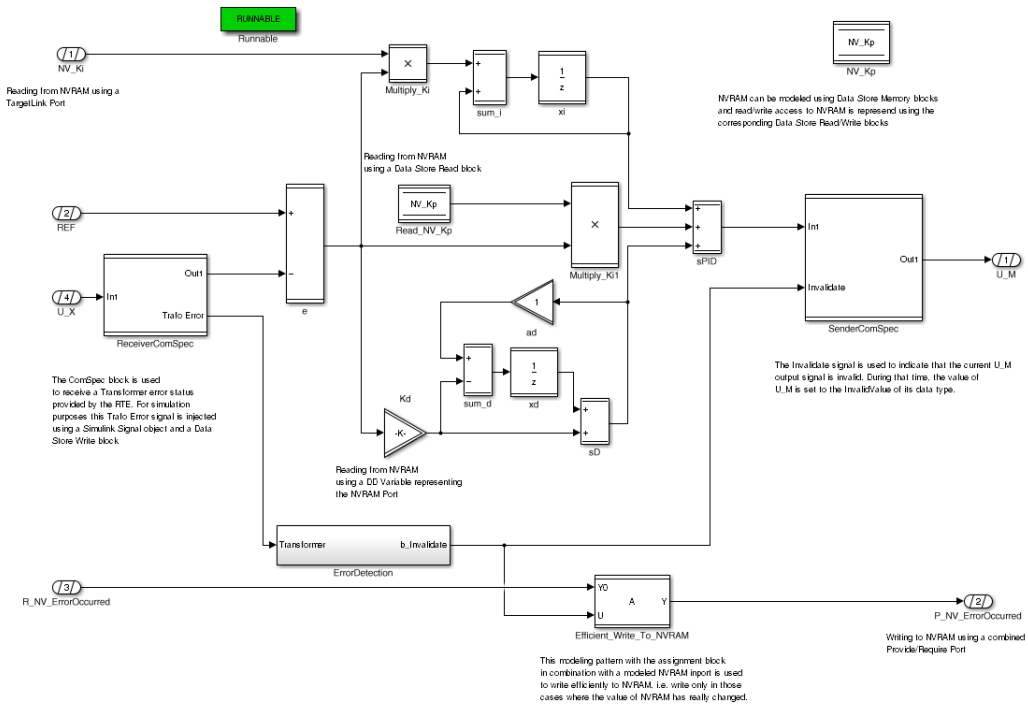
NVData Interfaces

TargetLink supports NvData communication and lets you specify NvData communication in the Data Dictionary.

- You can create DD **NvDataInterface** objects in the `/Pool/Autosar/Interfaces/` subtree.
- You can create **NvReceiverPort**, **NvSenderPort** and **NvSenderReceiverPort** objects in the `/Pool/Autosar/SoftwareComponents/<SoftwareComponent>/Ports/` subtree.
- You can create DD **NvDataElement** objects in the `/Pool/Autosar/Interfaces/<NvDataInterface>/NvDataElements/` subtree. TargetLink lets you create NvData elements with a simple structure or with a composite structure.

TargetLink offers various approaches for modeling NvData communication. The following approaches are used in this demo model:

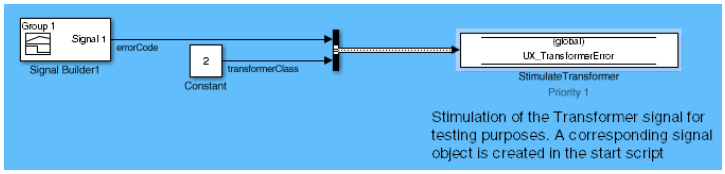
- An **InPort** block reads from NVRAM.
- A **Data Store Memory** block is used to model NVRAM. The associated **Data Store Read** and **Data Store Write** blocks are used to read and write to NVRAM.
- A **Gain** block is used to read a DD Variable object representing a read access to NVRAM.
- An **Assignment** block in combination with a combined buffer for read accesses and write accesses (combined Provide/Require port). It is used to write efficiently to NVRAM, e.g., to write only when the value of NVRAM really differs.



AUTOSAR transformer error handling in sender-receiver communication

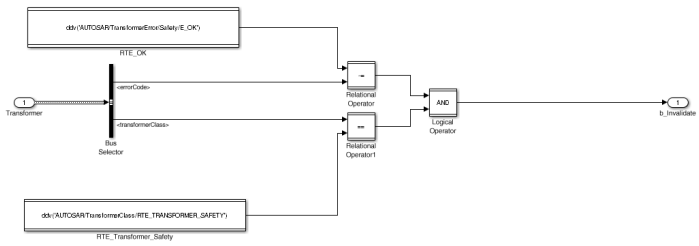
Because data transformation belongs to the RTE, it is out of scope for TargetLink. However, you can model transformer error logic with TargetLink and generate production code that reacts to transformer errors. TargetLink provides the DD **ErrorHandling** property at DD <Port> objects, which lets you control whether the generated code contains transformer errors.

This part of the demo shows you how to use AUTOSAR transformers to implement end-to-end communication protection in which an AUTOSAR transformer requests the status of a protected signal. The transformer error is simulated by using a Simulink Signal object and a Data Store Write block as shown in the following screenshot:

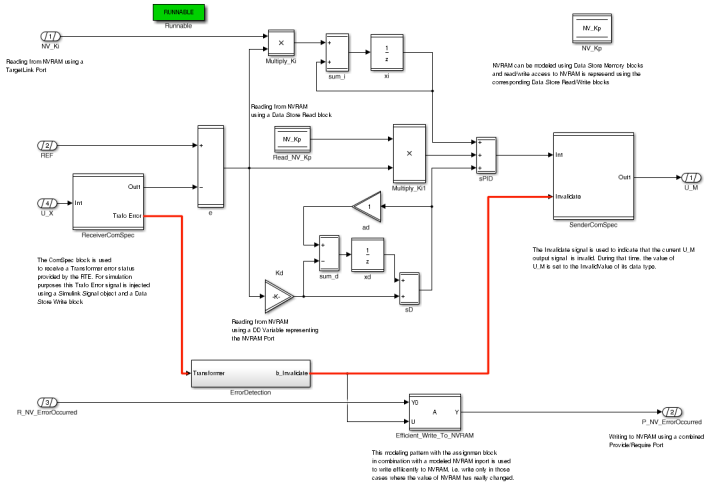


A look in the **pidcontroller** subsystem shows the error handling of the injected signal:

- A **ReceiverComSpec** block receives the transformer error status provided by the RTE.
- The error detection is handled in the **ErrorDetection** subsystem.



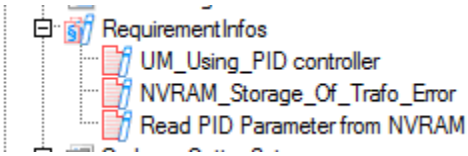
- If an error occurs, an invalidation value is fed to the **SenderComSpec** block, which invalidates the signal. This means that a specific invalid value is sent instead of the signal's original value.



Requirement information in the Data Dictionary

The Data Dictionary provides the `/Pool/RequirementInfos` subtree to store requirement information.

This demo provides the following DD **RequirementInfo** objects that act as a proxy for the requirements management system:



Because the **RequirementInfoAsCodeComment** Code Generator option is set to **On**, the production code generated for this demo containes requirement information such as the following:

```
/* TargetLink outport: pidcontroller/P_NV_ErrorOccurred
      # combined # Assignment:
pidcontroller/Efficient_Write_To_NVRAM - output calculation
      # combined # TargetLink inport:
pidcontroller/R_NV_ErrorOccurred

      Requirements:
      pidcontroller/P_NV_ErrorOccurred
      Description: Received End2End Communication Erorrs via the
Transformer must be logged in NVRAM
      Document    : Requirements Document
      Location    : REQ 2
      User Tag    : Transformer Errors are logged using NVRAM */
```

Refer to Basics on Using DD-Based Requirement Information (📖 TargetLink Interoperation and Exchange Guide).

Related topics

Basics

Basics on Reducing Rewrites to NVRAM (📖 TargetLink Classic AUTOSAR Modeling Guide)
Basics on Signal Invalidation (📖 TargetLink Classic AUTOSAR Modeling Guide)
Basics on Using DD-Based Requirement Information (📖 TargetLink Interoperation and Exchange Guide)

HowTos

How to Create Ports (📖 TargetLink Classic AUTOSAR Modeling Guide)
How to Model and Simulate Transformer Error Logic in Sender-Receiver Communication (📖 TargetLink Classic AUTOSAR Modeling Guide)
Modeling NvData Communication (📖 TargetLink Classic AUTOSAR Modeling Guide)

References

RequirementInfoAsCodeComment (📖 TargetLink Model Element Reference)

AR_COLLISION_DETECTION

Access

In the MATLAB Command Window, enter the following command:

```
tl_demos ar_collision_detection
```

You can also enter **`tl_demos`** to access all available demos in the MATLAB Help browser. Select a demo and click **Open this example** at the top right.

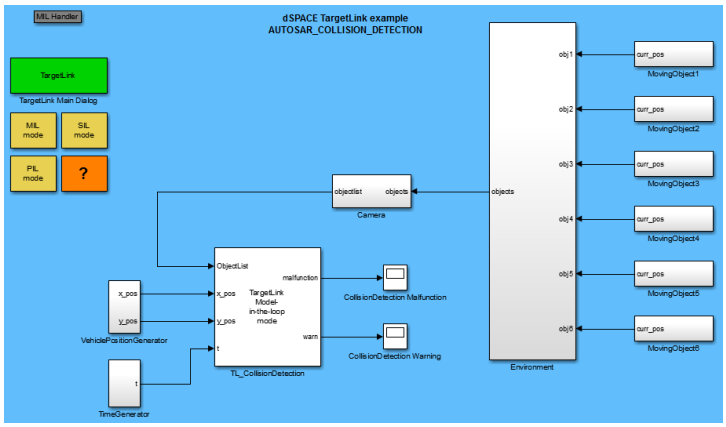
Description

The `ar_collision_detection` demo model shows you a collision detection system as it is used in cars. Tracking is implemented using asynchronous client-server communication combined with transformer error logic.

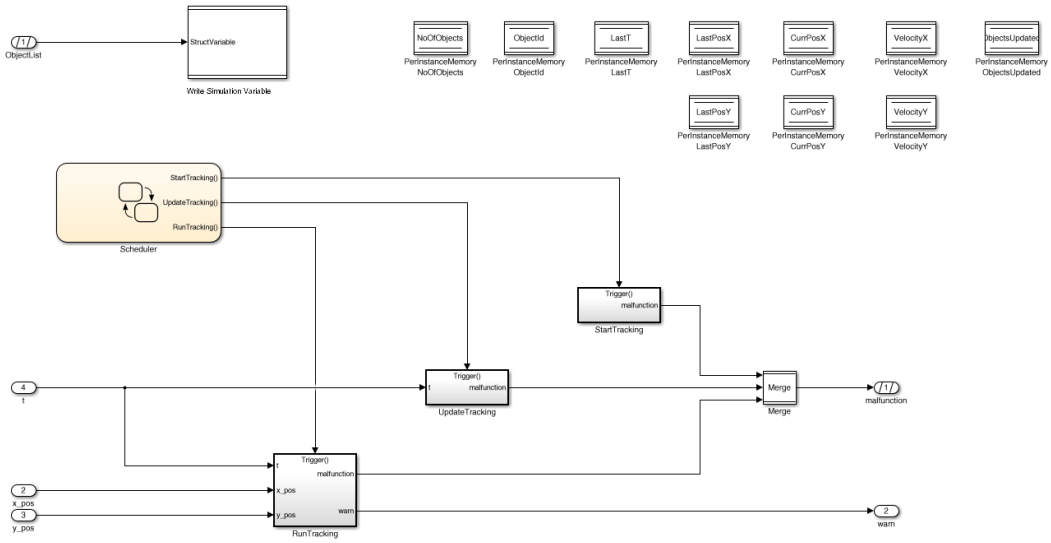
A **Camera** subsystem is used for object tracking and responds to the environment subsystem, which simulates six objects. An object list is created for the **TL_CollisionDetection** subsystem. The list must be continuously updated.

The **TL_CollisionDetection** subsystem requests an updated object list from the camera via an asynchronous call (**StartTracking** and **UpdateTracking** runnables). After the camera provides a new object list as a result of the asynchronous call, the **UpdateTracking** runnable is triggered. The runnable updates the position data of the detected objects. The actual collision detection is performed in the **RunTracking** runnable that determines the distance to the detected objects and issues a collision warning if necessary. As long as no new camera data is available, the **RunTracking** runnable operates in blind mode (**Update_CurrentPosition_for_Blind_Tracking** subsystem). This means the runnable updates the position data of the objects on the basis of the history.

The following illustration shows the root level of the `ar_collision_detection` demo.



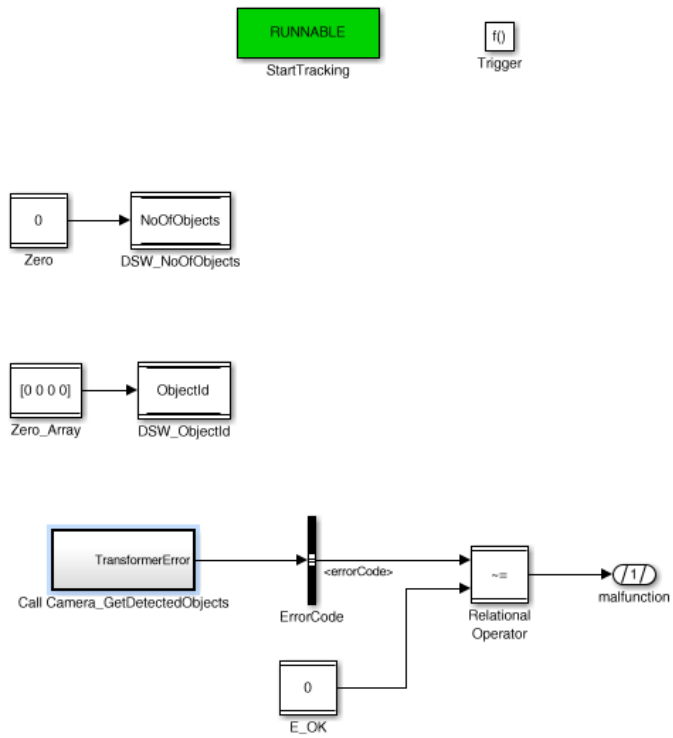
In addition, refer to the TL_CollisionDetection subsystem.



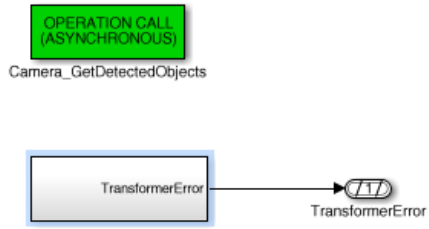
Asynchronous client-server communication

Different runnables are used to call the server operation via `Rte_Call` and retrieve the result via `Rte_Result`:

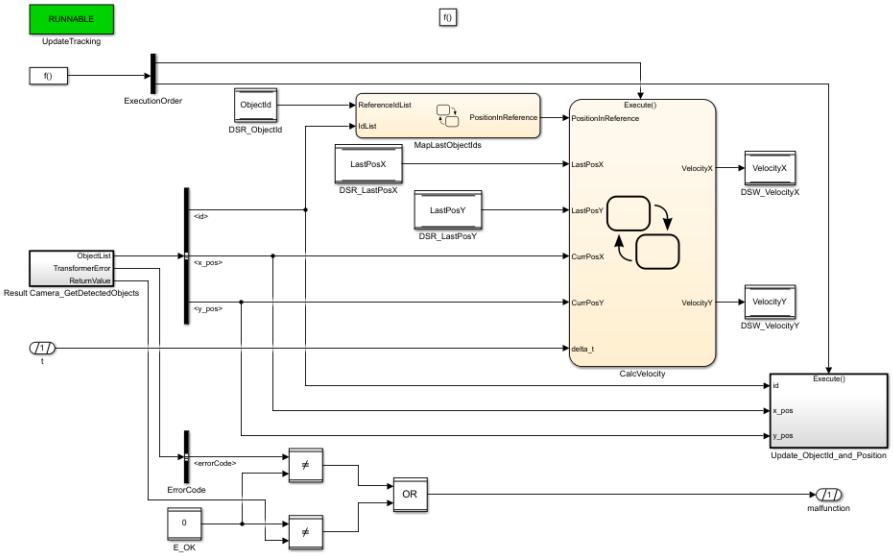
1. The StartTracking client runnable initiates the communication with a server runnable via an asynchronous call of the `Rte_Call` RTE API function. This is modeled via the asynchronous operation call subsystem called Call Camera_GetDetectedObjects:



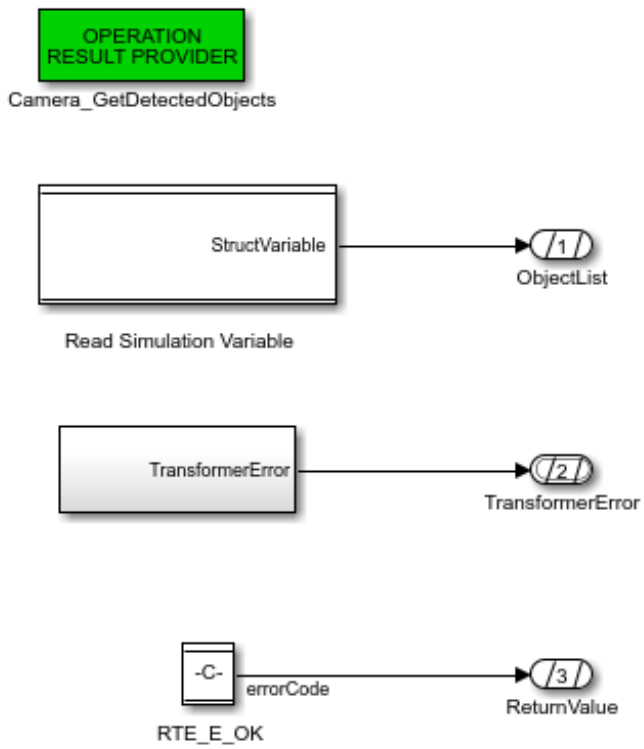
These are the contents of the asynchronous operation call subsystem:



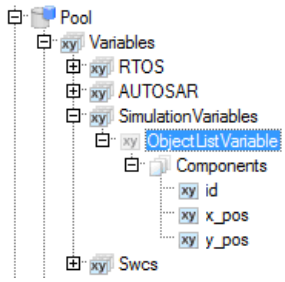
2. The **UpdateTracking** client runnable receives the result of the operation for further processing via a call of the `Rte_Result` RTE API function. This call includes the object list as a parameter. The call is modeled via the operation result provider subsystem called **Result Camera_GetDetectedObjects**:



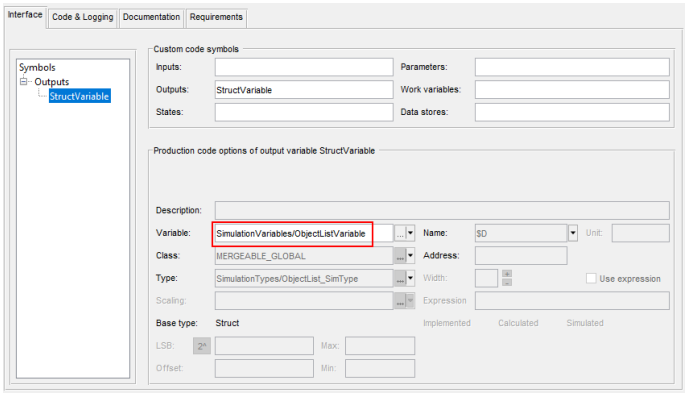
The operation result subsystem has no incoming signal line. Instead, the list of objects is read from a Simulink.Signal object via a Data Store Read (DSR) block that is masked by a **Custom Code** (type II) block.



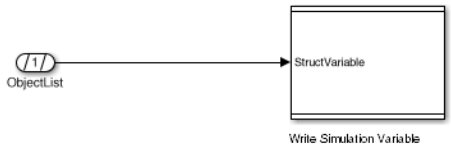
In MIL simulation mode, the Simulink.Signal object is used. For SIL and PIL simulations, a structured variable with global scope is used. This variable is specified in and generated from the Data Dictionary in DD-based code generation:



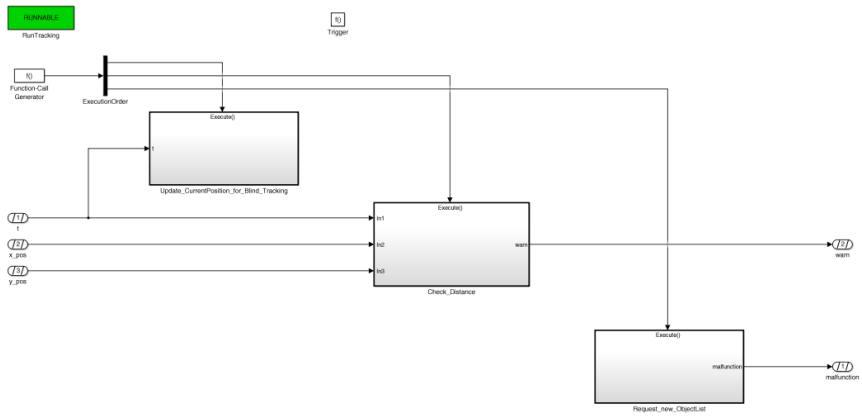
The variable is referenced at the **Custom Code** (type II) block.



On the root level of the model, the same modeling style is used. However, a Data Store Write block is used instead:



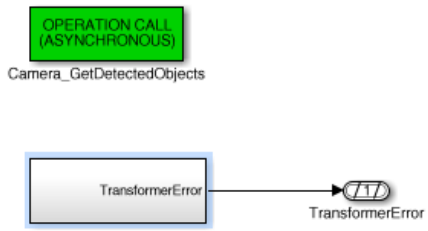
3. The **RunTracking** client runnable again initiates the communication with the server runnable via the same call of the `Rte_Call` RTE API function as the **StartTracking** runnable. This call is generated for the **Camera_GetDetectedObjects** asynchronous operation call subsystem contained in **Request_new_ObjectList**.



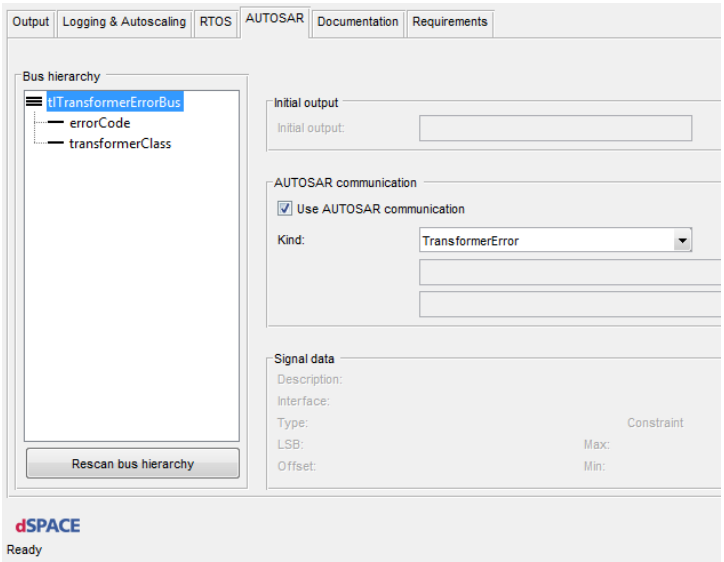
AUTOSAR transformer error handling in client-server communication

In this model, transformer error logic for client-server communication is modeled in all three runnables.

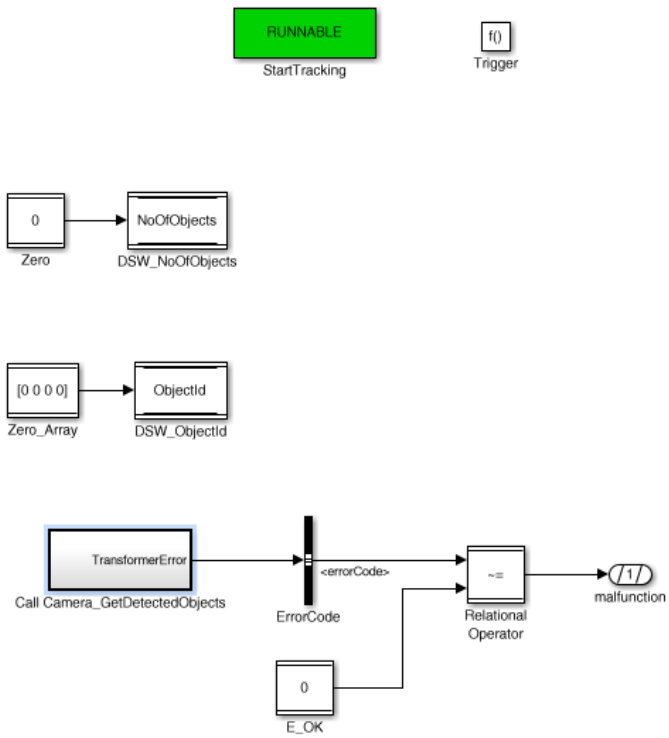
Each runnable contains an operation subsystem that contains a **Bus Outport** block modeling the transformer error. The following screenshot shows the contents of the asynchronous operation call subsystem called **Call Camera_GetDetectedObjects**:



Here, the `TransformerError` parameter is generated into the call of the `Rte_Call` RTE API function.

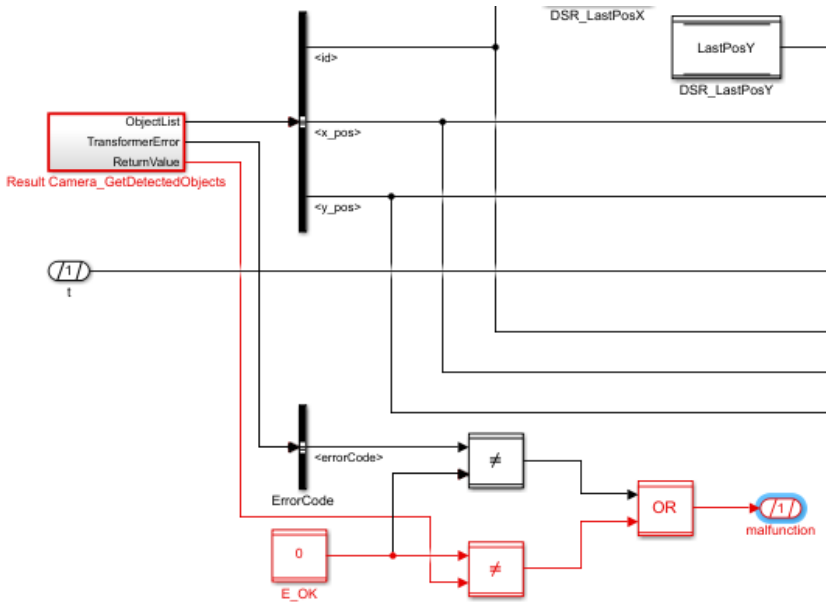


The transformer error logic is modeled on the right-hand side of the ErrorCode Bus Selector block:

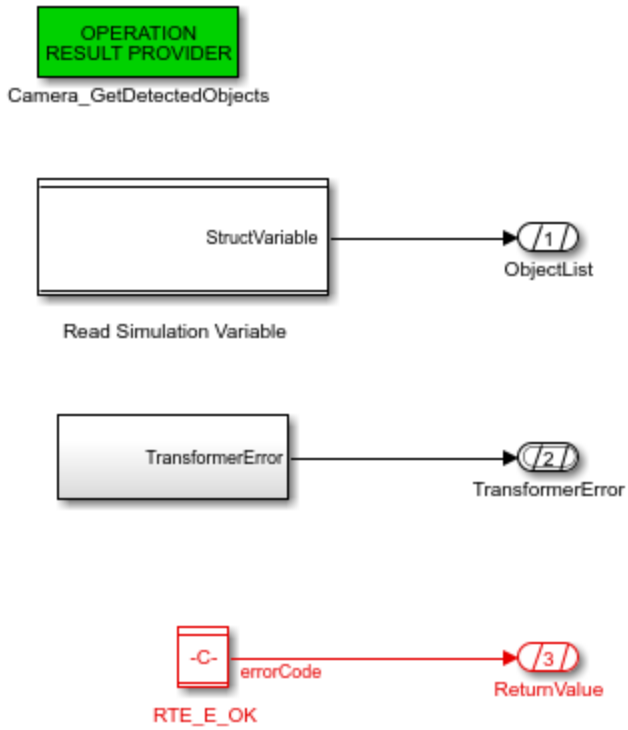


Rte_Result error handling

Error handling for fetching the operation result of an asynchronous server call via the `Rte_Result` RTE API function is modeled in the **UpdateTracking** subsystem (color added for clarity):



The subsystem uses the operation return value of the RTE API function as modeled in the **Result Camera_GetDetectedObjects** operation result provider subsystem (color added for clarity):



Related topics

Basics

Basics on Modeling Operation Calls on the Client Side of Client-Server Communication (📖 TargetLink Classic AUTOSAR Modeling Guide)

HowTos

How to Model Operation Calls via Asynchronous Operation Call Subsystems (📖 TargetLink Classic AUTOSAR Modeling Guide)
How to Model Operation Results via Operation Result Provider Subsystems (📖 TargetLink Classic AUTOSAR Modeling Guide)
How to Simulate Operation Calls in Asynchronous Client-Server Communication (📖 TargetLink Classic AUTOSAR Modeling Guide)

AR_MEMORY_MAPPING

Access

In the MATLAB Command Window, enter the following command:

```
tl_demos ar_memory_mapping
```

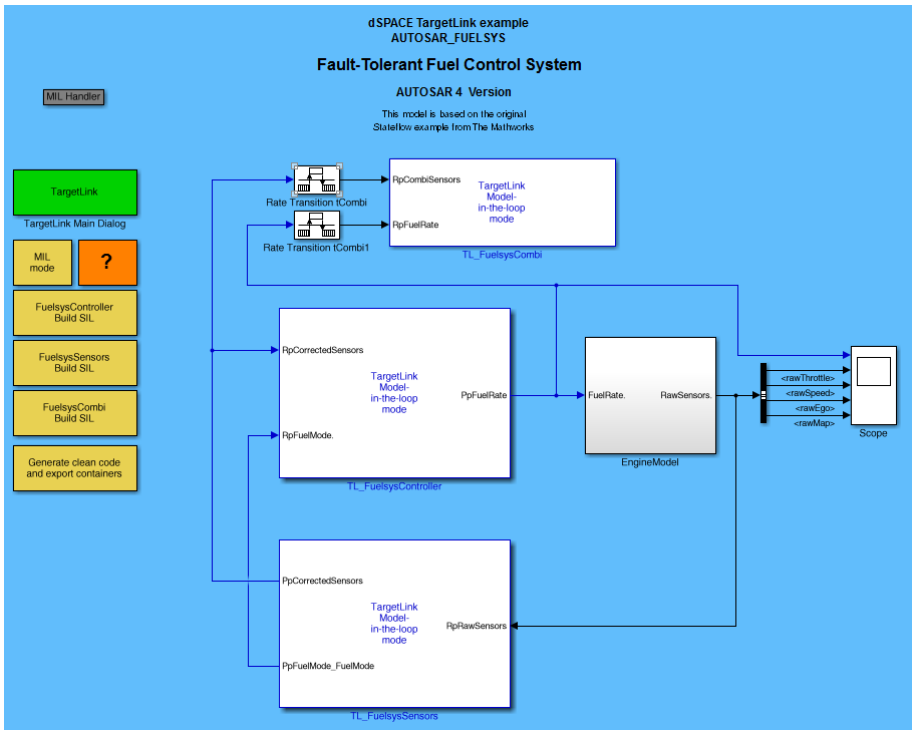
You can also enter **`tl_demos`** to access all available demos in the MATLAB Help browser. Select a demo and click **Open this example** at the top right.

Description

The `ar_memory_mapping` demo model shows you how to perform AUTOSAR memory mapping with TargetLink. The demo model contains all the relevant objects.

This demo model adjusts the AUTOSAR_FUELSYS demo model that implements a fault-tolerant fuel control system with software components.

The following illustration shows the root level of demo model.



AUTOSAR memory mapping with TargetLink

Use cases for AUTOSAR memory mapping In general, memory mapping is used to structure the memory's content and to accelerate access. Variables as well as functions can be properly mapped to the desired memory segments. In addition, it prevents unnecessary memory gaps (unused RAM space).

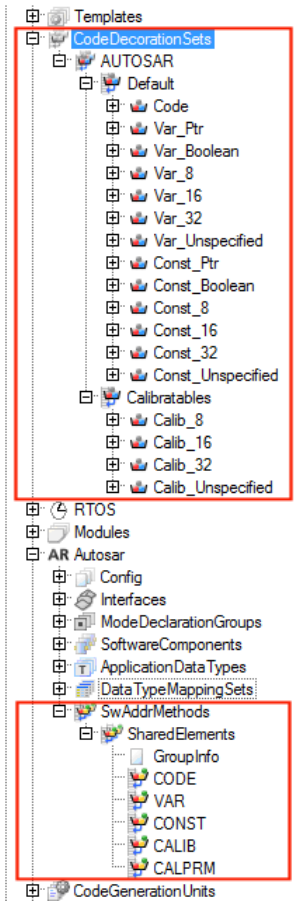
Data Dictionary objects for AUTOSAR memory mapping This demo model contains all the relevant objects to use AUTOSAR memory mapping with TargetLink. You can change the parameters or create new objects to experiment with the demo model. The following Data Dictionary objects relevant for AUTOSAR memory mapping are located in this demo:

- SwAddrMethod objects in the `/Pool/Autosar/SwAddrMethods/` subtree are provided for a high-level description. This is usually done with a system-level design tool such as SystemDesk. During container exchange or ARXML import, it is imported to the Data Dictionary.

- CodeDecorationSet objects are provided.

  The low-level description via DD CodeDecoration objects.

Description	AUTOSAR	Data Dictionary object
High-Level	`<SW-ADDR-METHOD>`	DD SwAddrMethod object
Low-Level	`<MEMORY-SECTION>`	DD CodeDecoration object



- SWCs need to be connected to CodeDecorationSet objects.

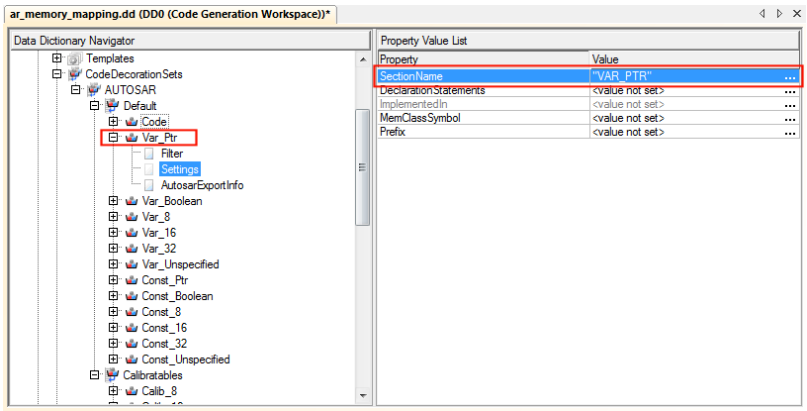
  To instruct TargetLink to export the `<Memory-Section>` element into the `<Resource-Consumption>` element of the correct software component's `<Implementation>` element, CodeDecorationSet objects are referenced via a MemorySectionsRef property of the SoftwareComponent object.

- CodeDecorationSet objects are referenced by VariableClasses, FunctionClasses, or in AUTOSAR/Config as DefaultCodeDecorationSet and thus are placed in the generated code.

- The Filter object of a CodeDecoration object is specified.

  Via the properties contained in the Filter object of a CodeDecoration object, it is specified to which code elements the decoration is to be applied.

- The Settings object of a CodeDecoration object is specified.



Via the properties contained in the Settings object of a CodeDecoration object, it is specified how the decoration appears in the production code.

- The AutosarExportInfo object of a code decoration is specified.

  Via the properties contained in the AutosarExportInfo child object of a CodeDecoration object, you can specify how the code decoration is used in AUTOSAR export.

Generating code and exporting ARXML files Executing Generate clean code and export containers on the left side of the model's root level has the following effects:

- The production code contains memory allocation keywords:

```
/* start of memory section 'CALIB_16' */
#define FuelsysController_START_SEC_CALIB_16
#include "FuelsysController_MemMap.h"
/*****************************************************************************************\
   CAL: global calibratable parameters (ROM) | Width: 16
\*****************************************************************************************/
extern CAL uint16 Ramp_Rate__Ki__x_table[5]; /*
   LSB: 2^-2 OFF: 0 MIN/MAX: 0 .. 2500
   Description: Speed axis of Ramp Rate (Ki) */
extern CAL uint16 Ramp_Rate__Ki__y_table[6]; /*
   LSB: 2^-15 OFF: 0 MIN/MAX: 0 .. 1
   Description: MAP axis of Ramp Rate (Ki) */
/* end of memory section 'CALIB_16' */
#define FuelsysController_STOP_SEC_CALIB_16
#include "FuelsysController_MemMap.h"

/* start of memory section 'VAR_16' */
#define FuelsysController_START_SEC_VAR_16
#include "FuelsysController_MemMap.h"
/*****************************************************************************************\
   UserSLGlobalInit: SLGlobalInit = { INIT_IN_RESTART_RUNNABLE } | Width: 16
\*****************************************************************************************/
extern sint16 X_SCtrl5_Discrete_Filter[2]; /* LSB: 2^-14 OFF: 0 MIN/MAX: -2 .. 1.99993896484375 */
extern sint16 X_SCtrl6_Discrete_Filter[2]; /* LSB: 2^-12 OFF: 0 MIN/MAX: -8 .. 7.999755859375 */

/* end of memory section 'VAR_16' */
#define FuelsysController_STOP_SEC_VAR_16
#include "FuelsysController_MemMap.h"
```

- The ARXML files are exported in a `/_ComponentFiles` subfolder.

```
<AR-PACKAGES>
  <AR-PACKAGE>
    <SHORT-NAME>Swcs</SHORT-NAME>
    <AR-PACKAGES>
      <AR-PACKAGE>
        <SHORT-NAME>FuelsysController</SHORT-NAME>
        <ELEMENTS>
          <SWC-IMPLEMENTATION UUID="ed16c695-b24d-492b-a11f-d727e89c76b7">
            <SHORT-NAME>FuelsysControllerImplementation</SHORT-NAME>
            <CODE-DESCRIPTORS>
                ...
            </CODE-DESCRIPTORS>
            <RESOURCE-CONSUMPTION>
              <SHORT-NAME>ResourceConsumption</SHORT-NAME>
              <MEMORY-SECTIONS>
                <MEMORY-SECTION UUID="da822095-8bf8-11e7-a00a-90b11c5d1ecd">
                  <SHORT-NAME>Var_16</SHORT-NAME>
                  <DESC>
                    <L-2 L="FOR-ALL">Used for variables which have to be aligned to 16 bit.</L-2>
                  </DESC>
                  <ALIGNMENT>16</ALIGNMENT>
                  <SW-ADDRMETHOD-REF DEST="SW-ADDR-METHOD">/SharedElements/SwAddrMethods/VAR</SW-ADDRMETHOD-REF>
                  <SYMBOL>VAR_16</SYMBOL>
                </MEMORY-SECTION>
                <MEMORY-SECTION UUID="da822099-8bf8-11e7-a00a-90b11c5d1ecd">
                  <SHORT-NAME>Calib_16</SHORT-NAME>
                  <DESC>
                    <L-2 L="FOR-ALL">Used for calibration variables which have to be aligned to 16 bit.</L-2>
                  </DESC>
                  <ALIGNMENT>16</ALIGNMENT>
                  <SW-ADDRMETHOD-REF DEST="SW-ADDR-METHOD">/SharedElements/SwAddrMethods/CALIB</SW-ADDRMETHOD-REF>
                  <SYMBOL>CALIB_16</SYMBOL>
                </MEMORY-SECTION>
              </MEMORY-SECTIONS>
            </RESOURCE-CONSUMPTION>
```

Preparing memory mapping manually Another example that helps you to prepare memory mapping *manually* is also available. The example shows the following aspects:

- Preparing the Data Dictionary for specifying memory sections as defined by AUTOSAR
- Specifying the memory sections for global variables and runnable code
- Specifying the memory sections for calibratable variables

Related topics

Basics

Basics on Code Decorations (📖 TargetLink Customization and Optimization Guide)
Basics on Memory Mapping in ClassicAUTOSAR (📖 TargetLink Classic AUTOSAR Modeling Guide)

Examples

Example of Adapting the AR_POSCONTROL Demo Model to use Memory Sections (Classic AUTOSAR) (📖 TargetLink Classic AUTOSAR Modeling Guide)

References

AutosarExportInfo (📖 TargetLink Data Dictionary Reference)
Filter (📖 TargetLink Data Dictionary Reference)
Settings (📖 TargetLink Data Dictionary Reference)

AR_ARRAY_OF_STRUCT_DATA

Access

In the MATLAB Command Window, enter the following command:

```
tl_demos ar_array_of_struct_data
```

You can also enter **`tl_demos`** to access all available demos in the MATLAB Help browser. Select a demo and click **Open this example** at the top right.

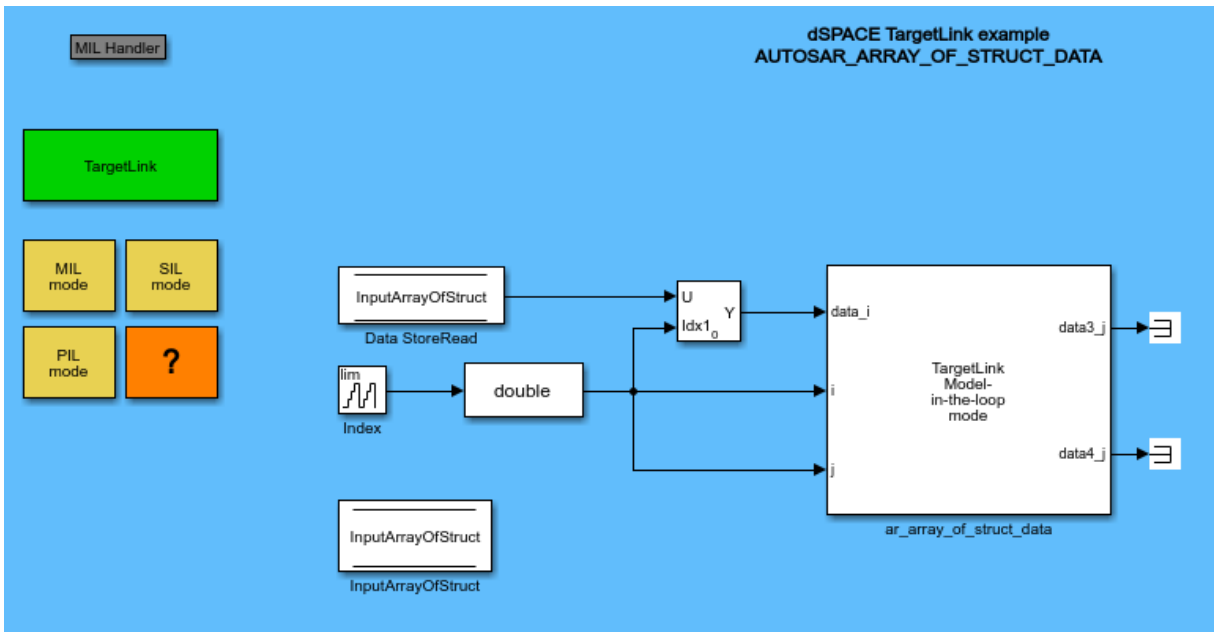
Description

The demo model demonstrates exemplarily how to access array elements from an array of struct via RTE API calls that are generated from a generic modeling pattern based on **Custom Code** (type II) blocks and data store blocks.

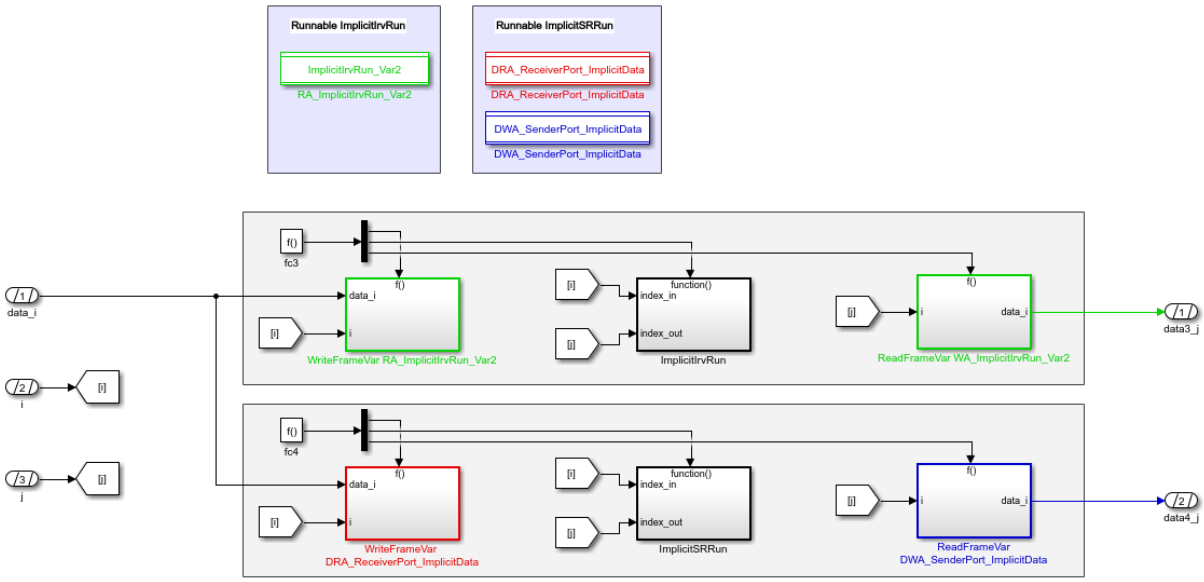
In the model, the array-of-struct variable is represented by the **Data Store Memory** block called `InputArrayOfStruct` that is placed on the root level of the model, outside the TargetLink subsystem.

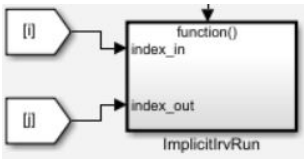
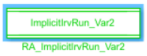
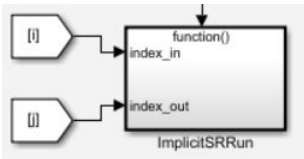
The **InputArrayOfStruct[i]** array element of the array-of-struct variable is selected and fed into the TargetLink subsystem as **data_i**, together with the indices **i** and **j**.

The following illustration shows the root level of the ar_array_of_struct_data demo.

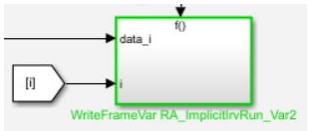
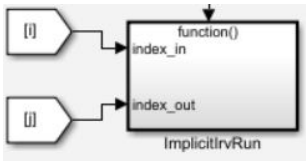
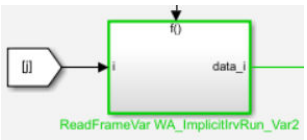
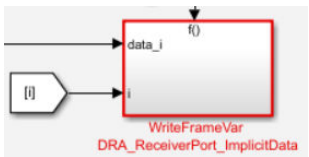
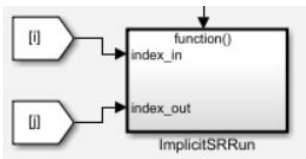
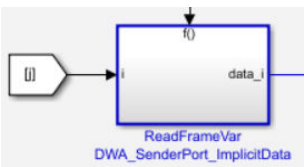


The TargetLink subsystem contains two runnable subsystems, each of which uses a communication kind as defined by Classic AUTOSAR. The AUTOSAR specification of the communication kind is made at **Data Store Memory** blocks that are associated with each runnable subsystem (colors were added for clarity):



Runnable Subsystem	Communication Kind	Associated Data Store Memories
	Interrunnable Communication	 This Data Store Memory block carries the AUTOSAR specification for interrunnable communication.
	Implicit Sender-Receiver Communication	This Data Store Memory block carries the AUTOSAR specification of a receiver port for implicit sender-receiver communication. This Data Store Memory block carries the AUTOSAR specification of a provide port for implicit sender-receiver communication.

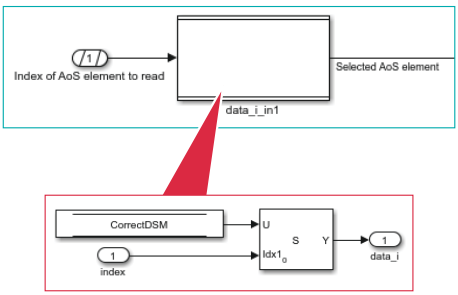
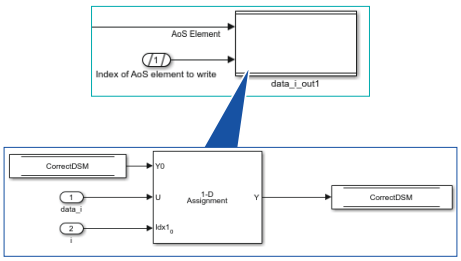
The other subsystems are used for simulation purposes:

Subsystem	Purpose	Associated Runnable
	Writing data_i to the data store called ImplicitIrvRun_Var2.	
	Reading the data written by the ImplicitIrvRun runnable to the data store called ImplicitIrvRun_Var2. This data is then routed out of the TargetLink subsystem via the Bus Outport block called `data3_j`.	
	Writing data_i to the data store called DRA_ReceiverPort_ImplicitData.	
	Reading the data written by the ImplicitSRRun runnable to the data store called DWA_SenderPort_ImplicitData. This data is then routed out of the TargetLink subsystem via the Bus Outport block called `data4_j`.	

Function call triggers ensure that the RTE variables are written before and read after the execution of the corresponding runnable code.

Generic modeling pattern for accessing array of structs

The demo model shows two generic modeling patterns for accessing arrays of structs. Both modeling patterns are based on a Custom Code (type II) block that accesses a data store:

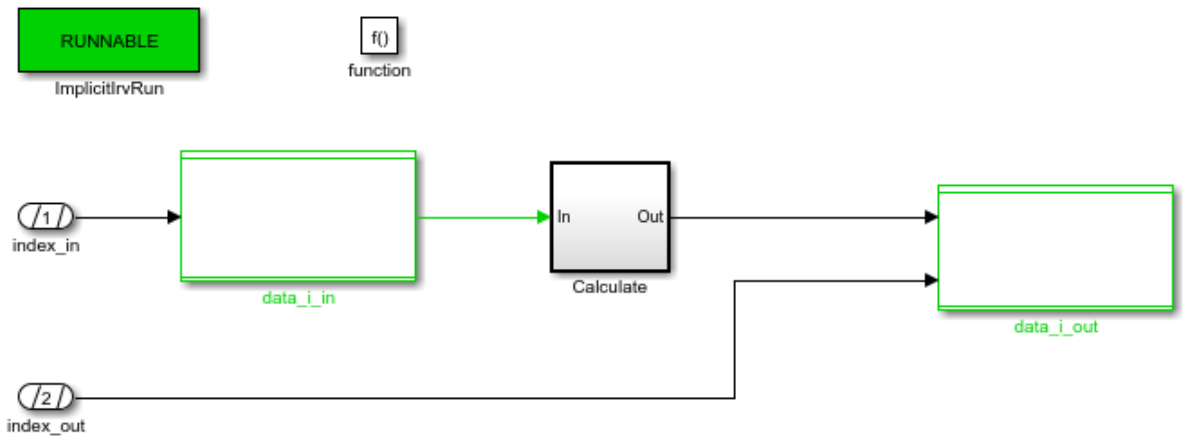
Read Access Modeling Pattern	Write Access Modeling Pattern
This Custom Code (type II) block has an index as the input and the selected array element as the output:  	This Custom Code (type II) block has an index and an array element as the input: 

Its custom code file looks as follows:
`/* fxp_output_begin */`
`data_i = data_store_signal[i];`
`/* fxp_output_end */`
	Its custom code file looks as follows:
	`/* fxp_output_begin */`
	`data_store_signal[i] = data_i;`
	`/* fxp_output_end */`

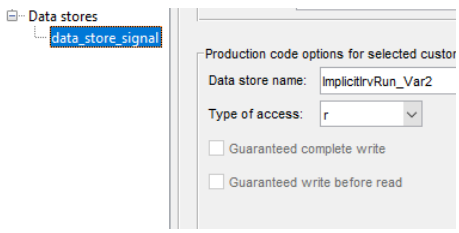
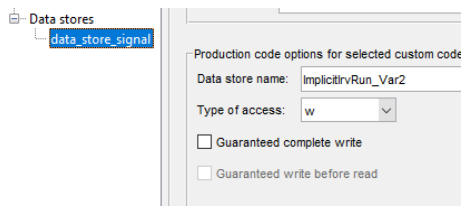
During code generation, TargetLink generates direct accesses to the RTE variables, or the correct RTE API functions and access points from the following information: ▪ The AUTOSAR specification at the associated Data Store Memory block ▪ The context of the Custom Code (type II) block	

Using the modeling pattern for interrunnable communication

The runnable subsystem called `ImplicitIrvRun` uses the generic modeling pattern for interrunnable communication:



The data store signals used at the **Custom Code** (type II) blocks are specified as follows:

data_i_in	data_i_out
	

The production code looks like this:

data_i_in

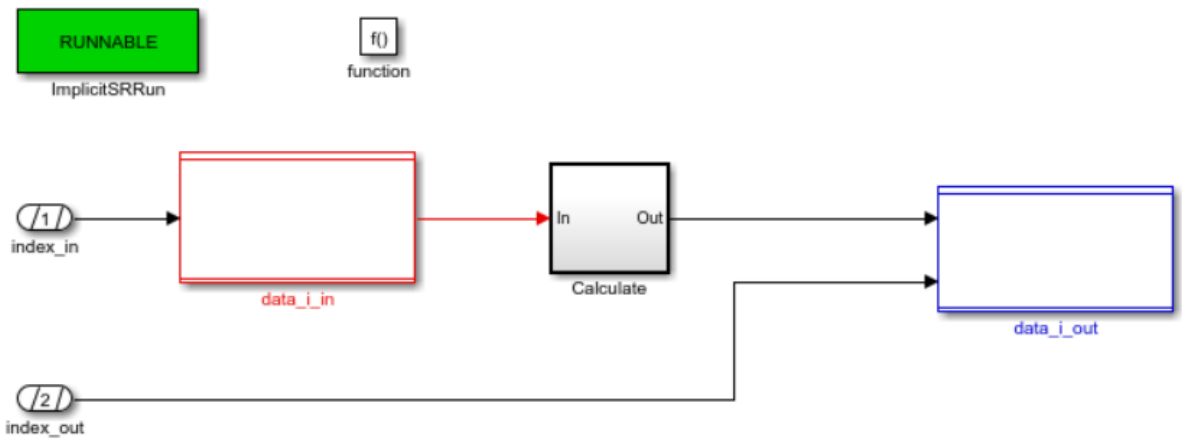
```
...

/* Custom code:
ar_array_of_struct_data/ImplicitIrvRun/data_i_in */
   p_Var2 = Rte_IrvIRead_ImplicitIrvRun_Var2();

...

   /* Custom code:
ar_array_of_struct_data/ImplicitIrvRun/data_i_in <<
output code >> */
   {
      Sa2_data_i = p_Var2[Index];
   }

...
```

data_i_out

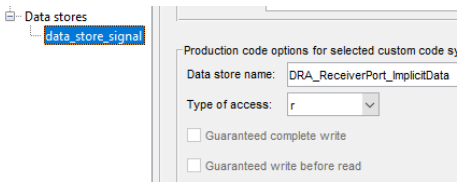
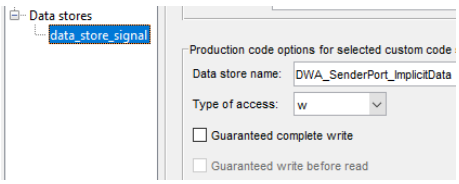
```
...

/* Custom code:
ar_array_of_struct_data/ImplicitIrvRun/data_i_out */
   p_Var2_a = Rte_IrvIWriteRef_ImplicitIrvRun_Var2();

...

   /* Custom code:
ar_array_of_struct_data/ImplicitIrvRun/data_i_out <<
output code >> */
   {
      p_Var2_a[Index_a] = Sa8_Out;
   }

...
```

Using the modeling pattern for sender-receiver communication

The runnable subsystem called `ImplicitSRRun` uses the generic modeling pattern for interrunnable communication:



The data store signals used at the **Custom Code** (type II) blocks are specified as follows:

data_i_in	data_i_out
	

The production code looks like this:

data_i_in

```
...

   /* Custom code:
ar_array_of_struct_data/ImplicitSRRun/data_i_in */
   p_ImplicitData
= Rte_IRead_ImplicitSRRun_ReceiverPort_ImplicitData();

...

   /* Custom code:
ar_array_of_struct_data/ImplicitSRRun/data_i_in <<
output code >> */
   {
      Sa3_data_i = p_ImplicitData[Index];
   }

...
```

data_i_out

```
...

   /* Custom code:
ar_array_of_struct_data/ImplicitSRRun/data_i_out */
   p_ImplicitData_a
= Rte_IWriteRef_ImplicitSRRun_SenderPort_ImplicitData();

...

   /* Custom code:
ar_array_of_struct_data/ImplicitSRRun/data_i_out << output
code >> */
   {
      p_ImplicitData_a[Index_a] = Sa9_Out;
   }

...
```

Related topics

Basics

Basics on Working With Array of Structs When Modeling Classic AUTOSAR in TargetLink (📖 TargetLink Classic AUTOSAR Modeling Guide)
Working with Array-of-Bus Signals (📖 TargetLink Preparation and Simulation Guide)

Examples

Example of Dynamically Accessing Array-of-Struct Variables via Data Store Blocks and Custom Code (Type II) Blocks (📖 TargetLink Preparation and Simulation Guide)

AR_INCREMENTAL_CODEGEN

Access

In the MATLAB Command Window, enter the following command:

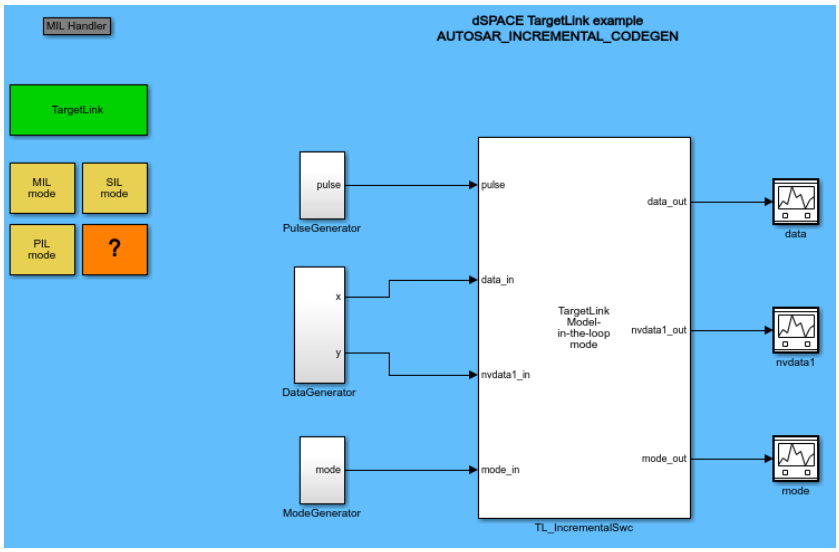
```
tl_demos ar_incremental_codegen
```

You can also enter **`tl_demos`** to access all available demos in the MATLAB Help browser. Select a demo and click **Open this example** at the top right.

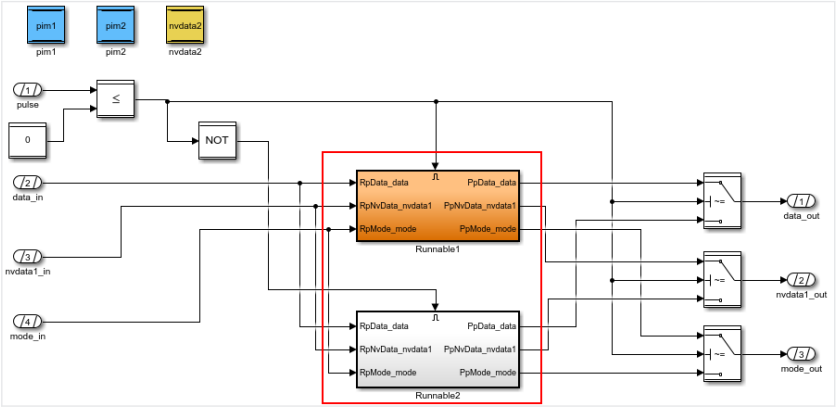
Description

The `AR_INCREMENTAL_CODEGEN` demo model shows you how to partition model and code in an AUTOSAR context, as described in Partitioning Model and Code for Classic AUTOSAR (📖 TargetLink Classic AUTOSAR Modeling Guide).

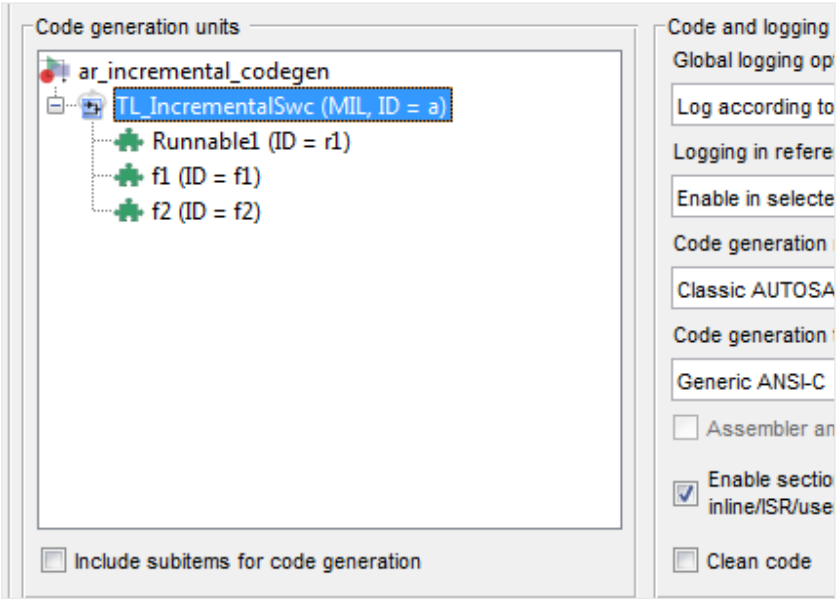
The following illustration shows the root level of the `AR_INCREMENTAL_CODEGEN` demo model.



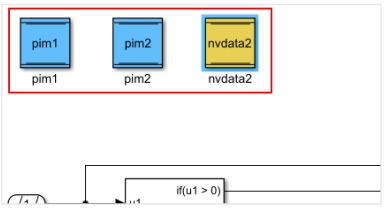
Different code generation units The TargetLink subsystem called TL_IncrementalSwc contains two runnable subsystems called Runnable1 and Runnable2.



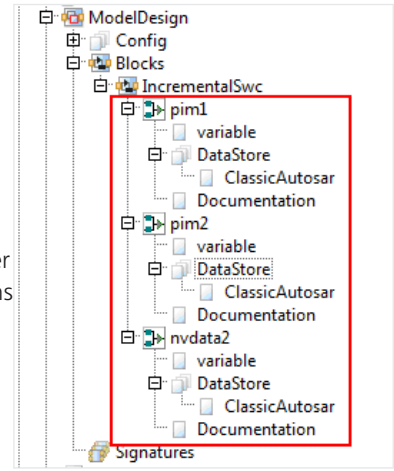
Runnable1 is configured for incremental code generation and forms a code generation unit (CGU) of its own. Runnable2 does not form a CGU of its own but contains the two subsystems f1 and f2 that are configured for incremental code generation. Accordingly, f1 and f2 are CGUs:



Data Store Memory block specifications in the Data Dictionary The demo also shows the specification of global data stores. The Data Store Memory blocks that are placed on the root level of the TargetLink subsystem reference a block specification in the Data Dictionary:



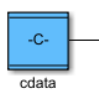
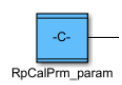
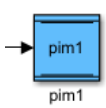
The blue blocks each reference a DD Variable object that is used to specify a per instance memory. The yellow block contains the AUTOSAR specification defined at the properties of the DD `nvdata2/DataStore/ClassicAutosar` object.



Providing global data stores via the Data Dictionary is required if you want to communicate accross CGU boundaries.

Partitioning model and code in an AUTOSAR context

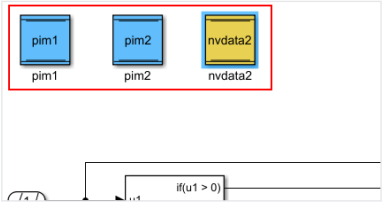
TargetLink lets you partition model and code in an AUTOSAR context, as described in Partitioning Model and Code for Classic AUTOSAR (📖 TargetLink Classic AUTOSAR Modeling Guide). The following table shows the different modeling styles that are featured in this demo model:

Modeling Style	Constraints	Modeling Examples
TargetLink port blocks whose AUTOSAR mode is set to `Classic`.	Must be placed in the same CGU as the Runnable block (Function block whose AUTOSAR mode is set to `Classic` and whose Role is set to `Runnable`).	Sender-receiver communication in Runnable1:
		NvData communication in Runnable1:
		Mode communication in Runnable1:
Data Store Memory blocks whose AUTOSAR mode is set to `Classic`	Can be placed in incremental CGUs. If the blocks are specified in the Data Dictionary, they can be used to communicate across CGU boundaries.	NvData communication in Runnable2/f2:
TargetLink blocks that reference a DD Variable object that is used in an AUTOSAR context	None	Shared calibration parameters 
		Parameter communication for calibration 
		Per instance memories (PIM): 
Function blocks whose AUTOSAR mode is set to `Classic`	Runnable subsystems can be configured for incremental code generation.	▪ Runnable1 is configured for incremental code generation and forms a code generation unit (CGU) of its own. ▪ Runnable2 does not form a CGU of its own.
	Operation subsystems must not be configured for incremental code generation.	Synchronous operation call subsystem:

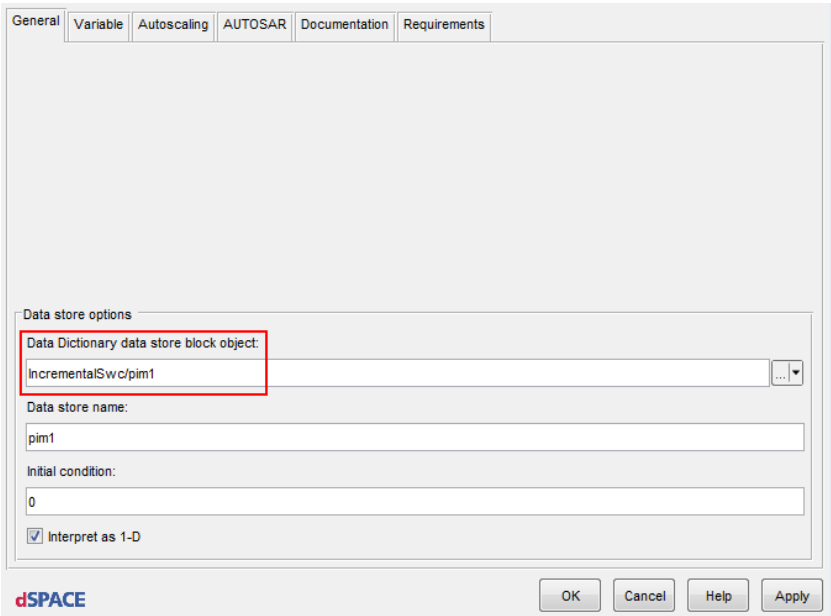
Specifying global data stores in the Data Dictionary

TargetLink lets you specify global data stores in the Data Dictionary. This is necessary to communicate accross CGU boundaries.

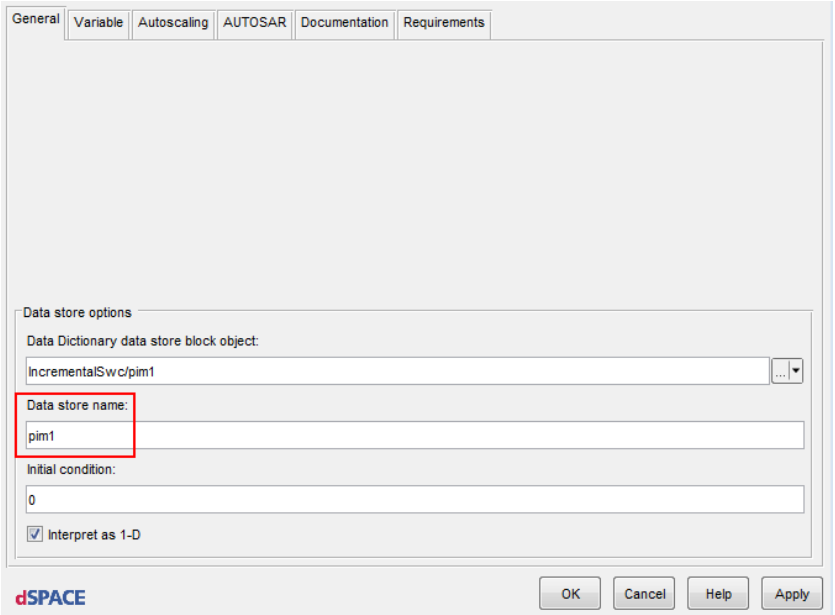
Data Store Memory blocks The following Data Store Memory blocks are located on the root level of TL_IncrementalSwc:



Each of these blocks references a DD Block object whose BlockType property is set to `TL_DataStoreMemory`. The following screenshot shows the block dialog of the pim1 block:

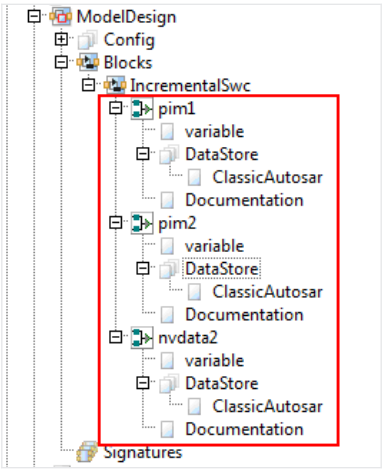


Additionally, each of these blocks specifies the name of the data store via the Data store name property:

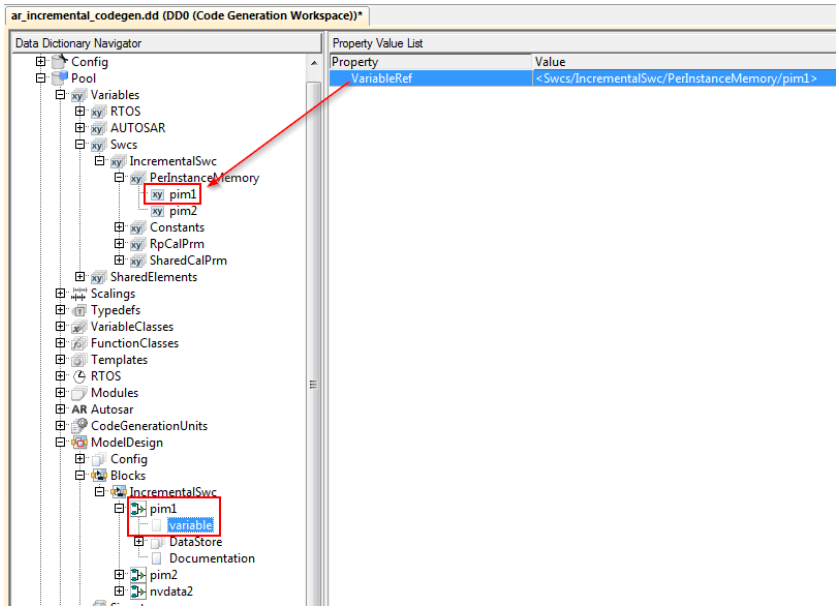


According to Simulink semantics, this name is used to consume the data store via Data Store Read or Data Store Write blocks that are located on the same level or below in the model hierarchy.

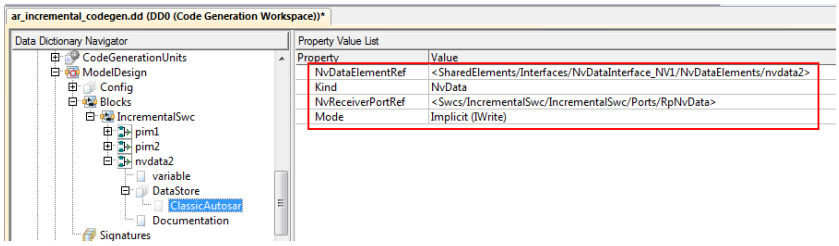
Block specifications in the Data Dictionary The following screenshot shows the three DD Block objects that are used to specify the global data stores:



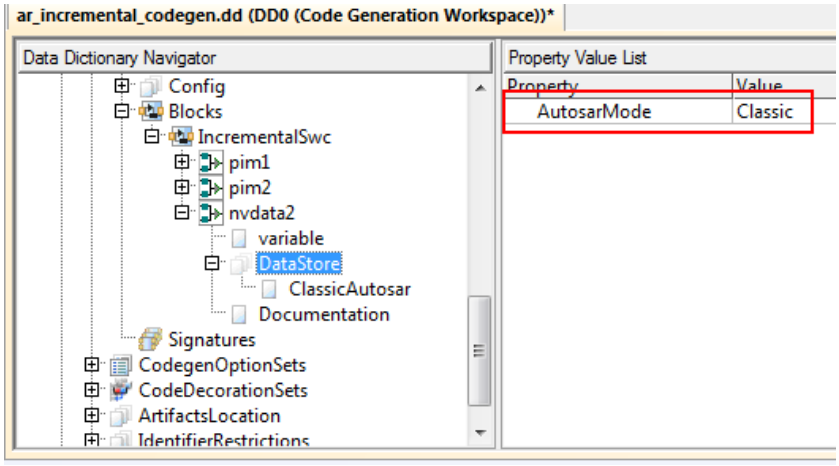
For pim1 and pim2 (blue blocks), the corresponding block variable is referenced at the DD BlockVariable object called `variable`. This is the case because TargetLink generates calls to the `Rte_Pim` RTE API function via DD Variable objects that reference a predefined variable class that references a suitable access function specification. The following screenshot shows the reference for pim1:



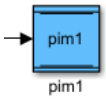
For nvdata2 (yellow block), the AUTOSAR specification is provided at the `DataStore/ClassicAutosar` child object. Refer to the following screenshot:



For TargetLink to use this specification, the AutosarMode property of the DataStore child object must be set to `Classic`:



Communication across CGU boundaries The global data stores that are specified in the Data Dictionary can be placed in the model hierarchy at the same level or below the Data Store Memory blocks. This enables communication across the boundaries of the individual CGUs:

Consuming Data Store Block	Code Generation Unit	Description
	Runnable1	The corresonding Data Store Memory block references a DD Variable object that is used to specify a per instance memory.
	Runnable1 and f2	
	f2	The corresponding Data Store Memory block carries the AUTOSAR specification at the properties of the DD `DataStore/ClassicAutosar` object.

Related topics

Basics

Comprehensive Modeling of RTE Code Pattern (📖 TargetLink Classic AUTOSAR Modeling Guide)
Modeling Client-Server Communication (📖 TargetLink Classic AUTOSAR Modeling Guide)
Modeling Modes (📖 TargetLink Classic AUTOSAR Modeling Guide)
Modeling Per Instance Memories (📖 TargetLink Classic AUTOSAR Modeling Guide)
Modeling Sender-Receiver Communication (📖 TargetLink Classic AUTOSAR Modeling Guide)
Partitioning Model and Code for Classic AUTOSAR (📖 TargetLink Classic AUTOSAR Modeling Guide)
Preparing SWCs for Measurement and Calibration (📖 TargetLink Classic AUTOSAR Modeling Guide)

HowTos

How to Create a Global Data Store via Data Store Memory Blocks (📖 TargetLink Preparation and Simulation Guide)
How to Model Shared Parameters for Calibration (📖 TargetLink Classic AUTOSAR Modeling Guide)
Modeling NvData Communication (📖 TargetLink Classic AUTOSAR Modeling Guide)

AAR_COMMUNICATION

Access

In the MATLAB Command Window, enter the following command:

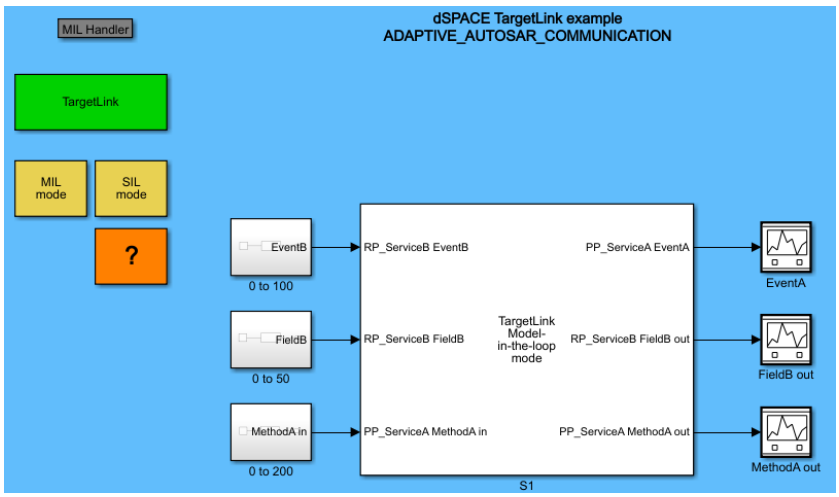
```
tl_demos aar_communication
```

You can also enter **`tl_demos`** to access all available demos in the MATLAB Help browser. Select a demo and click **Open this example** at the top right.

Description

The `AAR_COMMUNICATION` demo model shows you how to model communication according to the specification of Adaptive AUTOSAR for the functional clusters ara::com and ara::per in TargetLink as described in Modeling Communication as Defined by Adaptive AUTOSAR (📖 TargetLink Adaptive AUTOSAR Modeling Guide).

The following illustration shows the root level of the `aar_communication` demo model.

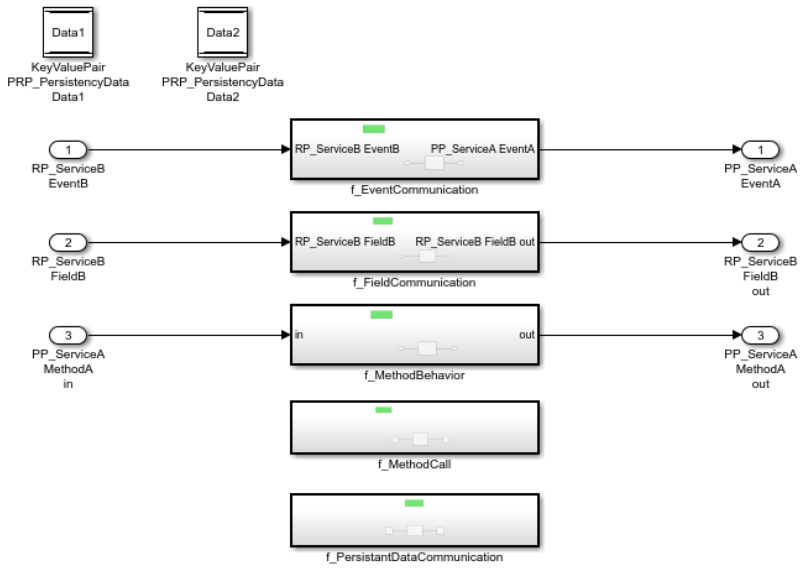


Different subsystems The TargetLink subsystem called S1 contains five subsystems specified for Adaptive AUTOSAR communication.

Each of the subsystems shows you one aspect of Adaptive AUTOSAR communication.

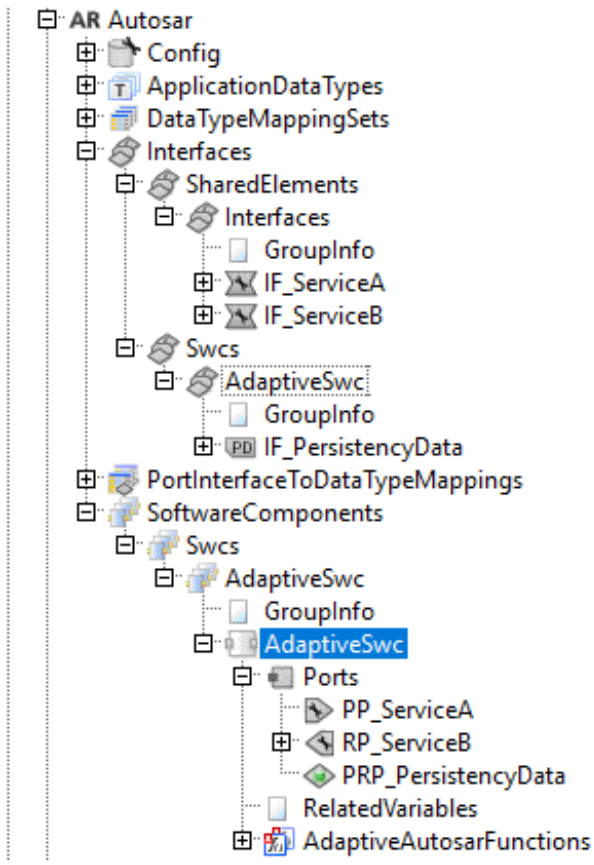
The following ways to communicate are shown:

- Event communication on page 167
- Field communication on page 168
- Method behavior on page 170
- Method call on page 171
- Persistent data communication on page 174



Data Dictionary specification

The Data Dictionary for this demo model contains one DD SoftwareComponent object named AdaptiveSwc:



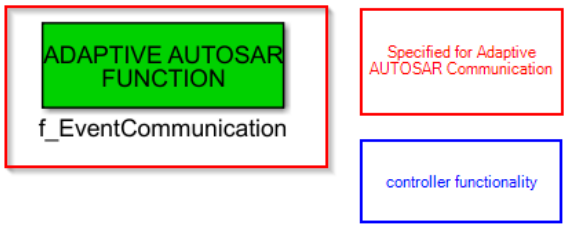
This software component has three ports that connect the software component with interfaces:

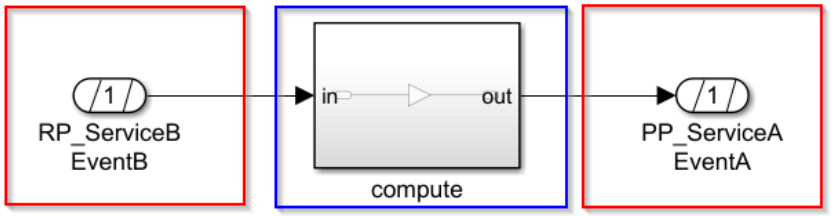
Port Name	DD <Port> Object	Interface Name	DD <Interface> Object
PP_ServiceA	ServiceProvidePort	IF_ServiceA	ServiceInterface
RP_ServiceB	ServiceRequirePort	IF_ServiceB	ServiceInterface
PRP_PersistencyData	PersistencyProvideRequirePort	IF_PersistencyData	PersistencyKeyValueDatabaseInterface

The Software Component has one DD AdaptiveAutosarFunction object for each subsystem in the model. These Adaptive AUTOSAR functions partition the code for each subsystem in a separate function as is shown in the descriptions of the subsystems below.

Event communication

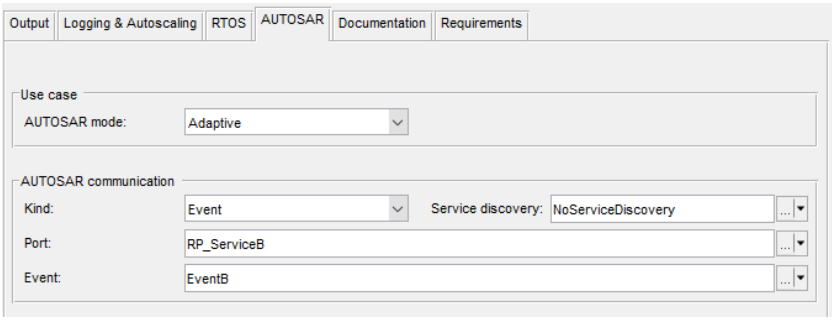
The f_EventCommunication subsystem includes model elements that are specified for event communication as defined by ara::com:





In TargetLink, event communication is modeled via port blocks that reside in Adaptive AUTOSAR Function subsystems or Method Behavior subsystems.

The InPort block named RP_ServiceB EventB in this subsystem is specified as follows:



This combination of an **InPort** block that references a DD **ServiceRequirePort** object and whose **Kind** property is set to **Event** lets you receive events as defined by ara::com. The resulting Adaptive AUTOSAR behavior code calls an automatically generated ARA adapter code function:

```
void f_EventCommunication(void)
{
   /* SLLocal: Default storage class for local variables | Width: 32 */
   IDT_int32 EventB;

   /* TargetLink inport: S1/f_EventCommunication/RP_ServiceB EventB */
   Receive_AdaptiveSwc_RP_ServiceB_EventB_NoServiceDiscovery_S1(&EventB);

   /* # combined # TargetLink outport: S1/f_EventCommunication/PP_ServiceA EventA
      # combined # Gain: S1/f_EventCommunication/compute/Gain */
   Send_AdaptiveSwc_PP_ServiceA_EventA_S1(EventB * 2);
}
```

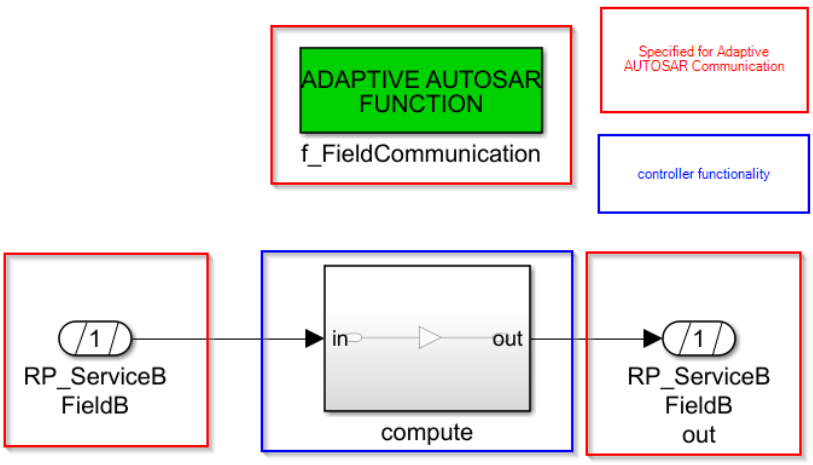
An **OutPort** block specified as the OutPort block named **PP_ServiceA EventA** shown in this subsystem leads to a call to an ARA adapter code function for sending events in the resulting behavior code as shown above:

```
Send_AdaptiveSwc_PP_ServiceA_EventA_S1(EventB * 2);
```

The controller functionality is modeled inside the inner subsystem. As this demo model focuses on modeling communication according to Adaptive AUTOSAR, only a **Gain** block is used. The code generated for these model elements is part of the Adaptive AUTOSAR behavior code as seen above.

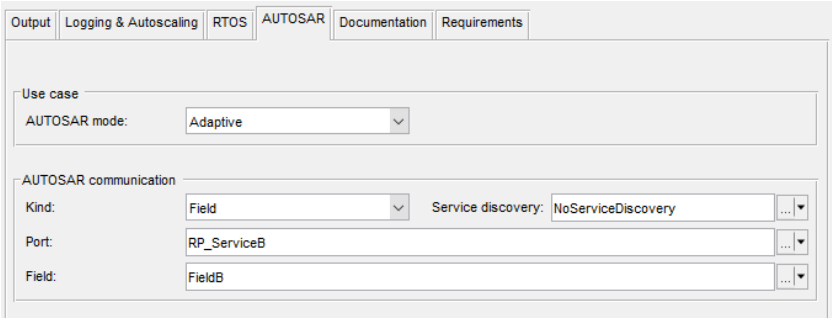
Field communication

The **f_FieldCommunication** subsystem includes model elements that are specified for field communication as defined by ara::com:



In TargetLink, field communication is modeled via port blocks that reside in Adaptive AUTOSAR Function subsystems or Method Behavior subsystems.

The InPort block named **RP_ServiceB FieldB** in this subsystem is specified as follows:



This combination of an InPort block that references a DD **ServiceRequirePort** object and whose **Kind** property is set to **Field** lets you get the value of a field as defined by ara::com by calling an automatically generated ARA adapter code function from the Adaptive AUTOSAR behavior code:

```
void f_FieldCommunication(void)
{
   /* SLLocal: Default storage class for local variables | Width: 32 */
   IDT_int32 FieldB;

   /* TargetLink inport: S1/f_FieldCommunication/RP_ServiceB FieldB */
   Get_AdaptiveSwc_RP_ServiceB_FieldB_NoServiceDiscovery_S1(&FieldB);

   /* # combined # TargetLink outport: S1/f_FieldCommunication/RP_ServiceB FieldB
out
      # combined # Gain: S1/f_FieldCommunication/compute/Gain */
   Set_AdaptiveSwc_RP_ServiceB_FieldB_NoServiceDiscovery_S1(FieldB * 2);
}
```

An **OutPort** block specified as the OutPort block named **RP_ServiceB FieldB out** shown in this subsystem leads to a call to an ARA adapter code function for setting fields in the resulting behavior code as shown above:`Set_AdaptiveSwc_RP_ServiceB_FieldB_NoServiceDiscovery_S1(FieldB * 2);`.

To get and/or set the values of a field, the DD **ArField** object, in this demo named **FieldB**, must be specified appropriately:

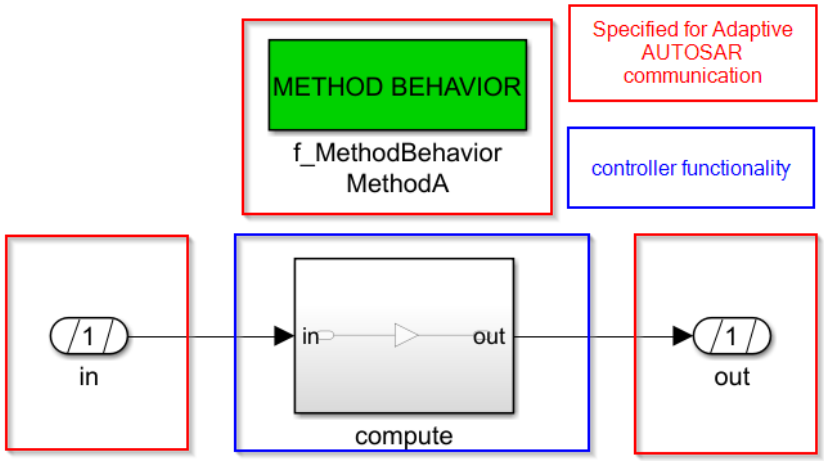
Property Value List

Property	Value
Description	""
HasGetter	on
HasNotifier	off
HasSetter	on

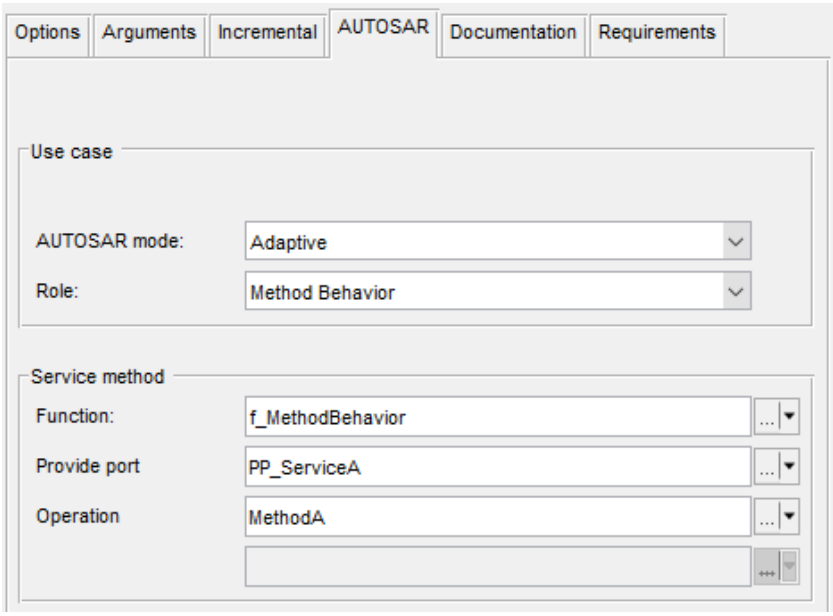
The controller functionality is modeled in the inner subsystem. Because this demo model focuses on modeling communication according to Adaptive AUTOSAR, only a **Gain** block is used. The code generated for these model elements is part of the Adaptive AUTOSAR behavior code as seen above.

Method behavior

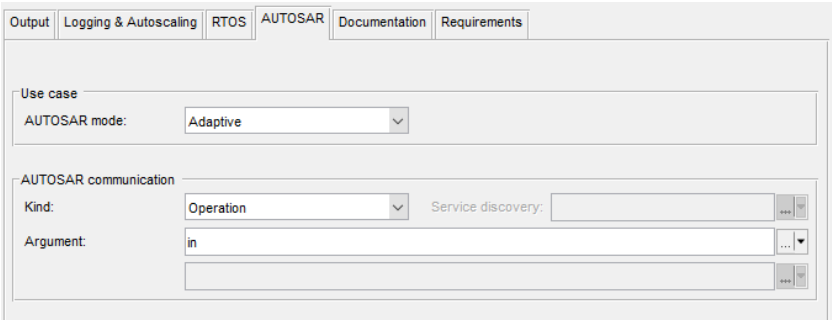
The f_MethodBehavior subsystem models the implementation of a method:



In TargetLink, method behaviors are modeled in Method Behavior subsystems. The method that is implemented and the Provide port that connects the Software Component with the interface the method belongs to are referenced via the Function block:



The use of method arguments is modeled via port blocks that reference the appropriate DD OperationArgument objects. The InPort block named in in this subsystem is specified as follows:



The controller functionality is modeled in the inner subsystem. Because this demo model focuses on modeling communication according to Adaptive AUTOSAR, only a Gain block is used. The code generated for these model elements is part of the Adaptive AUTOSAR behavior code.

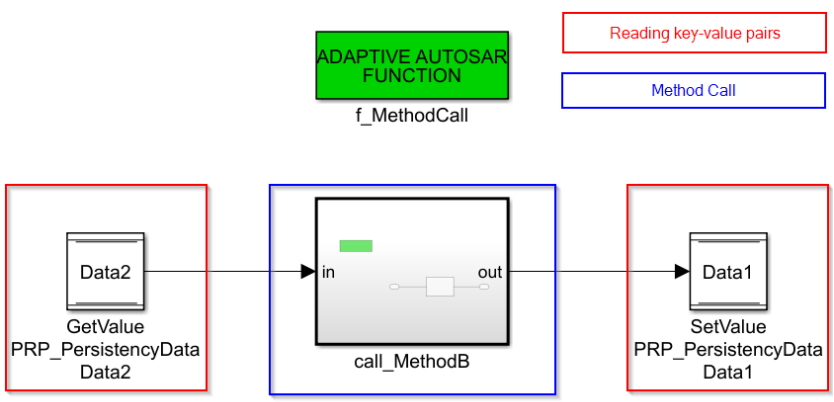
The behavior code generated for the Method Behavior subsystem and the inner subsystem shown in this demo looks as follows:

```
void f_MethodBehavior(IDT_int32 in, IDT_int32 * out)
{
   /* TargetLink outport: S1/f_MethodBehavior/out
      # combined # Gain: S1/f_MethodBehavior/compute/Gain */
   *out = in * 2;
}
```

Method call

The f_MethodCall subsystem combines two aspects of Adaptive AUTOSAR communication:

- Reading and writing key-value pairs from a key-value storage via Data Store blocks
- Calling a method via a Method Call subsystem



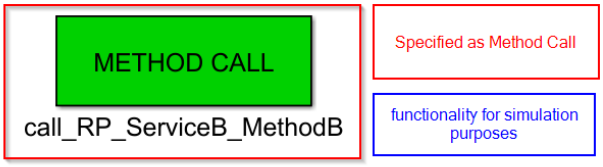
Reading and writing key-value pairs In TargetLink, reading and writing key-value pairs from a key-value storage is modeled via Data Store blocks. The Data Store Read block named GetValue PRP_PeristencyData Data2 is associated with the Data Store Memory block named KeyValuePair PRP_PersistencyData Data2 at the root level of the TargetLink subsystem. This Data Store Memory block references the DD PersistencyDataElement object

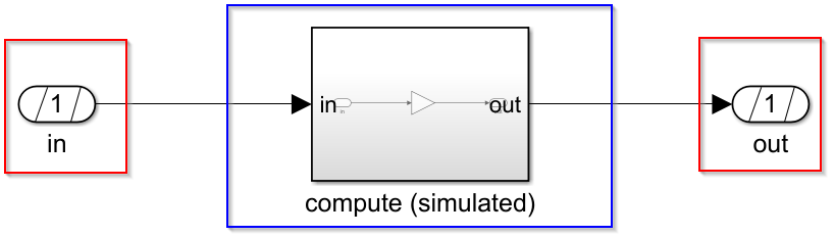
named Data2. The Data Store Read block shown here therefore models reading this data element. The resulting behavior code contains a call to an adapter code function that reads the value for the key `Data2` from the key-value storage and assigns it to the data element:

```
void f_MethodCall(void)
{
   /* SLLocal: Default storage class for local variables | Width: 32 */
   int32_t Data2;
   IDT_int32 out;

   /* Data store read: S1/f_MethodCall/GetValue PRP_PersistencyData Data2 */
   GetValue_AdaptiveSwc_PRP_PersistencyData_Data2_S1(&Data2);
   /* call of function: S1/f_MethodCall/call_MethodB */
   Call_AdaptiveSwc_RP_ServiceB_MethodB_NoServiceDiscovery_S1(Data2, &out);

   /* Data store write: S1/f_MethodCall/SetValue PRP_PersistencyData Data1 */
   SetValue_AdaptiveSwc_PRP_PersistencyData_Data1_S1(out);
}
```

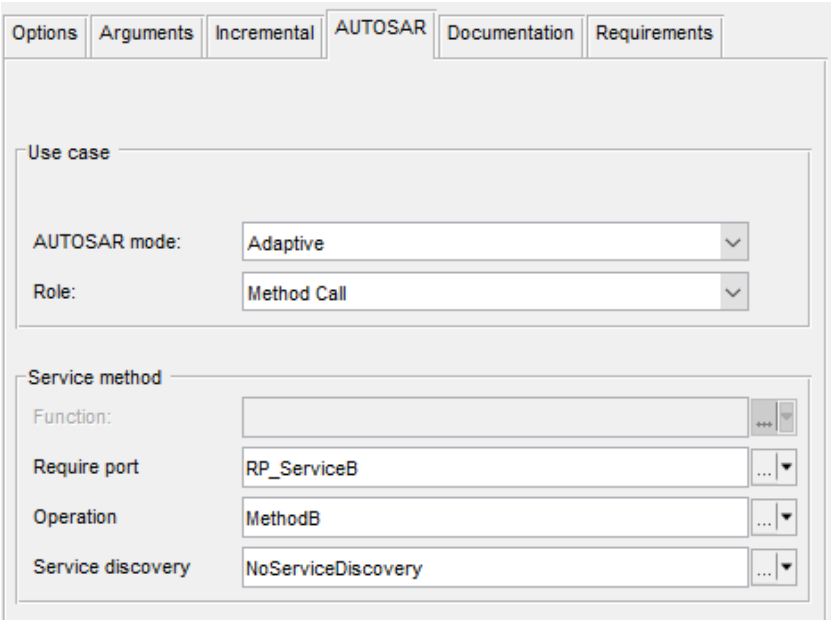
Accordingly, the call of the adapter code function to set the value for the key `Data1` is modeled via a Data Store Write block, named SetValue PRP_PersistencyData Data1 in this subsystem. This Data Store Write block is associated with a different Data Store Memory block, which references the DD PersistencyDataElement object Data1.

Method Call The subsystem named call_MethodB is a Method Call subsystem:

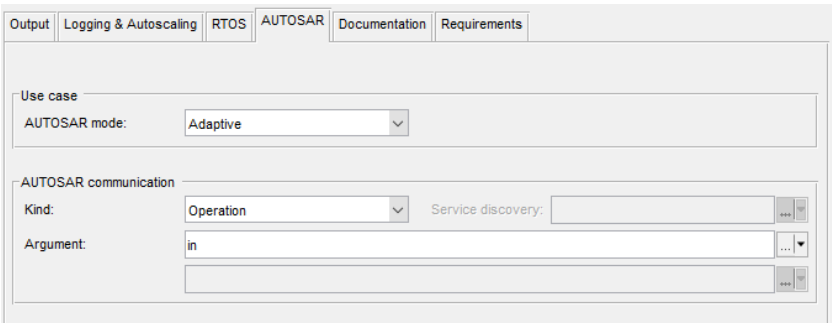




The method that is called and the Require port that connects the Software Component with the interface the method belongs to are referenced via the Function block:



The use of method arguments is modeled via port blocks that reference the appropriate DD **OperationArgument** objects.The InPort block named **in** in this subsystem is specified as follows:

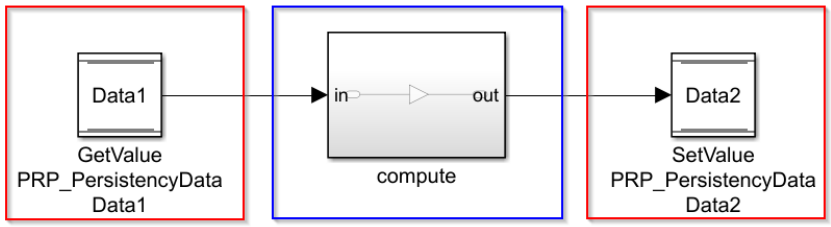


The inner subsystem named **compute (simulated)** is used to model the method for simulation purposes. Because this demo model focuses on modeling communication according to Adaptive AUTOSAR, only a **Gain** block is used. The code generated for these model elements is *not* part of the Adaptive AUTOSAR behavior code. Instead, a call of the method is generated in the behavior code:

```
void f_MethodCall(void)
{
   /* SLLocal: Default storage class for local variables | Width: 32 */
   int32_t Data2;
   IDT_int32 out;

   /* -- snip --
   /* call of function: S1/f_MethodCall/call_MethodB */
   Call_AdaptiveSwc_RP_ServiceB_MethodB_NoServiceDiscovery_S1(Data2, &out);

   /* -- snip --
}
```

Persistent data communication

The f_PersistentDataCommunication subsystem models reading and writing key-value pairs from a key-value storage via Data Store blocks :



In TargetLink, reading and writing key-value pairs from a key-value storage is modeled via Data Store blocks. The Data Store Read block named GetValue PRP_PeristencyData Data1 is associated with the Data Store Memory block named KeyValuePair PRP_PersistencyData Data1 at the root level of the TargetLink subsystem. This Data Store Memory block references the DD PersistencyDataElement object named Data1. The Data Store Read block shown here therefore models reading this data element. The resulting behavior code contains a call to an adapter code function that reads the value for the key `Data1` from the key-value storage and assigns it to the data element:

```
void f_PersistantDataCommunication(void)
{
   /* SLLocal: Default storage class for local variables | Width: 32 */
   int32_t Data1;

   /* Data store read: S1/f_PersistantDataCommunication/GetValue
PRP_PersistencyData Data1 */
   GetValue_AdaptiveSwc_PRP_PersistencyData_Data1_S1(&Data1);

   /* Data store write: S1/f_PersistantDataCommunication/SetValue
PRP_PersistencyData Data2
      # combined # Gain: S1/f_PersistantDataCommunication/compute/Gain */
   SetValue_AdaptiveSwc_PRP_PersistencyData_Data2_S1(Data1 * 2);
}
```

Accordingly, the call of the adapter code function to set the value for the key `Data2` is modeled via a Data Store Write block, named SetValue PRP_PersistencyData Data2 in this subsystem. Note that this Data Store Write block is associated with a different Data Store Memory block, which references the DD PersistencyDataElement object Data2.

The controller functionality is modeled in the inner subsystem. Because this demo model focuses on modeling communication according to Adaptive AUTOSAR, only a Gain block is used. The code generated for these model elements is part of the Adaptive AUTOSAR behavior code.

Related topics

Basics

Basics on Code Artifacts generated by TargetLink (📖 TargetLink Adaptive AUTOSAR Modeling Guide)
Basics on SIL Simulation for Adaptive AUTOSAR Models (📖 TargetLink Adaptive AUTOSAR Modeling Guide)
Modeling Communication as Defined by Adaptive AUTOSAR (📖 TargetLink Adaptive AUTOSAR Modeling Guide)
Overview of Supported Modeling Styles for Adaptive AUTOSAR (📖 TargetLink Adaptive AUTOSAR Modeling Guide)
Specifying Data Types (📖 TargetLink Adaptive AUTOSAR Modeling Guide)

Multirate Code Generation

MULTIRATE

Access

In the MATLAB Command Window, enter the following command:

```
tl_demos multirate
```

You can also enter **`tl_demos`** to access all available demos in the MATLAB Help browser. Select a demo and click **Open this example** at the top right.

Description

The **`Multirate`** demo model shows you tasks running at different sample times.

Task partitioning

- Automatically determined by sample rate
- Explicitly specified with the Task block

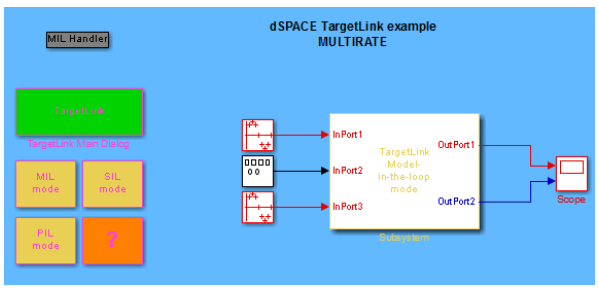
Messages for intertask communication

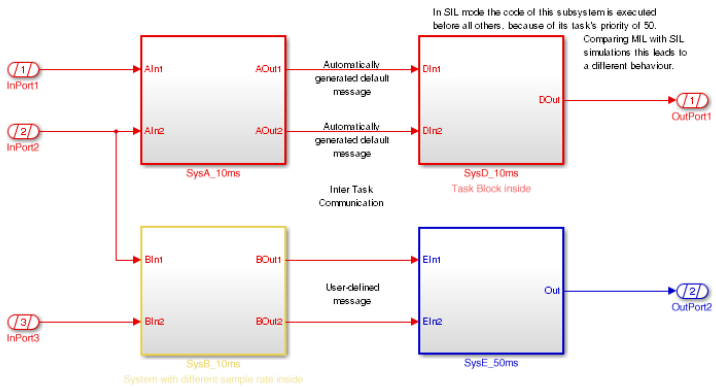
- Automatically, unprotected
- Explicitly protected

Protection of critical sections

- By means of the **Critical Section** block

The illustration below shows the root level of the Multirate demo model.

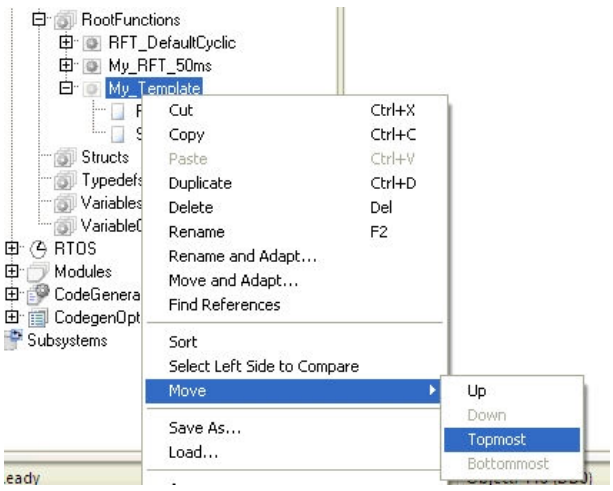




Root function template

Besides the predefined templates provided with the TargetLink Data Dictionary, you can specify your own templates to filter out specific subsystems in your multirate model. To do so, you have to add templates in the TargetLink Data Dictionary Manager.

TargetLink evaluates the templates in the order of their appearance in the TargetLink Data Dictionary. After TargetLink has found a template that filters out a subsystem, it does not evaluate the remaining templates below this one. You have to move a template to the topmost position in the RootFunctionTemplates sub object if you want TargetLink to evaluate it first.



Multirate subsystems (RTOS)

TargetLink allows you to create multirate models containing blocks with different sample times. Multirate models let you generate applications which comply with real-time operating systems (RTOS).

During the process of developing multirate models and generating production code for multitasking applications, RTOS data is read from and written to the following objects in the DD object tree.

- `/Pool/RTOS`
- `/Subsystems/RTOS`

Related topics

Basics

Mapping Multirate to Multitasking with TargetLink (📖 TargetLink Multirate Modeling Guide)

HowTos

How to Add a Root Function Template and Specify Its Properties (📖 TargetLink Multirate Modeling Guide)

Component-Based Development

Where to go from here

Information in this section

MODEL_REFERENCING..... 181
CG_FROM_DD..... 185
REPLACEABLE_DATA_ITEMS..... 187
MODULAR_DEVELOPMENT..... 192

MODEL_REFERENCING

Access

In the MATLAB Command Window, enter the following command:

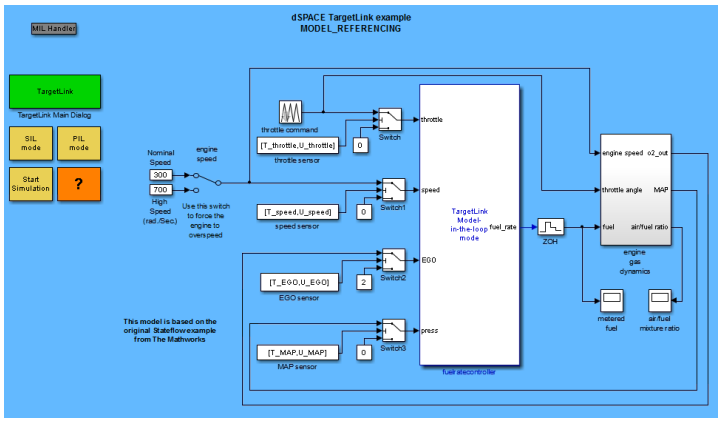
```
tl_demos model_referencing
```

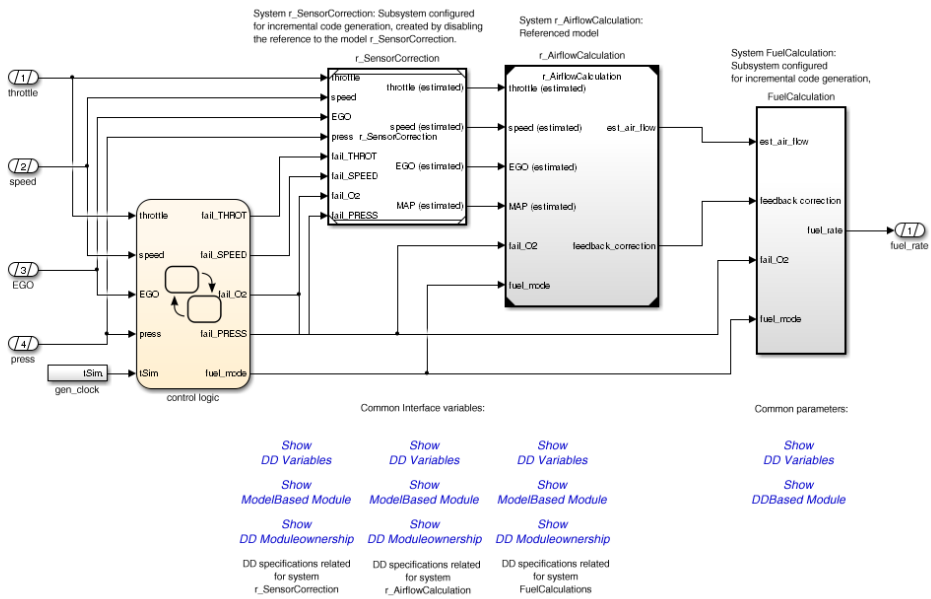
You can also enter **`tl_demos`** to access all available demos in the MATLAB Help browser. Select a demo and click **Open this example** at the top right.

Description

The **`model_referencing`** demo model shows you a fault-tolerant fuel control system using model referencing.

The illustration below shows the root level of the `model_referencing` demo model.



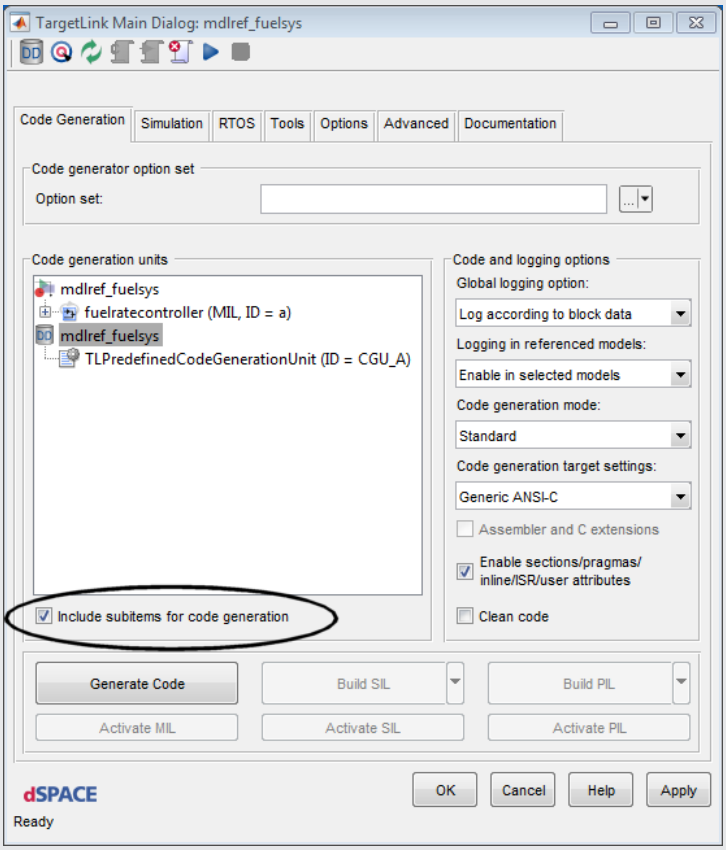


Additional information The TargetLink subsystem fuel rate controller is partitioned into the following sub items:

- FuelCalculation: Subsystem configured for incremental code generation
- r_SensorCorrection: Subsystem configured for incremental code generation, created by disabling the reference to the model r_SensorCorrection
- r_AirflowCalculation: Referenced model

Note

The demo model provides subsystems configured for incremental code generation and referenced models. Therefore, code must be generated from the innermost subsystem to the outermost one. You can force the Code Generator to do this via the Include subitems for code generation checkbox or via the following API function: `tl_generate_code(...,'IncludeSubItems','on');` However, the Include subitems for code generation data is not stored at the block, i.e., the information is lost when you close the block. This is why an error messages occurs if you start code generation via the Menu in the Simulink Editor.



Open the TargetLink Main Dialog to show the systems hierarchy. Functions related to the selected system are available via the context menu.

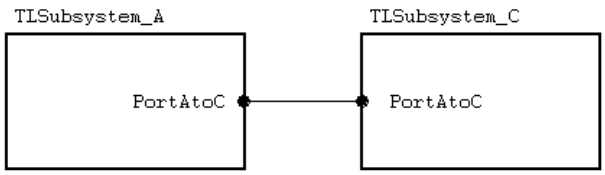
Common interface variables: The functions generated for the systems: r_SensorCorrection, r_AirflowCalculation and FuelCalculation exchange data via the global interface variables. The definition of the interface variables is generated into the module containing the definition of the function using these variables as its outputs. The system modeling the function is specified as an owner of this module.

Common parameters: The definitions of the calibratable variables defined in blocks residing in systems r_SensorCorrection and r_AirflowCalculation are

generated into one common dd-based module CalParameters.c This module is to be generated straight from the Data Dictionary. As a CodeGenerationUnit the predefined one is used.

Interface variables and modules sharing

It is usually helpful to split large TargetLink models into smaller, i.e., more manageable components such as TargetLink subsystems, referenced models, or subsystems configured for incremental code generation. These model components are typically connected via signal lines and therefore use identical interface variables.



TargetLink supports the splitting of models.

Parameter sharing

It is advantageous to have the functional code separated from the variable code (variable declaration/definition), because:

- Parameters change more frequently than functional code. For example, the functional code already exists as `*.obj`, and the parameters are still subject to change.
- Functional code does not vary as variable code usually does.
- Updating an ECU with new calibration data by programming the ECU's *flash memory* requires the parameters' code to be stored in a specific module.
- For inserting external functional code, the missing variable declarations/definitions must be provided by TargetLink or the TargetLink Data Dictionary.

Model referencing

Simulink model referencing is a mechanism that lets you include models in other models by using Model blocks. Each Model block represents a reference to another model and displays the inputs and outputs of the referenced model.

When you arrange large models by means of model referencing you benefit from advantages such as:

- Modular development

  Each referenced model can be implemented independently of the model that references it.
- Inclusion by reference

  The functionality of a model can be included in one or several code generation units without having to make redundant copies.

- Incremental loading

  Simulink does not load a referenced model on the host PC until you update or simulate the model in which it is used. This feature, called incremental loading, speeds up the loading process of a root model in a model referencing hierarchy.

Related topics

Basics

Basics on Model Referencing (📖 TargetLink Customization and Optimization Guide)
Decomposing Models for Distributed Development (📖 TargetLink Customization and Optimization Guide)

CG_FROM_DD

Access

In the MATLAB Command Window, enter the following command:

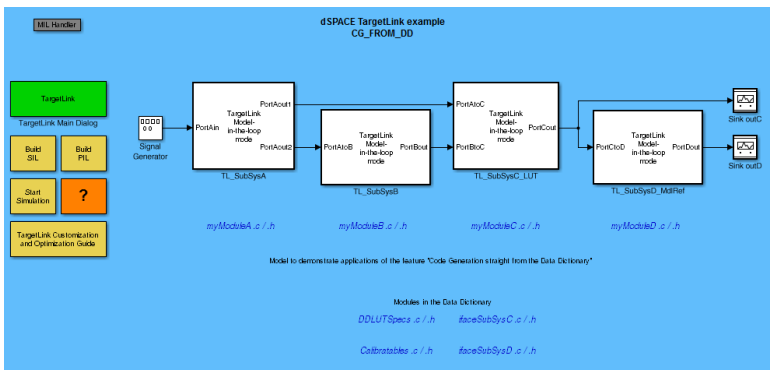
```
tl_demos cg_from_dd
```

You can also enter **`tl_demos`** to access all available demos in the MATLAB Help browser. Select a demo and click **Open this example** at the top right.

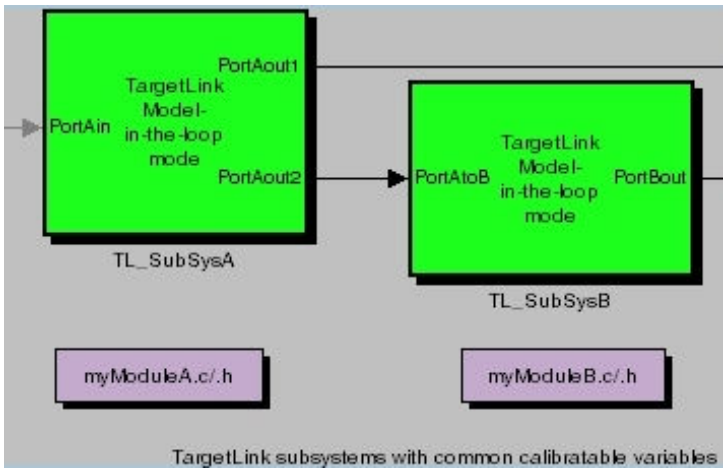
Description

The **`cg_from_dd`** demo model shows you applications of the feature *Code Generation straight from the TargetLink Data Dictionary*. Distributed function development is possible by partitioning models. TargetLink provides different ways to do this, i.e., it enables you to partition your models using model referencing, let other function developers work on subsystems and perform code generation for them incrementally, or generate component-based production code straight from the **TargetLink Data Dictionary**.

The *CG_from_DD* demo model demonstrates applications of the feature *Code Generation straight from the Data Dictionary*.



Distributed function development without model referencing



The TL_SubSysA and TL_SubSysB subsystems make you familiar with *distributed function development without model referencing*.

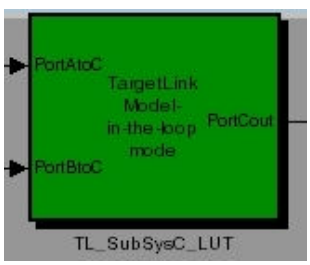
- For TL_SubSysA, code is generated into the model-based `myModuleA` module.
- For TL_SubSysB, code is generated into the model-based `myModuleB` module.
- The module ownership for these subsystems and modules (and others) is explicitly specified in the TargetLink Data Dictionary.

The TargetLink subsystem output variables are located in specific modules (see the TL_SubSys_IOs variables group in the TargetLink Data Dictionary).

The gain parameters (myGainA, myGainB) are collected in the DD-based module `Calibratables`:

- You can generate code for this module straight from the TargetLink Data Dictionary.
- You can generate an A2L file straight from the TargetLink Data Dictionary, including the look-up table variables from TL_SubSysC_LUT.

Look-up table specifications



The TL_SubSysC subsystem makes you familiar with *look-up table specifications*:

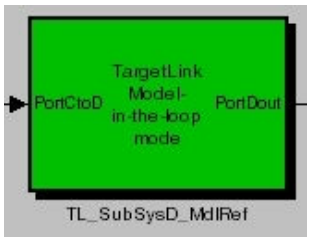
- LUT2D_SpecInBlock block: The look-up table is specified in the block dialog.
- LUT2D_SpecInDD block: The look-up table is specified in the TargetLink Data Dictionary.

The model-based module `LUTFunction` contains the functions of both tables. The DD-based module `DDLutSpecs` contains all variables.

After a **Build** node is created in the **TargetLink Data Dictionary**, you can generate an A2L file for the **Parameters_CGU** DD code generation unit object. The file contains the following variables:

- Variables related to the `Calibratables` module
- Variables related to the `DDLUTSpecs` module including LUT information

Distributed function development using model referencing



The **TL_SubSysD** subsystem makes you familiar with *distributed function development using model referencing*:

- The function is available in stand-alone form (`heaviside_stand_alone.slx`) and also in integrated form (`CG_from_DD.slx`).
- The interface code is separated from the function code. The `Ext_Heaviside_Fcn` module holds the function code, and the `Ext_Heaviside_Fcn_Interface` module holds the global interface.

You can easily check for interface violations in the integrated model.

REPLACEABLE_DATA_ITEMS

Access

In the MATLAB Command Window, enter the following command:

```
tl_demos Replaceable_Data_Items
```

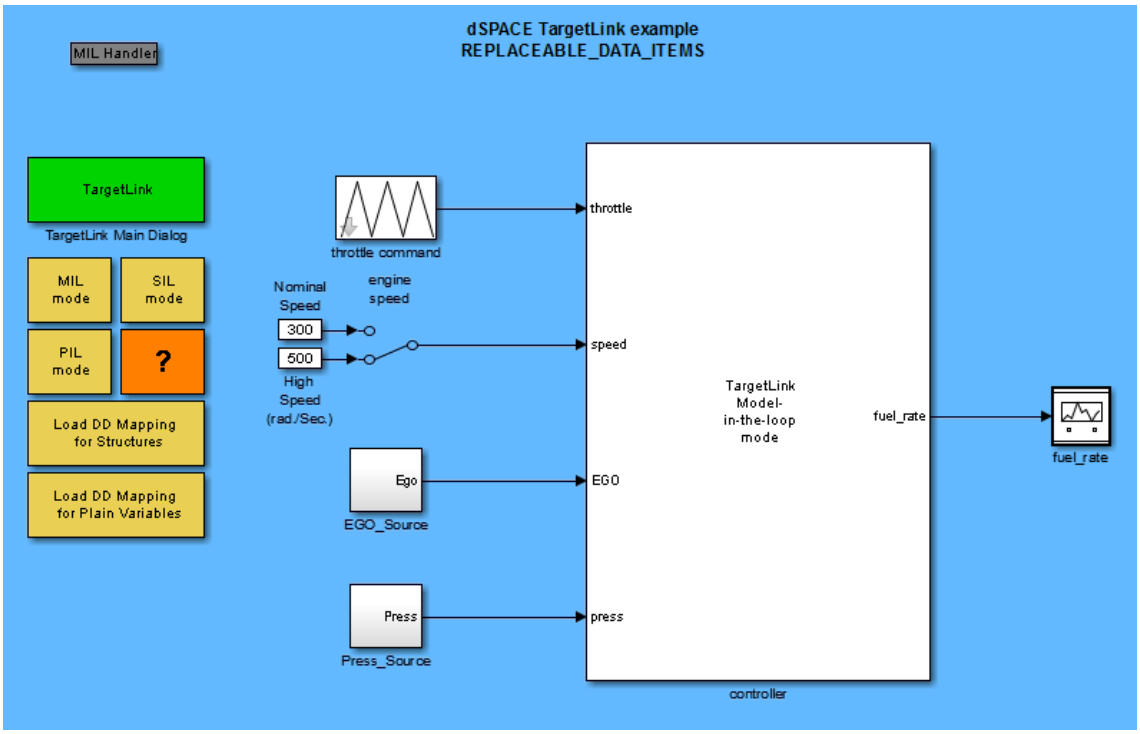
You can also enter **`tl_demos`** to access all available demos in the MATLAB Help browser. Select a demo and click **Open this example** at the top right.

Description

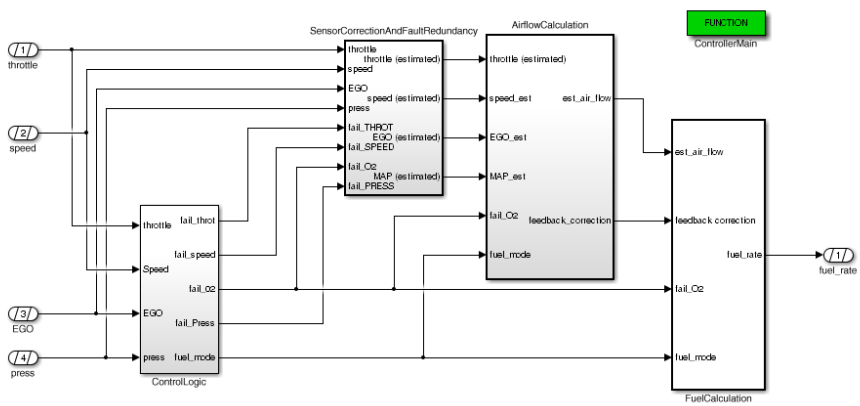
The `replaceable_data_items` demo model shows you modular, component-based development using Replaceable Data Items (RDI). Basically, RDIs serve two different purposes:

- *Efficient Interfaces*: Interfaces at the borders of incrementally generated subsystems or referenced models are implemented very efficiently by using RDI objects that are implemented as global macros and then by successive code generation which maps global macros to implementation variables. The RDI objects help maintain code efficiency when a large model is partitioned into smaller incrementally generated systems.

- *Flexible Parameters*: RDI objects are implemented as global macros in the generated production code. You can map them very flexibly to different implementation variables, such as calibration parameters and measurement signals in different contexts/projects, without having to modify the model or the production code. Instead, you can define an RDI mapping to specify how to implement RDI macros: e.g., as plain variables, components of a structure, etc.These variables can then be generated from the Data Dictionary.



The entire functionality is subdivided into 4 subsystems (shown in light blue) which are configured for incremental code generation and developed by using RDIs for input and output variables, calibration parameters and measurable signals. The RDI specifications reside in `/Pool/CodeGenerationUnitInterfaces` in the DD.



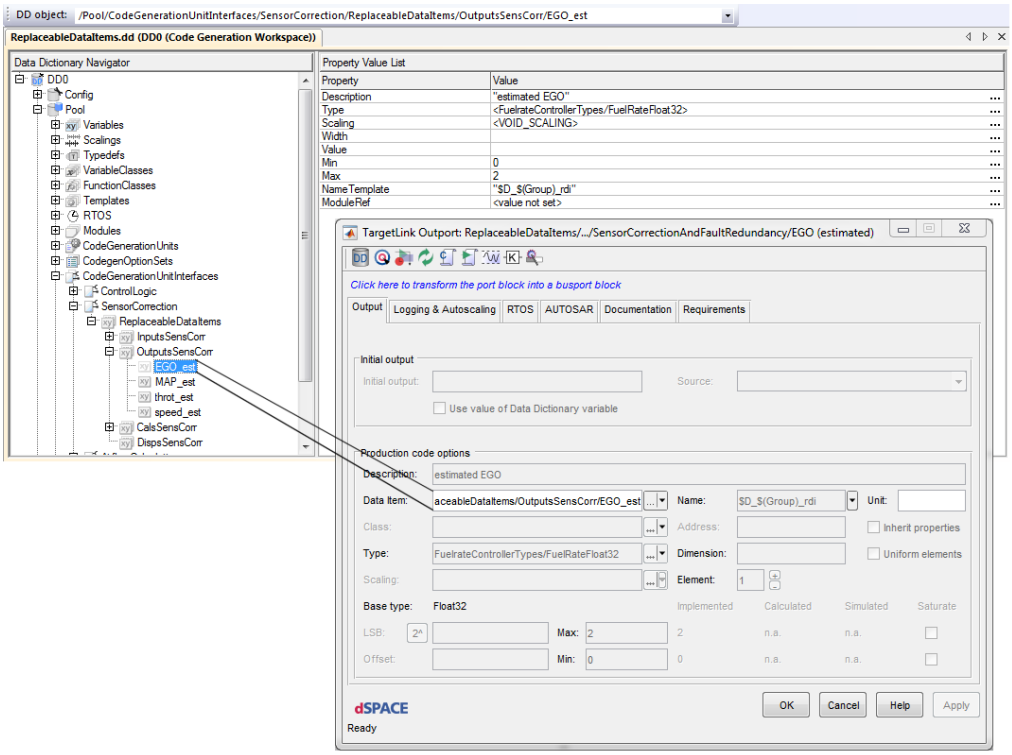
Interfaces between incrementally generated systems are implemented efficiently and there is still the flexibility to map all RDIs to different implementation variables at a later stage while leaving the model and the function code unchanged. Implementation variables for the RDIs of calibration parameters and measurable signals are generated along with the TargetLink **controller** subsystem on the basis of the user-defined mapping in `/Pool/DataItemMappings`).

Additional information Required steps for code generation/SIL/PIL simulation:

1. Generate code for the incremental systems first (before you generate code for the TargetLink controller Subsystem).
2. Double-click on one of the yellow buttons on the left to load a mapping of RDIs for Cals/Disps to different variables.
3. Generate code or build the host for the controller subsystem. Depending on the mapping from step 2, Cals/Disps are either plain variables or components of a structure. Model/code of the incrementally generated systems from step 1 remain unchanged.

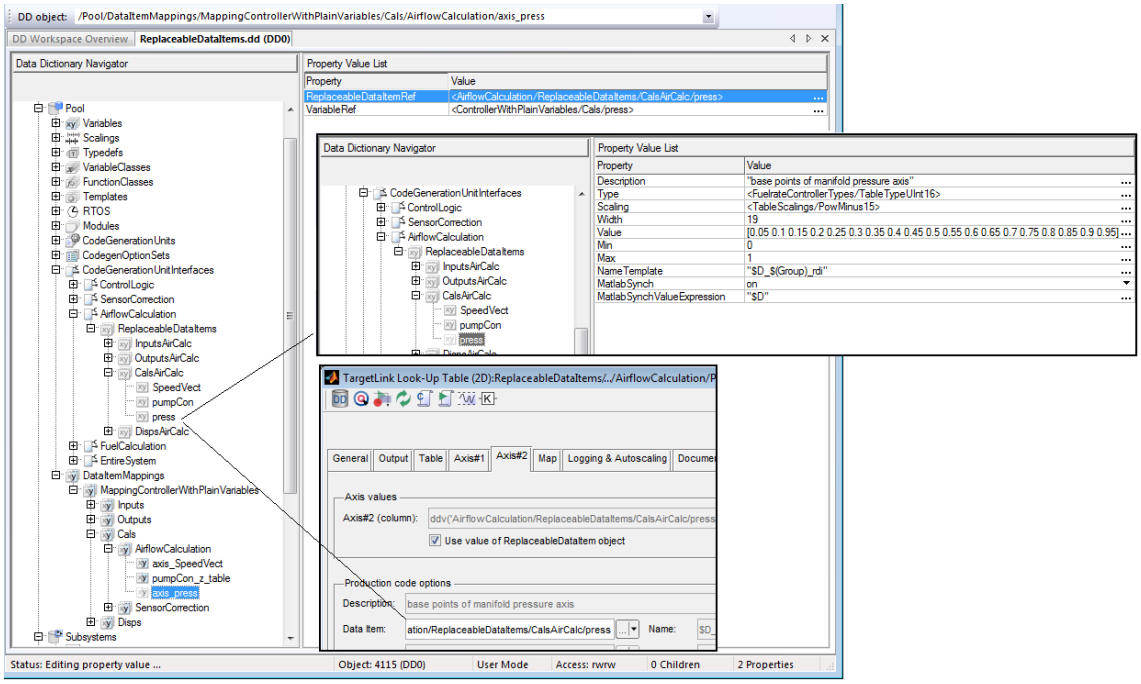
Modeling efficient interfaces

With efficient interfaces you can reduce the number of RAM variables that are usually required for the interface to referenced models and to subsystems configured for incremental code generation. Therefore, and because this interface type requires only little specification work in the TargetLink Data Dictionary (only the inport and outport blocks must reference RDI objects), this interface type is called efficient. Moreover, the mapping code is generated as part of the production code generated for the integration model.

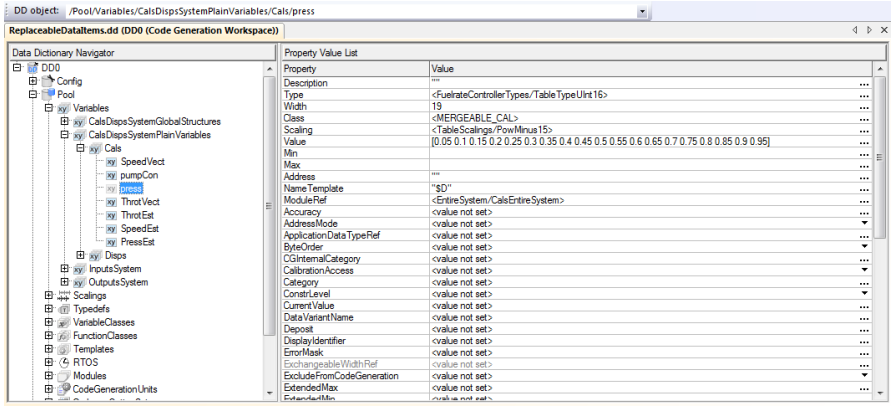


Modeling flexible parameters

Flexible parameters are abstract variables that let you map arbitrary block parameters by using RDI objects to specific implementations at a later stage. Typical use cases are referenced models that are reused across several different projects with different model parameters for each project. For this *explicit modeling*, you have to specify a ReplaceableDataItem (RDI) and a DataItemMapping (DIM) object in the TargetLink Data Dictionary.



The RDI objects' mapping to DIM objects must be unique and works only for Pool variables that are explicitly specified in the DD. The mapping code is generated as part of the production code generated for the integration model.



Related topics

Basics

Basics on Component-Based Development Using Abstract Interfaces (RDI Objects) (📖 TargetLink Customization and Optimization Guide)

HowTos

How to Map RDI Objects to Specific Implementation Variables (📖 TargetLink Customization and Optimization Guide)
How to Model Efficient Interfaces Using RDI Objects (📖 TargetLink Customization and Optimization Guide)
How to Model Flexible Parameters Using RDI Objects (📖 TargetLink Customization and Optimization Guide)

MODULAR_DEVELOPMENT

Access

In the MATLAB Command Window, enter the following command:

```
tl_demos modular_development
```

You can also enter **`tl_demos`** to access all available demos in the MATLAB Help browser. Select a demo and click **Open this example** at the top right.

Description

The **`modular_development`** demo model shows you how the artifacts generated for incremental code generation units, such as the production code or HTML documentation, can be integrated and used in the integration model. This demo model demonstrates how different developers can develop and generate code for their developer frame models and test them separately. Afterwards, this modular code is integrated into an additionally generated C code integration frame resulting from an integration model without the need to generate the modular code again. This is achieved by explicitly defining code generation locations in the model DD and then transferring this information to the integration DD of the integration model. So finally the integrator is able to use the artifacts already generated by the developers.

Incremental code generation means that a code generation unit is decomposed into several smaller code generation units that can be developed and tested separately.

In TargetLink, code generation units are:

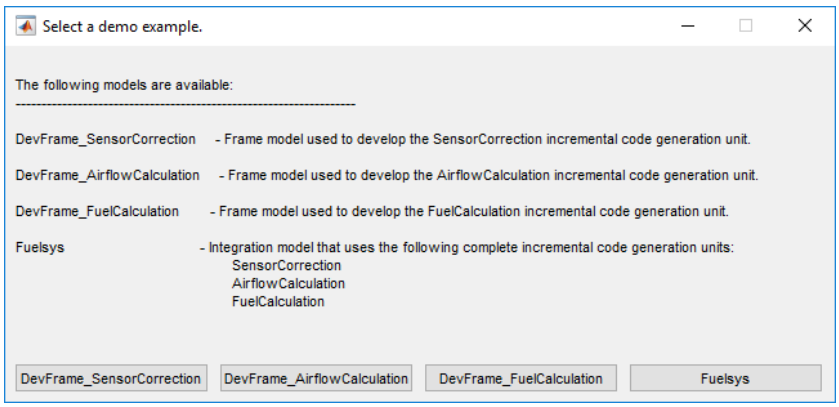
- TargetLink subsystems
- Referenced models
- Subsystems configured for incremental code generation
- DD **CodeGenerationUnit** objects

The demo consists of three developer frame models and one integration model. Every model has its own DD. The code generation unit modeled by the TargetLink fuelratecontroller subsystem is decomposed in three incremental code generation units. You start with the three developer frame models:

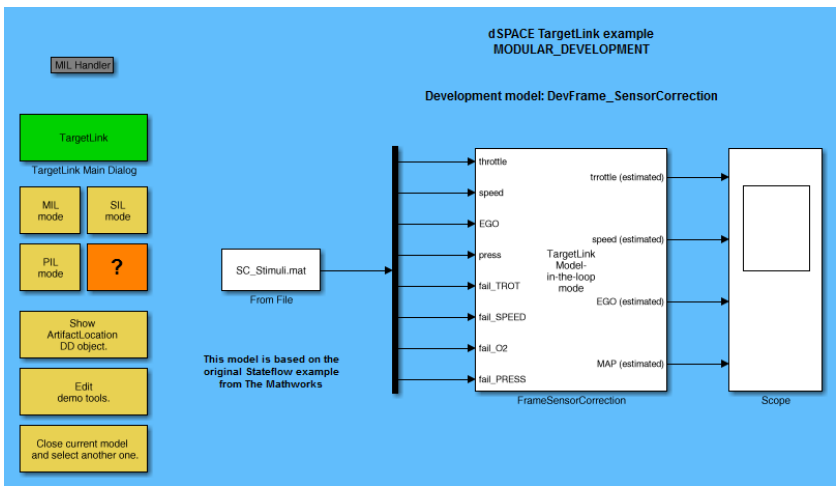
- DevFrame_SensorCorrection - modeled by a subsystem configured for incremental code generation
- DevFrame_AirflowCalculation - modeled by a subsystem configured for incremental code generation
- DevFrame_FuelCalculation - modeled by a referenced model that is also an incremental CGU

In each of the three developer frame models, you generate code for the incremental code generation unit. This leads to the artifacts being generated into the specified locations according to the ArtifactsLocation specifications in the DD. Afterwards, you can integrate the complete incremental code generation unit (incremental CGU) into the Fuelsys integration model.

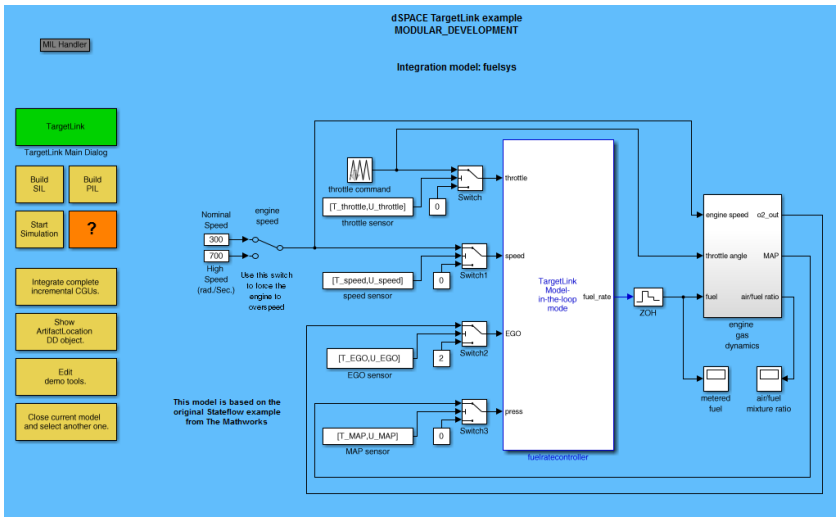
Start the demo model and select one of the developer frame models:



The following screenshot shows the root level of one of the developer frame models, `DevFrame_SensorCorrection`:



The following screenshot shows the root level of the Fuelsys integration model:



Incremental code generation for modular development

Usage of the demo model To generate code for the developer frame models and to use it in an integration model, you have to perform the following steps:

- Click Build SIL at the root level of each developer frame model to generate code and to generate the DD subsystems metadata of each CGU.
- Click Integrate complete incremental CGU in the Fuelsys integration model. The `tlDemoIntegrateIncrCGUs` script in the `Tools` folder of the demo is started. The script integrates the incremental code generation units and their existing artifacts in the integration model's environment according to the specifications made in the DD ProjectFolder and DD FolderStructure objects.
- In the TargetLink Main Dialog clear the Include subitems for code generation checkbox to use the previously generated files. Then click Build SIL to produce code for the integration frame.

Tip

Before you start the code generation in the integration model, you can use the `tl_clean` API function to delete TargetLink-generated files from the working directory of the integration model.
This is required if you already generated code for the integration model while the following conditions were met:

- The Include subitems for code generation checkbox was selected.
- The artifact locations had not yet been transferred from the developer frame model DDs to the integration model DD.

Structure of the frame models Each frame model consists of a TargetLink subsystem that contains the incremental code generation units (a referenced model or a subsystem configured for incremental code generation). The TargetLink subsystem is required for code generation and consistency checks.

Each frame model also has a `<CGU name>_Archive` folder with the following subfolders:

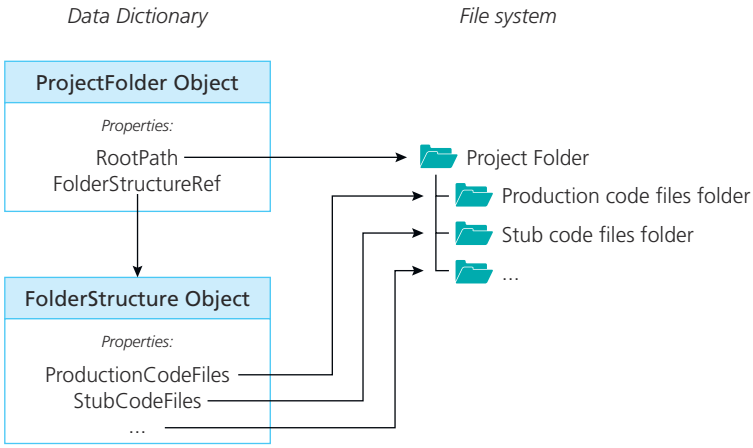
- `Config` contains a script for configuring the structure of the resulting CGU.
- `Library` contains either the referenced model or a Simulink library with the subsystem configured for incremental code generation.

Separating the artifact locations TargetLink creates several files (artifacts) during code generation and when building the simulation application. The artifacts necessary for the integration of the incremental CGU should be separated from the ones not necessary (e.g., for the TargetLink subsystem).

- In this demo model, the files necessary for the integration are located in `<CGU_Name>_Archive\GeneratedFiles`.
- The files generated for the TargetLink subsystem are located in `.\Temp`.

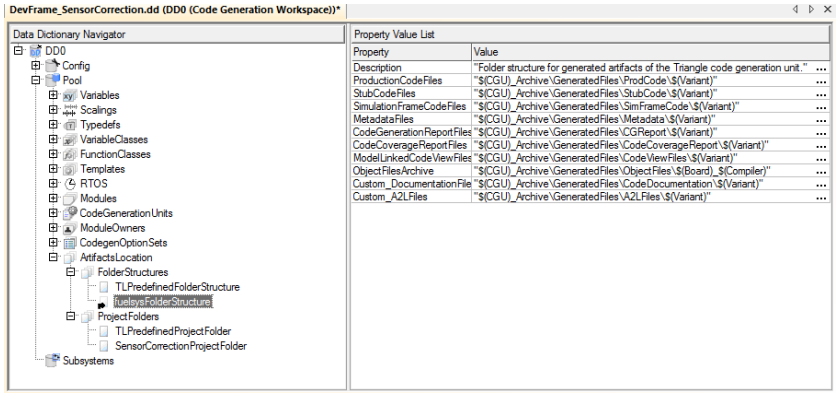
You can specify their location in the file system via the DD ProjectFolder and FolderStructure objects that are located in `/Pools/ArtifactsLocations`.

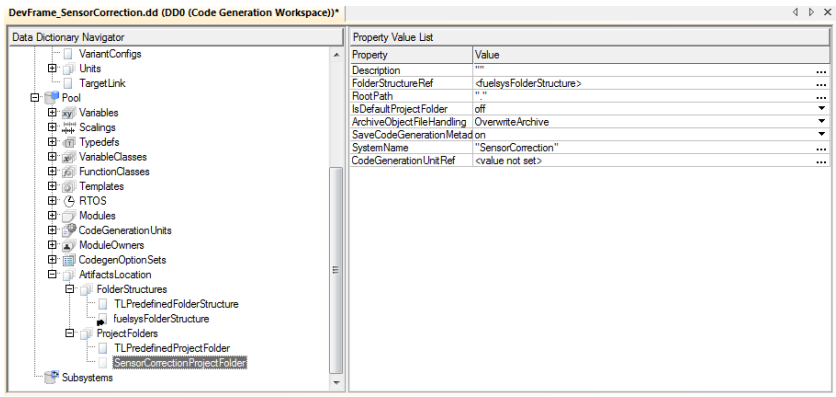
- In the DD ProjectFolder object, you specify the root path of the CGU-related artifacts.
- In the DD FolderStructure object that is referenced by the DD ProjectFolder object, you specify the locations of the artifacts relative to this root path.



On the root level of the models, double-click **Show ArtifactLocation DD object** to access them.

To extract the artifacts required for the integration, the incremental CGU has its own project folder and folder structures in the file system. The TargetLink subsystem of the frame model use the DD TLPredefinedProjectFolder and DD TLPredefinedFolderStructure objects and its artifacts become obsolete. Refer to the following screenshots:

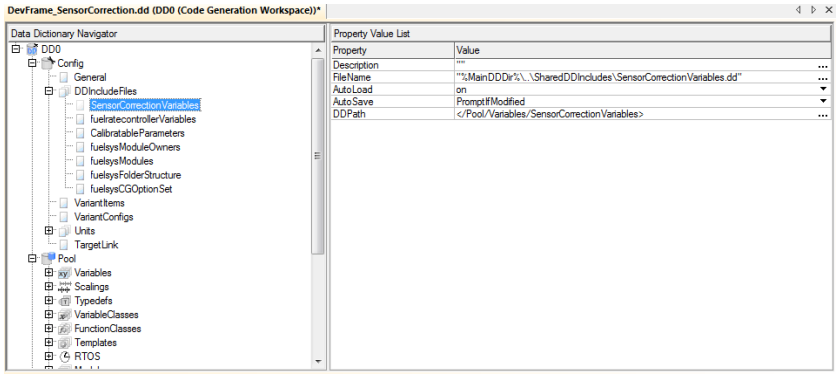




Note

When specifying the artifact locations for incremental code generation, you must set the `SaveCodeGenerationMetadata` property to on to facilitate the integration of the different model components within the integration model.

Specifying shared DD includes Complete Data Dictionaries can be augmented with DD IncludeFile objects, which specify that additional partial DD files are loaded when the complete Data Dictionary is opened. This is called DD file inclusion. Data in partial DD files can thus become part of a project, which lets you distribute and reuse settings, for example, data type specifications and identical Code Generator option settings. DD IncludeFile objects are located in `/Config/DDIncludeFiles` and can be grouped, which means that they can be arranged in a hierarchical order. The following screenshot shows the DD IncludeFile objects of the fuelsys integration model.



Additional demo tools You can access additional demo tools on the root level of the fuelsys integration model by clicking **Edit demo tools**. The tools are helpful scripts demonstrating functions like incremental documentation generation.

The following scripts are available:

- `tlDemoGenA2L` generates an A2L file for the specified code generation unit. In addition, it gets the location at which to generate the A2L file from the DD ProjectFolder and DD FolderStructure objects.
- `tlDemoGenDoc` incrementally generates HTML documentation of the TargetLink production code and gets the destination directory from the DD ProjectFolder and DD FolderStructure objects.
- `tlDemoIntegrateIncrCGUs` integrates the incremental code generation units and their artifacts into the integration model's environment by using the settings of the DD ProjectFolder and FolderStructure objects.

Refer to the detailed descriptions in the scripts.

Related topics

Basics

Basics on Specifying the Location of Artifacts Generated by TargetLink (📖 TargetLink Customization and Optimization Guide)

HowTos

How to Include Partial Data Dictionary Files (📖 TargetLink Data Dictionary Basic Concepts Guide)

References

tl_clean (📖 TargetLink API Reference)

Code Testing

Where to go from here

Information in this section

ONLINE_PARAMETER_MODIFICATION..... 199
SIGNAL_INJECTION..... 203
REAL_TIME_PLATFORM_SUPPORT..... 206
VECU_EVENT_SIMULATION..... 213

ONLINE_PARAMETER_MODIFICATION

Access

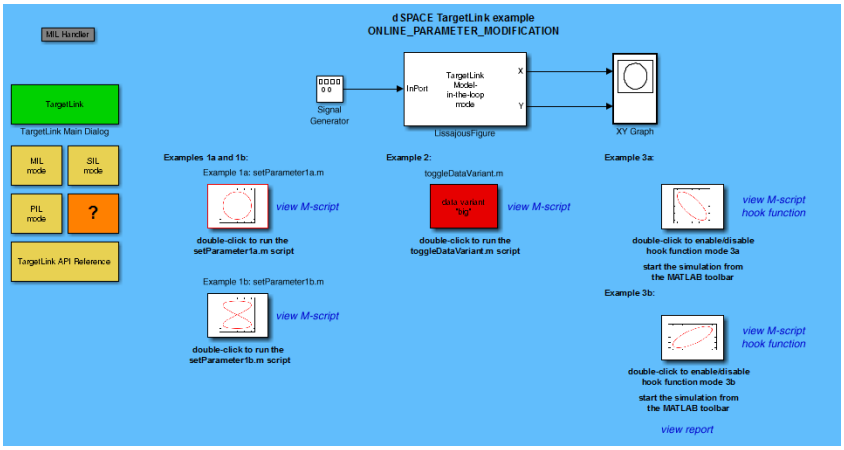
In the MATLAB Command Window, enter the following command:

```
tl_demos parameter_modification
```

You can also enter **`tl_demos`** to access all available demos in the MATLAB Help browser. Select a demo and click **Open this example** at the top right.

Description

The `online_parameter_modification` demo model shows you how to change calibration parameter values directly in memory without having to regenerate or recompile the code. The following illustration shows the root level of the online_parameter_modification demo model.



Additional information This model and the associated scripts show how to run tests for different calibration parameter values in SIL/PIL mode without having to generate or compile the code again. Instead, parameter values can be modified directly in memory before each simulation run in SIL/PIL mode is started. The tlSimInterface API function is used for this purpose. Example 1a and Example 1b demonstrate the use of the tlSimInterface API function to run a full simulation including the modification of parameters in memory in SIL/PIL mode. Double-click the icons on the left to run the simulations and inspect the scripts.

The M-scripts show how to:

- Connect to a simulation platform
- Reset a simulation application to initialize the variables properly
- Call the 'Restart Function' to initialize variables there
- Write parameters to memory that shall be modified (Online parameter modification)
- Start the simulation
- Disconnect from the simulation platform

Example details:

- Example 1a uses block variables whereas Example 1b makes use of DD variable objects. Example 1a and Example 1b differ only in the specification of the parameters to be modified. If a simulation run is started from the MATLAB tool bar instead of the scripts, MIL and SIL/PIL simulation results will differ because TargetLink's simulation S-Function automatically resets all variables to their initial values thereby resetting the modified parameters. Example 3 demonstrates how to avoid this.
- Example 2 essentially demonstrates the same steps as Example 1a and Example 1b. However, block variables are not modified directly, here. Instead, the tlSimInterface API function is used to modify the value of a variant switch variable in memory, thereby switching a variant configuration. Double-click on the icons on the left to run a simulation or inspect the script.

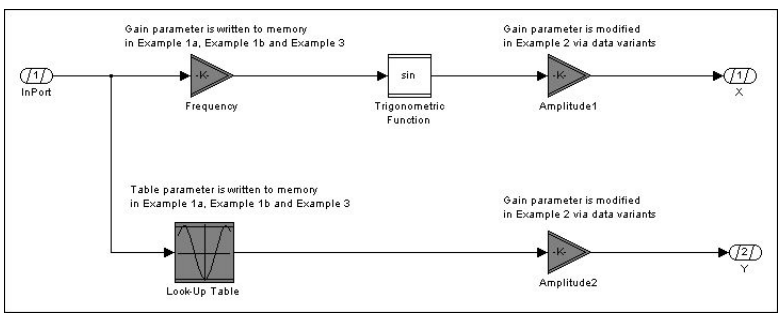
- Example 3a and 3b demonstrates how a pre_prodcode_sim_hook -function can be used to modify parameters a lot easier than in Example 1a, Example 1b and Example 2. Simulation runs can be started from the Simulink toolbar as usual. Note: - Reset of an application and call of the 'Restart Function' for proper variable initialization are carried out by TargetLink before the hook script is called. - The use of the pre_prodcode_sim_hook has the advantage that simulations can be started from the MATLAB tool bar as usual. - Double-click on the icons on the left to enable the hook script (a red foreground color signals the activation). Example 3a demonstrates how to modify parameters with the tlSimInterface API function. Example 3b demonstrates how to modify parameters with the tlSimParameterUpdate API function which must call from a pre_prodcode_sim_hook -function. After simulation a report is generated, which lists the updated parameters.

Online Parameter Modification

TargetLink allows you to modify parameter values of a simulation application without regenerating or recompiling the code. You can modify parameter values in a hook script by using the **`tlSimParameterUpdate()`** and **`tlSimInterface()`** API functions. Furthermore, you can use the **`tlSimInterface()`** API function also in M files to access the simulation application.

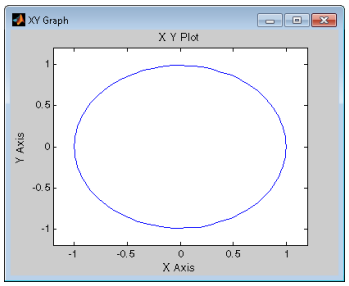
You can use online parameter modification to change the parameter values of the simulation application downloaded to the evaluation board (physical or virtual) for PIL mode or loaded to the memory of the Host PC for SIL mode. You can change initialized values in the simulation application to perform the simulation with parameter values different from those specified in the model.

The illustration below shows the **LissajousFigure** subsystem of the **`online_parameter_modification`** demo.

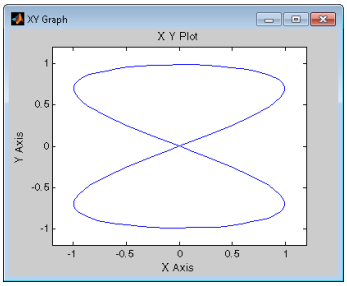


The model represents an oscilloscope in X-Y mode with two phase-shifted sinusoid inputs.

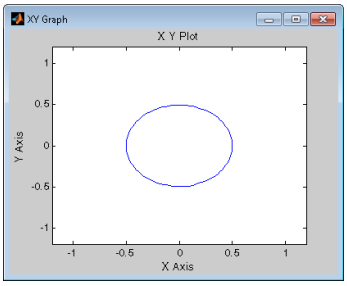
The simulation of this model creates a Lissajous figure in a plot window. With the default parameter values, the curve is a circle.



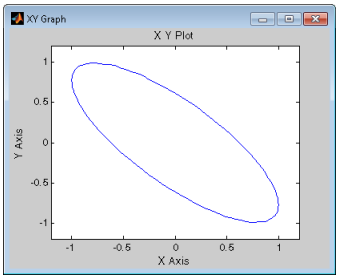
- The block displays the curve as a figure-8 shape.



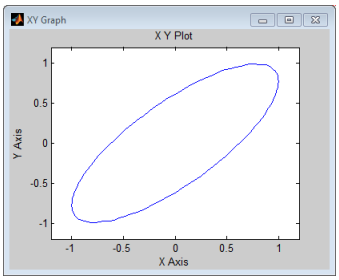
- The block creates a circle with a smaller radius.



- The block displays the curve as a left-slanted ellipse.



- The block displays the curve as a right-slanted ellipse.



Related topics

Basics

Basics on Modifying Parameter Values for Simulation (📖 TargetLink Preparation and Simulation Guide)

Examples of Implementing Online Parameter Modification (📖 TargetLink Preparation and Simulation Guide)

SIGNAL_INJECTION

Access

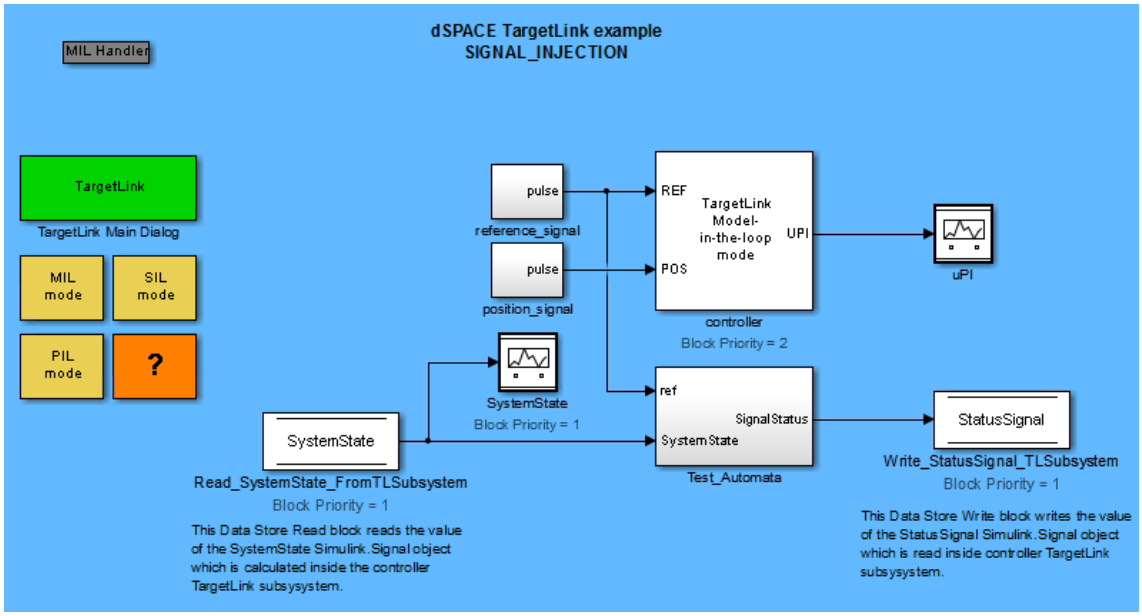
In the MATLAB Command Window, enter the following command:

```
tl_demos signal_injection
```

You can also enter **`tl_demos`** to access all available demos in the MATLAB Help browser. Select a demo and click **Open this example** at the top right.

Description

The signal_injection demo model demonstrates how to *inject* the value of a stimulus signal from outside of a TargetLink subsystem into a TargetLink Subsystem using Simulink.Signal objects without a connection line. For example, this lets you perform sophisticated tests where stimulus values are not predetermined a priori but calculated during each time step based partially on internal values of the control system. The illustration below shows the root level of the signal_injection demo model. This demo model also shows how to *tunnel* the value of a signal from inside of a TargetLink subsystem to the outside without a connection line.

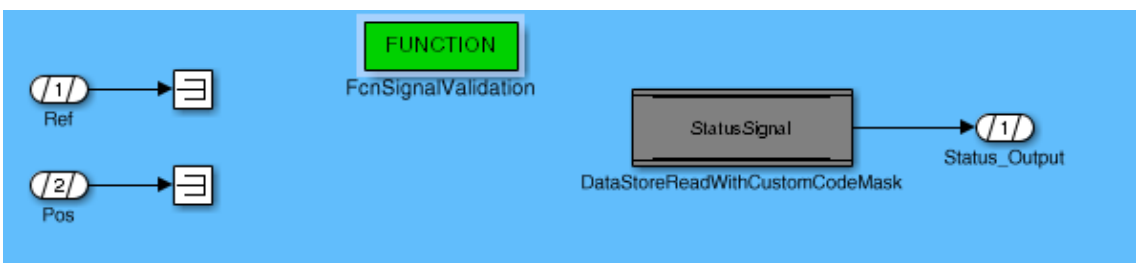


Additional information This demo model demonstrates a TargetLink code testing feature which allows you to inject the value of a stimulus signal from outside of a TargetLink subsystem into a TargetLink Subsystem using Simulink.Signal objects without a connection line. TargetLink provides the mechanism for SIL/PIL mode whereas standard Simulink features are used for MIL mode. The mechanism is very helpful to stimulate the outputs of subsystems for which only stub implementations exist during tests.

The demo model also demonstrates a code testing feature which allows you to tunnel the value of a signal from the inside of a TargetLink subsystem to the outside without a connection line. TargetLink provides the mechanism for SIL/PIL mode whereas standard Simulink features are used for MIL mode. The mechanism is very helpful for performing sophisticated tests where stimulus values are not predetermined a priori but calculated during each time step based partially on internal values of the control system.

Injecting signals

The new value for the **Status_Output** block of the subsystem is injected from outside the TargetLink subsystem. For that purpose, the **DataStoreReadWithCustomCodeMask** block is used. Essentially, it is a Simulink **Data Store Read** block with a TargetLink Custom Code mask on top of it.



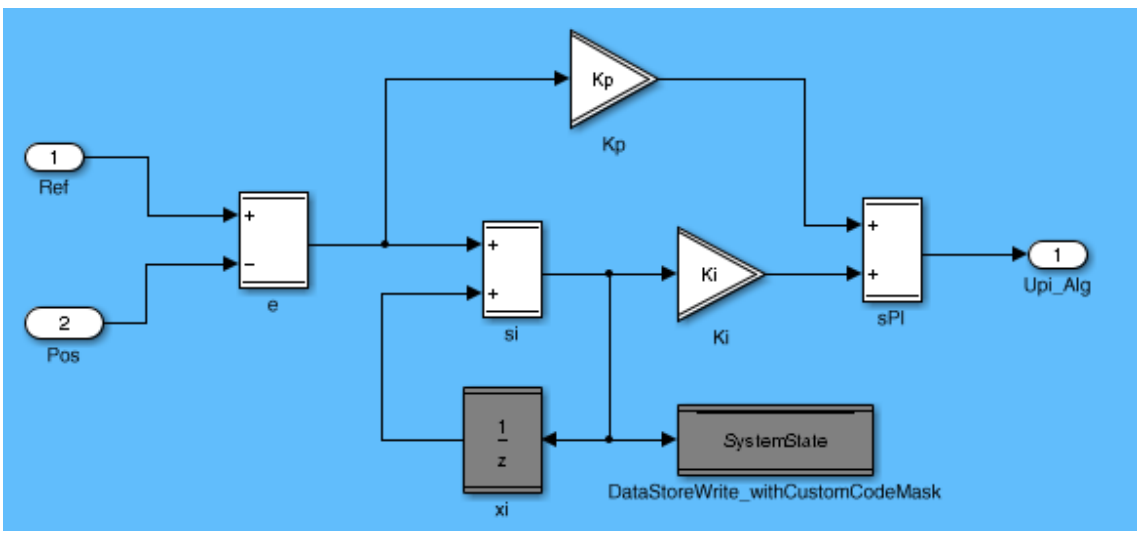
Additional information This is a stub subsystem whose Status_Output output signal is stimulated for testing purposes. TargetLink generates a stub implementation for the subsystem which is created in a separate module with an ownership specification that makes it stub code rather than production code. Naturally, on the real ECU the module is supposed to be replaced by the module with the real implementation.

The new value for the Status_Output block of the subsystem is injected from outside the TargetLink subsystem. For that purpose, the DataStoreReadWithCustomCodeMask block is used. Essentially, it is a Simulink Data Store Read block with a TargetLink Custom Code mask on top of it.

- In MIL mode (when the blocks are simulated), the Data Store Read functionality reads the current value from the StatusSignal Simulink.Signal object. The Simulink.Signal object is actually equivalent to a Data Store Memory block at the top hierarchy of the model. The Simulink.Signal object is created in the start.m script of the demo model.
- During SIL/PIL simulation mode, the current value is read from the StatusSignal C variable that is defined in the Custom Code mask of the DataStoreReadWithCustomCodeMask block. The block variables uses the TESTING/STUB_FOR_SIGNAL_INJECTION variable class which has the SimulationValueSource property set. As a consequence, TargetLink's simulation frame copies the current value of the StatusSignal Simulink.Signal object to the StatusSignal C variable in the code before the TargetLink subsystem is executed. Generally speaking, TargetLink's SIL/PIL simulation frame copies the values from Simulink.Signal objects to C variables if the SimluationValueSource property is set.

Tunneling signals

The new value for the xi block state variable is written to the SystemState Simulink.Signal object so that it can be tunneled out of the TargetLink subsystem. For that purpose, the DataStoreWrite_withCustomCodeMask block is used. Essentially, it is a Simulink Data Store Write block with a TargetLink Custom Code mask on top of it.



Additional information

- In MIL mode (when the blocks are simulated), the Data Store Write functionality writes the current value to the SystemState Simulink.Signal object. Outside the TargetLink subsystem, the value is read using a Data Store Read block. The Simulink.Signal object is actually equivalent to a Data Store Memory block at the top hierarchy of the model. The Simulink.Signal object is created in the start.m script of the demo model.
- During SIL/PIL simulation mode, the current value is written to the specified SystemState C variable that is defined in the Data Dictionary. The SystemState DD variable object has the SimluationValueDestination property set. As a consequence, TargetLink's SIL/PIL simulation frame copies the current value of the SystemState C variable to the SystemState Simulink.Signal object. Generally speaking, TargetLink's SIL/PIL simulation frame copies the values from C variables to Simulink.Signal objects if the SimluationValueDestination property is set.

Related topics

Basics

Basics on Injecting or Tunneling Signals During Simulation (📖 TargetLink Preparation and Simulation Guide)

REAL_TIME_PLATFORM_SUPPORT

Access

In the MATLAB Command Window, enter the following command:

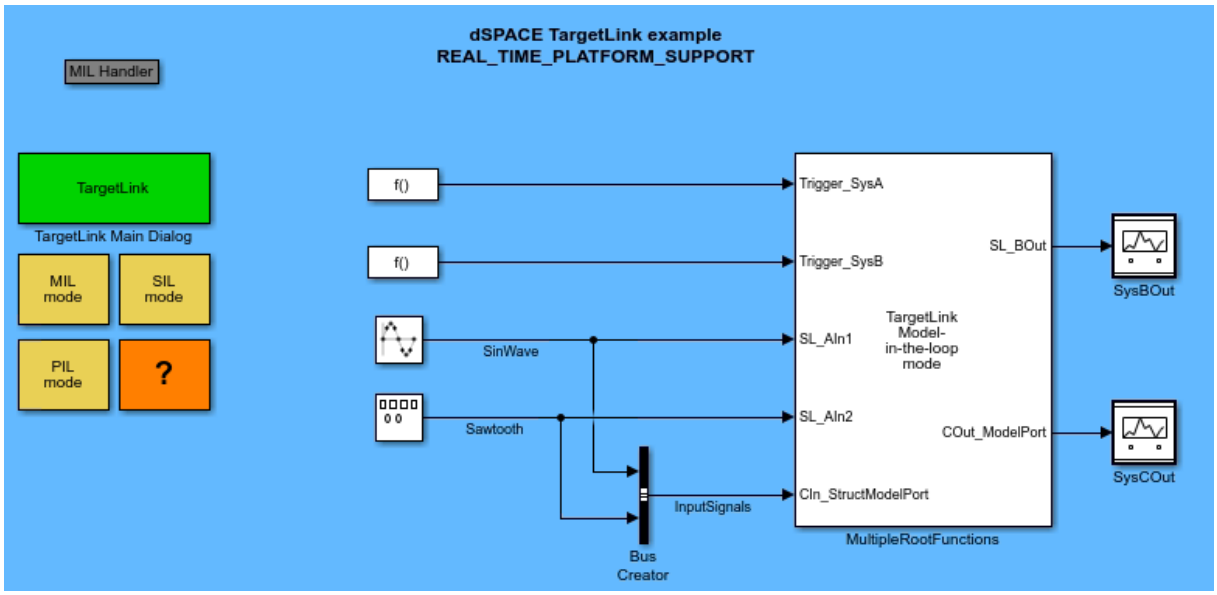
```
tl_demos real_time_platform_support
```

You can also enter **`tl_demos`** to access all available demos in the MATLAB Help browser. Select a demo and click **Open this example** at the top right.

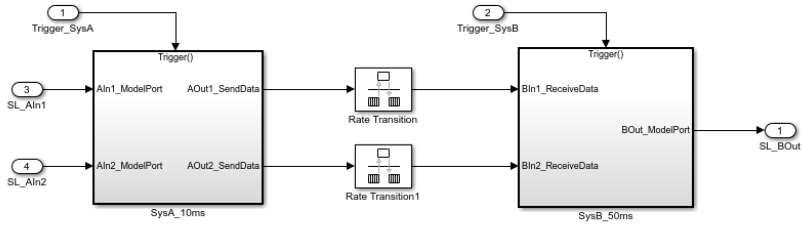
Description

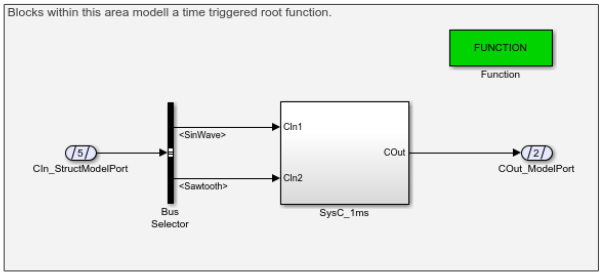
The `REAL_TIME_PLATFORM_SUPPORT` demo model shows you how to prepare a TargetLink subsystem for use as behavior model (Simulink implementation container (SIC)) in ConfigurationDesk. In particular, how to model its interface via model port variables, how to model multiple SIC runnable functions that can be connected to different tasks (asynchronous or time triggered) in ConfigurationDesk, and how to ensure a safe communication between them in a real-time simulation. You can execute a Simulink implementation container on a SCALEXIO AutoBox.

The following illustration shows the root level of the `real_time_platform_support` demo model.



The demo model contains a TargetLink subsystem prepared for use as a behavior model in ConfigurationDesk. It consists of externally triggered function-call subsystems which transfer data via **Rate Transition** blocks. The TargetLink **InPort** and **OutPort** blocks of the function-call-triggered subsystems leading signals to and from the TargetLink subsystem boundary are modeled as model ports of the behavior model. Additionally, the TargetLink subsystem contains blocks that do not exchange data with the function-call-triggered subsystems. They are connected to the TargetLink **InPort** and **OutPort** blocks at the top level of the TargetLink subsystem. The port blocks at the top level of the subsystem also represent the model ports of the behavior model. The production code and SIC can be generated for the TargetLink subsystem.





Preparing subsystems for use in ConfigurationDesk and exporting a SIC container

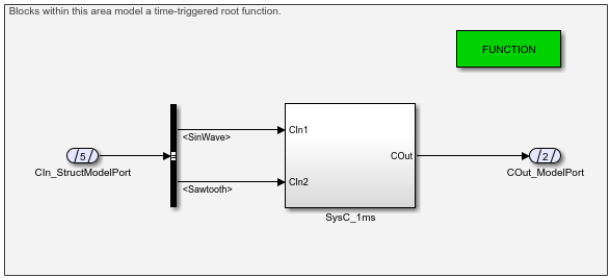
Modeling SIC runnable functions For the TargetLink subsystems , root functions are generated. During SIC file generation, TargetLink creates corresponding SIC runnable functions, which can be simulated in dSPACE ConfigurationDesk in different tasks with different sampling rates or asynchronously. Each root function generated for a TargetLink subsystem becomes an SIC runnable function.

The MultipleRootFunctions TargetLink subsystem contains time-based and asynchronous modeled parts that result in three root functions and corresponding SIC runnable functions: one time-triggered and two asynchronous ones.

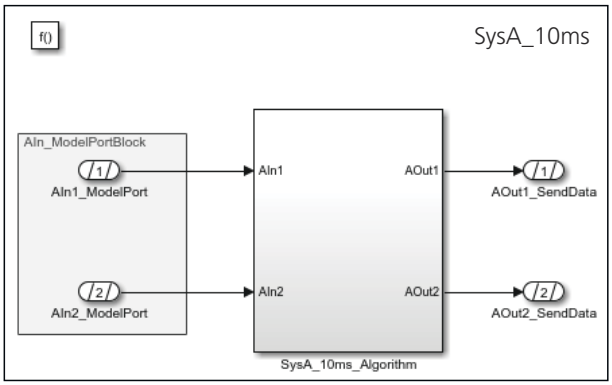
Note

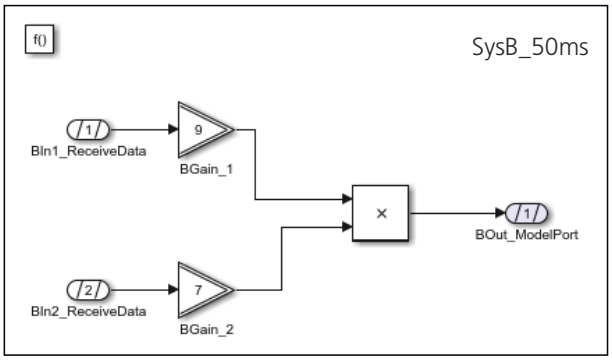
You can model SIC runnable functions that are time-triggered or asynchronous in the same TargetLink subsystem. However, the blocks for which a time-triggered SIC runnable function will be generated must be moved to a Function Call subsystem. Refer to Details on Generating SIC Files for ConfigurationDesk That Contain Several SIC Runnable Functions (📖 TargetLink Interoperation and Exchange Guide).

- Blocks representing a time-based root or SIC runnable function (`SysC_1ms`)



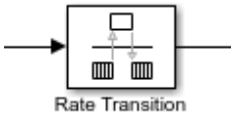
- Function-call subsystems representing asynchronous root or SIC runnable functions (`SysA_10ms and SysB_50ms`)

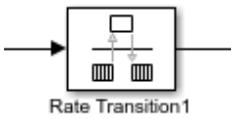




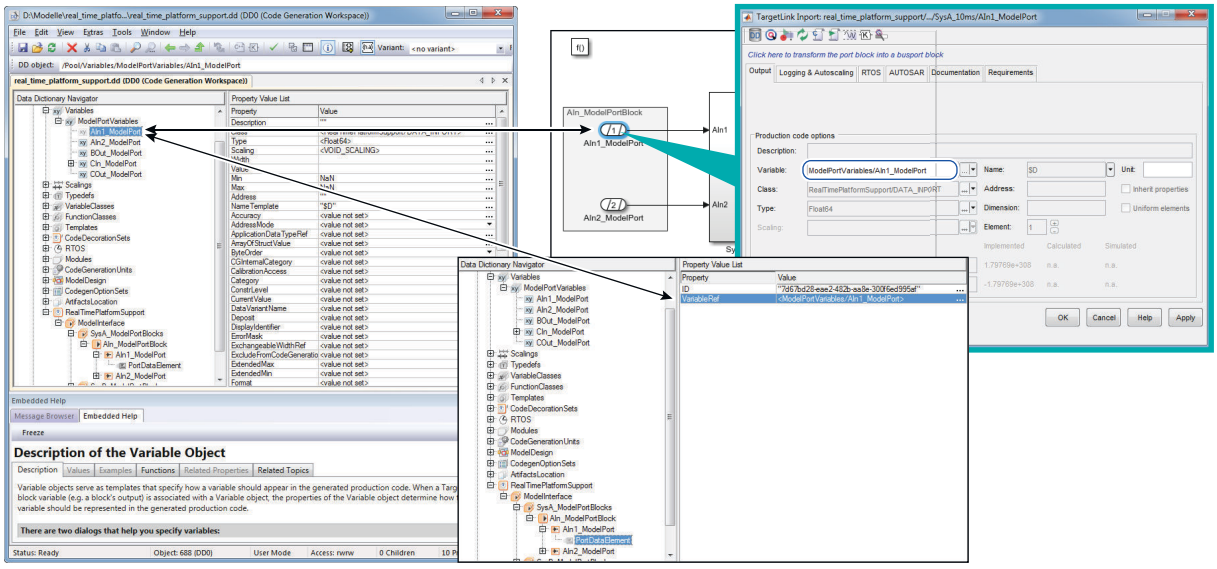
In ConfigurationDesk, you can connect both SIC runnables that are generated from the asynchronous root functions to tasks. You can connect them to tasks with asynchronous events and to tasks with timer events. If you connect them to tasks with timer events, the SIC runnables are also evaluated periodically.

Modeling safe communication To provide safe communication between the asynchronous SIC runnable functions `SysA_10ms and SysB_50ms`), i.e., between the corresponding tasks in ConfigurationDesk, two Rate Transition blocks are inserted in the signal flow. Their Ensure data integrity during data transfer property is selected.

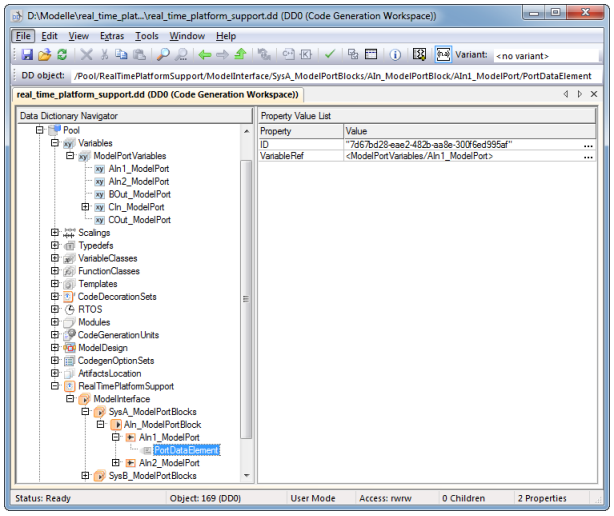




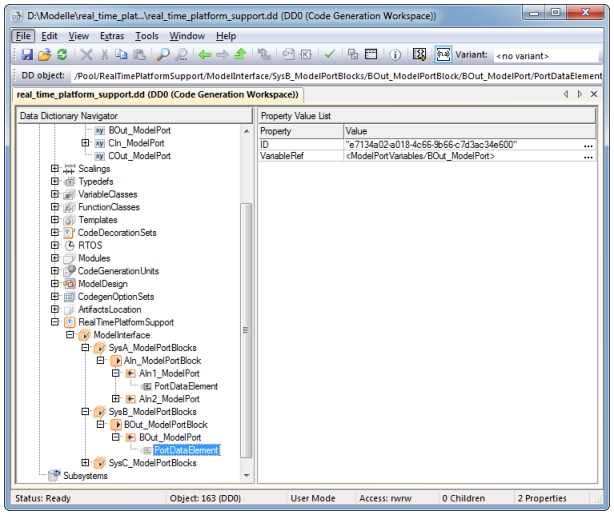
Modeling the SIC interface in the Data Dictionary The SIC interface is specified in the Data Dictionary using model port variables which are referenced at the TargetLink InPort and OutPort blocks, and DD `ModelPortBlocks/ModelPort` objects. Additionally, the RealTimePlatformSupport code generation option is set to on.



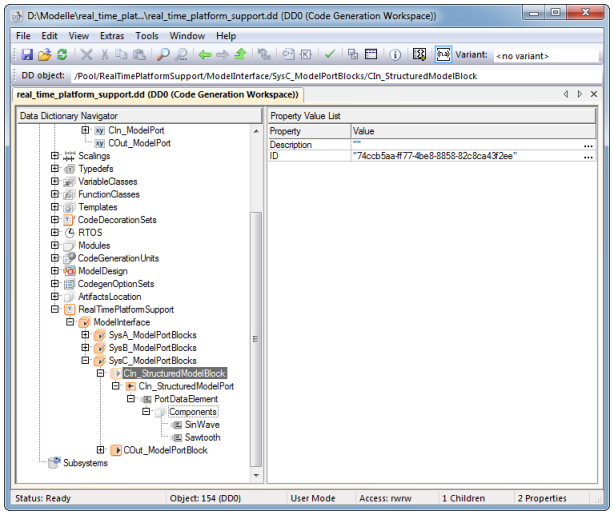
Modeling model port blocks with single and multiple ports for use in ConfigurationDesk in the Data Dictionary The demo model also shows you how to specify multiple and single model port blocks to be used in ConfigurationDesk. For multiple inports, such as the TargetLink InPort blocks (SL_AIn1 and SL_AIn2) in the SysA_10ms subsystem, corresponding DD ModelPortBlock objects are specified in `/Pool/RealTimePlatformSupport/ModelInterface/SysA_ModelPortBlocks/AIn_ModelPortBlock`.



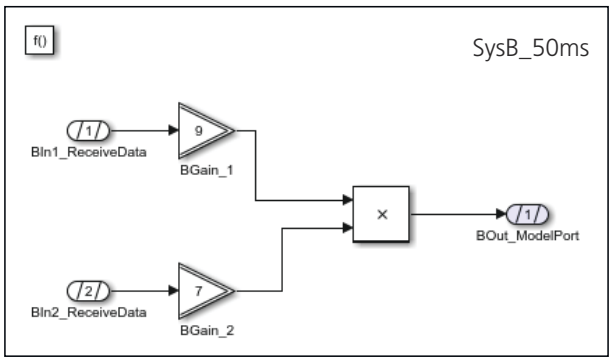
The TargetLink OutPort block (SL_BOut) in the SysB_50ms subsystem is a single port. The corresponding DD ModelPortBlock object is specified in: `/Pool/RealTimePlatformSupport/ModelInterface/SysB_ModelPortBlocks/BOut_ModelPortBlock`.



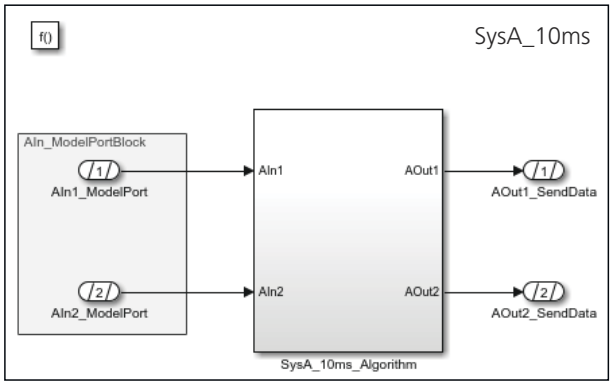
Modeling model port blocks with structured model ports for use in ConfigurationDesk in the Data Dictionary A structured model port is specified for the BusInport block (CIn_StructModelPort) at the root of the TargetLink subsystems. The corresponding DD ModelPortBlock object is specified in `/Pool/RealTimePlatformSupport/ModelInterface/SysC_ModelPortBlocks/CIn_StructuredModelBlock`.



Placing TargetLink port blocks that represent model port blocks for use in in ConfigurationDesk directly in the subsystem that models the function TargetLink port blocks that represent model ports to be used in ConfigurationDesk blocks can be placed directly in the subsystem that models the actual function (see `SysB_50ms`).



Placing TargetLink port blocks that represent model ports for use in ConfigurationDesk in a wrapper subsystem around the actual function To test the subsystem in real-time simulation without changes, the TargetLink port blocks that represent the model ports to be used in ConfigurationDesk are placed in a wrapper subsystem around the actual function subsystem (see `SysA_10ms`).



VECU_EVENT_SIMULATION

Access

In the MATLAB Command Window, enter the following command:

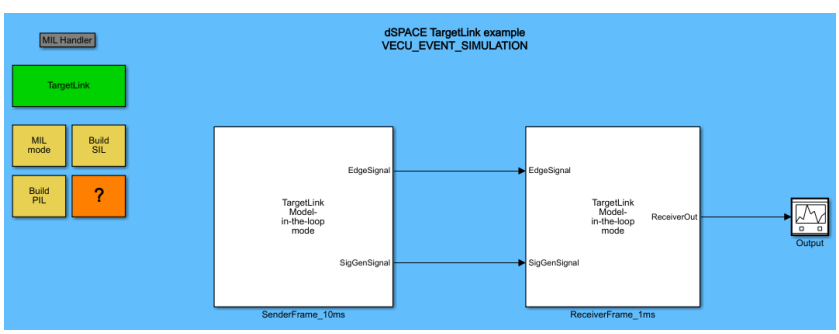
```
tl_demos vecu_event_simulation
```

You can also enter **`tl_demos`** to access all available demos in the MATLAB Help browser. Select a demo and click **Open this example** at the top right.

Description

The `VECU_EVENT_SIMULATION` demo model shows you how to simulate event communication via data signals that are communicated between different V-ECUs.

The following illustration shows the root level of the `VECU_EVENT_SIMULATION` demo model.

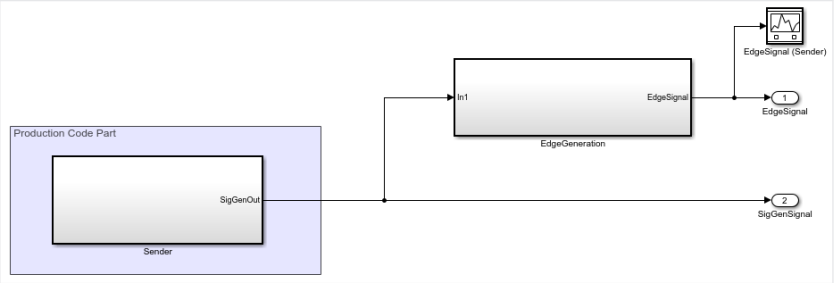


The demo model contains two TargetLink subsystems for generating V-ECU implementations. **SenderFrame_10ms** is used to model the event sender, **ReceiverFrame_1ms** is used to model the event receiver. Because V-ECU implementations do not support event communication, data communication is used instead: **SenderFrame_10ms** generates an edge data signal. **ReceiverFrame_1ms** detects this edge and generates an internal event that triggers a function.

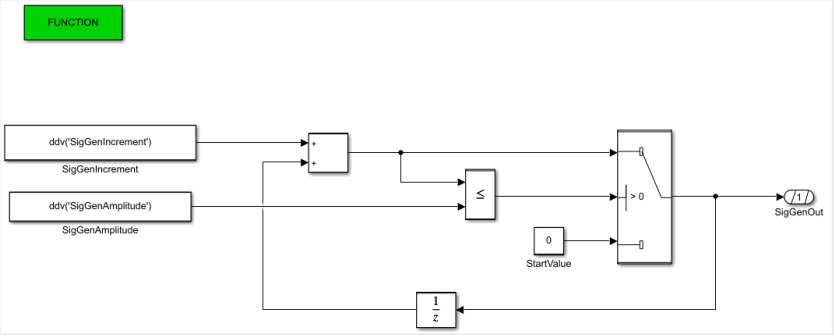
The sampling rate of ReceiverFrame_1ms is ten times faster than the sampling rate of SenderFrame_10ms. This ensures that the edge signal that is generated in the EdgeGeneration subsystem contained in SenderFrame_10ms can be reliably detected by ReceiverFrame_1ms.

Modeling the event sender

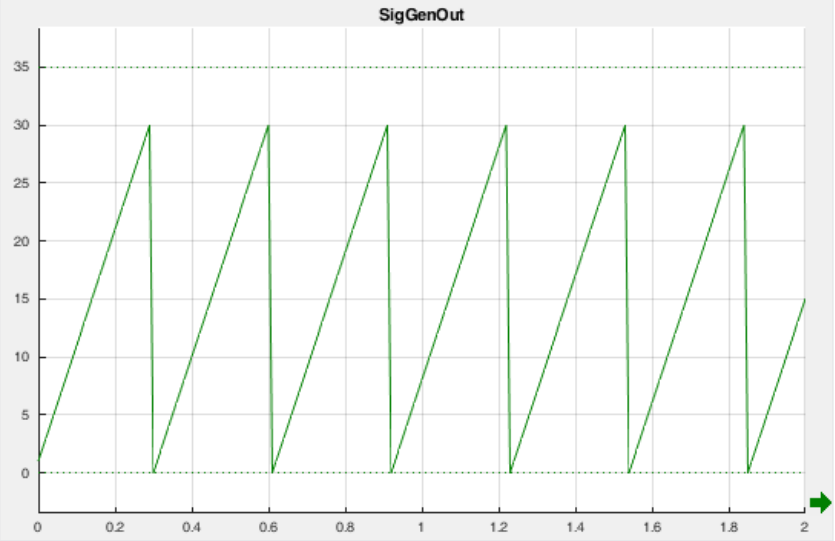
SenderFrame_10ms contains two subsystems:



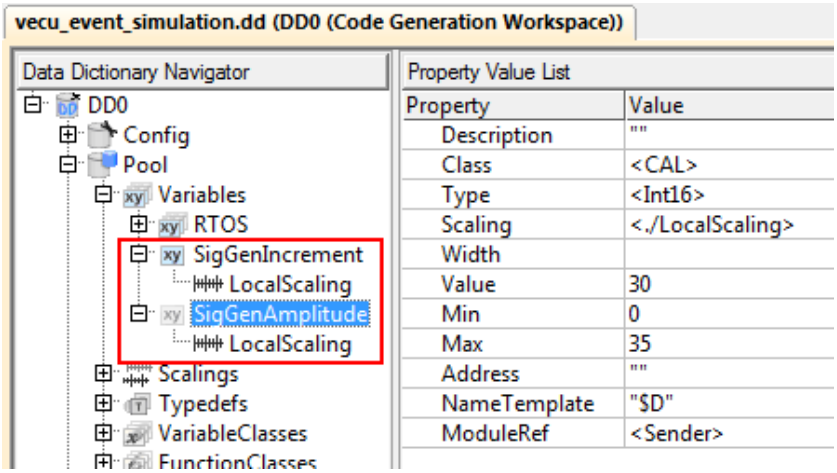
Sender This subsystem is used to generate the production code of the sender of data to be processed. It is configured for incremental code generation. Its code is generated into the Sender module:



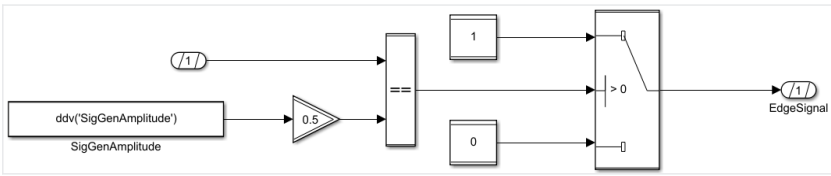
The Sender subsystem generates the SigGenOut sawtooth signal :



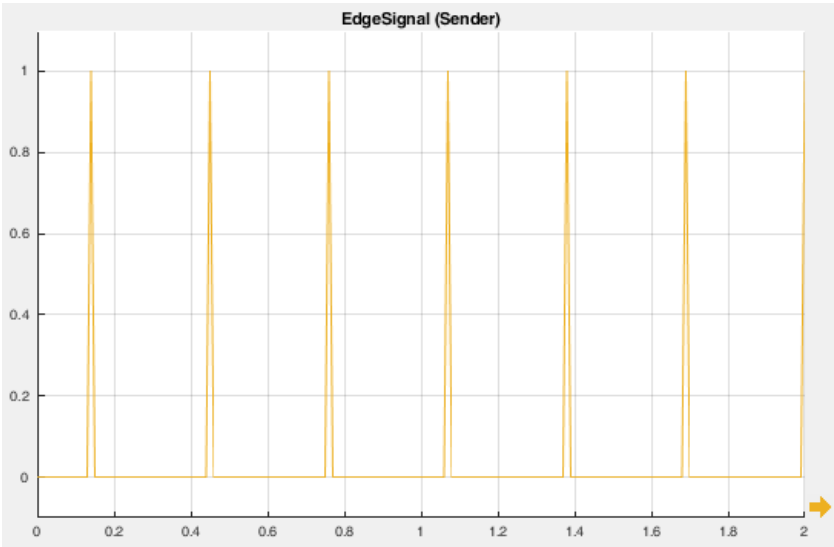
The two variables called SigGenIncrement and SigGenAmplitude are specified in the Data Dictionary. They are calibratable to facilitate experiments in a measurement and calibration tool such as ControlDesk:



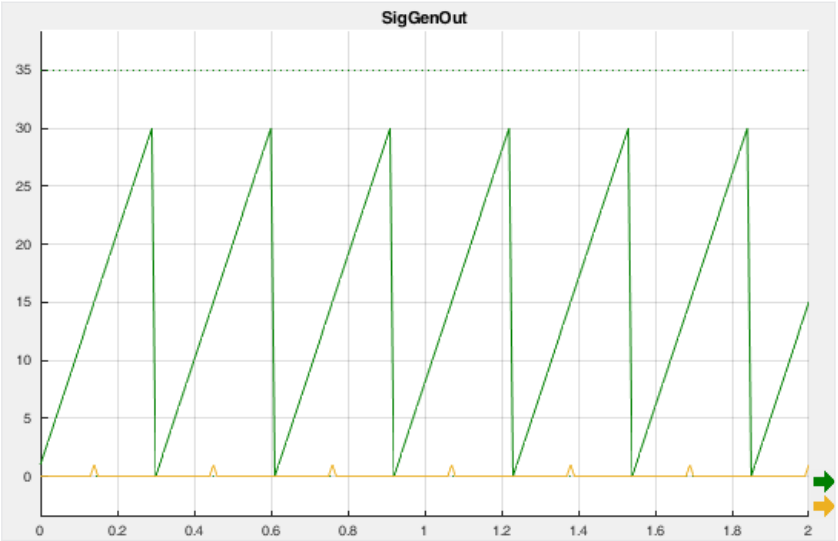
EdgeGeneration The EdgeGeneration subsystem generates the EdgeSignal that replaces the event in data communication:



Whenever the value of SigGenOut reaches half of the value of SigGenAmplitude (15 in this example), EdgeSignal is 1. For all other values of SigGenOut, EdgeSignal is 0. Refer to the following plot (color changed to orange):

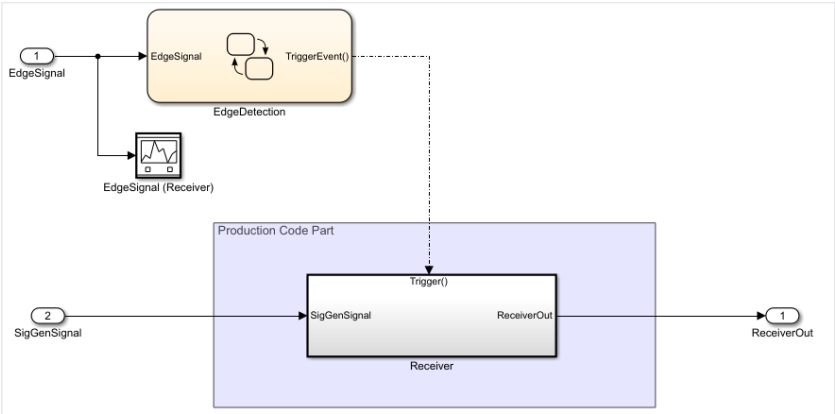


Superimposing it on SigGenOut shows that the event is generated whenever SigGenOut reached half of its maximum value:

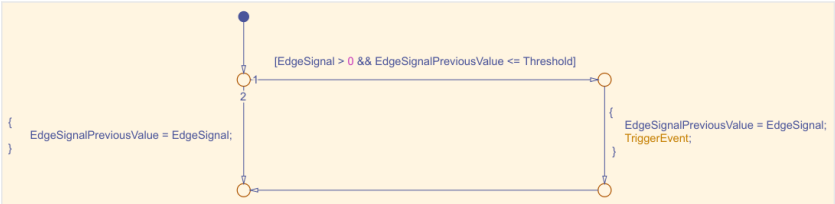


Modeling the event receiver

ReceiverFrame_1ms contains a Stateflow chart EdgeDetection and a function-call-triggered subsystem Receiver that is configured for incremental code generation:



EdgeDetection

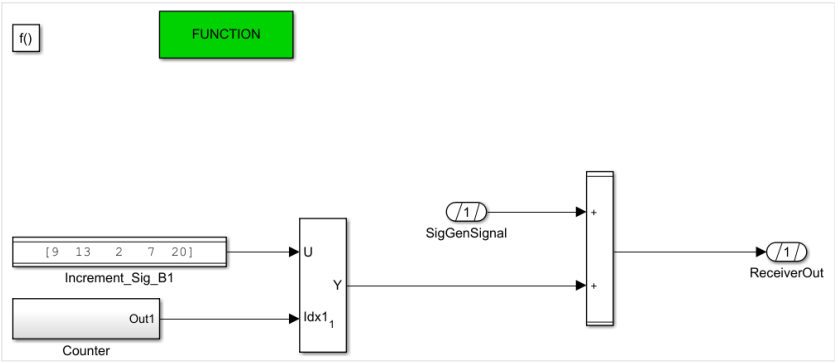


This Stateflow Chart is used to model an event scheduler. It provides the TriggerEvent, if all of the following conditions are met:

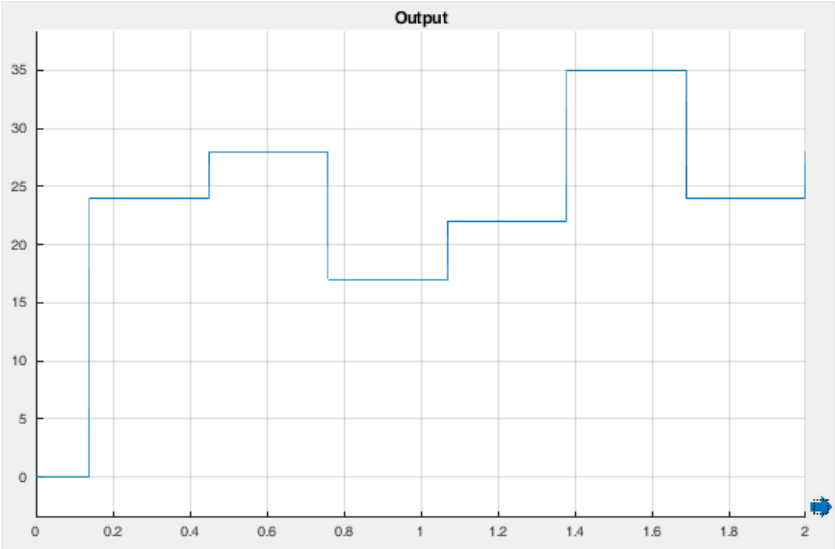
- EdgeSignal is greater than 0. Semantically, this means that the sawtooth signal has a rising edge and its current value is 15.
- EdgeSignalPreviousValue is smaller than or equal to Threshold. In this example, Threshold is set to 0. Semantically, this means that the value of EdgeSignal changed since the last step.

In all other cases, EdgeSignalPreviousValue is updated with EdgeSignal.

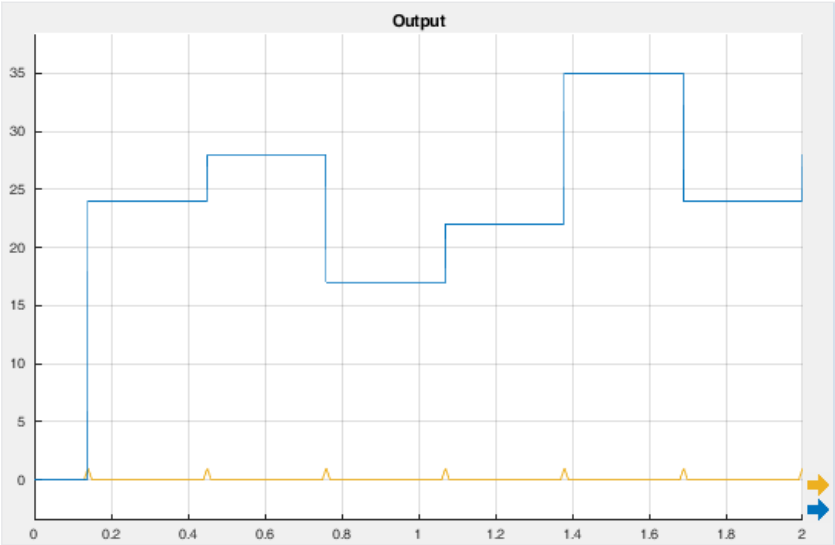
Receiver This subsystem is used to generate the production code of the receiver that processes the sender signal. It is configured for incremental code generation. Its code is generated into the Receiver module:



The Counter subsystem models a simple counter that is incremented by 1 each time the Receiver subsystem is triggered. The counter has a maximum value of 5, which matches the number of elements of the Increment_Sig_B1 vector. Accordingly, each time Receiver is triggered, another vector element is selected and added to SigGenSignal. The block variable of ReceiverOut is measurable to facilitate experiments in a measurement and calibration tool such as ControlDesk. The ReceiverOut signal is shown in the following plot (color changed to blue):



Superimposing it with **EdgeSignal** shows that the output changes whenever an edge is detected:



Using V-ECU implementations in the tool chain You can use the V-ECU implementations generated by TargetLink in the tool chain:

Product	Purpose
VEOS (Player)	Build V-ECUs to perform offline simulations on VEOS.
ConfigurationDesk	Build V-ECUs to perform real-time simulations on supported platforms.
ControlDesk	Perform experiments to measure and calibrate the variables.

Data Dictionary Releated Features

Where to go from here

Information in this section

DD_FILTER.....219
DD_ML_API.....221
DD_ML_IMPORTEXPORT.....222
DD_CUSTOM_COMMAND.....225

DD_FILTER

Access

In the MATLAB Command Window, enter the following command:

```
tl_demos dd_filter
```

You can also enter **`tl_demos`** to access all available demos in the MATLAB Help browser. Select a demo and click **Open this example** at the top right.

Description

The **`dd_filter`** demo shows you how to create XML filter rule sets based on DD files with the required specification. You can use filter rule sets in the Filter list of the Data Dictionary Manager. Filter rules let you *hide* specified objects and properties in different panes of the Data Dictionary Manager. This helps you focus on objects that are of interest for your current use case.

- The demo contains the **`tl_example_CreateDDFilterBasedOnDDFile`** script and three generic DD project files with the specification for typical filter use cases. The user is prompted to open a a complete Data Dictionary project file. You can create your own DD project files, or you can use the following predefined DD files:

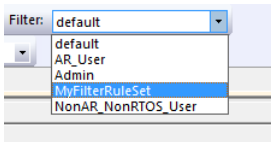
- AR_User - A filter rule set for AUTOSAR users.
- Admin - a filter rule set adapted for DD administrators.
- NonAR_NonRTOS_User - A filter rule set for both non-AUTOSAR and non-RTOS users. In addition, some internal objects (of no interest for normal users) are hidden, too.

For details, refer to Basics on Filter Rule Sets for the Data Model (📖 TargetLink Data Dictionary Basic Concepts Guide).

- At the end of the execution of the script, an XML file containing the filter rule set is created. The XML file can be shared among multiple users to make the filter rule set available to team members. The filter rule set has the name of the source DD file. If you want to rename the filter rule set displayed in the Filter list, open the generated XML file and change the name as shown in the following screenshot:

- Use the TargetLink Preferences dialog under Data Dictionary - Filter Rules or the `tl_pref('set', 'ddfilterruledirectory', <Directory>)` API function to set the preferences accordingly.
- You might have to call the `dsdd('ReloadFilterRuleSets')` API function so that the filters become visible after the preferences were adjusted.

You can then use the filter rule set in the Filter list of the Data Dictionary Manager.



Note

This demo does not contain any model. For further information, refer to the code comments of the script.

Related topics

Basics

Basics on Filter Rule Sets for the Data Model (📖 TargetLink Data Dictionary Basic Concepts Guide)

HowTos

How to Create Filter Rule Sets via DD Files (📖 TargetLink Data Dictionary Basic Concepts Guide)

DD_ML_API

Access

In the MATLAB Command Window, enter the following command:

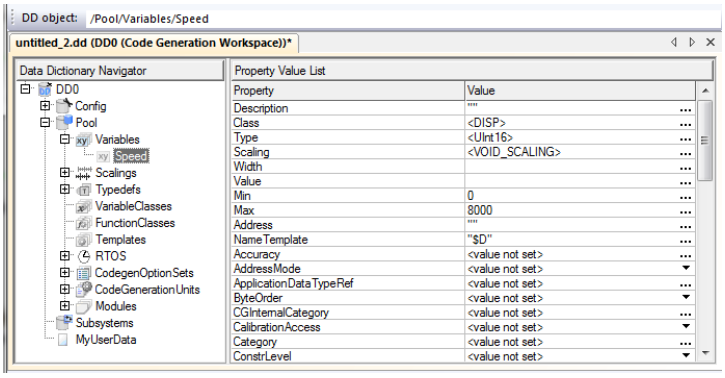
```
tl_demos dd_ml_api
```

You can also enter **`tl_demos`** to access all available demos in the MATLAB Help browser. Select a demo and click **Open this example** at the top right.

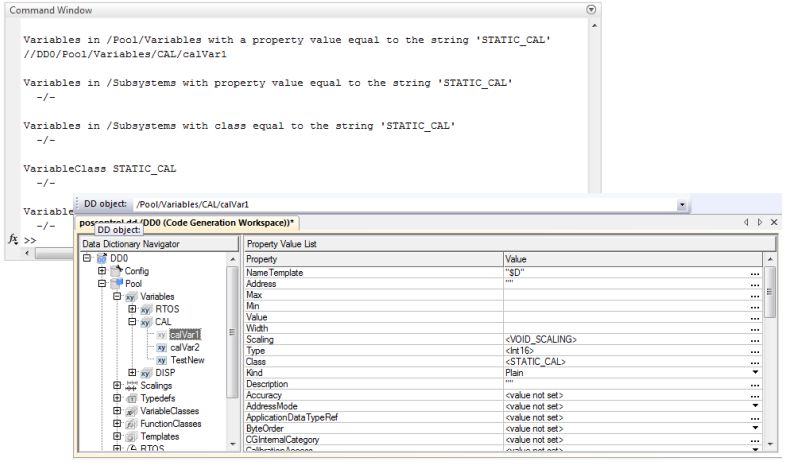
Description

This demo shows the basic use of the DD MATLAB API. The demo comes with 2 scripts:

- The **`simple.m`** script creates a DD Variable object (**Speed**) and a user-defined generic DD object (**MyUserData**) via dsdd commands. It then opens the Data Dictionary Manager and selects **Speed**.



- The **`findCalVariables.m`** script demonstrates how to find all DD Variable objects with **STATIC_CAL** class in a DD file. It loads the poscontrol.dd project file. Then it searches for the STATIC_CAL class and lists the result in the MATLAB Command Window. It also opens the Data Dictionary Manager and selects **calVar1**.



Note

This demo does not contain any model. For further information, refer to the code comments of the scripts.

Related topics

Basics

Managing Data with the DD MATLAB API (📖 TargetLink Data Dictionary Basic Concepts Guide)

DD_ML_IMPORTEXPORT

Access

In the MATLAB Command Window, enter the following command:

```
tl_demos dd_ml_importexport
```

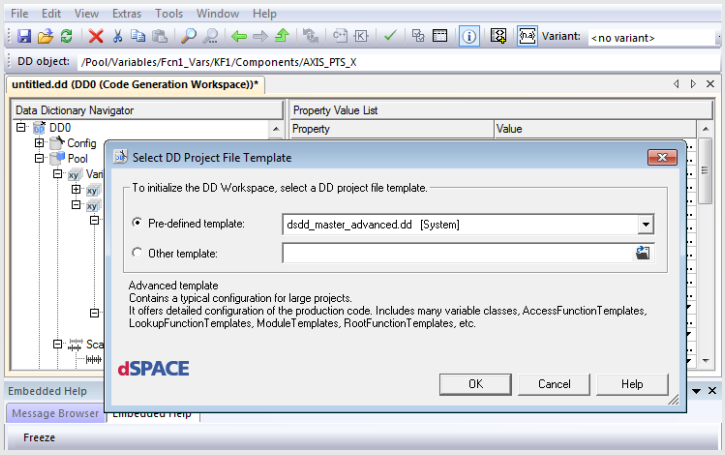
You can also enter **`tl_demos`** to access all available demos in the MATLAB Help browser. Select a demo and click **Open this example** at the top right.

Description

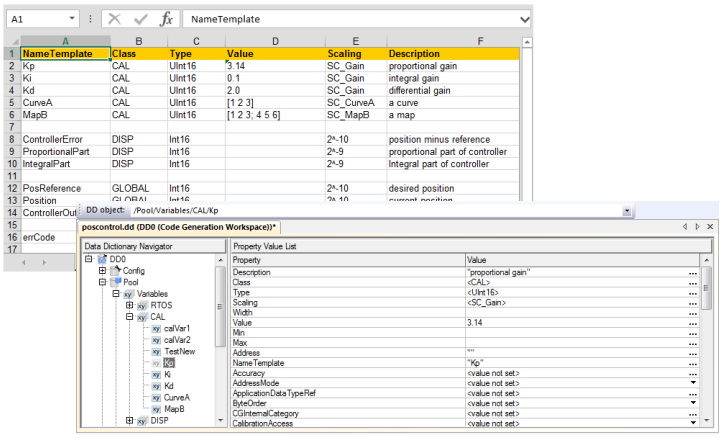
The `dd_ml_importexport` demo shows you various m-scripts that demonstrate how to import XLS and XML file content to the Data Dictionary. It contains four M scripts that show you how to import the data of XLS and XML files to the Data Dictionary. In addition, the demo contains one XLS and two XML sample files that can be used as source files for the import scripts.

Note

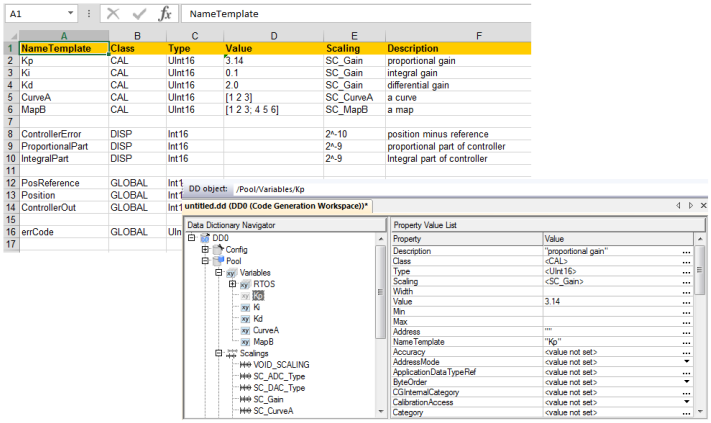
Before you run the scripts `xls2dd.m`, `xls2dd_simple.m` and `xml2dd_extended.m`, make sure one of the DD project file templates that reside in <TL_InstRoot>\dsdd\Config is loaded. You can also load the template by selecting the File - New - Use current DD workspace menu command in the Data Dictionary Manager. The `xml2dd.m` script does not need a template.



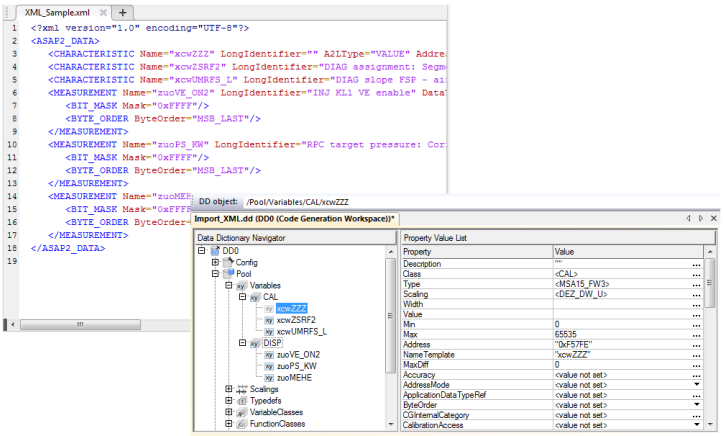
- XLS import:
  - The `xls2dd.m` script imports variables and scalings from an XLS file to the Data Dictionary. You are prompted to open an XLS file. You can use the `XLS_Sample.xls` file. The script then creates DD objects and groups the DD Variable objects according to the CAL class.



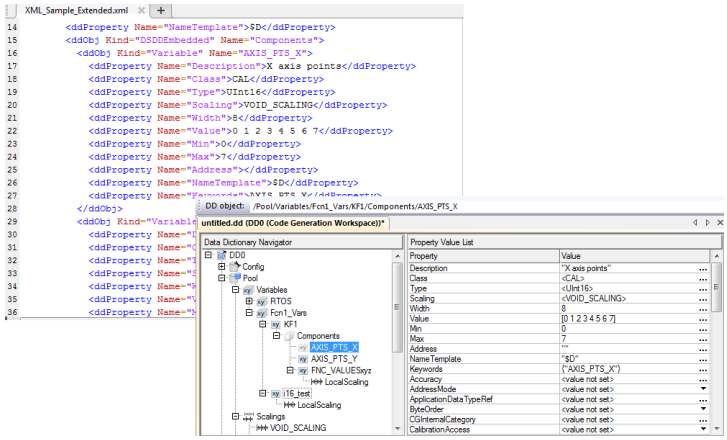
  - The `xls2dd_simple.m` script imports variables from an XLS file to the Data Dictionary (simplified version). The objects are created and the DD Variable objects are inserted directly in the Variables object tree.



- XML import:
  - The `xml2dd.m` script imports an XML file in "simple" format to the Data Dictionary: The XML elements of the sample file (`XML_Sample.xml`) are converted to generic DD objects with the name of the XML tag. Subelements result in the creation of child objects. The attributes of an XML element are mapped to string-type properties of the created object.



  - The `xml2dd_extended.m` script imports an XML file in "extended" format to the Data Dictionary: The XML sample file (`XML_Sample_Extended.xml`) already contains all the information needed to create the corresponding DD objects.



Note

- This demo does not contain any model. For further information, refer to the code comments of the scripts.
- This demo is called `DD_ML_ImportExport`. Do not confuse it with the `ML_ImportExport` demo in the original `Demos` folder of your TargetLink installation.

DD_CUSTOM_COMMAND

Access

In the MATLAB Command Window, enter the following command:

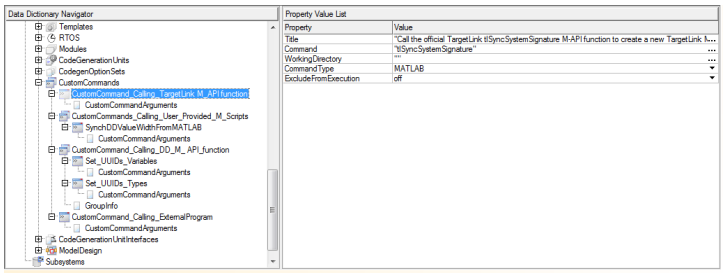
```
tl_demos dd_custom_command
```

You can also enter **`tl_demos`** to access all available demos in the MATLAB Help browser. Select a demo and click **Open this example** at the top right.

Description

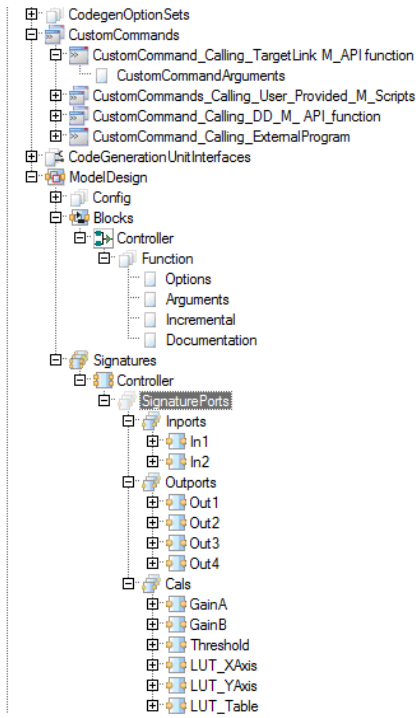
The `dd_custom_command` demo shows you four different custom command calls in the Pool area of the Data Dictionary Manager. You can call your own commands from within the Data Dictionary by specifying **CustomCommand** objects. You can also specify additional arguments. Custom commands can be external programs, your own M scripts or other preparatory scripts or functions you want to call. The dd_custom_command demo contains four different custom command calls in the Pool area of the Data Dictionary Manager.

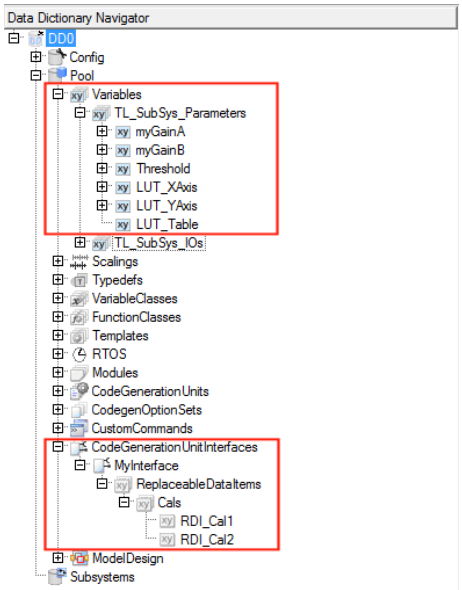
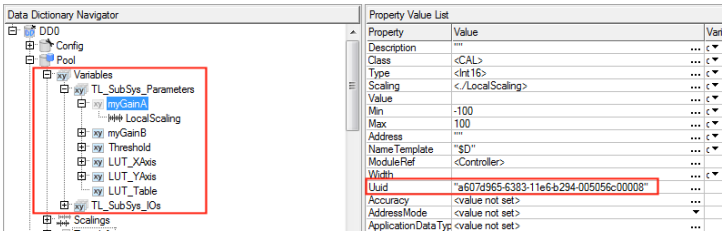
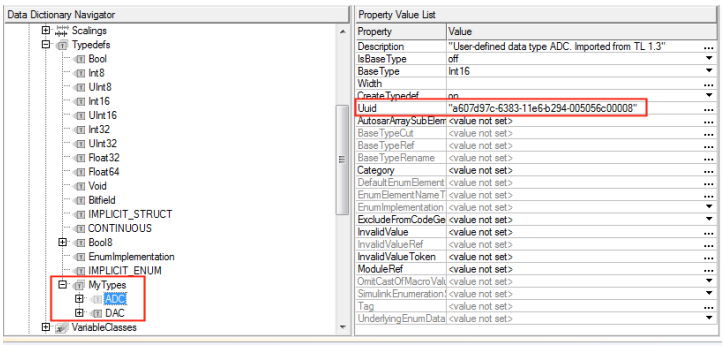
The illustration below shows the dd_custom_command demo in the Data Dictionary Manager.

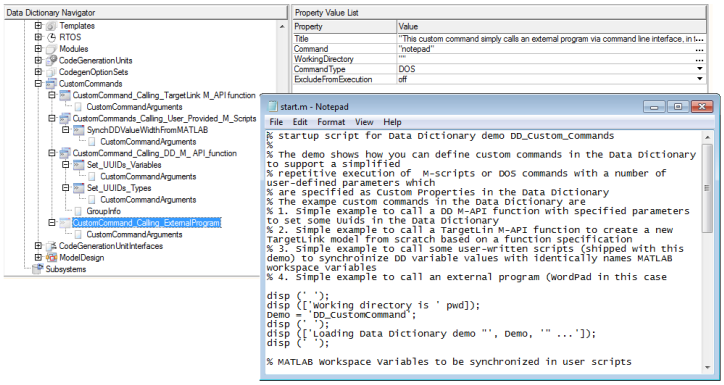


Calling external commands

The following table describes the different custom command calls in this demo. To run the custom command, select **Run Custom Command** in its context menu.

Custom command name	Description
Calling TargetLink M_API function	Calls the TargetLink `tlSyncSystemSignature` M-API function to create a new TargetLink model (Controller_Model) from scratch based on the provided Data Dictionary Objects, e.g. variables, a Function Block object and Port objects. In addition, a System Signature Synchronization Report (HTML) lists the newly created ports. 
Calling_User_Provided_M_scripts	Calls a user-provided example script (`tl_example_sync_DDValueWidth_From_MATLAB_Workspace`) to synchronize

Custom command name	Description
	MATLAB workspace values with the Value property of DD variables and Replaceable Data Items as shown below: 
Calling_DD_M_API_Function	▪ This first custom command directly calls a DD API function (`SetUuids`) to set the UUID properties of objects (initially, the UUID properties are empty), in this case for DD variables as shown below:  ▪ The second custom command directly calls an official DD API function (`SetUuids`) to set the UUID properties of objects, in this case for a group under /Pool/Typedefs as shown below: 

Custom command name	Description
Calling_ExternalProgram	This custom command simply calls an external program via command line interface, in this case notepad, where the file to be opened is passed as a command line argument (Windows Command prompt): 

Related topics

HowTos

How to Specify Custom Command Calls (📖 TargetLink Data Dictionary Basic Concepts Guide)

MATLAB Code

MATLAB_CODE_STATEFLOW

Access

In the MATLAB Command Window, enter the following command:

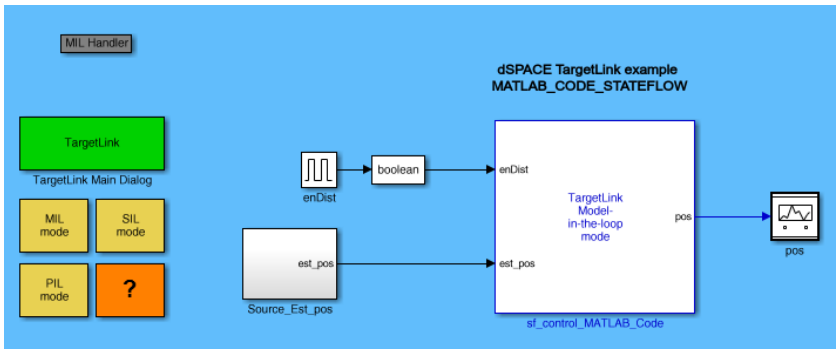
```
tl_demos matlab_code_stateflow
```

You can also enter **`tl_demos`** to access all available demos in the MATLAB Help browser. Select a demo and click **Open this example** at the top right.

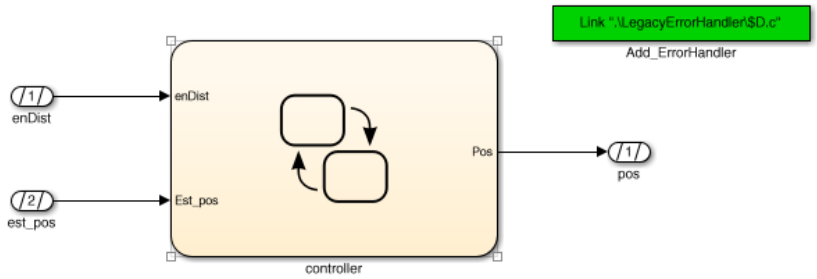
Description

The MATLAB_CODE_STATEFLOW demo model demonstrates fixed-point code generation of MATLAB functions. The MATLAB functions in this demo are located in a Stateflow chart. The demo shows how to specify code generation settings for MATLAB functions and local functions as well as top-level and local variables in the Property Manager and in the Data Dictionary. In addition, it demonstrates how to embed external (handwritten) code via an external function call.

The following screenshot shows the root level of the MATLAB_CODE_STATEFLOW demo model.

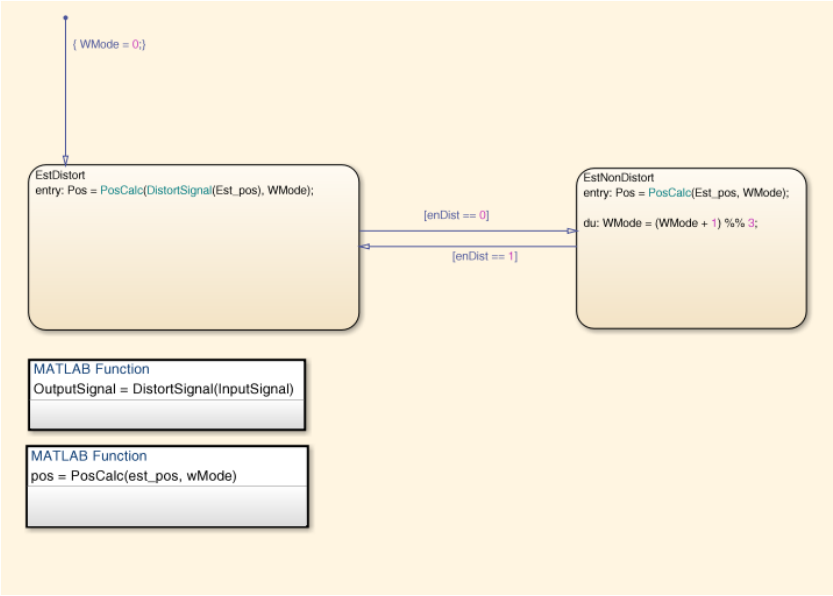


The sf_control_MATLAB_Code subsystem contains the control stateflow chart and the Add_ErrorHandler Addfile block as shown in the following screenshot:



The controller Stateflow chart contains two states and two MATLAB functions.

The two states `EstDistort` and `EstNonDistort` are used to access the MATLAB functions `PosCalc` and `DistortSignal`.



MATLAB functions in Stateflow

Using persistent variables in MATLAB code You can use persistent variables in MATLAB functions as shown in the PosCalc MATLAB function. To do so, mark them as `persistent`:

```
PosCalc
function pos = PosCalc(est_pos, wMode)
% function pos = PosCalc(est_pos, WMode)
% Function to calculate pos
% Inputs:
%         est_pos: estimated positoin
%         wMode: Mode for weighting
%
% Outputs:
%         pos: calculated position
%

% define state of the system
persistent old_pos;
% Initialize old_pos state
if isempty(old_pos)
    old_pos = 0.0;
end
```

Accessing Stateflow constant and Stateflow parameter variables in MATLAB code In a MATLAB function it is also possible to access Stateflow variables like constants and parameters directly. Code generation settings like data types/scalings for those variables are specified in the Property Manager.

The `SFConstant` and `SFParameter` variables are accessed in the `PosCalc` MATLAB function as shown in the following illustration:

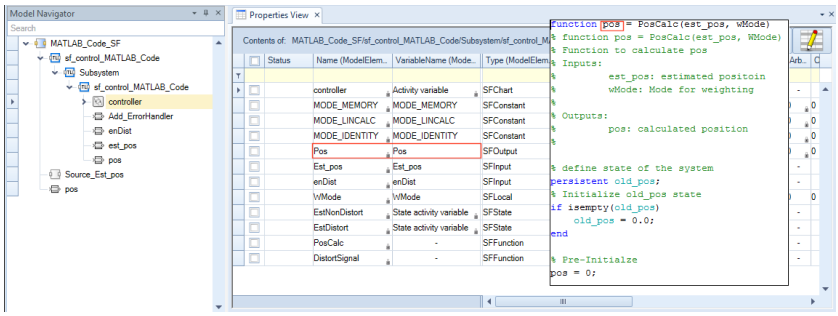
Contents of: MATLAB_Code_SF/sf_control_MATLAB_Code/Subsystem/sf_control_MATLAB_Code/c

Status	Name (ModelElem...	VariableName (Mode...	Type (ModelElement)
	controller	Activity variable	SFChart
	MODE_MEMORY	MODE_MEMORY	SFConstant
	MODE_LINCALC	MODE_LINCALC	SFConstant
	MODE_IDENTITY	MODE_IDENTITY	SFConstant
	Pos	Pos	SFOutput
	Est_pos	Est_pos	SFInput
	enDist	enD	
	WMode	WM	
	EstNonDistort	Stat	
	EstDistort	Stat	
	PosCalc		
	DistortSignal		

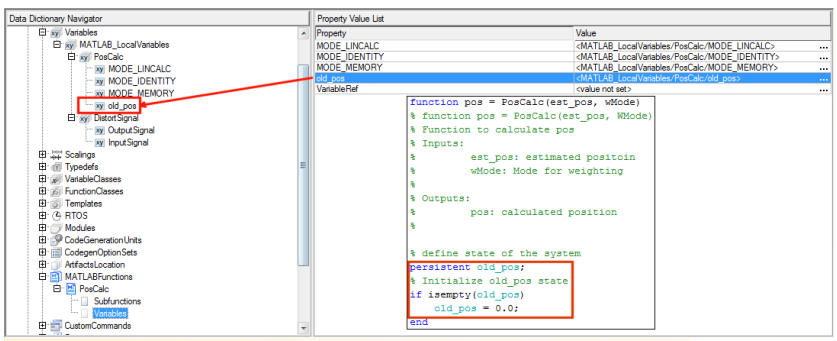
```
% MODE_IDENTITY, MODE_MEMORY, MODE_LINCALC defined as Stateflow Constants

% Different Treatment for the different modes
if (wMode == MODE_IDENTITY)
   % Return current estimate
   pos = DistortSignal(est_pos);

elseif (wMode == MODE_MEMORY)
   % Return old value
   pos = old_pos;

elseif (wMode == MODE_LINCALC)
   % Return pos based on weighting of currnent and old value
   pos = ParPos * est_pos + (1 - ParPos) * old_pos;
```

Generating fixed-point code from MATLAB code

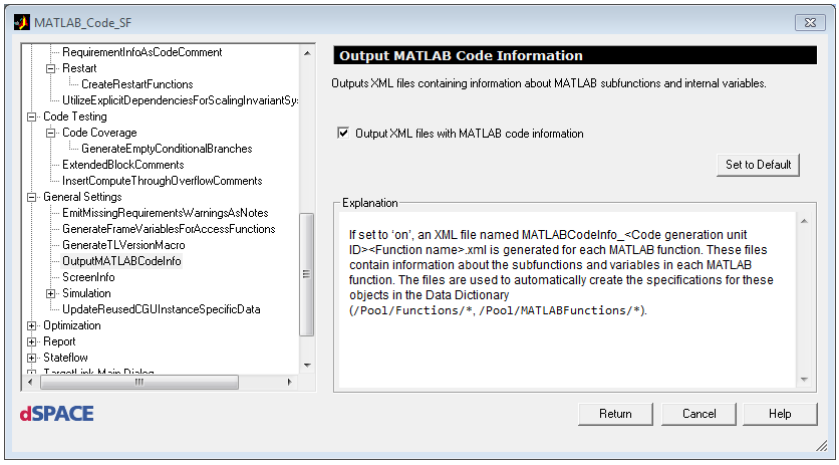
Specifiying code generation settings for top-level MATLAB functions. The exact C code function interface for a top-level MATLAB function is specified in the TargetLink Property Manager. The data types/scalings/variable classes for all interface variables as well as function classes are specified:



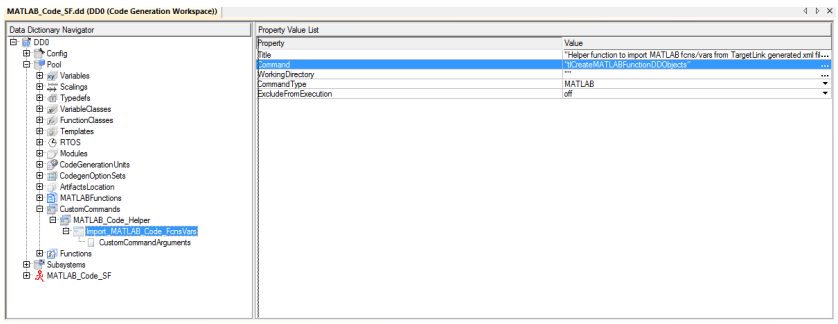
Specifying code generation settings for local variables and local functions In order to specify code generation settings for local variables as well as local functions, MATLAB function objects need to be created in the Data Dictionary. At these objects, references to Data Dictionary variables are created to define code generation settings for local variables in a MATLAB function. The matching between variables/local functions in the MATLAB code on one hand and objects in the Data Dictionary on the other hand is based on identical names.



Automatic data object creation as an assistance function Because the specification of code generation settings for local variables as well as MATLAB local functions depend on existing Data Dictionary objects, TargetLink provides a helper functionality to automatically create these objects. For that purpose, the `OutputMATLABCodeInfo` Code Generator option must be set and code must have been successfully generated. As a consequence, XML files are produced during code generation that can then be imported to the Data Dictionary using the `tlCreateMATLABFunctionDDObjects` API function. To simplify the process, it is recommended to use a Data Dictionary CustomCommand to perform the API function call from the Data Dictionary. The `OutputMATLABCodeInfo` Code Generator option can export variable names and local functions from each MATLAB function to an XML file. In this demo model, the XML files are already in the folder of the demo model.



Importing the specifications to the Data Dictionary The information in the XML files should be imported into the Data Dictionary to obtain the specifications. The import is handled by the `tlCreateMATLABFunctionDDObjects` API function. In this demo, a DDCustomCommand helper function calls the API function and delivers additional parameters.



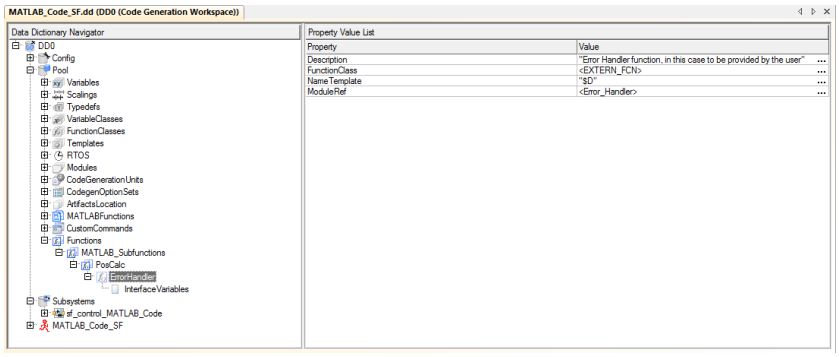
Calling legacy C code functions from MATLAB code and executing MATLAB code in MIL simulation mode only

Using a MATLAB local function to call externally provided C code In order to call external C code functions and run MATLAB code only in MIL simulation mode, the following steps are necessary:

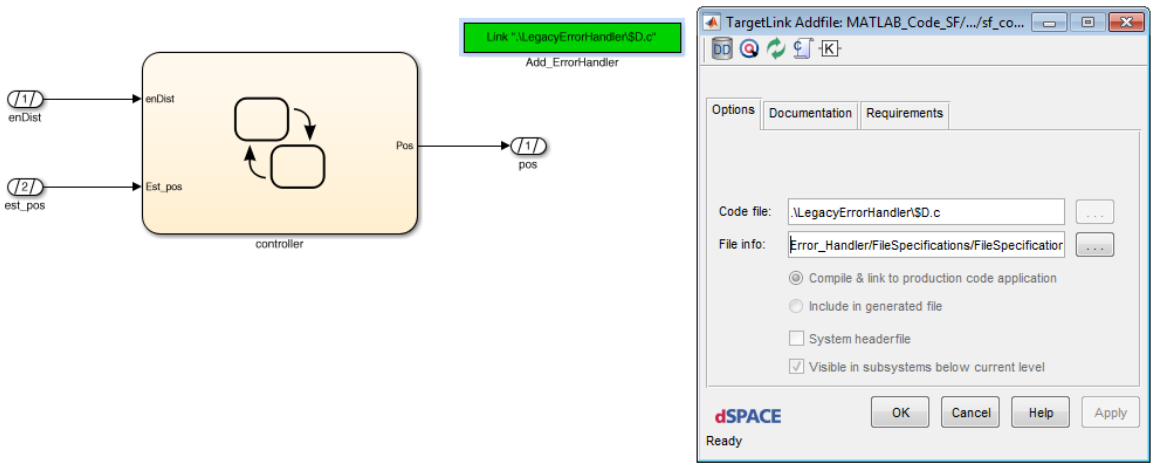
1. A MATLAB local function (`ErrorHandler` in this case) has to be defined, for which a C code implementation exists
2. In the Data Dictionary, specify a MATLAB local function object with the name (`ErrorHandler`) of the MATLAB local function.
3. The `FunctionClass` property of the MATLAB local function needs to be set to `EXTERN_FCN`.

As a consequence, TargetLink will not generate an implementation for the `ErrorHandler` MATLAB local function but call the externally provided C code. Moreover, TargetLink will ignore the entire content of the `ErrorHandler` local function (**BlackBox** behavior). The MATLAB code is only executed in MIL-mode.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% subfunction: ErrorHandler
%
% This MATLAB function has an external function class specified in the DD.
%
% As a consequence, TargetLink will not generate an implementation for the
% ErrorHandler MATLAB subfunction but call externally provided C code.
% Moreover, TargetLink will ignore the entire content of the ErrorHandler
% subfunction (Black Box behavior).
% The MATLAB Code of ErrorHandler is only executed in MIL mode.

function ErrorHandler()

% Just do something, e.g. throw error in MIL-Mode or something else
% error('Improper Value for wMode');
```



In order to compile and link the externally provided C code implementation for `ErrorHandler`, an **Addfile** block in the parent subsystem specifies the external code file:



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.....236
A.....237
B.....240
C.....241
D.....244
E.....246
F.....247
G.....248
I.....248
L.....250
M.....251
N.....253
O.....254
P.....255
R.....256
S.....258
T.....260
U.....262
V.....262
W.....263

Numerics

1-D look-up table A look-up table that maps one input value (x) to one output value (y).

2-D look-up table A look-up table that maps two input values (x,y) to one output value (z).

A

Abstract interface An interface that allows you to map a project-specific, physical specification of an interface (made in the TargetLink Data Dictionary) to a logical interface of a modular unit. If the physical interface changes, you do not have to change the Simulink subsystem or the partial DD file and therefore neither the generated code of the modular unit.

Access function (AF) A C function or function-like preprocessor macro that encapsulates the access to an interface variable.

See also read/write access function and variable access function.

Acknowledgment Notification from the RTE that a data element or an event message have been transmitted.

Activating RTE event An RTE event that can trigger one or more runnables.

See also activation reason.

Activation reason The activating RTE event that actually triggered the runnable.

Activation reasons can group several RTE events.

Active page pointer A pointer to a data page. The page referenced by the pointer is the active page whose values can be changed with a calibration tool.

Adaptive AUTOSAR Short name for the AUTOSAR *Adaptive Platform* standard. It is based on a service-oriented architecture that aims at on-demand software updates and high-end functionalities. It complements Classic AUTOSAR.

Adaptive AUTOSAR behavior code Code that is generated for model elements in Adaptive AUTOSAR Function subsystems or Method Behavior subsystems. This code represents the behavior of the model and is part of an adaptive application. Must be integrated in conjunction with ARA adapter code.

Adaptive AUTOSAR Function A TargetLink term that describes a C++ function representing a partial functionality of an adaptive application. This function can be called in the C++ code of an adaptive application. From a higher-level perspective, Adaptive AUTOSAR functions are analogous to runnables in Classic AUTOSAR.

Adaptive AUTOSAR Function subsystem An atomic subsystem used to generate code for an Adaptive AUTOSAR Function. It contains a **Function** block whose **AUTOSAR mode** is set to `Adaptive` and whose **Role** is set to `Adaptive AUTOSAR Function`.

ANSI C Refers to C89, the C language standard ANSI X3.159-1989.

Application area An optional DD object that is a child object of the DD root object. Each **Application** object defines how an ECU program is built from the generated subsystems. It also contains some experiment data, for example, a list of variables to be logged during simulations and results of code coverage tests.

Build objects are children of Application objects. They contain all the information about the binary programs built for a certain target platform, for example, the symbol table for address determination.

Application data type Abstract type for defining types from the application point of view. It allows you to specify physical data such as measurement data. Application data types do not consider implementation details such as bit-size or endianness.

Application data type (ADT) According to AUTOSAR, application data types are used to define types at the application level of abstraction. From the application point of view, this affects physical data and its numerical representation. Accordingly, application data types have physical semantics but do not consider implementation details such as bit width or endianness.

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

See also implementation data type (IDT).

Application layer The topmost layer of the ECU software. The application layer holds the functionality of the ECU software and consists of atomic software components (atomic SWCs).

ARA adapter code Adapter code that connects Adaptive AUTOSAR behavior code with the Adaptive AUTOSAR API or other parts of an adaptive application.

Array-of-struct variable An array-of-struct variable is a structure that either is non-scalar itself or that contains at least one non-scalar substructure at any nesting depth. The use of array-of-struct variables is linked to arrays of buses in the model.

Artifact A file generated by TargetLink:

- Code coverage report files
- Code generation report files
- Metadata files
- Model-linked code view files
- Production code files
- Simulation application object files
- Simulation frame code files
- Stub code files

Artifact location A folder in the file system that contains an artifact. This location is specified relatively to a project folder.

ASAP2 File Generator A TargetLink tool that generates ASAP2 files for the parameters and signals of a Simulink model as specified by the corresponding TargetLink settings and generated in the production code.

ASCII In production code, strings are usually encoded according to the ASCII standard. The ASCII standard is limited to a set of 127 characters implemented by a single byte. This is not sufficient to display special characters of different languages. Therefore, use another character encoding, such as UTF-8, if required.

Asynchronous operation call subsystem A subsystem used when modeling *asynchronous* client-server communication. It is used to generate the call of the `Rte_Call` API function and for simulation purposes.

See also operation result provider subsystem.

Asynchronous server call returns event An RTE event that specifies whether to start or continue the execution of a runnable after the execution of a server runnable is finished.

Atomic software component (atomic SWC) The smallest element that can be defined in the application layer. An atomic SWC describes a single functionality and contains the corresponding algorithm. An atomic SWC communicates with the outside only via the interfaces at the SWC's ports. An atomic SWC is defined by an internal behavior and an implementation.

Atomic software component instance An atomic software component (atomic SWC) that is actually used in a controller model.

AUTOSAR Abbreviation of *AUT*omotive *O*pen *S*ystem *AR*chitecture. 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
- Multiport Switch (Data Input port)
- Probe
- Sink
- Signal Conversion
- Switch (Data Input port)
- Unit Delay
- Stateflow Data
- MATLAB Function Data

C

Calibratable variable Variable whose value can be changed with a calibration tool during run time.

Calibration Changing the calibration parameter values of ECUs.

Calibration parameter Any ECU variable type that can be calibrated. The term *calibration parameter* is independent of the variable type's dimension.

Calprm Defined in a calprm interface. Calprms represent calibration parameters that are accessible via a measurement and calibration system.

Calprm interface An interface that is provided or required by a software component (SWC) via a port (AUTOSAR).

Calprm software component A special software component (SWC) that provides calprms. Calprm software components have no internal behavior.

Canonical In the DD, array-of-struct variables are specified canonically. Canonical means that you specify one array element as a representative for all array elements.

Catalog file (CTLG) A description of the content of an SWC container. It contains file references and file category information, such as source code files (`C` and `H`), object code files (such as `O` or `OBJ`), variable description files (`A2L`), or AUTOSAR files (`ARXML`).

Characteristic table (Classic AUTOSAR) A look-up table as described by Classic AUTOSAR whose values are measurable or calibratable. See also compound primitive data type

Classic AUTOSAR Short name for the AUTOSAR *Classic Platform* standard that complements Adaptive AUTOSAR.

Classic initialization mode The initialization mode used when the Simulink diagnostics parameter Underspecified initialization detection is set to `Classic`.

See also simplified initialization mode.

Client port A require port in client-server communication as described by Classic AUTOSAR. In the Data Dictionary, client ports are represented as DD ClientPort objects.

Client-server interface An interface that describes the operations that are provided or required by a software component (SWC) via a port (AUTOSAR).

Code generation mode One of three mutually exclusive options for generating TargetLink standard production code, AUTOSAR-compliant production code or RTOS-compliant (multirate RTOS/OSEK) production code.

Code generation unit (CGU) The smallest unit for which you can generate code. These are:

- TargetLink subsystems
- Subsystems configured for incremental code generation
- Referenced models
- DD CodeGenerationUnit objects

Code output style definition file To customize code formatting, you can modify a code output style definition file (XML file). By modifying this file, you can change the representation of comments and statements in the code output.

Code output style sheets To customize code formatting, you can modify code output style sheets (XSL files).

Code section A section of generated code that defines and executes a specific task.

Code size Amount of memory that an application requires specified in RAM and ROM after compilation with the target cross-compiler. This value helps to determine whether the application generated from the code files fits in the ECU memory.

Code variant Code variants lead to source code that is generated differently depending on which variant is selected (i.e., varianted at code generation time). For example, if the Type property of a variable has the two variants Int16 and Float32, you can generate either source code for a fixed-point ECU with one variant, or floating-point code with the other.

Compatibility mode The default operation mode of RTE generators. The object code of an SWC that was compiled against an application header generated in compatibility mode can be linked against an RTE generated in compatibility mode (possibly by a different RTE generator). This is due to using standardized data structures in the generated RTE code.

See also vendor mode.

Compiler inlining The process of replacing a function call with the code of the function body during compilation by the C compiler via inline expansion.

This reduces the function call overhead and enables further optimizations at the potential cost of larger code size.

Composition A structuring element in the application layer. A composition consists of software components and their interconnections via ports.

Compound primitive data type A primitive application data type (ADT) as defined by Classic AUTOSAR whose category is one of the following:

- COM_AXIS
- CUBOID
- CUBE_4
- CUBE_5
- CURVE
- MAP
- RES_AXIS
- VAL_BLK
- STRING

Compute-through-overflow (CTO) Calculation method for additions and subtraction where overflows are allowed in intermediate results without falsifying the final result.

Concern A concept in component-based development. It describes the idea that components separate their concerns. Accordingly, they must be developed in such a way that they provide the required functionality, are flexible and easy to maintain, and can be assembled, reused, or replaced by newer, functionally equivalent components in a software project without problems.

Config area A DD object that is a child object of the DD root object. The Config object contains configuration data for the tools working with the TargetLink Data Dictionary and configuration data for the TargetLink Data Dictionary itself. There is only one Config object in each DD workspace. The configuration data for the TargetLink Data Dictionary is a list of included DD files, user-defined views, data for variant configurations, etc. The data in the Config area is typically maintained by a Data Dictionary administrator.

ConfigurationDesk A dSPACE software tool for implementing and building real-time applications (RTA).

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

Constrained range limits User-defined minimum (Min) or maximum (Max) values that the user ensures will never be exceeded. The Code Generator relies on these ranges to make the generated production code more efficient. If no

Min/Max values are entered, the implemented range limits are used during production code generation.

Constrained type A DD Typedef object whose Constraints subtree is specified.

Container A bundle of files. The files are described in a catalog file that is part of the container. The files of a container can be spread over your file system.

Container Manager A tool for handling containers.

Container set file (CTS) A file that lists a set of containers. If you export containers, one container set file is created for every TargetLink Data Dictionary.

Conversion method A method that describes the conversion of a variable's integer values in the ECU memory into their physical representations displayed in the Measurement and Calibration (MC) system.

Custom code Custom code consists of C code snippets that can be included in production code by using custom code files that are associated with custom code blocks. TargetLink treats this code as a black box. Accordingly, if this code contains custom code variables you must specify them via custom code symbols.. See also external code.

Custom code symbol A variable that is used in a custom code file. It must be specified on the Interface page of custom code blocks.

Customer-specific C function An external function that is called from a Stateflow diagram and whose interface is made known to TargetLink via a scripting mechanism.

D

Data element Defined in a sender-receiver interface. Data elements are information units that are exchanged between sender ports, receiver ports and sender-receiver ports. They represent the data flow.

Data page A structure containing all of the calibratable variables that are generated during code generation.

Data prototype The generic term for one of the following:

- Data element
- Operation argument
- Calprm
- Interrunnable variable (IRV)
- Shared or PerInstance Calprm
- Per instance memory

Data receive error event An RTE event that specifies to start or continue the execution of a runnable related to receiver errors.

Data received event An RTE event that specifies whether to start or continue the execution of a runnable after a data element is received by a receiver port or sender-receiver port.

Data semantics The communication of data elements with last-is-best semantics. Newly received data elements overwrite older ones regardless of whether they have been processed or not.

Data send completed event An RTE event that specifies whether to start or continue the execution of a runnable related to a sender acknowledgment.

Data transformation A transformation of the data of inter-ECU communication, such as end-to-end protection or serialization, that is managed by the RTE via transformers.

Data type map Defines a mapping between implementation data types (represented in TargetLink by DD **Typedef** objects) and application data types.

Data type mapping set Summarizes all the data type maps and mode request type maps of a software component (SWC).

Data variant One of two or more differing data values that are generated into the same C code and can be switched during ECU run time using a calibratable variant ID variable. For example, the **Value** property of a gain parameter can have the variants 2, 3, and 4.

DataItemMapping (DIM) A DataItemMapping object is a DD object that references a ReplaceableDataItem (RDI) and a DD variable. It is used to define the DD variable object to map an RDI object to, and therefore also the implementation variable in the generated code.

DD child object The DD object below another DD object in the DD object tree.

DD data model The DD data model describes the object kinds, their properties and constraints as well as the dependencies between them.

DD file A DD file (*.dd) can be a DD project file or a partial DD file.

DD object Data item in the **Data Dictionary** that can contain DD child objects and DD properties.

DD object tree The tree that arranges all DD objects according to the DD data model.

DD project file A file containing the DD objects of a DD workspace.

DD root object The topmost DD object of the DD workspace.

DD subtree A part of the DD object tree containing a DD object and all its descendants.

DD workspace An independent organizational unit (central data container) and the largest entity that can be saved to file or loaded from a DD project file. Any number of DD workspaces is supported, but only the first (DD0) can be used for code generation.

Default enumeration constant Represents the default constant, i.e., the name of an enumerated value that is used for initialization if an initial value is required, but not explicitly specified.

Direct reuse The Code Generator adds the instance-specific variables to the reuse structure as leaf struct components.

E

ECU Abbreviation of *electronic control unit*.

ECU software The ECU software consists of all the software that runs on an ECU. It can be divided into the basic software, run-time environment (RTE), and the application layer.

ECU State Manager A piece of software that manages modes. An ECU state manager is part of the basic software.

Enhanceable Simulink block A Simulink® block that corresponds to a TargetLink simulation block, for example, the **Gain** block.

Enumerated value An enumerated value consists of an enumeration constant and a corresponding underlying integer value (enumeration value).

Enumeration constant An enumeration constant defines the name for an enumerated value.

Enumeration data type A data type with a specific name, a set of named enumerated values and a default enumeration constant.

Enumeration value An enumeration value defines the integer value for an enumerated value.

Event message Event messages are information units that are defined in a sender-receiver interface and exchanged between sender ports or receiver ports. They represent the control flow. On the receiver side, each event message is related to a buffer that queues the received messages.

Event semantics Communication of data elements with first-in-first-out semantics. Data elements are received in the same order they were sent. In simulations, TargetLink behaves as if data semantics was specified, even if you specified event semantics. However, TargetLink generates calls to the correct RTE API functions for data and event semantics.

ExchangeableWidth A DD object that defines code variants or improves code readability by using macros for signal widths.

Exclusive area Allows for specifying critical sections in the code that cannot preempt/interrupt each other. An exclusive area can be used to specify the mutual exclusion of runnables.

Executable application The generic term for offline simulation applications and real-time applications.

Explicit communication A communication mode in Classic AUTOSAR. The data is exchanged whenever data is required or provided.

Explicit object An explicit object is an object in production code that the Code Generator created from a direct specification made at a DD object or at a model element. For comparison, see implicit object.

Extern C Stateflow symbol A C symbol (function or variable) that is used in a Stateflow chart but that is defined in an external code module.

External code Existing C code files/modules from external sources (e.g., legacy code) that can be included by preprocessor directives and called by the C code generated by TargetLink. Unlike Custom code, external code is used as it is.

External container A container that is owned by the tool with that you are exchanging a software component but that is not the tool that triggers the container exchange. This container is used when you import files of a software component which were created or changed by the other tool.

F

Filter An algorithm that is applied to received data elements.

Fixed-Point Library A library that contains functions and macros for use in the generated production code.

Function AF The short form for an access function (AF) that is implemented as a C function.

Function algorithm object Generic term for either a MATLAB local function, the interface of a MATLAB local function or a local MATLAB variable.

Function class A class that represents group properties of functions that determine the function definition, function prototypes and function calls of a function in the generated production code. There are two types of function classes: predefined function class objects defined in the `/Pool/FunctionClasses` group in the DD and implicit function classes (default function classes) that can be influenced by templates in the DD.

Function code Code that is generated for a modular unit that represents functionality and can have abstract interfaces to be reused without changes in different contexts, e.g. in different integration models.

Function inlining The process of replacing a function call with the code of the function body during code generation by TargetLink via inline expansion. This reduces the function call overhead and enables further optimizations at the potential cost of larger code size.

Function interface An interface that describes how to pass the inputs and outputs of a function to the generated production code. It is described by the function signature.

Function subsystem A subsystem that is atomic and contains a Function block. When generating code, TargetLink generates it as a C function.

Functional Mock-up Unit (FMU) An archive file that describes and implements the functionality of a model based on the Functional Mock-up Interface (FMI) standard.

G

Global data store The specification of a DD DataStoreMemoryBlock object that references a variable and is associated with either a Simulink.Signal object or Data Store Memory block. The referenced variable must have a module specification and a fixed name and must be global and non-static. Because of its central specification in the Data Dictionary, you can use it across the boundaries of CGUs.

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 thatcan execute binary code compiled for the corresponding microprocessor. This allows the ISS to behave in the same way as the simulated microprocessor.

Integration model A model or TargetLink subsystem that contains modular units which it integrates to make a larger entity that provides its functionality.

Interface Describes the data elements, NvData, event messages, operations, or calibration parameters that are provided or required by a software component (SWC) via a port (AUTOSAR).

Internal behavior An element that represents the internal structure of an atomic software component (atomic SWC). It is characterized by the following entities and their interdependencies:

- Exclusive area
- Interrunnable variable (IRV)
- Per instance memory
- Per instance parameter
- Runnable
- RTE event
- Shared parameter

Interrunnable variable (IRV) Variable object for specifying communication between the runnables in one atomic software component (atomic SWC).

Interrupt service routine (ISR) function A function that implements an ISR and calls the step functions of the subsystems that are assigned by the user or by the TargetLink Code Generator during multirate code generation.

Intertask communication The flow of data between tasks and ISRs, tasks and tasks, and between ISRs and ISRs for multirate code generation.

Is service A property of an interface that indicates whether the interface is provided by a basic software service.

ISV Abbreviation for instance-specific variable.

L

Leaf bus element A leaf bus element is a subordinate bus element that is not a bus itself.

Leaf bus signal See also leaf bus element.

Leaf struct component A leaf struct component is a subordinate struct component that is not a struct itself.

Legacy function A function that contains a user-provided C function.

Library subsystem A subsystem that resides in a Simulink® library.

Local container A container that is owned by the tool that triggers the container exchange.

The tool that triggers the exchange transfers the files of a software component to this container when you export a software component. The external container is not involved.

Local MATLAB variable A variable that is generated when used on the left side of an assignment or in the interface of a MATLAB local function. TargetLink does not support different data types and sizes on local MATLAB variables.

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

M

Macro A literal representing a C preprocessor definition. Macros are used to provide a fixed sequence of computing instructions as a single program statement. Before code compilation, the preprocessor replaces every occurrence of the macro by its definition, i.e., by the code that it stands for.

Macro AF The short form for an access function (AF) that is implemented as a function-like preprocessor macro.

MATLAB code elements MATLAB code elements include MATLAB local functions and local MATLAB variables. MATLAB code elements are not available in the Simulink Model Explorer or the Property Manager.

MATLAB local function A function that is scoped to a MATLAB main function and located at the same hierarchy level. MATLAB local functions are treated like MATLAB main functions and have the same properties as the MATLAB main function by default.

MATLAB main function The first function in a MATLAB function file.

Matrix AF An access function resulting from a DD AccessFunction object whose VariableKindSpec property is set to `APPLY_TO_MATRIX`.

Matrix signal Collective term for 2-D signals implemented as matrix variable in production code.

Matrix variable Collective term for 2-D arrays in production code that implement 2-D signals.

Measurement Viewing and analyzing the time traces of calibration parameters and measurement variables, for example, to observe the effects of ECU parameter changes.

Measurement and calibration system A tool that provides access to an ECU for measurement and calibration. It requires information on the calibration parameters and measurement variables with the ECU code.

Measurement variable Any variable type that can be measured but not calibrated. The term *measurement variable* is independent of a variable type's dimension.

Memory mapping The process of mapping variables and functions to different memory sections.

Memory section A memory location to which the linker can allocate variables and functions.

Message Browser A TargetLink component for handling fatal (F), error (E), warning (W), note (N), and advice (A) messages.

MetaData files Files that store metadata about code generation. The metadata of each code generation unit (CGU) is collected in a DD Subsystem object that is written to the file system as a partial DD file called `<CGU>_SubsystemObject.dd`.

Method Behavior subsystem An atomic subsystem used to generate code for a method implementation. From the TargetLink perspective, this is an Adaptive AUTOSAR Function that can take arguments.

It contains a Function block whose AUTOSAR mode is set to `Adaptive` and whose Role is set to `Method Behavior`.

Method Call subsystem An atomic subsystem that is used to generate a method call in the code of an Adaptive AUTOSAR Function. The subsystem contains a Function block whose AUTOSAR mode is set to `Adaptive` and whose Role is set to `Method Call`. The subsystem interface is used to generate the function interface while additional model elements that are contained in the subsystem are only for simulation purposes.

Microcontroller family (MCF) A group of microcontroller units with the same processor, but different peripherals.

Microcontroller unit (MCU) A combination of a specific processor with additional peripherals, e.g. RAM or AD converters. MCUs with the same processor, but different peripherals form a microcontroller family.

MIL simulation A simulation method in which the function model is computed (usually with double floating-point precision) on the host computer as an executable specification. The simulation results serve as a reference for SIL simulations and PIL simulations.

MISRA Organization that assists the automotive industry to produce safe and reliable software, e.g., by defining guidelines for the use of C code in automotive electronic control units or modeling guidelines.

Mode An operating state of an ECU, a single functional unit, etc..

Mode declaration group Contains the possible operating states, for example, of an ECU or a single functional unit.

Mode manager A piece of software that manages modes. A mode manager can be implemented as a software component (SWC) of the application layer.

Mode request type map An entity that defines a mapping between a mode declaration group and a type. This specifies that mode values are instantiated in the software component (SWC)'s code with the specified type.

Mode switch event An RTE event that specifies to start or continue the execution of a runnable as a result of a mode change.

Model Compare A dSPACE software tool that identifies and visualizes the differences in the contents of Simulink/TargetLink models (including Stateflow). It can also merge the models.

Model component A model-based code generation unit (CGU).

Model element A model in MATLAB/Simulink consists of model elements that are TargetLink blocks, Simulink blocks, and Stateflow objects, and signal lines connecting them.

Model port A port used to connect a behavior model in ConfigurationDesk. In TargetLink, multiple model ports of the same kind (data in or data out) can be grouped in a model port block.

Model port block A block in ConfigurationDesk that has one or more model ports. It is used to connect the behavior model in ConfigurationDesk.

Model port variable A DD Variable object that represents a model port of a behavior model in ConfigurationDesk.

Model-dependent code elements Code elements that (partially) result from specifications made in the model.

Model-independent code elements Code elements that can be generated from specifications made in the Data Dictionary alone.

Modular unit A submodel containing functionality that is reusable and can be integrated in different integration models. The production code for the modular unit can be generated separately.

Module A DD object that specifies code modules, header files, and other arbitrary files.

Module specification The reference of a DD Module object at a Function Block (TargetLink Model Element Reference) block or DD object. The resulting code elements are generated into the module. See also production code and stub code.

ModuleOwnership A DD object that specifies an owner for a module (module owner) or module group, i.e. the owning code generation unit (CGU) that generates the production code for it or declares the module as external code that is not generated by TargetLink.

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 thats holds all relevant information about an ECU application, for example, for code generation.

TargetLink simulation block A block that processes signals during simulation. In most cases, it is a block from standard Simulink libraries but carries additional information required for production code generation.

TargetLink subsystem A subsystem from the TargetLink block library that defines a section of the Simulink model for which code must be generated by TargetLink.

Task A code section whose execution is managed by the real-time operating system. Tasks can be triggered periodically or based on events. Each task can call one or more SIC runnable functions.

Task function A function that implements a task and calls the functions of the subsystems which are assigned to the task by the user or via the TargetLink Code Generator during multirate code generation.

Term function A function that contains the code to be executed when the simulation finishes or the ECU application terminates.

Terminate function A runnable that finalizes a SWC, for example, by calling code that has to run before the application shuts down.

Timing event An RTE event that specifies to start or continue the execution of a runnable at constant time intervals.

tllib A TargetLink block library that is the source for creating TargetLink models graphically. Refer to How to Open the TargetLink Block Library (TargetLink Orientation and Overview Guide).

Transformer The Classic AUTOSAR entity used to perform a data transformation.

TransformerError The parameter passed by the run-time environment (RTE) if an error occurred in a data transformation. The `Std_TransformerError` is a struct whose components are the transformer class and the error code. If the error is a hard error, a special runnable is triggered via the TransformerHardErrorEvent to react to the error.

In AUTOSAR releases prior to R19-11 this struct was named `Rte_TransformerError`.

TransformerHardErrorEvent The RTE event that triggers the runnable to be used for responding to a hard TransformerError in a data transformation for client-server communication.

Type prefix A string written in front of the variable type of a variable definition/declaration, such as `MyTypePrefix Int16 MyVar`.

U

Unicode The most common standard for extended character sets is the Unicode standard. There are different schemes to encode Unicode in byte format, e.g., UTF-8 or UTF-16. All of these encodings support all Unicode characters. Scheme conversion is possible without losses. The only difference between these encoding schemes is the memory that is required to represent Unicode characters.

User data type (UDT) A data type defined by the user. It is placed in the Data Dictionary and can have associated constraints.

Utility blocks One of the categories of TargetLink blocks. The blocks in the category keep TargetLink-specific data, provide user interfaces, and control the simulation mode and code generation.

V

Validation Summary Shows unresolved model element data validation errors from all model element variables of the Property View. It lets you search, filter, and group validation errors.

Value copy AF An access function (AF) resulting from DD AccessFunction objects whose AccessFunctionKind property is set to `READ_VALUE_COPY` or `WRITE_VALUE_COPY`.

Variable access function An access function (AF) that *encapsulates the access* to a variable for reading or writing.

Variable class A set of properties that define the role and appearance of a variable in the generated production code, e.g. CAL for global calibratable variables.

VariantConfig A DD object in the Config area that defines the code variants and data variants to be used for simulation and code generation.

VariantItem A DD object in the DD Config area used to variant individual properties of DD Variable and ExchangeableWidth objects. Each variant of a property is associated with one variant item.

V-ECU implementation container (VECU) A file that consists of all the files required to build an offline simulation application (OSA) to use for simulation with VEOS.

V-ECU Manager A component of TargetLink that allows you to configure and generate a V-ECU implementation.

Vendor mode The operation mode of RTE generators that allows the generation of RTE code which contains vendor-specific adaptations, e.g., to reduce resource consumption. To be linkable to an RTE, the object code of an SWC must have been compiled against an application header that matches the RTE code generated by the specific RTE generator. This is the case because the data structures and types can be implementation-specific.

See also compatibility mode.

VEOS A dSPACE software platform for the C-code-based simulation of virtual ECUs and environment models on a PC.

Virtual ECU (V-ECU) Software that emulates a real ECU in a simulation scenario. The virtual ECU comprises components from the application and the basic software, and provides functionalities comparable to those of a real ECU.

Virtual ECU testing Offline and real-time simulation using virtual ECUs.

Virtual evaluation board (virtual EVB) A combination of an instruction set simulator (ISS) and a simulated periphery. This combination can be used for validation of generated production code in PIL simulation mode.

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.

A

AR_ARRAY_OF_STRUCT_DATA demo model 152
AR_COLLISION_DETECTION demo model 139
AR_MEMORY_MAPPING demo model 147
AR_NVDATA_TRANSFORMER demo model 135
AR_Poscontrol demo model 119
AUTODOC_GENERATION demo model 67
AUTOSAR Fuelsys demo model 130
AUTOSAR memory mapping 148

B

BUS_CC 85
BUS_STRUCT demo model 50

C

Calling external commands 226
CG_from_DD demo model 185
Common Program Data folder 8
CommonProgramDataFolder 8
Custom_Blocks demo model 14
custom_enhancement demo model 15

D

DCMC demo model 128
DD_CUSTOM_COMMAND demo model 225
DD_FILTER demo model 219
DD_ML_API demo model 221
DD_ML_IMPORTEXPORT demo model 222
demo model
- AR_ARRAY_OF_STRUCT_DATA 152
- AR_COLLISION_DETECTION 139
- AR_MEMORY_MAPPING 147
- AR_NVDATA_TRANSFORMER 135
- AR_Poscontrol 119
- AUTODOC_GENERATION 67
- AUTOSAR Fuelsys 130
- BUS_CC 85
- BUS_STRUCT 50
- CG_from_DD 185
- Custom_Blocks 14
- custom_enhancement 15
- DCMC 128
- DD_CUSTOM_COMMAND 225
- DD_FILTER 219
- DD_ML_API 221
- DD_ML_IMPORTEXPORT 222
- DYNAMIC_LUT 117
- EMBEDDING_EXTERNAL_CODE 64
- Enablsys 31
- ENUM_BLINKER 81
- ENUM_TYPES 78
- Fuelsys 99
- Fuelsys_Float 101
- FUNCTION_REUSE 56
- FUNCTION_VARIANTS 75
- MATLAB_CODE_STATEFLOW 229
- MATRIX 44
- Model_Referencing 181
- MODULAR_DEVELOPMENT 192
- MULTIPLE_INSTANCES_REFMODEL 60
- Multirate 177
- Online Parameter Modification 199
- PIPT1 22
- Poscontrol 35
- Prelookup_usr 113
- Preparation 9
- REPLACEABLE_DATA_ITEMS 187
- Signal injection 203
- Stateflow 97
- Struct_Interfaces 41
- Table1D 103
- TABLE1D_FCN 112
- Table1D_USR 106
- Table1D_USR_LOCAL 108
- Table2D 105
- Table2D_USR 110
- VARIABLE_INITIALIZATION 72
- VARIABLE_VECTOR_WIDTH 55
- VFCP 25

Documents folder 8
DocumentsFolder 8
DYNAMIC_LUT demo model 117

E

EMBEDDING_EXTERNAL_CODE demo model 64
Enablsys demo model 31
ENUM_BLINKER demo model 81
ENUM_TYPES demo model 78

F

Fuelsys demo model 99
Fuelsys_Float demo model 101
FUNCTION_REUSE demo model 56
FUNCTION_VARIANTS demo model 75

L

Local Program Data folder 8
LocalProgramDataFolder 8

M

MATLAB_CODE_STATEFLOW 229
MATRIX demo model 44
Model_Referencing demo model 181
MODULAR_DEVELOPMENT demo model 192
MULTIPLE_INSTANCES_REFMODEL demo model 60
Multirate demo model 177

P

PIPT1 demo model 22
Poscontrol demo model 35
Prelookup_usr demo model 113
Preparation demo model 9

R

REPLACEABLE_DATA_ITEMS demo model 187

S

Stateflow demo model 97
Struct_Interfaces demo model 41

T

Table1D demo model 103
TABLE1D_FCN demo model 112
Table1D_USR demo model 106
Table1D_USR_LOCAL demo model 108
Table2D demo model 105
Table2D_USR demo model 110
TargetLink feature
- applying block output variables 27
- applying record layouts to customize table data 104
- applying user-defined data types 26
- applying variant coding 28
- ASAP 2 file generation 37
- client-server communication using get operations 129
- client-server communication using set operations 130
- custom code 15
- custom look-up functions 107, 111
- enabled subsystems 32
- enhancing unsupported Simulink blocks 16
- Feature 1 65, 68, 73, 77, 79, 140, 194
- Feature 2 78
- floating-point code generation 102
- function-call-triggered subsystems 33
- functions shared between subsystems 112
- logging 24
- mergeable variables 34
- mode-dependent activation of runnables 133
- model referencing 184
- modeling interrunnable communication 124
- modeling sender-receiver communication 123
- multirate subsystems (RTOS) 178
- name macros 23
- nonscalar inputs/outputs 109
- online parameter modification 201
- package information 131
- parameter sharing 184
- pointers to structures 42
- preparing bitwise implementation 13
- preparing deadzone implementation 11
- preparing model referencing 9
- preparing S-function implementation 10
- preparing SWC parameters for calibration 133
- preparing user library 12
- preparing user library (alternative methods) 12
- Property Manager 100

referencing value property of DD variable objects 28
root function template 178
sender-receiver communication with struct-based data elements 132
signal injection 204
simulating runnable triggering 126
Stateflow 100
Stateflow support 98
SWC-internal parameters for calibration 125
TargetLink properties in stateflow 98
tunneling signals 205
usage of access function templates 29
using Function block 36
using work variables as loop indeces in custom code templates 19
variable classes 22
width-inheriting variables 110
width-invariant custom code template 18
workspace variables 105
worst-case autoscaling 24

V

VARIABLE_INITIALIZATION demo model 72
VARIABLE_VECTOR_WIDTH demo model 55
VFCP demo model 25